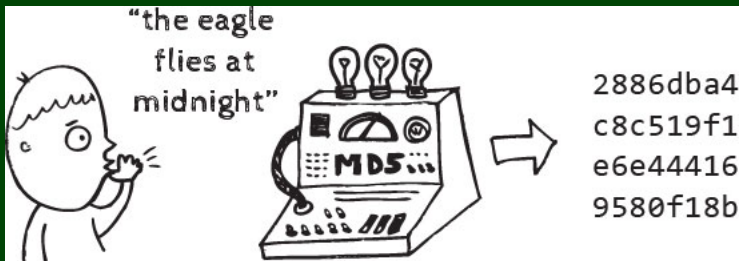


CS 2

Introduction to Programming Methods

Hashing: Part 2



Hash Tables

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



Another Consideration?

What do we do when λ (the load factor) gets too large?

- 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 λ 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

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.

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.

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 }
```


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)

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**

								38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

(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**

8								38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

(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**

8	109							38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

(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**

8	109	10						38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

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

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**

8	109	10						38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

(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 $\mathcal{O}(n)$
- find?

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**

8	109	10						38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

(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 $\mathcal{O}(n)$
- find? We have to retrace our steps. If the insert chain was k long, then $\text{find} \in \mathcal{O}(k)$.
- delete?

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**

8	109	10						38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

(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 $\mathcal{O}(n)$
- find? We have to retrace our steps. If the insert chain was k long, then $\text{find} \in \mathcal{O}(k)$.
- delete? We don't have a choice; we **must** use lazy deletion. What happens if we delete 19 and then do $\text{find}(109)$ in our example?

8	109	10						38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

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**

8	109	10						38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

(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 $\mathcal{O}(n)$
- find? We have to retrace our steps. If the insert chain was k long, then $\text{find} \in \mathcal{O}(k)$.
- delete? We don't have a choice; we **must** use lazy deletion. What happens if we delete 19 and then do $\text{find}(109)$ in our example?

8	109	10						38	X
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

Which Criteria Does Linear Probing Meet?

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

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.

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.

8	109	10	101	20				38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

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

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.

8	109	10	101	20				38	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

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

In linear probing, we expect to get $\mathcal{O}(\lg n)$ size clusters.

This is really bad! But, how bad, really?

Load Factor & Space Usage

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

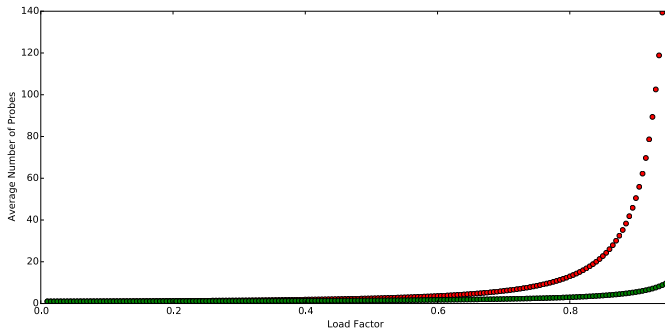
Average Number of Probes

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

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

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

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

Strategy #2: Quadratic Probing

```
1 i = 0;
2 while (index in use) {
3     try (h(key) + i2) % |T|
4 }
```


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

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

Strategy #2: Quadratic Probing

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

Example

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

									89
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

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

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

Strategy #2: Quadratic Probing

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

Example

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

								18	89
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

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

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

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3   try (h(key) + i2) % |T|  
4 }
```

Example

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

49								18	89
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

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

$$\xrightarrow{i=1} 58 + 1^2 \equiv 9$$

$$\xrightarrow{i=2} 58 + 2^2 \equiv 2$$

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

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

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3   try (h(key) + i2) % |T|  
4 }
```

Example

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

49		58						18	89
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

$$h(79) \xrightarrow{i=0} 79 + 0^2 \equiv 9$$

$$\xrightarrow{i=1} 79 + 1^2 \equiv 0$$

$$\xrightarrow{i=2} 79 + 2^2 \equiv 3$$

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

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

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3   try (h(key) + i2) % |T|  
4 }
```

Example

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

49		58	79					18	89
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3     try (h(key) + i2) % |T|  
4 }
```

Example

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

						76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

$$h(76) \xrightarrow{i=0} 76 + 0^2 \equiv_7 6$$

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3     try (h(key) + i2) % |T|  
4 }
```

Example

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

					40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

$$h(40) \xrightarrow{i=0} 40 + 0^2 \equiv 5$$

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3     try (h(key) + i2) % |T|  
4 }
```

Example

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

48					40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

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

$$\xrightarrow{i=1} 48 + 1^2 \equiv_7 0$$

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3     try (h(key) + i2) % |T|  
4 }
```

Example

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

48		5			40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

$$h(5) \xrightarrow{i=0} 5 + 0^2 \equiv_7 5$$

$$\xrightarrow{i=1} 5 + 1^2 \equiv_7 6$$

$$\xrightarrow{i=2} 5 + 2^2 \equiv_7 2$$

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3     try (h(key) + i2) % |T|  
4 }
```

Example

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

48		5	55		40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

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

$$\xrightarrow{i=1} 55 + 1^2 \equiv_7 0$$

$$\xrightarrow{i=2} 55 + 2^2 \equiv_7 3$$

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3     try (h(key) + i2) % |T|  
4 }
```

Example

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

48		5	55		40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

$$h(47) \xrightarrow{i=0} 47 + 0^2 \equiv_7 5$$

$$\xrightarrow{i=1} 47 + 1^2 \equiv_7 6$$

$$\xrightarrow{i=2} 47 + 2^2 \equiv_7 2$$

$$\xrightarrow{i=3} 47 + 3^2 \equiv_7 0$$

$$\xrightarrow{i=4} 47 + 4^2 \equiv_7 0$$

$$\xrightarrow{i=5} 47 + 5^2 \equiv_7 2$$

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3   try (h(key) + i2) % |T|  
4 }
```

Example

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

48		5	55		40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

$$h(47) \xrightarrow{i=0} 47 + 0^2 \equiv_7 5$$

$$\xrightarrow{i=1} 47 + 1^2 \equiv_7 6$$

$$\xrightarrow{i=2} 47 + 2^2 \equiv_7 2$$

$$\xrightarrow{i=3} 47 + 3^2 \equiv_7 0$$

$$\xrightarrow{i=4} 47 + 4^2 \equiv_7 0$$

$$\xrightarrow{i=5} 47 + 5^2 \equiv_7 2$$

We will never get a 1 or a 4!

Strategy #2: Quadratic Probing

```
1 i = 0;  
2 while (index in use) {  
3     try (h(key) + i2) % |T|  
4 }
```

Example

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

48		5	55		40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

$$h(47) \xrightarrow{i=0} 47 + 0^2 \equiv_7 5$$

$$\xrightarrow{i=1} 47 + 1^2 \equiv_7 6$$

$$\xrightarrow{i=2} 47 + 2^2 \equiv_7 2$$

$$\xrightarrow{i=3} 47 + 3^2 \equiv_7 0$$

$$\xrightarrow{i=4} 47 + 4^2 \equiv_7 0$$

$$\xrightarrow{i=5} 47 + 5^2 \equiv_7 2$$

We will never get a 1 or a 4!

This means we will never be able to insert 47. What's going on?

48		5	55		40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

Why Does `insert(47)` Fail?

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

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

48		5	55		40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

Why Does `insert(47)` Fail?

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

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

So, quadratic probing doesn't always **fill the table**.

48		5	55		40	76
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]

Why Does `insert(47)` Fail?

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

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

So, quadratic probing doesn't always **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 **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

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

39			29					9	19
T[0]	T[1]	T[2]	T[3]	T[4]	T[5]	T[6]	T[7]	T[8]	T[9]

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**.

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.

When λ is too big, create a bigger table and copy over the items

When λ 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?

When λ 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 λ 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: $\mathcal{O}(n)$
- n inserts / calls to the hash function: $n \times \mathcal{O}(1) = \mathcal{O}(n)$
- But this is amortized $\mathcal{O}(1)$

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 project05, you will use both types.
- In Java, each Object has an equals method and a hashCode method

```
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`

```
1 int result = 17; // start at a prime
2 foreach field f
3   int fieldHashCode =
4     boolean: (f ? 1: 0)
5     byte, char, short, int: (int) f
6     long: (int) (f ^ (f >>> 32))
7     float: Float.floatToIntBits(f)
8     double: Double.doubleToLongBits(f), then above
9     Object: object.hashCode()
10    result = 31 * result + fieldHashCode;
11 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
 - λ depends on type of table
- What we skipped: perfect hashing, universal hash functions, hopscotch hashing, cuckoo hashing