CSE332: Data Abstractions
Lecture 11: Hash Tables

Dan Grossman
Spring 2012
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

---

client

E int

$\longrightarrow$

hash table library

$\longrightarrow$

table-index

$\longrightarrow$

collision?

$\longrightarrow$

collision resolution

0

$\cdots$

TableSize – 1
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, $\lambda$, of a hash table is

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

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

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

• Each unsuccessful \texttt{find} compares against _____ items
• Each successful \texttt{find} compares against _____ items
More rigorous chaining analysis

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

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

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by random finds, then on average:
- Each unsuccessful \texttt{find} compares against $\lambda$ items
- Each successful \texttt{find} compares against $\lambda/2$ items

So we like to keep $\lambda$ 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 $(h(key) + 1) \% \text{TableSize}$. If full,
  – try $(h(key) + 2) \% \text{TableSize}$. If full,
  – try $(h(key) + 3) \% \text{TableSize}$. If full...

• Example: insert 38, 19, 8, 109, 10

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>1</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>
</tr>
<tr>
<td>3</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>
</tr>
<tr>
<td>5</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>
</tr>
<tr>
<td>7</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>/</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alternative: Use empty space in the table

- Another simple idea: 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
Alternative: Use empty space in the table

- Another simple idea: If $h(key)$ is already full,
  - try $(h(key) + 1) \% \text{TableSize}$. If full,
  - try $(h(key) + 2) \% \text{TableSize}$. If full,
  - try $(h(key) + 3) \% \text{TableSize}$. If full...

- Example: insert 38, 19, 8, 109, 10
**Alternative: Use empty space in the table**

- Another simple idea: If \( h(key) \) is already full,
  - try \( (h(key) + 1) \% \text{TableSize} \). If full,
  - try \( (h(key) + 2) \% \text{TableSize} \). If full,
  - try \( (h(key) + 3) \% \text{TableSize} \). If full...

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>109</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>38</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
Alternative: Use empty space in the table

- Another simple idea: 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

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
  - \( i^{\text{th}} \) probe was \((h(\text{key}) + i) \mod \text{TableSize}\)
- In general have some probe function \( f \) and use \( h(\text{key}) + f(i) \mod \text{TableSize} \)

Open addressing does poorly with high load factor \( \lambda \)
- 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]
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 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 $\lambda$ and has no trouble with $\lambda > 1$
Quadratic probing

• We can avoid primary clustering by changing the probe function
  \((h(key) + f(i)) \mod \text{TableSize}\)

• A common technique is 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

<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>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>89</td>
<td></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

0
1
2
3
4
5
6
7
8
9
### Quadratic Probing Example

Table Size = 10

Insert:
- 89
- 18
- 49
- 58
- 79

<table>
<thead>
<tr>
<th>0</th>
<th>49</th>
</tr>
</thead>
<tbody>
<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 \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

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

TableSize = 7

<table>
<thead>
<tr>
<th>Insert</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>(76 % 7 = 6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>(40 % 7 = 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>(48 % 7 = 6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>(5 % 7 = 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>(55 % 7 = 6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>(47 % 7 = 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)

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

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

TableSize = 7

Insert:
- 76 \ (76 \ % \ 7 = 6)
- 40 \ (40 \ % \ 7 = 5)
- 48 \ (48 \ % \ 7 = 6)
- 5 \ (5 \ % \ 7 = 5)
- 55 \ (55 \ % \ 7 = 6)
- 47 \ (47 \ % \ 7 = 5)

Doh!: For all \( n \), \( (n^2 + 5) \ % \ 7 \) is 0, 2, 5, or 6
- Excel shows it takes “at least” 50 probes and a pattern
- Proof uses induction and \( (n^2+5) \ % \ 7 = ((n-7)^2+5) \ % \ 7 \)
  - In fact, for all \( c \) and \( k \), \( (n^2+c) \ % \ k = ((n-k)^2+c) \ % \ k \)
From Bad News to Good News

• Bad news:
  – Quadratic probing can cycle through the same full indices, never terminating despite table not being full

• Good news:
  – If TableSize is prime and $\lambda < 1/2$, then quadratic probing will find an empty slot in at most $\text{TableSize}/2$ probes
  – So: If you keep $\lambda < 1/2$ and TableSize is prime, no need to detect cycles
    
    – Proof is posted in lecture11.txt
      • Also, slightly less detailed proof in textbook
      • Key fact: For prime $T$ and $0 < i, j < T/2$ where $i \neq j$,
        $(k + i^2) \% T \neq (k + j^2) \% T$ (i.e., no index repeat)
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:

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

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:
    • \( h(key) = key \mod p \)
    • \( g(key) = q - (key \mod q) \)
    • \( 2 < q < p \)
    • \( p \) and \( q \) are prime
More double-hashing facts

• Assume “uniform hashing”
  – Means probability of $g(\text{key1}) \% p == g(\text{key2}) \% p$ is $1/p$

• Non-trivial facts we won’t prove:
  Average # of probes given $\lambda$ (in the limit as $\text{TableSize} \to \infty$)
  – 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
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
Hashing and comparing

• Need to emphasize a critical detail:
  – We initially hash \( E \) to get a table index
  – While chaining or probing we compare to \( E \)
    • Just need equality testing (i.e., “is it what I want”)

• 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)
  ```

• 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

```java
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()!=(PolarPoint)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 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 \( \text{compare}(a,c) < 0 \)
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