CSE373: Data Structures & Algorithms
Lecture 12: Hash Collisions

Dan Grossman
Fall 2013
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

```
TableSize - 1
```

```
<table>
<thead>
<tr>
<th>client</th>
<th>int</th>
<th>hash table library</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>int</td>
<td>table-index collision? collision resolution</td>
</tr>
</tbody>
</table>
```

hash table

0

...
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`

<table>
<thead>
<tr>
<th>0</th>
<th>/</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>/</td>
</tr>
<tr>
<td>9</td>
<td>/</td>
</tr>
</tbody>
</table>
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
  – Maybe 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 `find` compares against ____ items
• Each successful `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 $(h(key) + 1) \mod \text{TableSize}$. 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

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>1</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>
</tr>
<tr>
<td>3</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>
</tr>
<tr>
<td>5</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>
</tr>
<tr>
<td>7</td>
<td>/</td>
<td></td>
<td></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>
<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>
</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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>/</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>/</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>9</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
**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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>109</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>
</tr>
<tr>
<td>3</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>
</tr>
<tr>
<td>5</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>
</tr>
<tr>
<td>7</td>
<td>/</td>
<td></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>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>19</td>
<td></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 $h(key)$ 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
  - $i^{th}$ probe was $(h(key) + i) \mod TableSize$
- 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 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
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 $\to \infty$)
  – 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 (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
  \((h(key) + f(i)) \% \text{TableSize}\)

• A common technique is quadratic probing:
  \(f(i) = i^2\)
  – So probe sequence is:
    • 0\(^{th}\) probe: \(h(key) \% \text{TableSize}\)
    • 1\(^{st}\) probe: \((h(key) + 1) \% \text{TableSize}\)
    • 2\(^{nd}\) probe: \((h(key) + 4) \% \text{TableSize}\)
    • 3\(^{rd}\) probe: \((h(key) + 9) \% \text{TableSize}\)
    • ...
    • \(i^{th}\) probe: \((h(key) + i^2) \% \text{TableSize}\)

• 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></td>
</tr>
<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></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>89</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TableSize = 10
Insert:
- 89
- 18
- 49
- 58
- 79
Quadratic Probing Example

<table>
<thead>
<tr>
<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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>89</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>

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

<table>
<thead>
<tr>
<th>0</th>
<th>49</th>
</tr>
</thead>
<tbody>
<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>
### Quadratic Probing Example

<table>
<thead>
<tr>
<th>Table Size = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Insert:</strong></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>

<table>
<thead>
<tr>
<th>0</th>
<th>49</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>58</td>
</tr>
<tr>
<td>3</td>
<td>79</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>
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

Table Size $= 7$

Insert:
- 76 $(76 \mod 7 = 6)$
- 40 $(40 \mod 7 = 5)$
- 48 $(48 \mod 7 = 6)$
- 5 $(5 \mod 7 = 5)$
- 55 $(55 \mod 7 = 6)$
- 47 $(47 \mod 7 = 5)$
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>
</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

TableSize = 7

<table>
<thead>
<tr>
<th>Insert</th>
<th></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>

<table>
<thead>
<tr>
<th>0</th>
<th>48</th>
</tr>
</thead>
<tbody>
<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>
Another Quadratic Probing Example

TableSize = 7

Insert:

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

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 \( n \), \( ((n\times n) + 5) \% 7 \) 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

• 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 $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in at most $\frac{\text{TableSize}}{2}$ probes
  – So: If you keep $\lambda < \frac{1}{2}$ and TableSize is prime, no need to detect cycles

  – Optional: Proof is posted in lecture12.txt
    • Also, slightly less detailed proof in textbook
    • Key fact: For prime $T$ and $0 < i, j < T/2$ where $i \neq j$,
      $(k + i^2) \mod T \neq (k + j^2) \mod T$ (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: $h(key) \mod \text{TableSize}$
- $1^{st}$ probe: $(h(key) + g(key)) \mod \text{TableSize}$
- $2^{nd}$ probe: $(h(key) + 2*g(key)) \mod \text{TableSize}$
- $3^{rd}$ 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 $g(key1) \% p == g(key2) \% p$ is $1/p$

• Non-trivial facts we won’t prove:
  Average # of probes given $\lambda$ (in the limit as TableSize $\rightarrow \infty$)
  – Unsuccessful search (intuitive): $\frac{1}{1-\lambda}$
  – Successful search (less intuitive): $rac{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
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