CSE 332: Data Structures & Parallelism
Lecture 11: More Hashing

Ruth Anderson
Autumn 2018
Today

- Dictionaries
  - Hashing
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

```
E  int  table-index  collision?  collision resolution
client    hash table library    …
```

TableSize –1
Hashing Choices

1. Choose a Hash function
2. Choose TableSize
3. Choose a Collision Resolution Strategy from these:
   - Separate Chaining
   - Open Addressing
     • Linear Probing
     • Quadratic Probing
     • Double Hashing

• Other issues to consider:
  – Deletion?
  – What to do when the hash table gets “too full”?
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

- 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
Open Addressing: Linear Probing

- 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: Linear Probing

- 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

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

- 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>0</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>109</td>
</tr>
<tr>
<td>2</td>
<td>10</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 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: $(h(key) + i) \mod \text{TableSize}$
- In general have some probe function $f$ and:
  - $i^{th}$ probe: $(h(key) + f(i)) \mod \text{TableSize}$

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”
Open Addressing: Linear Probing

What about \texttt{find}? If value is in table? If not there? Worst case?

What about \texttt{delete}?

How does open addressing with linear probing compare to separate chaining?
Open Addressing: 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 |

- 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 probe sequences
- Called *primary clustering*
- Saw the start of a cluster in our linear probing 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 $\rightarrow \infty$)
  – 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)
Analysis in chart form

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

• By comparison, separate chaining performance is linear in $\lambda$ and has no trouble with $\lambda > 1$
Open Addressing: Linear probing

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

- For linear probing:
  \[f(i) = i\]

- So probe sequence is:
  - 0th probe: \[h(key) \mod \text{TableSize}\]
  - 1st probe: \[(h(key) + 1) \mod \text{TableSize}\]
  - 2nd probe: \[(h(key) + 2) \mod \text{TableSize}\]
  - 3rd probe: \[(h(key) + 3) \mod \text{TableSize}\]
  - ...
  - ith probe: \[(h(key) + i) \mod \text{TableSize}\]
Open Addressing: Quadratic probing

- We can avoid primary clustering by changing the probe function…

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

- For 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

Table Size = 10
Insert:
89
18
49
58
79

ith probe: \((h(key) + i^2) \mod \text{TableSize}\)
Quadratic Probing Example

TableSize = 10

insert(89)
Quadratic Probing Example

TableSize = 10

insert(89)

insert(18)
Quadratic Probing Example

TableSize = 10
insert(89)
insert(18)
insert(49)
**Quadratic Probing Example**

TableSize = 10

- insert(89)
- insert(18)
- insert(49)

49 % 10 = 9 collision!

(49 + 1) % 10 = 0

- insert(58)

<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>49</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>18</td>
<td>89</td>
</tr>
</tbody>
</table>
Quadratic Probing Example

Table Size = 10

- Insert (89)
- Insert (18)
- Insert (49)
- Insert (58)

58 % 10 = 8 collision!

(58 + 1) % 10 = 9 collision!

(58 + 4) % 10 = 2

Insert (79)
Quadratic Probing Example

Table Size = 10

- Insert(89)
- Insert(18)
- Insert(49)
- Insert(58)
- Insert(79)

79 % 10 = 9 collision!
(79 + 1) % 10 = 0 collision!
(79 + 4) % 10 = 3
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  \( (47 \% 7 = 5) \)

ith probe: \( h(key) + i^2 \) \% TableSize
Another Quadratic Probing Example

TableSize = 7

Insert:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<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>76</td>
<td>76</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>48</td>
<td>48</td>
<td>48</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>55</td>
<td>55</td>
</tr>
<tr>
<td>5</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>47</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>76</td>
<td>76</td>
<td>76</td>
<td>76</td>
<td>76</td>
</tr>
</tbody>
</table>

ith probe: \((h(\text{key}) + i^2) \mod \text{TableSize}\)
### Another Quadratic Probing Example

TableSize = 7

**Insert:**

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></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>40</td>
</tr>
<tr>
<td>6</td>
<td>76</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Hash (mod TableSize)</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>

**ith probe:** \(h(key) + i^2 \mod \text{TableSize}\)
Another Quadratic Probing Example

TableSize = 7

Insert:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>48</td>
<td></td>
</tr>
<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>40</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>76</td>
<td></td>
</tr>
</tbody>
</table>

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

ith probe: \( h(key) + i^2 \mod \text{TableSize} \)
Another Quadratic Probing Example

TableSize = 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></td>
</tr>
<tr>
<td>3</td>
<td></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>

\[ \text{ith probe: } (h(key) + i^2) \% \text{ TableSize} \]
### Another Quadratic Probing Example

TableSize = 7

<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>

Insert:

<table>
<thead>
<tr>
<th>76</th>
<th>(76 % 7 = 6)</th>
</tr>
</thead>
<tbody>
<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

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>

(47 + 1) % 7 = 6 collision!
(47 + 4) % 7 = 2 collision!
(47 + 9) % 7 = 0 collision!
(47 + 16) % 7 = 0 collision!
(47 + 25) % 7 = 2 collision!

10/22/2018
Another Quadratic Probing Example

insert(47) will always fail here. Why?

<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>

For all $i$, $(5 + i^2) \mod 7$ is 0, 2, 5, or 6

Proof uses induction and

$(5 + i^2) \mod 7 = (5 + (i - 7)^2) \mod 7$

In fact, for all $c$ and $k$,

$(c + i^2) \mod k = (c + (i - k)^2) \mod k$
From bad news to good news

Bad News:
• After TableSize quadratic probes, we cycle through the same indices

Good News:
• If TableSize is prime and λ < ½, then quadratic probing will find an empty slot in at most TableSize/2 probes
• So: If you keep λ < ½ and TableSize is prime, no need to detect cycles
• Proof posted in lecture11.txt (slightly less detailed proof in textbook)
  – For prime T and 0 ≤ i, j ≤ T/2 where i ≠ j,
    \[(h(key) + i^2) \mod T \neq (h(key) + j^2) \mod T\]
    That is, if T is prime, the first T/2 quadratic probes map to different locations
Quadratic Probing: Success guarantee for $\lambda < \frac{1}{2}$

- If size is prime and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in size/2 probes or fewer.

  - show for all $0 \leq i, j \leq \text{size}/2$ and $i \neq j$
    
    $$(h(x) + i^2) \mod \text{size} \neq (h(x) + j^2) \mod \text{size}$$

  - by contradiction: suppose that for some $i \neq j$:
    $$(h(x) + i^2) \mod \text{size} = (h(x) + j^2) \mod \text{size}$$

    $\Rightarrow \quad i^2 \mod \text{size} = j^2 \mod \text{size}$$

    $\Rightarrow \quad (i^2 - j^2) \mod \text{size} = 0$

    $\Rightarrow \quad [(i + j)(i - j)] \mod \text{size} = 0$

    BUT size does not divide $(i-j)$ or $(i+j)$

How can $i+j = 0$ or $i+j = \text{size}$ when:

$i \neq j$ and $0 \leq i, j \leq \text{size}/2$?

Similarly how can $i-j = 0$ or $i-j = \text{size}$?

First size/2 probes will be distinct, and if less than half of table is full then after size/2 probes you will find one of those empty spots

Size would need to divide one of these
Clustering reconsidered

• Quadratic probing does not suffer from primary clustering: As we resolve collisions we are not merely growing “big blobs” by adding one more item to the end of a cluster, we are looking \( i^2 \) locations away, for the next possible spot.

• But quadratic probing does not help resolve collisions between keys that initially hash to the same index
  – Any 2 keys that initially hash to the same index will have the same series of moves after that looking for any empty spot
  – 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 $key$, $h(key) == g(key)$

$$(h(key) + f(i)) \% TableSize$$

– For double hashing:

$$f(i) = i \times g(key)$$

– So probe sequence is:

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

– Detail: Make sure $g(key)$ can’t be $0$
Open Addressing: Double Hashing

<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></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>

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

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

13
28
33
147
43

Hash Functions:

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

\[ \text{ith probe: } (h(key) + i \times g(key)) \mod \text{TableSize} \]
Double Hashing

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

13
28
33
147
43

T = 10 (TableSize)

Hash Functions:

h(key) = key mod T

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

ith probe: (h(key) + i*g(key)) % TableSize
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 → \( g(33) = 1 + 3 \mod 9 = 4 \)
- 147
- 43
Double Hashing

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

- 13
- 28
- 147
- 43

Hash Functions:

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

\[ \text{ith probe: } (h(key) + i \times g(key)) \mod \text{TableSize} \]

<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>13</td>
<td></td>
<td></td>
<td></td>
<td>33</td>
<td>28</td>
<td>147</td>
</tr>
</tbody>
</table>

147 \( \Rightarrow \) \( g(147) = 1 + 14 \mod 9 = 6 \)
Double Hashing

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

- 13
- 28
- 33
- 147

\[ g(147) = 1 + 14 \mod 9 = 6 \]

\[ g(43) = 1 + 4 \mod 9 = 5 \]

\[ \text{We have a problem:} \]

\[ 3 + 0 = 3 \quad 3 + 5 = 8 \quad 3 + 10 = 13 \]

\[ 3 + 15 = 18 \quad 3 + 20 = 23 \]

Hash Functions:

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

\[ T = 10 \ (\text{TableSize}) \]

ith probe: \((h(key) + i \times g(key)) \mod \text{TableSize}\)
Double-hashing analysis

- **Intuition**: Since each probe is “jumping” by $g(\text{key})$ each time, we “leave the neighborhood” and “go different places from other initial collisions”

But, as in quadratic probing, we could still have a problem where we are not "safe" due to an infinite loop despite room in table

- It is known that this cannot happen in at least one case:
  
  For primes $p$ and $q$ such that $2 < q < p$
  
  $h(\text{key}) = \text{key} \mod p$
  
  $g(\text{key}) = q - (\text{key} \mod q)$
Yet another reason to use a prime TableSize

- So, for double hashing
  \[ i^{th} \text{ probe: } (h(\text{key}) + i \times g(\text{key})) \mod \text{TableSize} \]
- Say \( g(\text{key}) \) divides Tablesize
  - That is, there is some integer \( x \) such that \( x \times g(\text{key}) = \text{Tablesize} \)
  - After \( x \) probes, we’ll be back to trying the same indices as before
- Ex:
  - Tablesize=50
  - \( g(\text{key}) = 25 \)
  - Probing sequence:
    - \( h(\text{key}) \)
    - \( h(\text{key}) + 25 \)
    - \( h(\text{key}) + 50 = h(\text{key}) \)
    - \( h(\text{key}) + 75 = h(\text{key}) + 25 \)
- Only 1 & itself divide a prime
More double-hashing facts

• Assume “uniform hashing”
  – Means probability of $g(key_1) \mod p = g(key_2) \mod p$ is $1/p$

• Non-trivial facts we won’t prove:
  Average # of probes given $\lambda$ (in the limit as 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
Charts

Uniform Hashing

- Red line: uniform hashing not found
- Blue line: uniform hashing found

Load Factor

Linear Probing

- Purple line: linear probing not found
- Green line: linear probing found

Load Factor
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 table fills

Why use it:
- Less memory allocation?
  - Some run-time overhead for allocating linked list (or whatever) nodes; open addressing 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 over

• With separate 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 probably won’t grow more than 20-30 times, and then calculate after that
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
  – Iterate over old table: O(n)
  – n inserts / calls to the hash function: n \cdot O(1) = O(n)

• 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); saving h(key) only helps by a constant factor
Hashing and comparing

- Our use of int key can lead to us overlooking a critical detail:
  - We initially hash \( E \) to get a table index
  - While chaining or probing we need to determine if this is the \( E \) that I am looking for. Just need equality testing.

- So a hash table needs a hash function and a equality testing
  - In the Java library each object has an `equals` method and a `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
Example
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 $\text{compare}(a, c) < 0$
A Generally Good hashCode()

```java
int result = 17; // start at a prime

for each 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;

return result;
```
Final word on hashing

• The hash table is one of the most important data structures
  – Efficient find, insert, and delete
  – Operations based on sorted order are not so efficient
  – Useful in many, many real-world applications
  – Popular topic for job interview questions
• Important to use a good hash function
  – Good distribution, Uses enough of key’s values
  – Not overly expensive to calculate (bit shifts good!)
• Important to keep hash table at a good size
  – Prime #
  – Preferable $\lambda$ depends on type of table
• 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