CSE 3332

Data Abstractions
Hashing: Part 2

"the eagle flies at midnight"

MD5...

2886dba4
c8c519f1
e6e44416
9580f18b
Hash Tables

- Provides $O(1)$ core Dictionary operations \textbf{(on average)}
- We call the key space the “universe”: $U$ and the Hash Table $T$
- We should use this data structure \textbf{only} when we expect $|U| >> |T|$
- (Or, the key space is non-integer values.)

Another Consideration?

\textbf{What do we do when } $\lambda$ \textbf{(the load factor)} \textbf{gets too large?}
Hashing Choices

1. Choose a hash function

2. Choose a table size

3. Choose a collision resolution strategy
   - Separate Chaining
   - Linear Probing
   - Quadratic Probing
   - Double Hashing
   - Other issues to consider:

4. Choose an implementation of deletion

5. Choose a $\lambda$ that means the table is “too full”

We discussed the first few of these last time. We’ll discuss the rest today.
Definition (Collision)

A collision is when two distinct keys map to the same location in the hash table.

A good hash function attempts to avoid as many collisions as possible, but they are inevitable.

How do we deal with collisions?

There are multiple strategies:
- Separate Chaining
- Open Addressing
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
Open Addressing

Definition (Open Addressing)

Open Addressing is a type of collision resolution strategy that resolves collisions by choosing a different location when the natural choice is full.
Open Addressing

Definition (Open Addressing)

Open Addressing is a type of collision resolution strategy that resolves collisions by choosing a different location when the natural choice is full.

There are many types of open addressing. Here’s the key ideas:

- We **must** be able to duplicate the path we took.
- We want to use **all** the spaces in the table.
- We want to avoid putting lots of keys close together.
Open Addressing

**Definition (Open Addressing)**

Open Addressing is a type of collision resolution strategy that resolves collisions by choosing a different location when the natural choice is full.

There are many types of open addressing. Here’s the key ideas:

- We **must** be able to duplicate the path we took.
- We want to use **all** the spaces in the table.
- We want to avoid putting lots of keys close together.

It turns out some of these are difficult to achieve...

**Strategy #1: Linear Probing**

```python
i = 0;
while (index in use) {
    try (h(key) + i) % |T|
}
```
Open Addressing

Definition (Open Addressing)

Open Addressing is a type of collision resolution strategy that resolves collisions by choosing a different location when the natural choice is full.

There are many types of open addressing. Here’s the key ideas:

- We must be able to duplicate the path we took.
- We want to use all the spaces in the table.
- We want to avoid putting lots of keys close together.

It turns out some of these are difficult to achieve...

Strategy #1: Linear Probing

```java
i = 0;
while (index in use) {
    try (h(key) + i) % |T|
}
```

Example

Insert 38, 19, 8, 109, 10 into a hash table with hash function \( h(x) = x \) and linear probing

(Items with the same hash code are the same color)
Definition (Open Addressing)

**Open Addressing** is a type of collision resolution strategy that resolves collisions by choosing a different location when the natural choice is full.

There are many types of open addressing. Here’s the key ideas:

- We **must** be able to duplicate the path we took.
- We want to use **all** the spaces in the table.
- We want to avoid putting lots of keys close together.

It turns out some of these are difficult to achieve...

**Strategy #1: Linear Probing**

1. `i = 0;`
2. `while (index in use) {`
   3. `try (h(key) + i) % |T|`
4. `}`

**Example**

Insert 38, 19, 8, 109, 10 into a hash table with hash function $h(x) = x$ and **linear probing**

(Items with the same hash code are the same color)
**Open Addressing**

**Definition (Open Addressing)**

Open Addressing is a type of collision resolution strategy that resolves collisions by choosing a different location when the natural choice is full.

There are many types of open addressing. Here’s the key ideas:

- **We must** be able to duplicate the path we took.
- We want to use all the spaces in the table.
- We want to avoid putting lots of keys close together.

It turns out some of these are difficult to achieve...

**Strategy #1: Linear Probing**

```plaintext
i = 0;
while (index in use) {
    try (h(key) + i) % |T|
}
```

**Example**

Insert 38, 19, 8, 109, 10 into a hash table with hash function \( h(x) = x \) and linear probing

(Items with the same hash code are the same color)
Open Addressing

Definition (Open Addressing)

Open Addressing is a type of collision resolution strategy that resolves collisions by choosing a different location when the natural choice is full.

There are many types of open addressing. Here’s the key ideas:

- We **must** be able to duplicate the path we took.
- We want to use **all** the spaces in the table.
- We want to avoid putting lots of keys close together.

It turns out some of these are difficult to achieve...

Strategy #1: Linear Probing

```java
i = 0;
while (index in use) {
    try (h(key) + i) % |T|
}
```

Example

Insert 38, 19, 8, 109, 10 into a hash table with hash function
\[ h(x) = x \] and **linear probing**

|---|----|------|------|------|------|------|------|------|------|------|------|

(Items with the same hash code are the same color)
Open Addressing

Definition (Open Addressing)

Open Addressing is a type of collision resolution strategy that resolves collisions by choosing a different location when the natural choice is full.

There are many types of open addressing. Here’s the key ideas:

- We **must** be able to duplicate the path we took.
- We want to use **all** the spaces in the table.
- We want to avoid putting lots of keys close together.

It turns out some of these are difficult to achieve...

Strategy #1: Linear Probing

```
1 i = 0;
2 while (index in use) {
3     try (h(key) + i) % |T|
4 }
```

Example

Insert 38, 19, 8, 109, 10 into a hash table with hash function 
\( h(x) = x \) and **linear probing**

(Items with the same hash code are the same color)
Strategy #1: Linear Probing

```java
int i = 0;
while (index in use) {
    try (h(key) + i) % |T|
}
```

Example

Insert 38, 19, 8, 109, 10 into a hash table with hash function
\[ h(x) = x \] and **linear probing**

Other Operations with Linear Probing

- **insert?** Finds the **next** open spot. The worst case is \( O(n) \)
- **find?**
Strategy #1: Linear Probing

```java
i = 0;
while (index in use) {
    try (h(key) + i) % |T|
}
```

Example

Insert 38, 19, 8, 109, 10 into a hash table with hash function

\[ h(x) = x \]

and linear probing

(Items with the same hash code are the same color)

Other Operations with Linear Probing

- **insert**: Finds the **next** open spot. The worst case is \( O(n) \)
- **find**: We have to retrace our steps. If the insert chain was \( k \) long, then \( \text{find} \in O(k) \).
- **delete**?
Strategy #1: Linear Probing

1. \( i = 0; \)
2. \( \text{while (index in use)} \{ \)
3. \( \quad \text{try (h(key) + i) \% |T|} \)
4. \( \}\)

Example

Insert 38, 19, 8, 109, 10 into a hash table with hash function \( h(x) = x \) and linear probing

<table>
<thead>
<tr>
<th>8</th>
<th>109</th>
<th>10</th>
<th>38</th>
<th>19</th>
</tr>
</thead>
</table>
Strategy #1: Linear Probing

```plaintext
i = 0;
while (index in use) {
    try (h(key) + i) % |T|
}
```

Example

Insert 38, 19, 8, 109, 10 into a hash table with hash function $h(x) = x$ and linear probing

<table>
<thead>
<tr>
<th>8</th>
<th>109</th>
<th>10</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>38</th>
<th>19</th>
</tr>
</thead>
</table>

(Items with the same hash code are the same color)

Other Operations with Linear Probing

- **insert?** Finds the next open spot. The worst case is $O(n)$
- **find?** We have to retrace our steps. If the insert chain was $k$ long, then find $\in O(k)$.
- **delete?** We don’t have a choice; we must use lazy deletion. What happens if we delete 19 and then do find(109) in our example?

<table>
<thead>
<tr>
<th>8</th>
<th>109</th>
<th>10</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>38</th>
<th>X</th>
</tr>
</thead>
</table>
Which Criteria Does Linear Probing Meet?

- We want to use all the spaces in the table.

Primary Clustering

Primary Clustering is when different keys collide to form one big group. In linear probing, we expect to get $O(\log n)$ size clusters. This is really bad! But, how bad, really?
Which Criteria Does Linear Probing Meet?

- We want to use all the spaces in the table.
  Yes! Linear probing will fill the whole table.
- We want to avoid putting lots of keys close together.
Analyzing Linear Probing

Which Criteria Does Linear Probing Meet?

- We want to use all the spaces in the table.
  
  **Yes! Linear probing will fill the whole table.**

- We want to avoid putting lots of keys close together.
  
  **Uh... not so much**

Primary Clustering

**Primary Clustering** is when different keys collide to form one big group.

<table>
<thead>
<tr>
<th>8</th>
<th>109</th>
<th>10</th>
<th>101</th>
<th>20</th>
<th></th>
<th>36</th>
<th>19</th>
</tr>
</thead>
</table>

Think of this as “clusters of many colors”. Even though these keys are all different, they end up in a giant cluster.
Analyzing Linear Probing

Which Criteria Does Linear Probing Meet?

- We want to use all the spaces in the table.
  Yes! Linear probing will fill the whole table.
- We want to avoid putting lots of keys close together.
  Uh... not so much

Primary Clustering

Primary Clustering is when different keys collide to form one big group.

<table>
<thead>
<tr>
<th>6</th>
<th>109</th>
<th>10</th>
<th>101</th>
<th>20</th>
<th></th>
<th></th>
<th>36</th>
<th>19</th>
</tr>
</thead>
</table>
Load Factor & Space Usage

Note that $\lambda \leq 1$, and we will eventually get to $\lambda = 1$.

Average Number of Probes

Unsuccessful Search

$$\frac{1}{2} \left(1 + \frac{1}{(1 - \lambda)^2}\right)$$

Successful Search

$$\frac{1}{2} \left(1 + \frac{1}{(1 - \lambda)}\right)$$
There's nothing theoretically wrong with open addressing that forces primary clustering. We’d like a different (easy to compute) function to probe with. That is:

**Open Addressing In General**

Choose a new function $f(x)$ and then probe with

$$(h(key) + f(i)) \mod |T|$$
There's nothing theoretically wrong with open addressing that forces primary clustering. We'd like a different (easy to compute) function to probe with. That is:

**Open Addressing In General**

Choose a new function $f(x)$ and then probe with

$$(h(key) + f(i)) \mod |T|$$

**Strategy #2: Quadratic Probing**

```plaintext
i = 0;
while (index in use) {
  try (h(key) + i^2) % |T|
}
```
There's nothing theoretically wrong with open addressing that forces primary clustering. We’d like a different (easy to compute) function to probe with. That is:

**Open Addressing In General**

Choose a new function $f(x)$ and then probe with

$$(h(key) + f(i)) \mod |T|$$

**Strategy #2: Quadratic Probing**

```
1  i = 0;
2  while (index in use) {
3      try (h(key) + i^2) % |T|
4  }
```

**Example**

Insert 89, 18, 49, 58, 79 into a hash table with hash function $h(x) = x$ and **quadratic probing**

<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><strong>89</strong></td>
</tr>
</tbody>
</table>
There's nothing theoretically wrong with open addressing that forces primary clustering. We'd like a different (easy to compute) function to probe with. That is:

### Open Addressing In General

Choose a new function $f(x)$ and then probe with

$$(h(key) + f(i)) \mod |T|$$

### Strategy #2: Quadratic Probing

1. $i = 0$
2. while (index in use) {
   3. try $(h(key) + i^2) \mod |T|$
3. }

#### Example

Insert 89, 18, 49, 58, 79 into a hash table with hash function $h(x) = x$ and **quadratic probing**

<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><strong>18</strong></td>
<td></td>
<td><strong>89</strong></td>
</tr>
</tbody>
</table>
There's nothing theoretically wrong with open addressing that forces primary clustering. We'd like a different (easy to compute) function to probe with. That is:

**Open Addressing In General**

Choose a new function \( f(x) \) and then probe with

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

**Strategy #2: Quadratic Probing**

```plaintext
1 i = 0;
2 while (index in use) {
3    try (h(key) + i^2) \mod |T|
4 }
```

**Example**

Insert 89, 18, 49, 58, 79 into a hash table with hash function \( h(x) = x \) and quadratic probing

\[
h(58) \xrightarrow{i=0} 58 + 0^2 \equiv 8
\]

\[
h(58) \xrightarrow{i=1} 58 + 1^2 \equiv 9
\]

\[
h(58) \xrightarrow{i=2} 58 + 2^2 \equiv 2
\]
There's nothing theoretically wrong with open addressing that forces primary clustering. We’d like a different (easy to compute) function to probe with. That is:

**Open Addressing In General**

Choose a new function $f(x)$ and then probe with

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

**Strategy #2: Quadratic Probing**

1. $i = 0$
2. while (index in use) {
   3. try $(h(\text{key}) + i^2) \mod |T|$
3. }

**Example**

Insert 89, 18, 49, 58, 79 into a hash table with hash function $h(x) = x$ and **quadratic probing**

\[
\begin{align*}
    h(79) & \xRightarrow{i=0} 79 + 0^2 \equiv 9 \\
    & \xRightarrow{i=1} 79 + 1^2 \equiv 0 \\
    & \xRightarrow{i=2} 79 + 2^2 \equiv 3
\end{align*}
\]
There's nothing theoretically wrong with open addressing that forces primary clustering. We'd like a different (easy to compute) function to probe with. That is:

**Open Addressing In General**

Choose a new function $f(x)$ and then probe with

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

**Strategy #2: Quadratic Probing**

```plaintext
1 i = 0;
2 while (index in use) {
3     try (h(key) + i^2) \mod |T|
4 }
```

**Example**

Insert 89, 18, 49, 58, 79 into a hash table with hash function $h(x) = x$ and **quadratic probing**.
**Strategy #2: Quadratic Probing**

```java
i = 0;
while (index in use) {
    try (h(key) + i\(^2\)) \% |T|
}
```

**Example**

Insert 76, 40, 48, 5, 55, 47 into a hash table with hash function \( h(x) = x \) and **quadratic probing**

<table>
<thead>
<tr>
<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>76</td>
<td></td>
</tr>
</tbody>
</table>

\[
h(76) \overset{i=0}{\rightarrow} 76 + 0^2 \equiv_{7} 6
\]
Another Quadratic Probing Example

Strategy #2: Quadratic Probing

```
1 i = 0;
2 while (index in use) {
3     try (h(key) + i^2) % |T|
4 }
```

Example

Insert 76, 40, 48, 5, 55, 47 into a hash table with hash function

\[ h(x) = x \] and **quadratic probing**

\[
\begin{array}{cccccccc}
\end{array}
\]

\[ h(40) \overset{i=0}{\rightarrow} 40 + 0^2 \equiv 5 \]
Strategy #2: Quadratic Probing

```java
i = 0;
while (index in use) {
    try (h(key) + i^2) % |T|
}
```

Example
Insert 76, 40, 48, 5, 55, 47 into a hash table with hash function \( h(x) = x \) and \textbf{quadratic probing}

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td>76</td>
<td></td>
</tr>
</tbody>
</table>

\[
h(48) \xrightarrow{i=0} 48 + 0^2 \equiv_7 6 \\
\xrightarrow{i=1} 48 + 1^2 \equiv_7 0
\]

We will never get a 1 or a 4! This means we will never be able to insert 47. What's going on?
Strategy #2: Quadratic Probing

```java
i = 0;
while (index in use) {
    try (h(key) + i^2) % |T|
}
```

Example

Insert 76, 40, 48, 5, 55, 47 into a hash table with hash function \( h(x) = x \) and **quadratic probing**

<table>
<thead>
<tr>
<th>48</th>
<th>5</th>
<th>40</th>
<th>76</th>
</tr>
</thead>
</table>

\[
egin{align*}
h(5) & \rightarrow 5 + 0^2 \equiv_7 5 \\
in=1 & \rightarrow 5 + 1^2 \equiv_7 6 \\
in=2 & \rightarrow 5 + 2^2 \equiv_7 2
\end{align*}
\]
Another Quadratic Probing Example

Strategy #2: Quadratic Probing

```
1 i = 0;
2 while (index in use) {
3     try (h(key) + i^2) % |T|
4 }
```

Example

Insert 76, 40, 48, 5, 55, 47 into a hash table with hash function $h(x) = x$ and **quadratic probing**

We will never get a 1 or a 4! This means we will never be able to insert 47. What's going on?

<table>
<thead>
<tr>
<th>48</th>
<th>5</th>
<th>55</th>
<th>40</th>
<th>76</th>
</tr>
</thead>
</table>

$h(55)_{\rightarrow}^{i=0} 55 + 0^2 \equiv_7 6$

$i=1 \rightarrow 55 + 1^2 \equiv_7 0$

$i=2 \rightarrow 55 + 2^2 \equiv_7 3$
Another Quadratic Probing Example

Strategy #2: Quadratic Probing

```java
int i = 0;
while (index in use) {
    try (h(key) + i^2) % |T|
}
```

Example

Insert 76, 40, 48, 5, 55, 47 into a hash table with hash function $h(x) = x$ and quadratic probing.

<table>
<thead>
<tr>
<th>48</th>
<th>5</th>
<th>55</th>
<th>40</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>48</td>
<td>5</td>
<td>55</td>
<td>40</td>
<td>76</td>
</tr>
</tbody>
</table>

$h(47)$

- $i=0$ $47 + 0^2 \equiv_7 5$
- $i=1$ $47 + 1^2 \equiv_7 6$
- $i=2$ $47 + 2^2 \equiv_7 2$
- $i=3$ $47 + 3^2 \equiv_7 0$
- $i=4$ $47 + 4^2 \equiv_7 0$
- $i=5$ $47 + 5^2 \equiv_7 2$
Strategy #2: Quadratic Probing

```java
i = 0;
while (index in use) {
    try (h(key) + i^2) % |T|
}
```

Example

Insert 76, 40, 48, 5, 55, 47 into a hash table with hash function $h(x) = x$ and **quadratic probing**

```
<table>
<thead>
<tr>
<th>48</th>
<th>5</th>
<th>55</th>
<th>40</th>
<th>76</th>
</tr>
</thead>
</table>
```

$h(47) \quad i=0 \rightarrow 47 + 0^2 \equiv_7 5$

$i=1 \rightarrow 47 + 1^2 \equiv_7 6$

$i=2 \rightarrow 47 + 2^2 \equiv_7 2$

$i=3 \rightarrow 47 + 3^2 \equiv_7 0$

$i=4 \rightarrow 47 + 4^2 \equiv_7 0$

$i=5 \rightarrow 47 + 5^2 \equiv_7 2$

We will never get a 1 or a 4!
Another Quadratic Probing Example

Strategy #2: Quadratic Probing

```java
i = 0;
while (index in use) {
    try (h(key) + i^2) % |T|
}
```

Example

Insert 76, 40, 48, 5, 55, 47 into a hash table with hash function $h(x) = x$ and **quadratic probing**

<table>
<thead>
<tr>
<th></th>
<th>48</th>
<th></th>
<th>5</th>
<th>55</th>
<th></th>
<th>40</th>
<th>76</th>
</tr>
</thead>
<tbody>
<tr>
<td>T[0]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>T[4]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$h(47) \quad i=0 \rightarrow 47 + 0^2 \equiv_7 5$

$\quad i=1 \rightarrow 47 + 1^2 \equiv_7 6$

$\quad i=2 \rightarrow 47 + 2^2 \equiv_7 2$

$\quad i=3 \rightarrow 47 + 3^2 \equiv_7 0$

$\quad i=4 \rightarrow 47 + 4^2 \equiv_7 0$

$\quad i=5 \rightarrow 47 + 5^2 \equiv_7 2$

We will never get a 1 or a 4!

This means we will never be able to insert 47. What’s going on?
Why Does \text{insert}(47) Fail?

For all $i$, $(5 + i^2) \mod 7 \in \{0, 2, 5, 6\}$. The proof is by induction. This actually generalizes:

For all $c, k$, $(c + i^2) \mod k = (c + (i - k)^2) \mod k$
Quadratic Probing: Table Coverage

Why Does $\text{insert}(47)$ Fail?

For all $i$, $(5 + i^2) \mod 7 \in \{0, 2, 5, 6\}$. The proof is by induction. This actually generalizes:

$$\text{For all } c, k, \ (c + i^2) \mod k = (c + (i - k)^2) \mod k$$

So, quadratic probing doesn’t always fill the table.
Why Does \textit{insert}(47) Fail?

For all $i$, $(5 + i^2) \mod 7 \in \{0, 2, 5, 6\}$. The proof is by induction. This actually generalizes:

$$\text{For all } c, k, (c + i^2) \mod k = (c + (i - k)^2) \mod k$$

So, quadratic probing doesn’t always \textbf{fill the table}.

The Good News!

If $|T|$ is prime and $\lambda < \frac{1}{2}$, then quadratic probing will find an empty slot in at most $\frac{|T|}{2}$ probes. So, if we keep $\lambda < \frac{1}{2}$, we don’t need to detect cycles. The proof will be posted on the website.

So, does quadratic probing completely fix \textit{clustering}?
With linear probing, we saw **primary clustering** (keys hashing near each other). Quadratic Probing fixes this by “jumping”. Unfortunately, we still get **secondary clustering**: 

**Secondary Clustering** is when different keys hash to the same place and follow the same probing sequence.

```
39    
```

Think of this as long probing chains of the same color. The keys all start at the same place; so, the chain gets really long.

We can avoid secondary clustering by using a probe function that depends on the key.
Strategy #3: Double Hashing

```java
i = 0;
while (index in use) {
    try (h(key) + i*g(key)) % |T|
}
```

We insist \( g(x) \neq 0 \).

---

Example

Insert 13, 28, 33, 147, 43 into a hash table with:

- \( h(x) = x \)
- \( g(x) = 1 + \left( \frac{x}{|T|} \right) \mod (|T| - 1) \)

using **double hashing**
Strategy #3: Double Hashing

```
1 i = 0;
2 while (index in use) {
3     try (h(key) + i*g(key)) % |T|
4 }
```

We insist $g(x) \neq 0$.

Example

Insert 13, 28, 33, 147, 43 into a hash table with:

- $h(x) = x$
- $g(x) = 1 + \left(\frac{x}{|T|}\right) \mod (|T| - 1)$

using double hashing
Strategy #3: Double Hashing

```plaintext
i = 0;
while (index in use) {
    try (h(key) + i*g(key)) % |T|,
}
```

We insist \( g(x) \neq 0 \).

Example

Insert 13, 28, 33, 147, 43 into a hash table with:

- \( h(x) = x \)
- \( g(x) = 1 + \left( \frac{x}{|T|} \right) \mod (|T| - 1) \)

using **double hashing**

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

\[
\begin{align*}
h(33) &: \quad i=0 \\
      &\rightarrow 33 + 0 \equiv 3 \\
&\rightarrow 33 + 1(1 + 3 \mod 9) \equiv 7
\end{align*}
\]
Strategy #3: Double Hashing

```
i = 0;
while (index in use) {
    try (h(key) + i * g(key)) % |T|
}
```

We insist $g(x) \neq 0$.

Example

Insert 13, 28, 33, 147, 43 into a hash table with:

- $h(x) = x$
- $g(x) = 1 + \left( \frac{x}{|T|} \right) \mod (|T| - 1)$

using double hashing

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

$\begin{align*}
h(147) & \overset{i=0}{\longrightarrow} 147 + 0 \equiv 7 \\
& \overset{i=1}{\longrightarrow} 147 + 1(1 + 14 \mod 9) \equiv 3 \\
& \overset{i=1}{\longrightarrow} 147 + 2(1 + 14 \mod 9) \equiv 9
\end{align*}$
Strategy #3: Double Hashing

```java
i = 0;
while (index in use) {
    try (h(key) + i * g(key)) % |T|
}
```

We insist \( g(x) \neq 0 \).

Example

Insert 13, 28, 33, 147, 43 into a hash table with:
- \( h(x) = x \)
- \( g(x) = 1 + \left( \frac{x}{|T|} \right) \mod (|T| - 1) \)

using **double hashing**

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

\[
h(43) \overset{i = 0}{\rightarrow} 43 + 0 \equiv 3
\]

\[
\overset{i = 1}{\rightarrow} 43 + 1(1 + 4 \mod 9) \equiv 8
\]

\[
\overset{i = 1}{\rightarrow} 43 + 2(1 + 4 \mod 9) \equiv 3
\]

\[
\overset{i = 1}{\rightarrow} 43 + 3(1 + 4 \mod 9) \equiv 8
\]
Strategy #3: Double Hashing

```java
i = 0;
while (index in use) {
    try (h(key) + i * g(key)) % |T| } 
```

We insist \( g(x) \neq 0 \).

---

Example

Insert 13, 28, 33, 147, 43 into a hash table with:

- \( h(x) = x \)
- \( g(x) = 1 + \left( \frac{x}{|T|} \right) \mod (|T| - 1) \)

using **double hashing**

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

\[
h(43) \quad \overset{i=0}{\rightarrow} \quad 43 + 0 \equiv 3 \\
\quad \overset{i=1}{\rightarrow} \quad 43 + 1(1 + 4 \mod 9) \equiv 8 \\
\quad \overset{i=1}{\rightarrow} \quad 43 + 2(1 + 4 \mod 9) \equiv 3 \\
\quad \overset{i=1}{\rightarrow} \quad 43 + 3(1 + 4 \mod 9) \equiv 8 
\]

We got stuck again!
**Double Hashing Analysis**

### Filling the Table

Just like with Quadratic Probing, we sometimes hit an infinite loop with double hashing. We will not get an infinite loop in the case with primes $p, q$ such that $2 < q < p$:

- $h(\text{key}) = \text{key mod } p$
- $g(\text{key}) = q - (\text{key mod } q)$

### Uniform Hashing

For double hashing, we assume **uniform hashing** which means:

$$\Pr[g(\text{key}_1) \mod p = g(\text{key}_2) \mod p] = \frac{1}{p}$$

### Average Number of Probes

<table>
<thead>
<tr>
<th>Unsuccessful Search</th>
<th>Successful Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\frac{1}{1 - \lambda}$</td>
<td>$\frac{1}{\lambda} \ln \left( \frac{1}{1 - \lambda} \right)$</td>
</tr>
</tbody>
</table>

This is way better than linear probing.
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 is Tricky!
- Clustering issues
- Doesn’t always use the whole table
- Why Use it?
  - Less memory allocation
  - Easier data representation

Now, let’s move on to resizing the table.
Rehashing

When $\lambda$ is too big, create a bigger table and copy over the items.
When $\lambda$ is too big, create a bigger table and copy over the items

**When To Resize**

- With separate chaining, we decide when to resize (should be $\lambda \leq 1$)
- With open addressing, we need to keep $\lambda < \frac{1}{2}$

**New Table Size?**

- Like always, we want around "twice as big"...
- But it should still be prime
- So, choose the next prime about twice as big

**How To Resize**

- Go through table, do standard insert for each into new table:
  - Iterate over old table:
    - $O(n)$
  - $n$ inserts/calls to the hash function:
    - $n \times O(1) = O(n)$
  - But this is amortized $O(1)$
When $\lambda$ is too big, create a bigger table and copy over the items

When To Resize
- With separate chaining, we decide when to resize (should be $\lambda \leq 1$)
- With open addressing, we need to keep $\lambda < \frac{1}{2}$

New Table Size?
- Like always, we want around “twice as big”
- ... but it should still be prime
- So, choose the next prime about twice as big

How To Resize
When $\lambda$ is too big, create a bigger table and copy over the items

**When To Resize**
- With separate chaining, we decide when to resize (should be $\lambda \leq 1$)
- With open addressing, we need to keep $\lambda < \frac{1}{2}$

**New Table Size?**
- Like always, we want around “twice as big”
- ... but it should still be prime
- So, choose the next prime about twice as big

**How To Resize**
Go through table, do standard insert for each into new table:
- Iterate over old table: $O(n)$
- $n$ inserts / calls to the hash function: $n \times O(1) = O(n)$
- But this is amortized $O(1)$
A hash function isn’t enough! We have to compare items:
- With separate chaining, we have to loop through the list checking if the item is what we’re looking for
- With open addressing, we need to know when to stop probing
A hash function isn’t enough! We have to compare items:

- With separate chaining, we have to loop through the list checking if the item is what we’re looking for
- With open addressing, we need to know when to stop probing

We have two options for this: equality testing or comparison testing.

- In Project 2, you will use both types.
- In Java, each Object has an equals method and a hashCode method

```java
1 class Object {
2     boolean equals(Object o) {...}
3     int hashCode() {...}
4     ...
5 }
```
For any class, it must be the case that:

- If `a.equals(b)`, then `a.hashCode() == b.hashCode()`

- If `a.compareTo(b) == 0`, then `a.hashCode() == b.hashCode()`

- If `a.compareTo(b) < 0`, then `b.compareTo(a) > 0`

- If `a.compareTo(b) == 0`, then `b.compareTo(a) == 0`

- If `a.compareTo(b) < 0` and `b.compareTo(c) < 0`, then `a.compareTo(c) < 0`
A Good Hashcode

```java
int result = 17; // start at a prime
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;
return result;
```
Hash Tables are one of the most important data structures
- Efficient find, insert, and delete
- 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 keys values
- Not overly expensive to calculate (bit shifts good!)

Important to keep hash table at a good size
- Prime Size
- $\lambda$ depends on type of table

What we skipped: perfect hashing, universal hash functions, hopscotch hashing, cuckoo hashing