CSE373: Data Structures & Algorithms
Lecture 14: Hash Collisions

Linda Shapiro
Winter 2015
Announcements

• Wednesday: Review List and go over answers to HW 4. It may not be turned in later than 2:30 Wednesday.
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

---

![Diagram of hash table process]

**E** → **int** → **table-index** → **collision?** → **collision resolution** → **hash table library** → **hash table**

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 \textbf{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. tree
  – Move-to-front upon access
  – 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

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

Under chaining, the average number of elements per bucket is ___
More rigorous chaining analysis

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

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

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 $\text{find}$ 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 find compares against $\lambda$ 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 `find` compares against $\lambda$ items
- Each successful `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: No lists; Use empty space in the table

- Another simple idea: If $h(\text{key})$ is already full,
  - try $(h(\text{key}) + 1) \% \text{TableSize}$. If full,
  - try $(h(\text{key}) + 2) \% \text{TableSize}$. If full,
  - try $(h(\text{key}) + 3) \% \text{TableSize}$. 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>
<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>38</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>
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 $h(key)$ 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

<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>/</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

<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>
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

| 0 | 8 |
| 1 | 109 |
| 2 | 10 |
| 3 | / |
| 4 | / |
| 5 | / |
| 6 | / |
| 7 | / |
| 8 | 38 |
| 9 | 19 |
Probing hash tables

Trying the next spot is called probing (also called open addressing)

- 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 $O(1)$
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 \( \text{TableSize} \to \infty \))
  – 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 (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
  \[(h(\text{key}) + f(i)) \mod \text{TableSize}\]

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

- Intuition: Probes quickly “leave the neighborhood”
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>
</table>

TableSize=10

Insert:
89
18
49
58
79
Quadratic Probing Example

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

Table Size = 10

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

<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>18</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>
Quadratic Probing Example

Table Size = 10

Insert:
89
18
49
58
79
Quadratic Probing Example

TableSize=10
Insert:
89
18
49
58
79

<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>
### Quadratic Probing Example

Table Size = 10

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

<table>
<thead>
<tr>
<th>0</th>
<th>Table Size = 10</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 \quad (76 \% 7 = 6)
40 \quad (40 \% 7 = 5)
48 \quad (48 \% 7 = 6)
5 \quad (5 \% 7 = 5)
55 \quad (55 \% 7 = 6)
47 \quad (47 \% 7 = 5)
## Another Quadratic Probing Example

Table Size = 7

<table>
<thead>
<tr>
<th>Insert</th>
<th>Hash Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>76</td>
<td>6</td>
</tr>
<tr>
<td>40</td>
<td>5</td>
</tr>
<tr>
<td>48</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>55</td>
<td>6</td>
</tr>
<tr>
<td>47</td>
<td>5</td>
</tr>
</tbody>
</table>

0 | 76

1

2

3

4

5

6
Another Quadratic Probing Example

TableSize = 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:
- 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

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

Table Size = 7

Insert:
76  \quad (76 \mod 7 = 6)
40  \quad (40 \mod 7 = 5)
48  \quad (48 \mod 7 = 6)
5   \quad ( 5 \mod 7 = 5)
55  \quad (55 \mod 7 = 6)
47  \quad (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)

Doh!: For all \( n \), \((n*n) + 5) \ % \ 7\ is \ 0, \ 2, \ 5, \ or \ 6\n
- Excel shows takes “at least” 50 probes and a pattern.
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
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 $f(i) = i \times g(key)$

Probe sequence:

- 0th probe: $h(key) \ % \ TableSize$
- 1st probe: $(h(key) + g(key)) \ % \ TableSize$
- 2nd probe: $(h(key) + 2 \times g(key)) \ % \ TableSize$
- 3rd probe: $(h(key) + 3 \times g(key)) \ % \ TableSize$
- ...
- ith probe: $(h(key) + i \times g(key)) \ % \ TableSize$

Detail: Make sure $g(key)$ cannot be 0
Double-hashing analysis

- **Intuition:** Because each probe is “jumping” by $g(\text{key})$ 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:
    - $h(\text{key}) = \text{key} \mod p$
    - $g(\text{key}) = q - (\text{key} \mod q)$
    - $2 < q < p$
    - $p$ and $q$ are prime
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 probing, 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
Summary

- Hashing gives us approximately $O(1)$ behavior for both insert and find.
- Collisions are what ruin it.
- There are several different collision strategies.
  - **Chaining** just uses linked lists pointed to by the hash table bins.
  - **Probing** uses various methods for computing the next index to try if the first one is full.
  - **Rehashing** makes a new, bigger table.
  - If the table is kept reasonably empty (small load factor), and the hash function works well, we will get the kind of behavior we want.