Hash Tables

Lecture 9 - EECS 214

Exam Logistics

- Exam next Monday (5/7/2018) in the usual time and place.
  - You will have the entire time slot to take the exam, but there's a class right after, so we can't let you go over, sorry.
- Practice exam out on Canvas. Solutions to be posted in a few days.
- Review session Wednesday (5/2) from 8 - 10 pm in Tech LR2
- Second review session this Saturday (5/5) from 6 - 8 pm, also in Tech LR2

Dictionaries

Data structures that hold an association between pairs of objects: a **key** and a **value**.

Also called:

- ***map***
- ***mapping***
- ***associative array***

Simplified Interface:

`Dictionary.Store(`*`key, val`*`)`

- Adds the key to the dictionary with the associated value
- For now, assume that duplicate keys just update the value

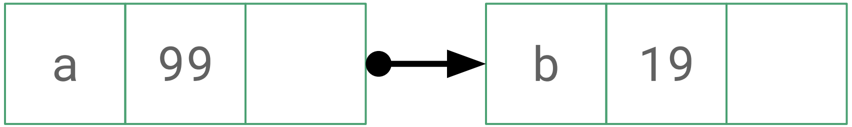
`Dictionary.Lookup(`*`key`*`)`

- Returns the value associated with the key (or null, broadly)

Dictionaries - Association List

- The simplest possible implementation of a dictionary is a **linked list** of **key/value** pairs

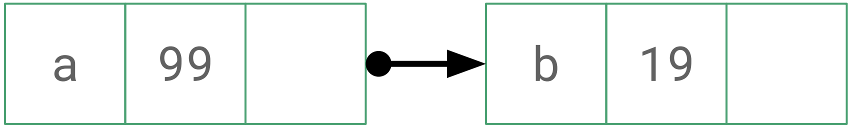
```
class AssocList {
    int key;
    object value;
    AssocList next;
}
```



Dictionaries - Association List

- Not super great: `O(n)` lookup time
- We looked at **binary search trees last week**, those have `O(log n)` lookup time
- Today (and next time), we'll look at data structures that have `O(1)` lookup
  - *maaagggiiiccc*

```
class AssocList {
    int key;
    object value;
    AssocList next;
}
```



Dictionaries - Array

Just kidding! We've already seen a data structure that has `O(1)` lookup.

- Arrays can be thought of as a **special kind of dictionary**
- Keys are always **"small" integers**: `{ 0,1,...,n }`
- Excellent performance: `O(1)` for all operations
- Limited utility because of limited keys

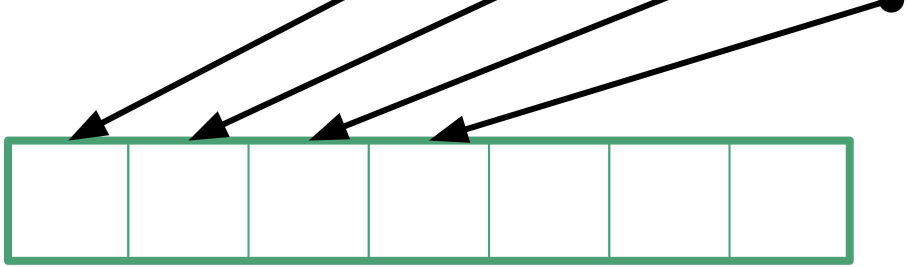
.4	99	1	10	.3	19	.7

Pretending Everything is an Array.com

Direct Addressing

- **Assign numbers** to all possible keys in advance
- Make an array as the set of all keys
- **Store value** in the appropriate cell

Fruit	Key
Dragonfruit	0
Persimmon	1
Kumquat	2
Lemon	3



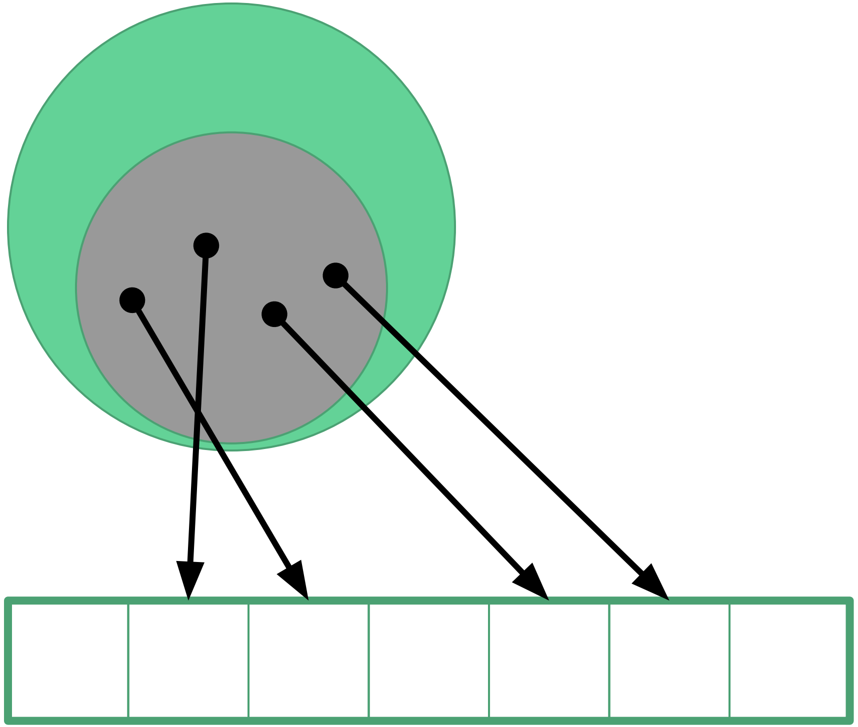
Direct Addressing

- More formally, let **U** be the ***universe*** of keys
- Define a one-to-one and onto "***hash***" function:

`h: U → { 0,1,...,|U| - 1 }`

- For any key u ∈ U, store the value in

`a[h(u)]`

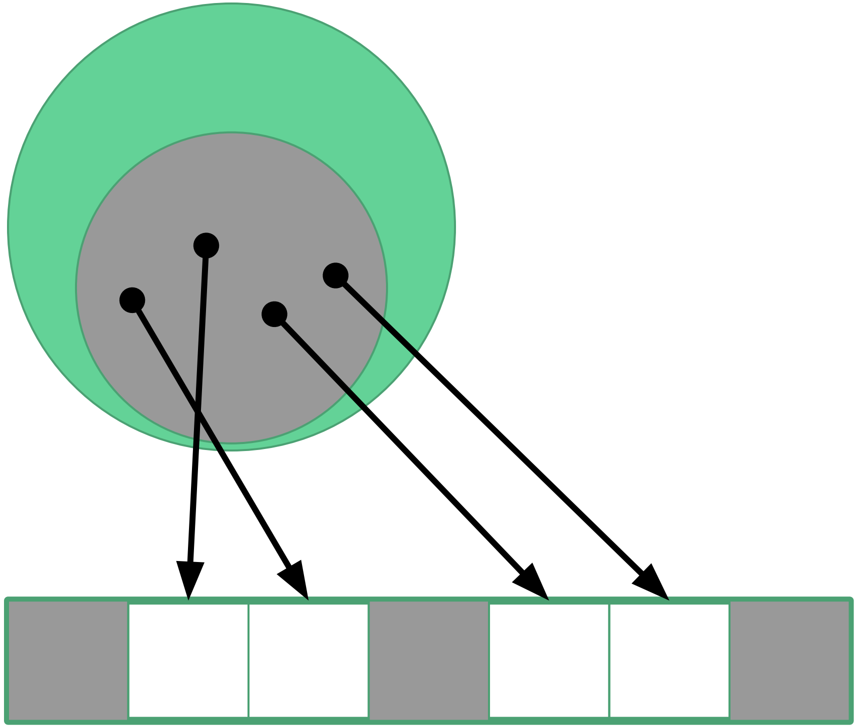


Direct Addressing

- The array is **as big as the key space**

This is a problem when:

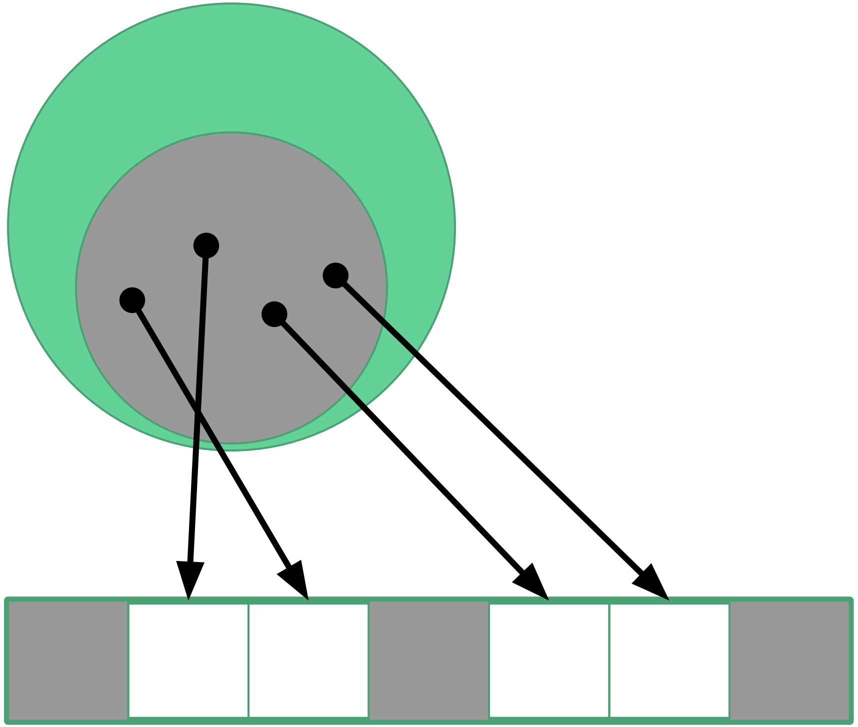
- The dictionary is **sparse**
  - Many fewer entries than possible keys
  - The array will be way bigger than it needs to be
- The key space is **infinite**
- 👏 **pigeonhole** principle 👏



Direct Addressing

What if we just use a smaller array and **hope for the best**?

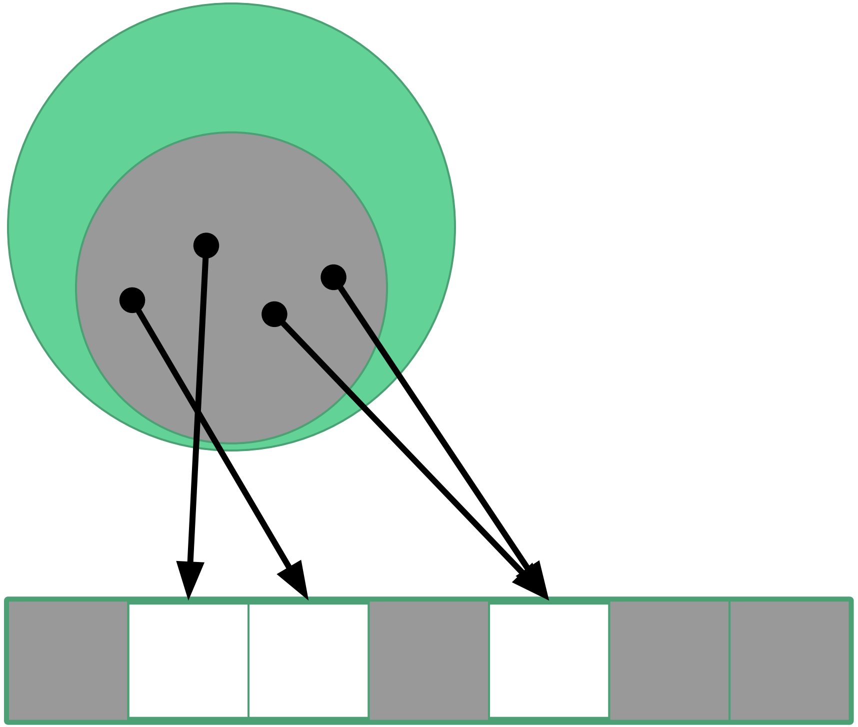
Aside: Hoping for the best works surprisingly often in computer science. Take 336 for more on this.



Direct Addressing - Shrinking the Array

If the array is smaller than the key space:

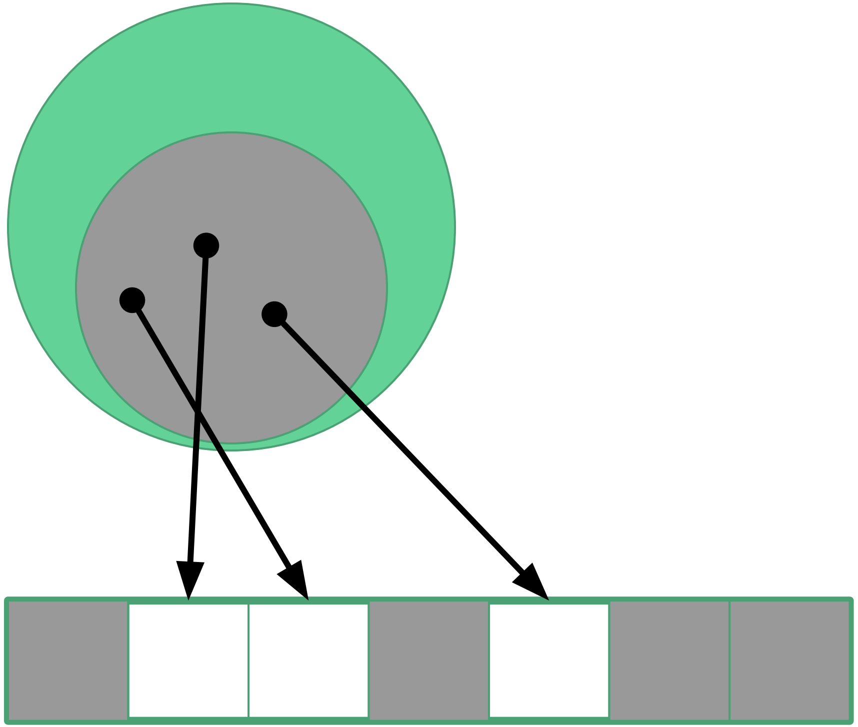
- **Multiple keys** will necessarily be assigned to the same cell
  - The hashing function is no longer one-to-one



Direct Addressing - Shrinking the Array

If the array is smaller than the key space:

- **Multiple keys** will necessarily be assigned to the same cell
  - The hashing function is no longer one-to-one
- This is fine if only one of those keys is actually used



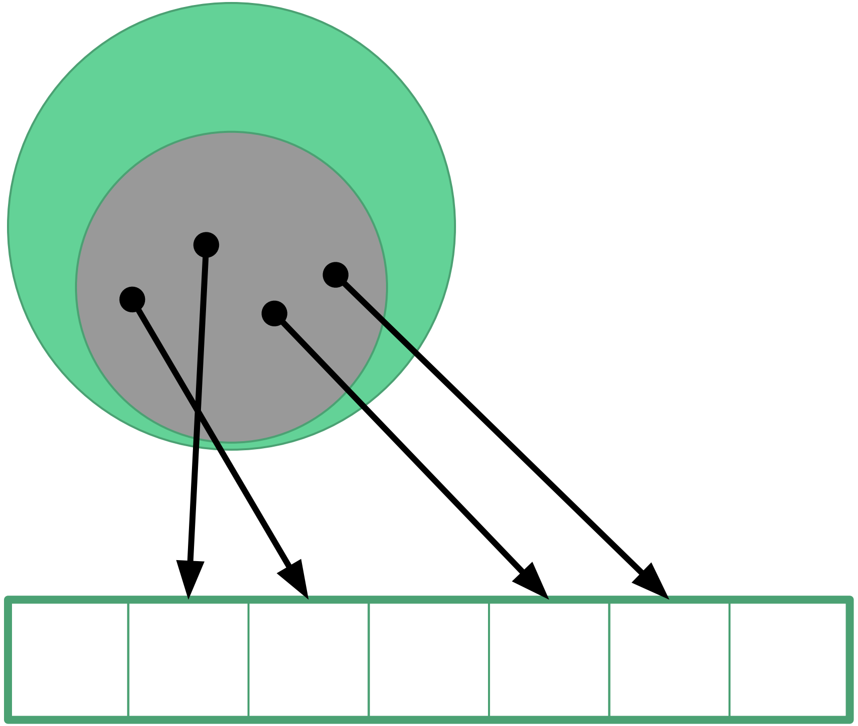
Hash Tables

A ***hash table*** is a dictionary data structure that:

- Maps keys into an ***m* element array**
- Here, `m << |U|`
- Using a **hash function:**

`h: U → { 0, 1,...,m - 1 }`

- Designed to distribute keys **uniformly** over array cells
    - a.k.a “Buckets” for some fucking reason

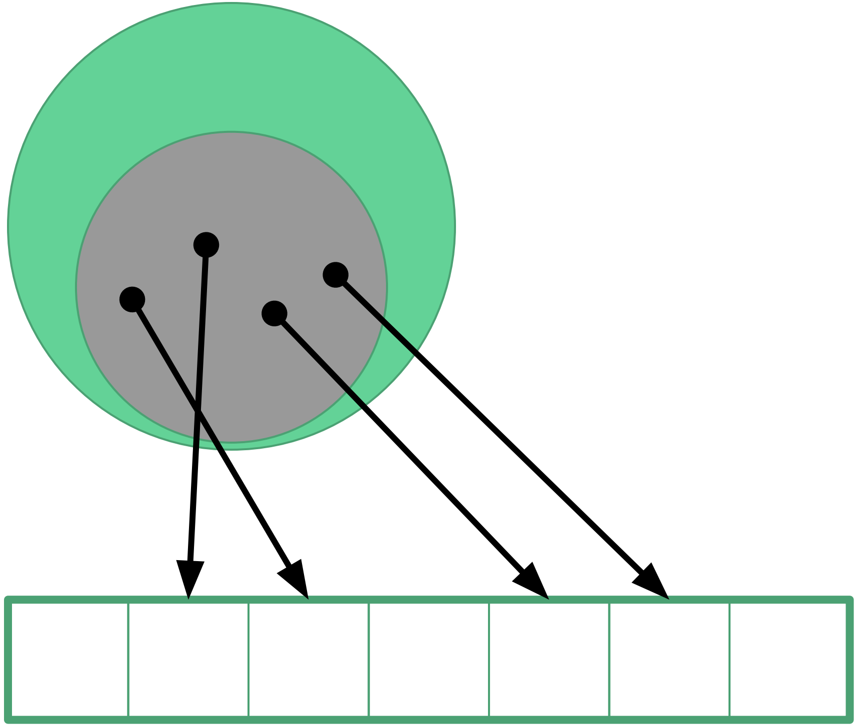


Hash Tables

```
void Store(key, value) {
    array[h(key)] = value;
}

object Lookup(key) {
    return array[h(key)];
}
```

Shit's **O(1)** (assuming h is O(1)).



Hash Tables - Hash Functions

A simple hash function for *ASCII strings*:

- Take each character of the string
- Turn it into its ASCII code
- Add them all up, then take the modulus

```
int ShittyHash(string s) {
    int sum = 0;
    foreach (char c in s) {
        sum += (int) c;
    }
    return sum % TABLE_SIZE;
}
```

Hash Tables - Hash Functions

A simple hash function for **`object`**s, e.g.

- Take the `object`'s address in memory
- Take the modulus

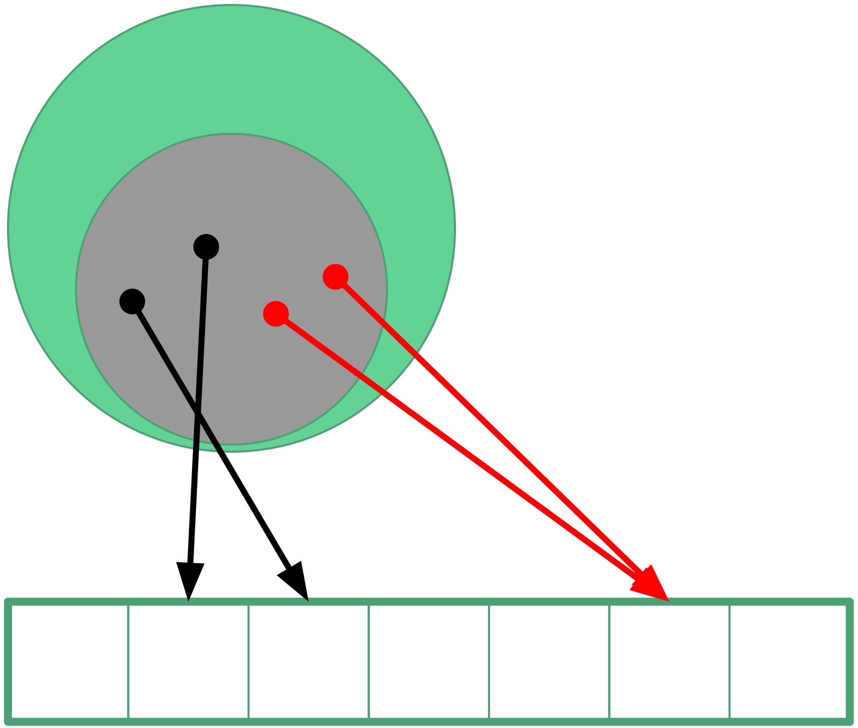
Note: this isn't possible in C#, which doesn't let you directly manipulate memory addresses. We'll pretend code in good ol' C.

```
int ObjectHash(void* o) {
    int x = (int) o;
    return x % TABLE_SIZE;
}
```

Hash Tables - Collisions

Great cool but what happens if two keys get mapped to the **same array cell**?

- Called a ***hash collision***



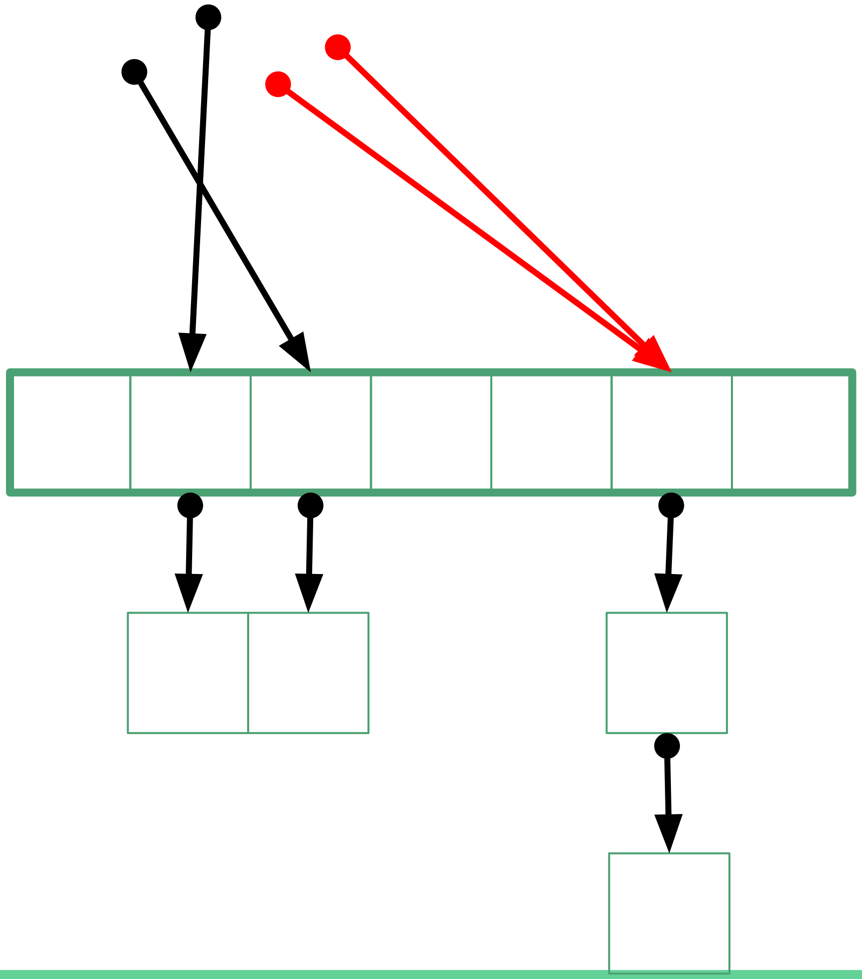
Hash Tables - Chaining

Great cool but what happens if two keys get mapped to the **same array cell**?

- Called a ***hash collision***

Easy solution:

- Store a wee linked list in each cell, and add elements to these
- Each node holds a key/value pair for the keys that map to that particular cell

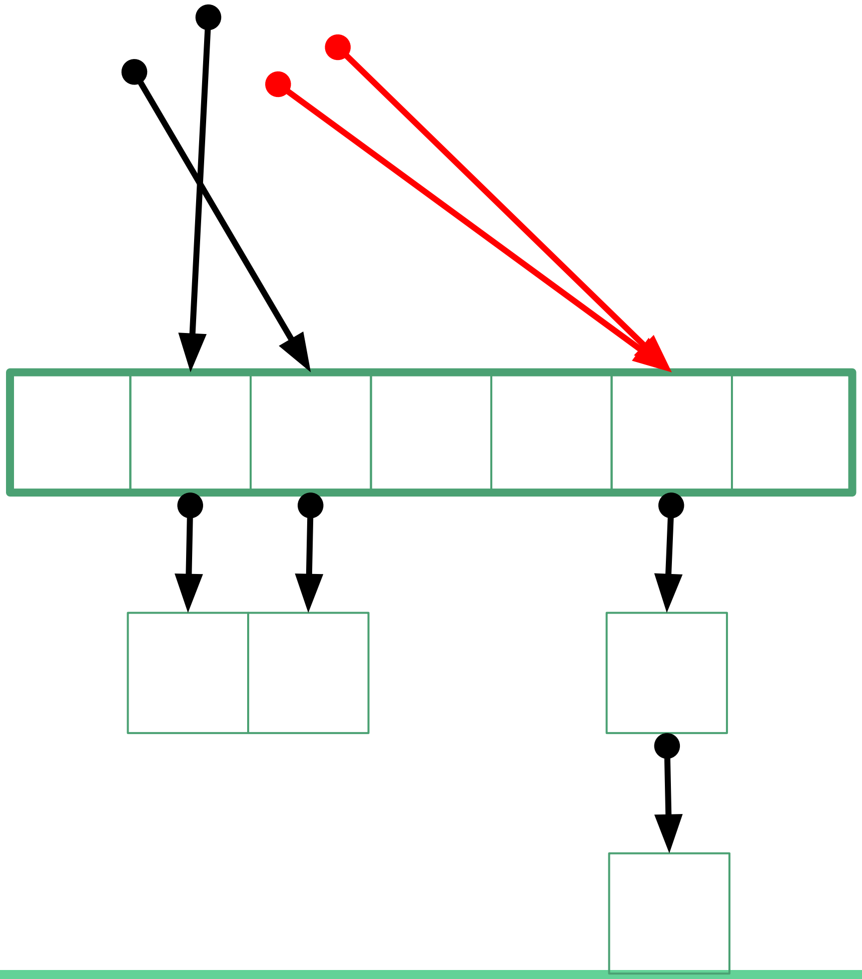


Hash Tables - Chaining

```
void Store(key, value) {
    array[h(key)].Store(key, value);
}

object Lookup(key) {
    return array[h(key)].Lookup(key);
}
```

Is this shit O(1)?

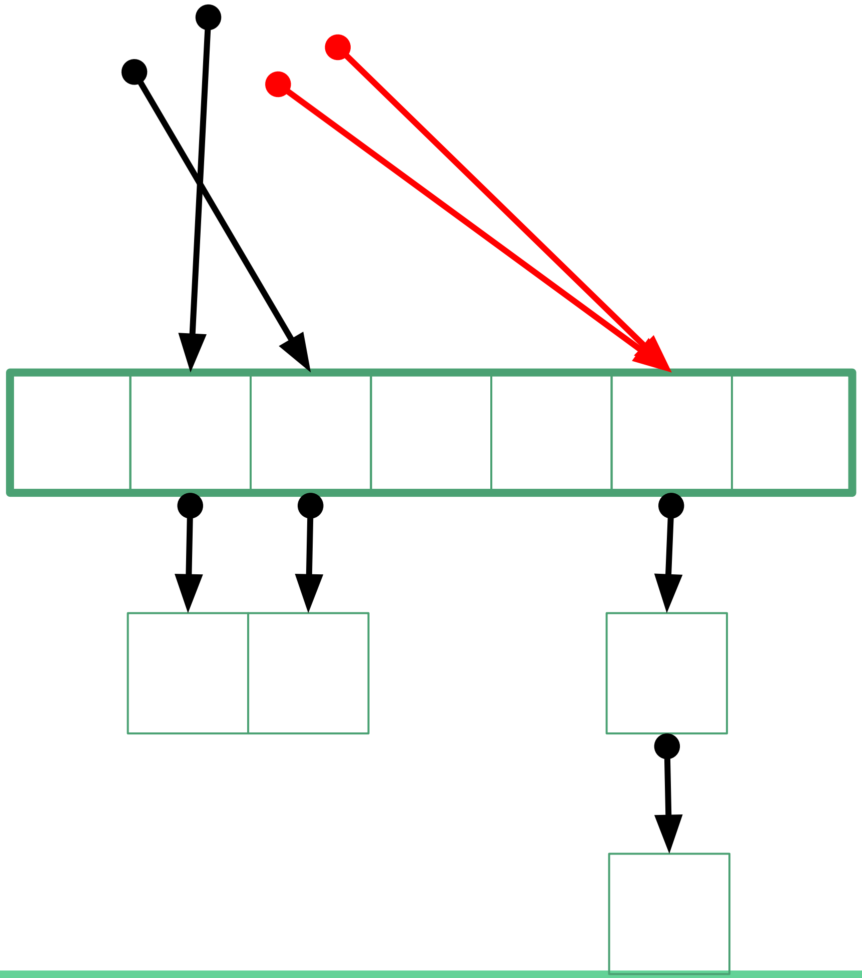


Hash Tables - Chaining

```
void Store(key, value) {
    array[h(key)].Store(key, value);
}

object Lookup(key) {
    return array[h(key)].Lookup(key);
}
```

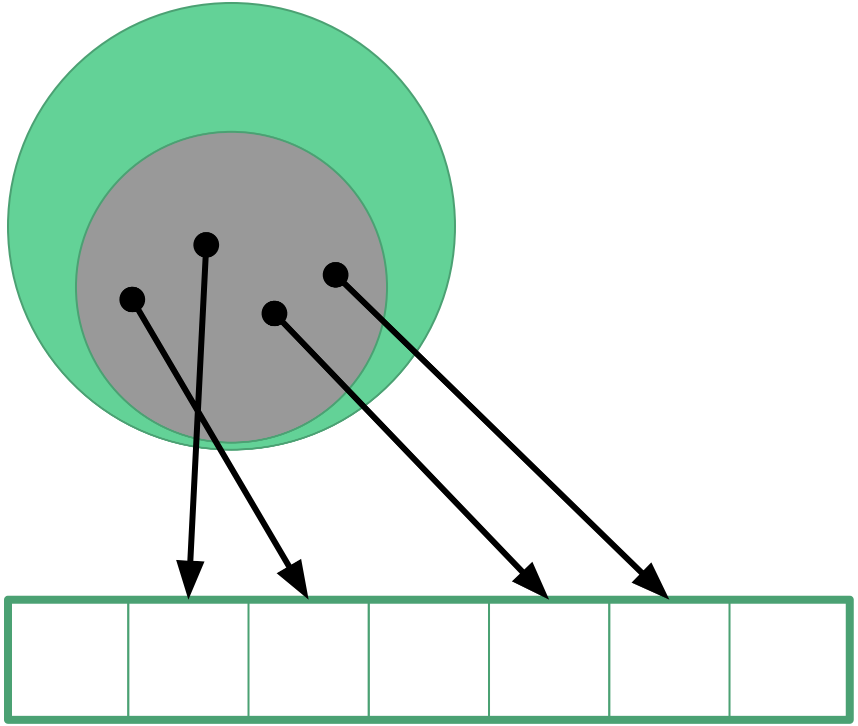
This shit is definitely not O(1).



Hash Tables - *Average* Performance

In practice, hash tables **generally perform well**.

What's the theoretical explanation?

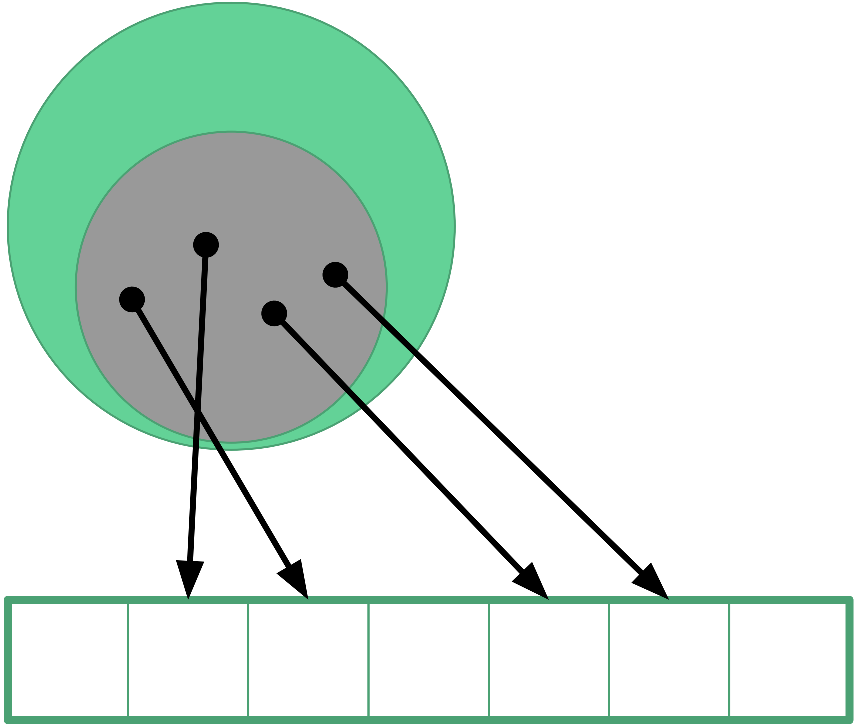


Hash Tables - *Average* Performance

What's the theoretical explanation?

Some assumptions:

- Simple, ***uniform*** hashing
  - Keys are **evenly distributed** over the array
- **Efficient** hash function
  - We assume that computing the hash of a key is $O(1)$
  - Or, at least, that it's not dependent on the number of keys

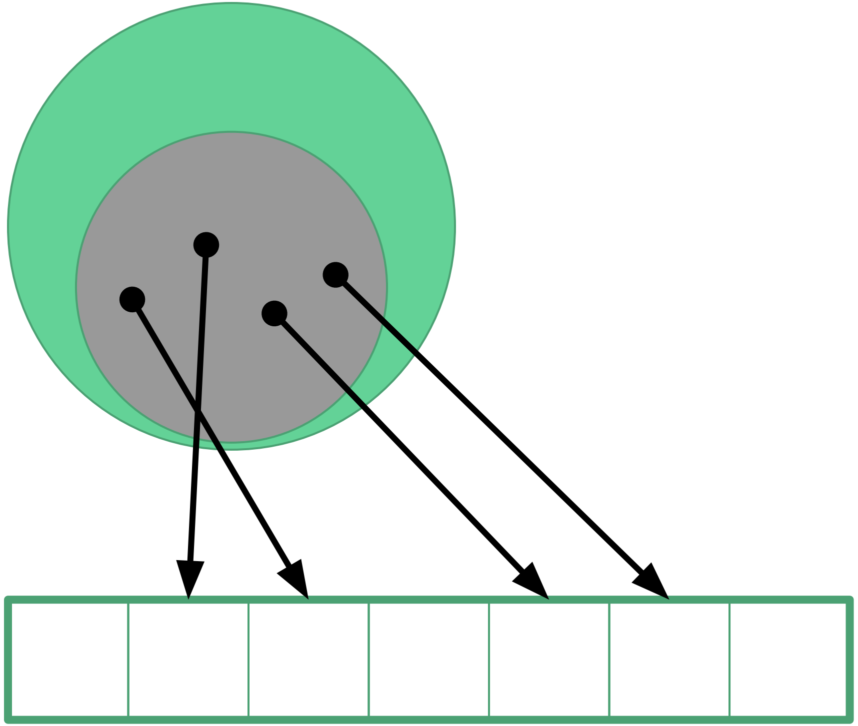


Hash Tables - *Average* Performance

Definition: the ***load factor*** of a hash table is

$$\alpha = n / m$$

- The **ratio** of *number of keys* stored in the table
- To the *size of the array*



Hash Tables - Performance

Theorem 1: an **unsuccessful** search takes average time $\Theta(1 + \alpha)$

Proof:

- The hash table has ***n* keys**, which means that the **total number of list nodes** in the whole table is *n*
- The table has ***m lists*** (one for each cell in the array)
- The average length of the lists in the array is thus n / m
- Search time is time to **compute hash** + time to **search list:**

$$\Theta(1) + \Theta(\alpha) = \Theta(1 + \alpha)$$

Hash Tables - Performance

Theorem 2: a **successful** search takes average time $\Theta(1 + \alpha)$

Proof:

Hash Tables - Performance

Lemma: the total number of list nodes searched while inserting *n* items into an empty hash table is on average

$$n(n - 1) / 2m$$

Proof:

- When we insert the i^{th} key, we perform an unsuccessful search
- At that point, there are $i - 1$ keys
- Load factor α is $(i - 1) / m$
- But the **expected number of searches** is $\boldsymbol{\alpha}$

So the **total number** of searches is, if you think way back to high school math:

$$(1 / m) * (n(n - 1) / 2) = n(n - 1) / 2m$$

(Google docs is shitty and won't let me write LaTeX, sorry)

Hash Tables - Performance

Theorem 2: a **successful** search takes average time $\Theta(1 + \alpha)$

Proof:

- The number of elements searched for a successful search is
  - the number of elements searched when the key was originally inserted
  - + 1
- The **average** number of elements searched on insertion is
  - the total number of elements searched **while inserting everything**
  - **divided by** the number of elements in the table

$$(1 / n)\ (n(n - 1)/2m) = (n - 1)/2m = n/2m - 1/2m =$$

$$\mathbf{\alpha/2 - 1/2m}$$

Hash Tables - Performance

Theorem 2: a **successful** search takes average time $\Theta(1 + \alpha)$

Proof:

- So, the average total number of elements searched is

$$1 + \alpha/2 - 1/2m$$

- So, the **total execution time** is

$$\Theta(1) + \Theta(1 + \alpha/2 - 1/2m) = \Theta(1 + \alpha)$$

Hash Tables - Performance

But wait… if the average performance is $\Theta(1 + \alpha)$…

And, since $\alpha = n/m$… Doesn't that make the whole thing $O(n)$?

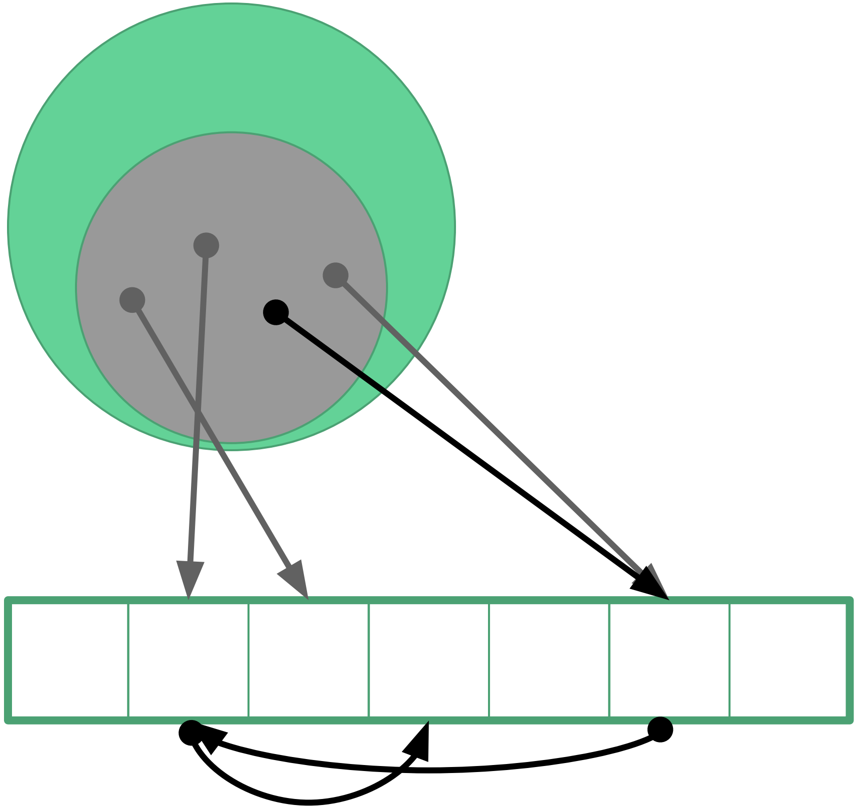
Actually…

- Though time **depends on *n***
- It also **depends on *m***
- **Compensate** for increases in *n* by increasing *m* as well

Open Addressing

Another solution to collisions:

- Instead of using a linked list, we can try **successive locations** in the hash tables
- *Important:* elements of the array need to be **key/value** pairs instead of just the values

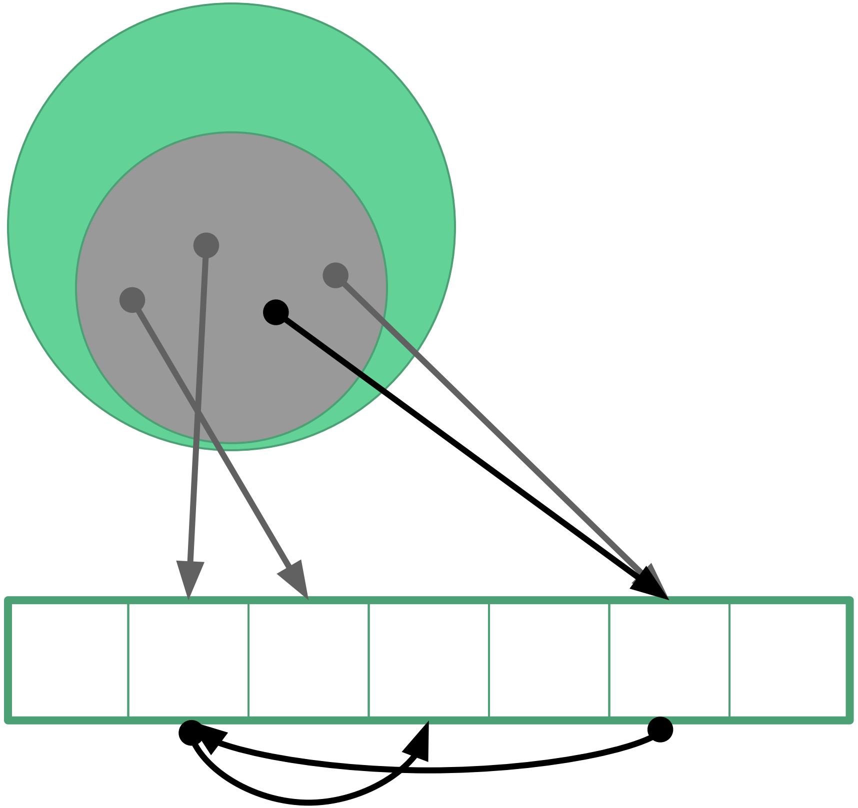


Open Addressing

Redefine our hash function ***h*** to take a **second argument:**

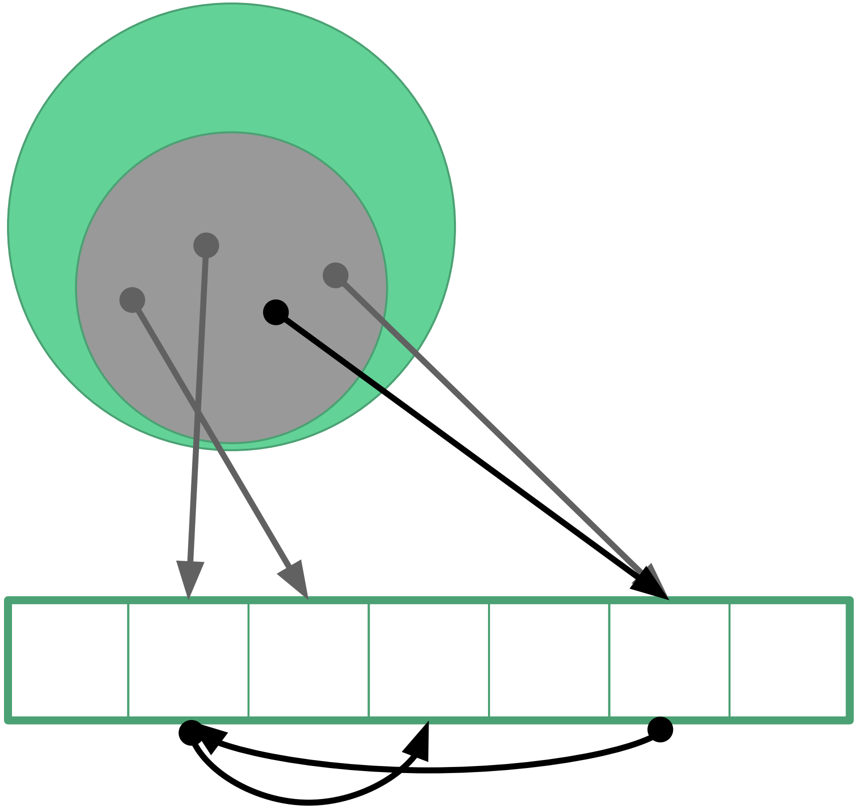
h: U x M ➜ M

- M = { 0, 1, ... m - 1 }
- The second argument specifies **which attempt** we're on as we try to hash something



Open Addressing - Store

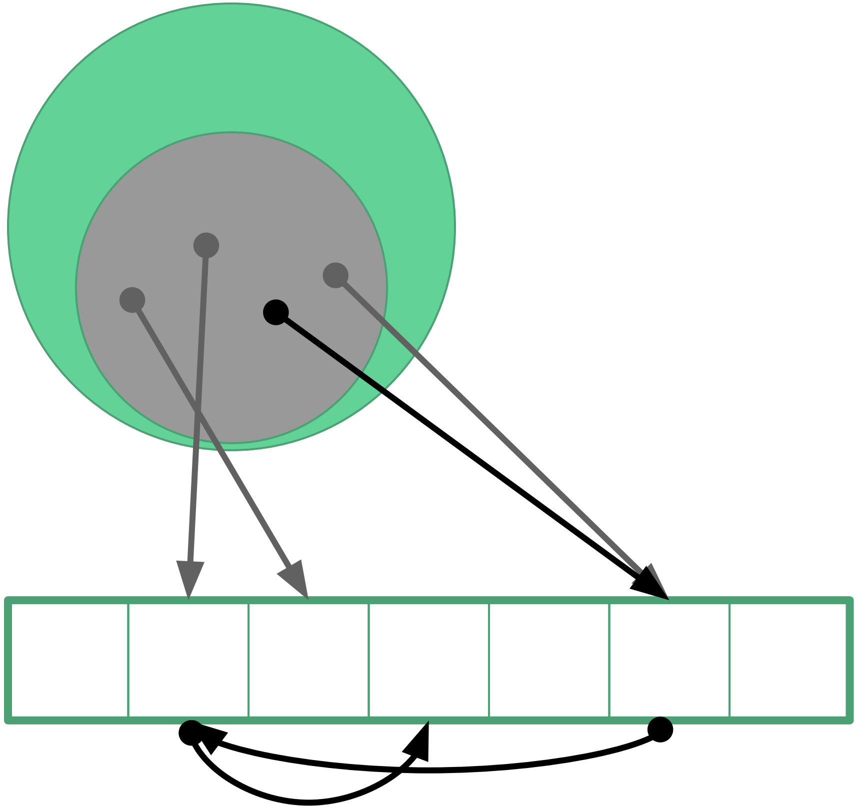
```
void Store(key, val) {
     for (int i=0; i<m; i++) {
          int j = h(key, i);
          var comp = array[j];
          if (comp.key==key) {
               comp.value=val;
               return;
          } else if (comp.key==null) {
               comp.key = key;
               comp.value = val;
               return;
          }
     }
     throw new Exception(“Table Full”);
}
```



Open Addressing - Store

```
void Store(key, val) {
    for (int i=0; i<m; i++) {
        int j = h(key, i);
        ...
```

Food for thought: why do we stop trying **after *m* iterations?**

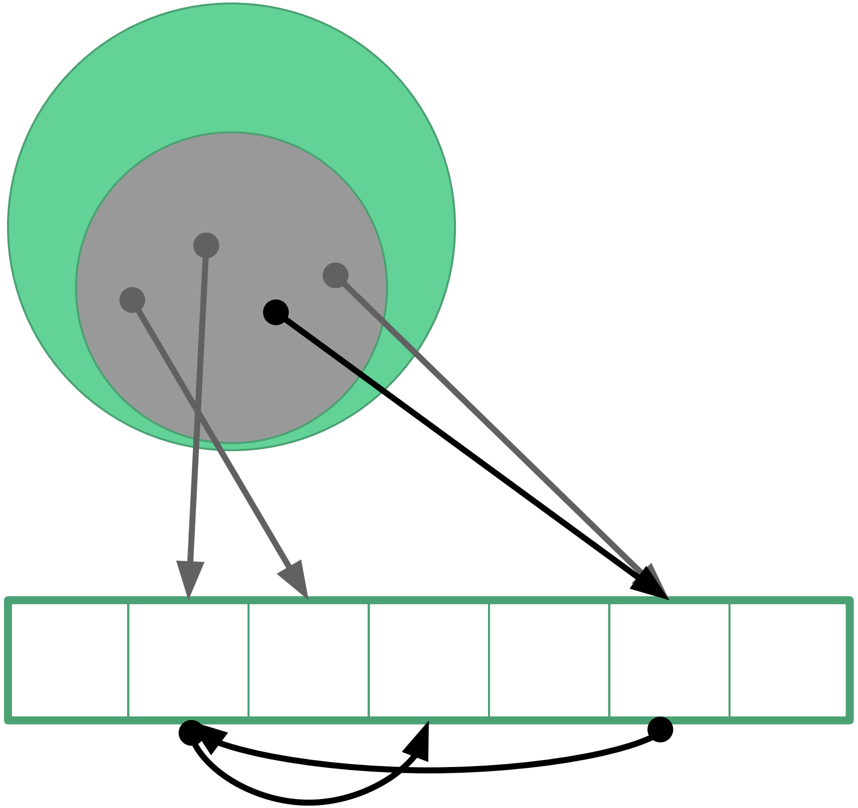


Open Addressing - Store

```
void Store(key, val) {
    for (int i=0; i<m; i++) {
        ...
```

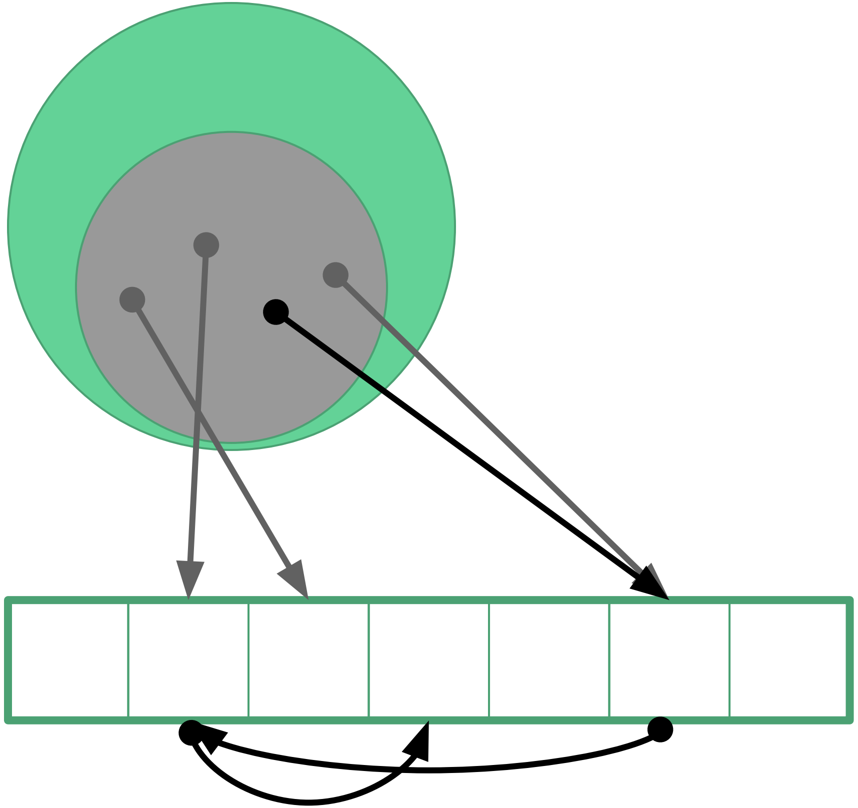
Food for thought: why do we stop trying **after *m* iterations?**

Because the array has only *m* elements, we must have tried them all at least once.



Open Addressing - Lookup

```
void Lookup(key, value) {
     for (int i=0; i<m; i++) {
          int j = h(key, i);
          if (array[j].key == key)
               return array[j].value;
          if (array[j].key == null)
               return null;
     }
     return null;
}
```



Probe Sequences

- Our two-argument hash function defines a sequence of cells to check

- We call this the ***probe sequence***

In practice, these hash functions are built from:

- a **one-argument hash function**
- some **extra magic**

Probe Sequences - Linear Probing

The "dumbest" possible hash function:

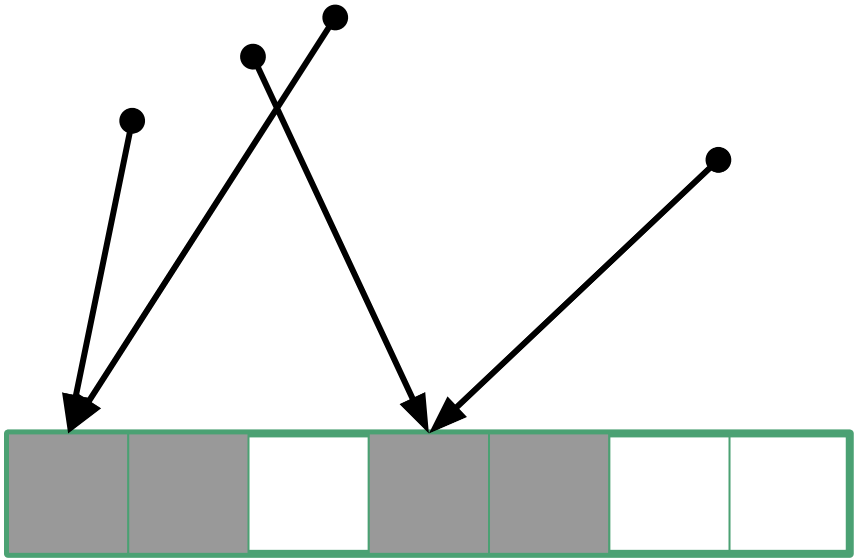
- Start with the location defined by a **normal hash function**, then search **consecutive cells**
- Make sure to **wrap around** to the beginning of the array if you reach the end

$$h_{linear}(k, i) = (h_{normal}(k) \mathbf{+ i}) \ mod \ m$$

Probe Sequences - Clustering

Problem:

- We end up with **long runs** of occupied cells
- Any new key that hashes to any **part of one of those runs** has to iterate *all the way to the end*
- Performance suffers



Probe Sequences - Quadratic Probing

A “slightly smarter” hash function:

- Make things more complicated by adding a **quadratic polynomial** to the equation

$$h_{\text{quadratic}}(k,\ i) = (h_{\text{normal}}(k) + \mathbf{c_1 i} + \mathbf{c_2 i^2})\ \mathit{mod}\ m$$

- Makes probe sequences more **complicated**, and less **prone to clustering**

Probe Sequences - Quadratic Probing

A "slightly smarter" hash function:

- Make things more complicated by adding a **quadratic polynomial** to the equation

$$h_{quadratic}(k, i) = (h_{normal}(k) + \mathbf{c_1 i} + \mathbf{c_2 i^2}) \ mod \ m$$

- Makes probe sequences more **complicated**, and less **prone to clustering**
- However, two keys with the **same h_{normal}** will still probe the same sequence
    - www.yikes.gov

Probing Sequences - Double Hashing

We can solve these problems by taking the combination of two hashing functions:

$$h_{fancy}(k, i) = (h_1(k) + \mathbf{h_2(k)i})\ mod\ m$$

- Two keys that map to the same h_1 value will still have different probe sequences
    - If their h_2 values are different, of course
- *Important: this requires **m** to be a prime number*
    - $h_2(k)\ mod\ m$ should never be 0, i.e., $h_2(k)$ should never be a multiple of *m*
    - This will guarantee the probe sequence always includes the **whole array**

Probe Sequences - Prime Numbers

Why do we care so much about prime numbers?

- Assume a hash table with *10 elements*
- Now, assume that for some key *k*:
  - $h_1(k) = 5;\ h_2(k) = 3$

Probe Sequences - Prime Numbers

Why do we care so much about prime numbers?

- Assume a hash table with *8 elements*
- Now, assume that for some key *k*:
  - $h_1(k) = 5;\ h_2(k) = 3$

$$h(k,i) = (h_1(k) + h_2(k)i) \bmod 8$$

```
h(k,0) = (5 + 0*3)mod 8 = 5
h(k,1) = (5 + 3)mod 8 = 0
h(k,2) = 3
h(k,3) = 6
h(k,4) = 1
h(k,5) = 4
h(k,6) = 7
h(k,7) = 2
```

Every slot is probed exactly once!

Probe Sequences - Prime Numbers

If we *change things around* a bit:

- Assume a hash table with *8 elements*
- Now, assume that for some key *k*:
  - $h_1(k) = 5$
  - $\mathbf{h_2(k) = 4}$

$$h(k,i) = (h_1(k) + h_2(k)i) \bmod 8$$

```
h(k,0) = (5 + 0*4)mod 8 = 5
h(k,1) = (5 + 4)mod 8 = 1
h(k,2) = 5
h(k,3) = 1
h(k,4) = 5
h(k,5) = 1
h(k,6) = 5
h(k,7) = 1
```

We **keep probing the same slots**! Boo.

Probe Sequences - Prime Numbers

- Number theory says that we'll have a problem if $h_2(k)$ and m **share a common factor**
- The easiest way to prevent this is to make sure that ***m* is prime**

Open Addressing - Performance

Theorem: assuming **uniform hashing**, the **average number of probes** for an *unsuccessful* search is at most:

$$1/(1 - \alpha)$$

Theorem: assuming uniform hashing, the **average number of probes** for a *successful* search is at most:

$$(1/\alpha)(\ln(1/(1 - \alpha))) + 1/\alpha$$

Proof: beyond the scope of this course, but you can find it in the CLR book.