Lecture 9
Hash tables

EECS-214

Dictionaries

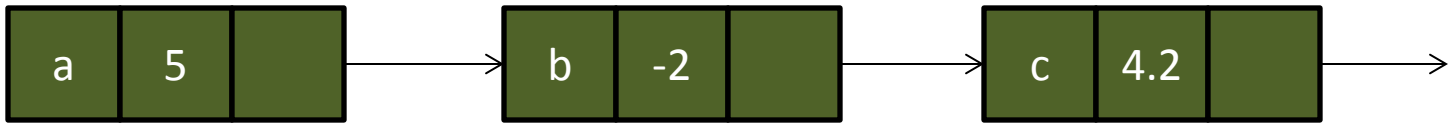
- A.k.a. **associative array**, **map**, **mapping**
- Data structure that holds associations between pairs of objects called **keys** and **values**

Simplified interface (we'll talk more about dictionaries later):

- dict.Store(key, value)
  - Adds key to dictionary with associated value
  - We'll assume that if the key is already present, this changes its value
- dict.Lookup(key)
  - Returns value associated with key in dict

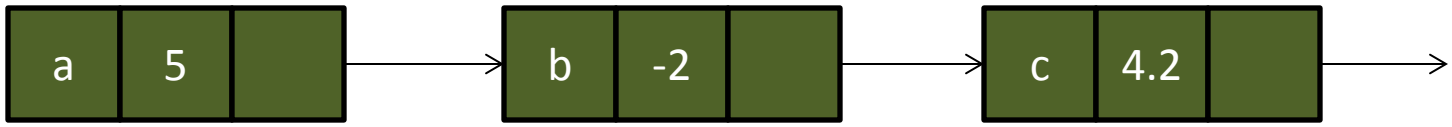
Implementing dictionaries with association lists

- The simplest possible implementation of dictionaries is as a **linked list** of key/value pairs
- Each cell has three fields
  - Key
  - Value
  - Link to next cell



Implementing dictionaries with association lists

- This isn't a great representation because lookup requires $O(n)$ time
- Last time, we looked at tree structures that can do lookup in $O(\log n)$ time
- Today we'll look at structures for $O(1)$ time



Arrays as dictionaries

- Arrays can be thought of as a **special kind** of dictionary
  - Keys are always **intervals of integers**: $\{0, 1, 2, \ldots n\}$

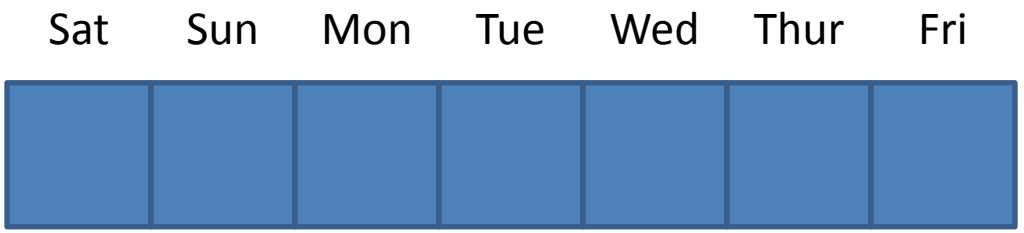
- Excellent performance: $\boldsymbol{O(1)}$ for all operations
- Limited utility because of limited keys

why can’t everything be like arrays?

Direct addressing

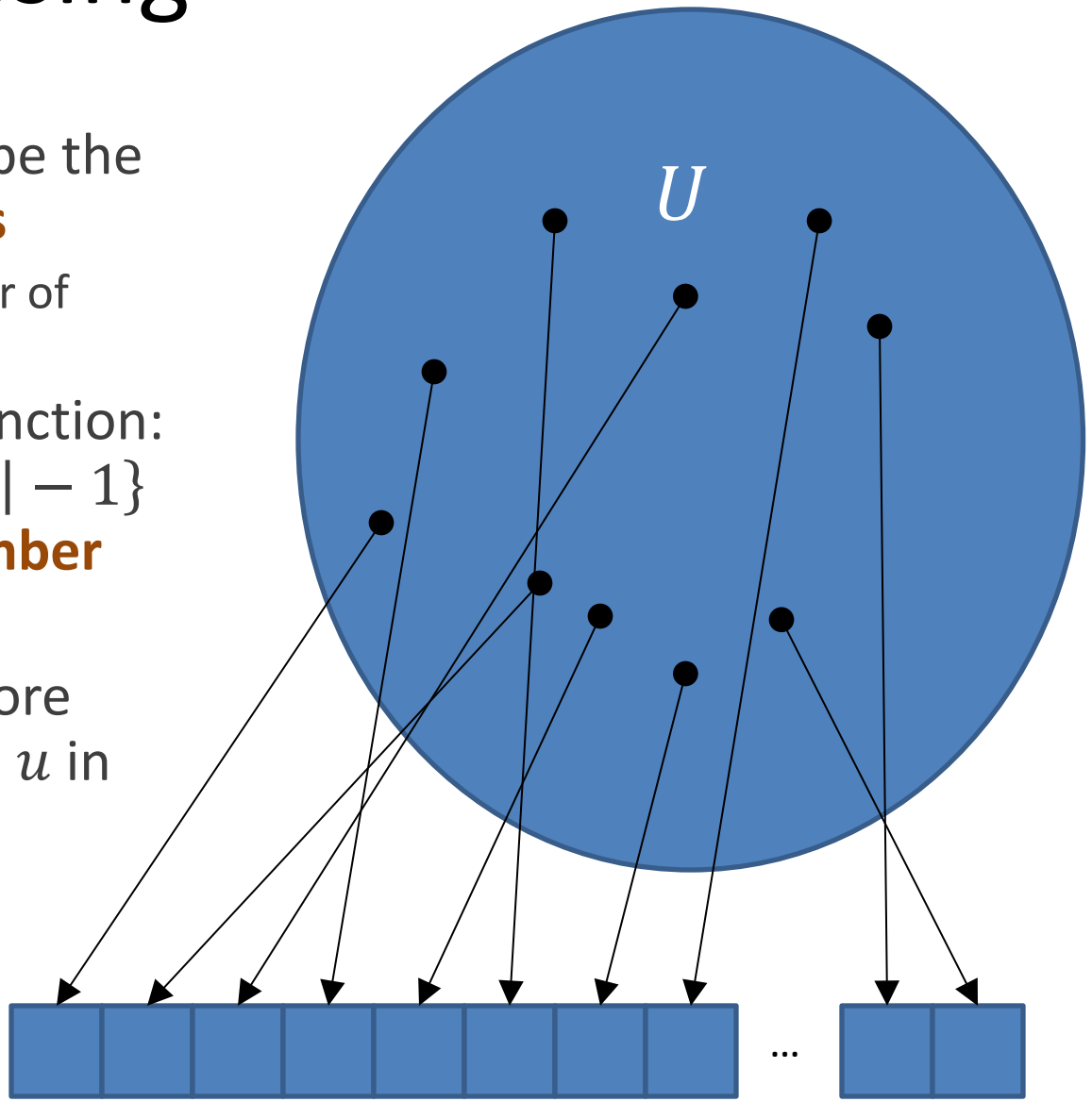
- **Assign numbers** to all possible keys in advance
- Make an array as big as the set of all keys
- **Store value** associated with key **in array slot** corresponding to the key's number
  - Value associated with Monday is in a[2]

Key	Integer
Saturday	0
Sunday	1
Monday	2
Tuesday	3
Wednesday	4
Thursday	5
Friday	6



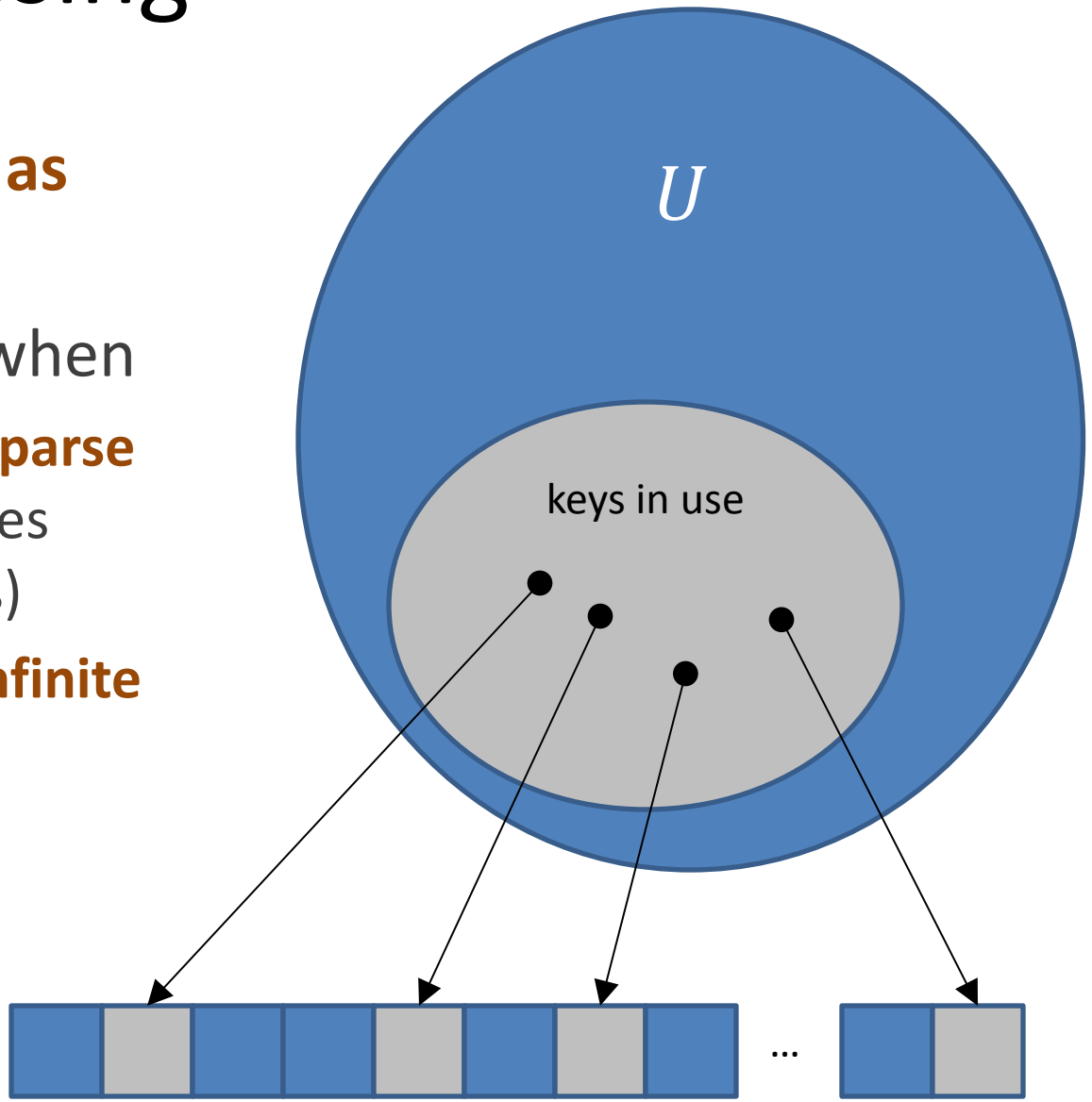
Direct addressing

- More formally, let U be the **universe (set) of keys**
  - Let $|U|$ be the number of elements in U
- Define a 1:1, onto, function: $h : U \rightarrow \{0, 1, \ldots |U| - 1\}$ that **associates a number with every key**
- For any key $u \in U$, store value associated with u in **a[$h[u]$]**



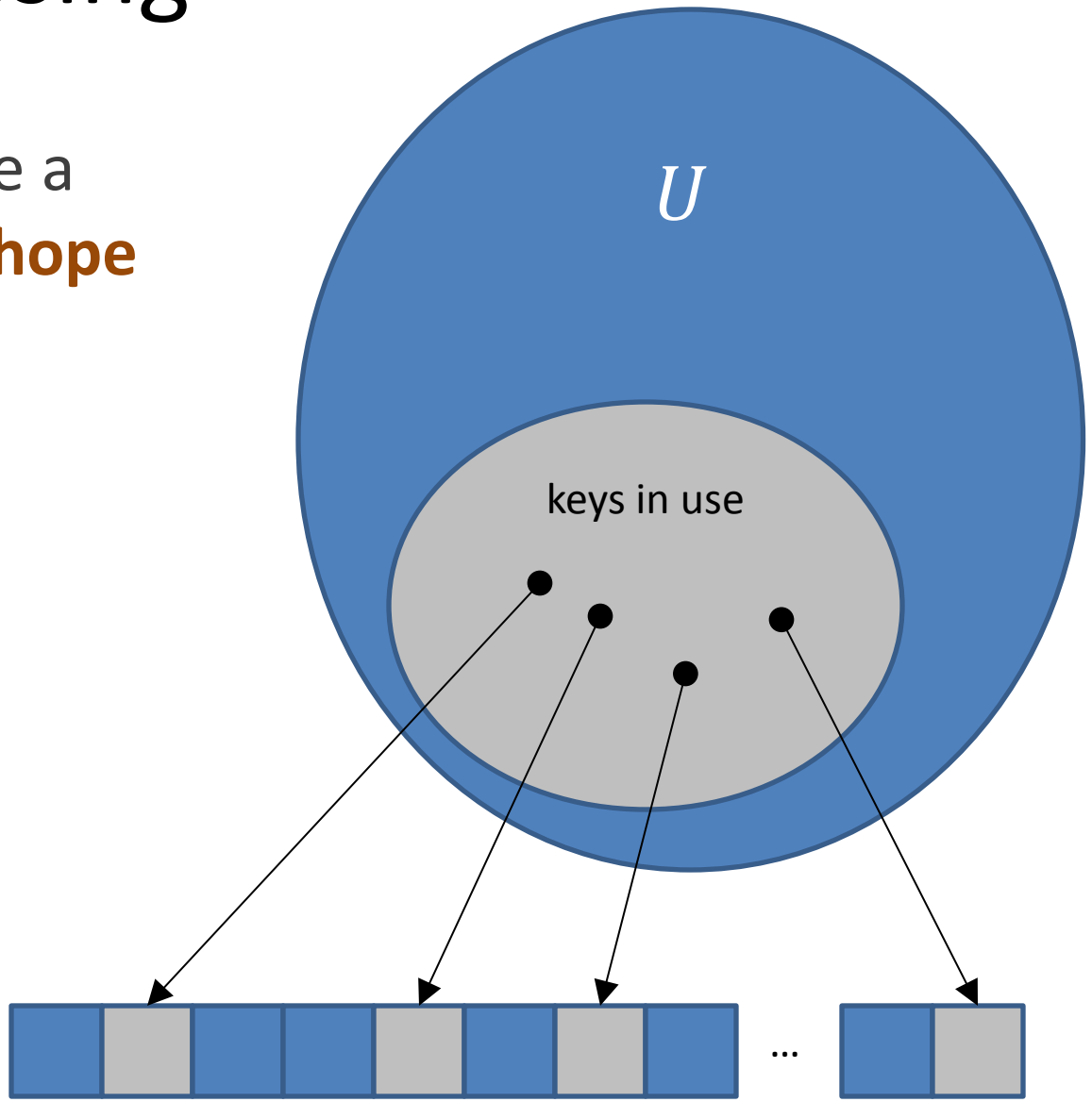
Direct addressing

- 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 key space is **infinite**



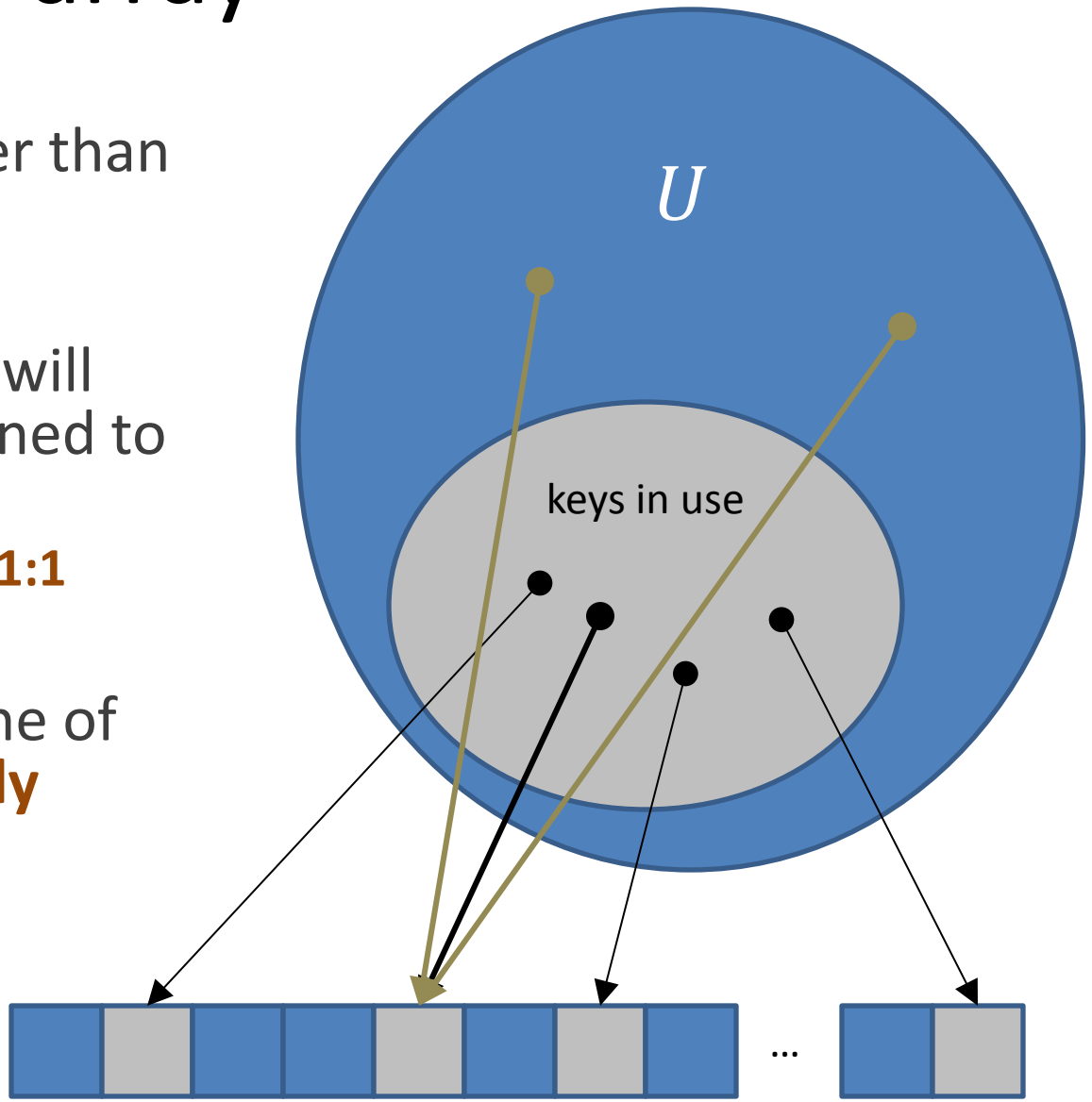
Direct addressing

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



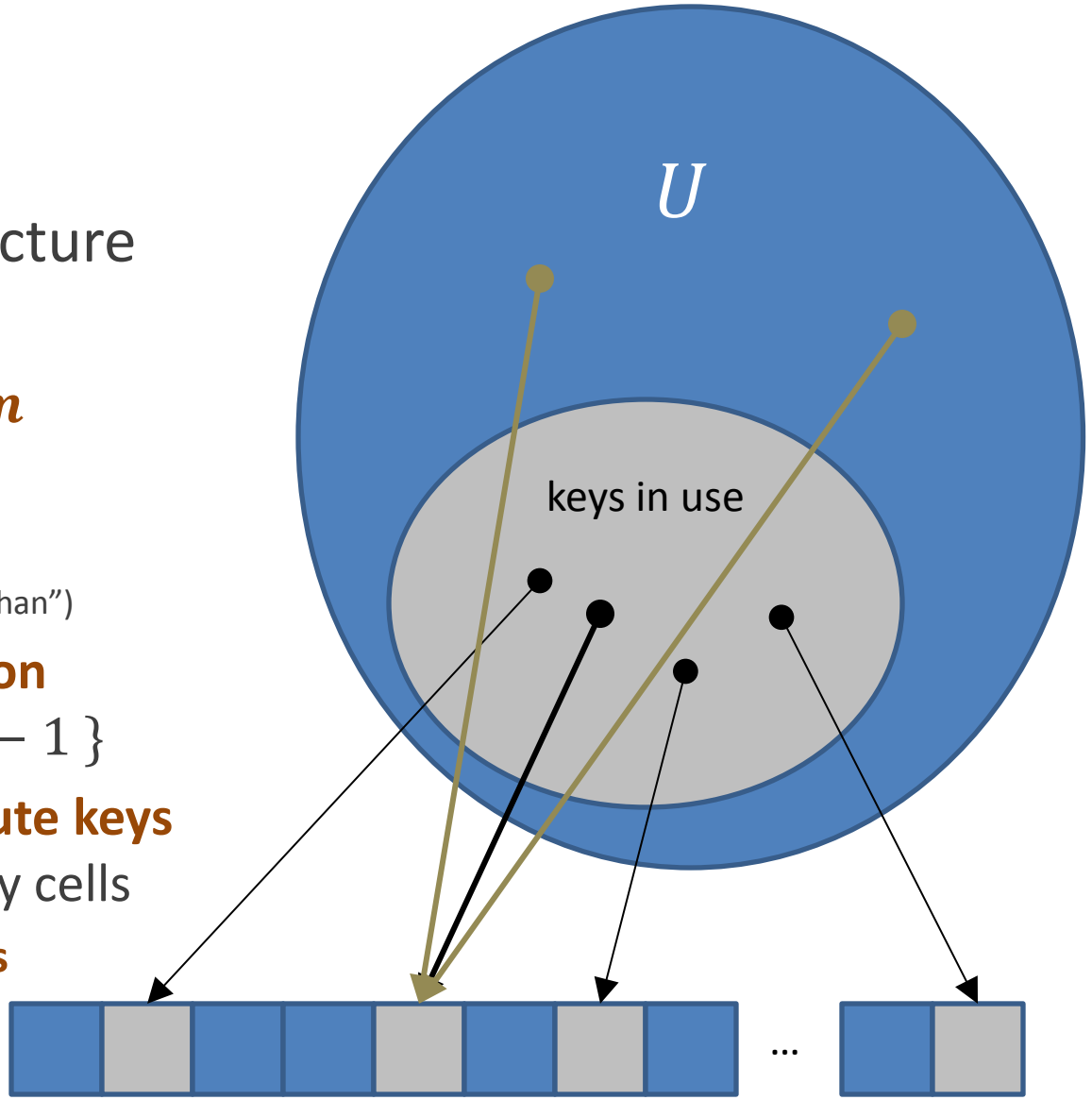
Shrinking the array

- If the array is smaller than the key space
- Then **multiple keys** will necessarily be assigned to the **same cell**
  - h will **no longer** be **1:1**
- That's 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**
    - Where $m \ll |U|$
    - ("≪" means "much less than")
  - Using a **hash function** $h : U \to \{0, 1, \ldots m - 1\}$
  - Designed to **distribute keys uniformly** over array cells
    - a.k.a. **hash buckets**



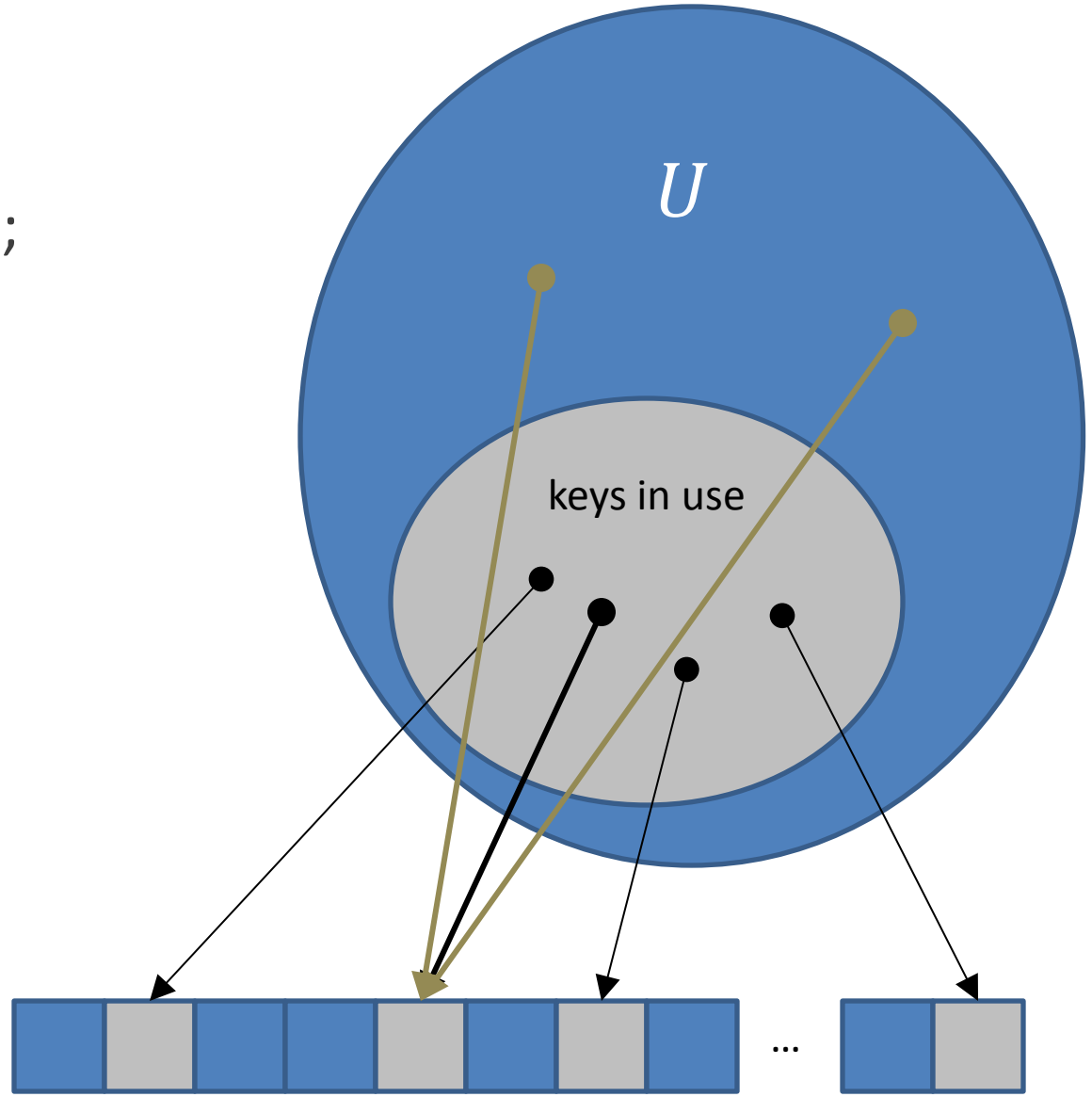
Hash tables

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

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

$O(1)$ time!

(assuming **h is $O(1)$** itself)



Example hash function

A simple hash function for **strings**:

- Take **each character** in the string
- Convert it to its ASCII or Unicode integer **numeric code**
- **Add them** all up
- Take the sum **mod m**
  - "mod m" means "the remainder after dividing by m"
    - (Except it's somewhat different for negative numbers, but we can ignore that)

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

Example hash function

A simple hash function for objects in memory:

- X's **address in memory**
  - Remember it can be thought of as an integer
- **Mod m**

- Can't write this in C# because C# won't let you manipulate addresses directly
  - C to the rescue!

```
int AddressHash(void *x) {
   return ((int)x)%tableSize;
}
```

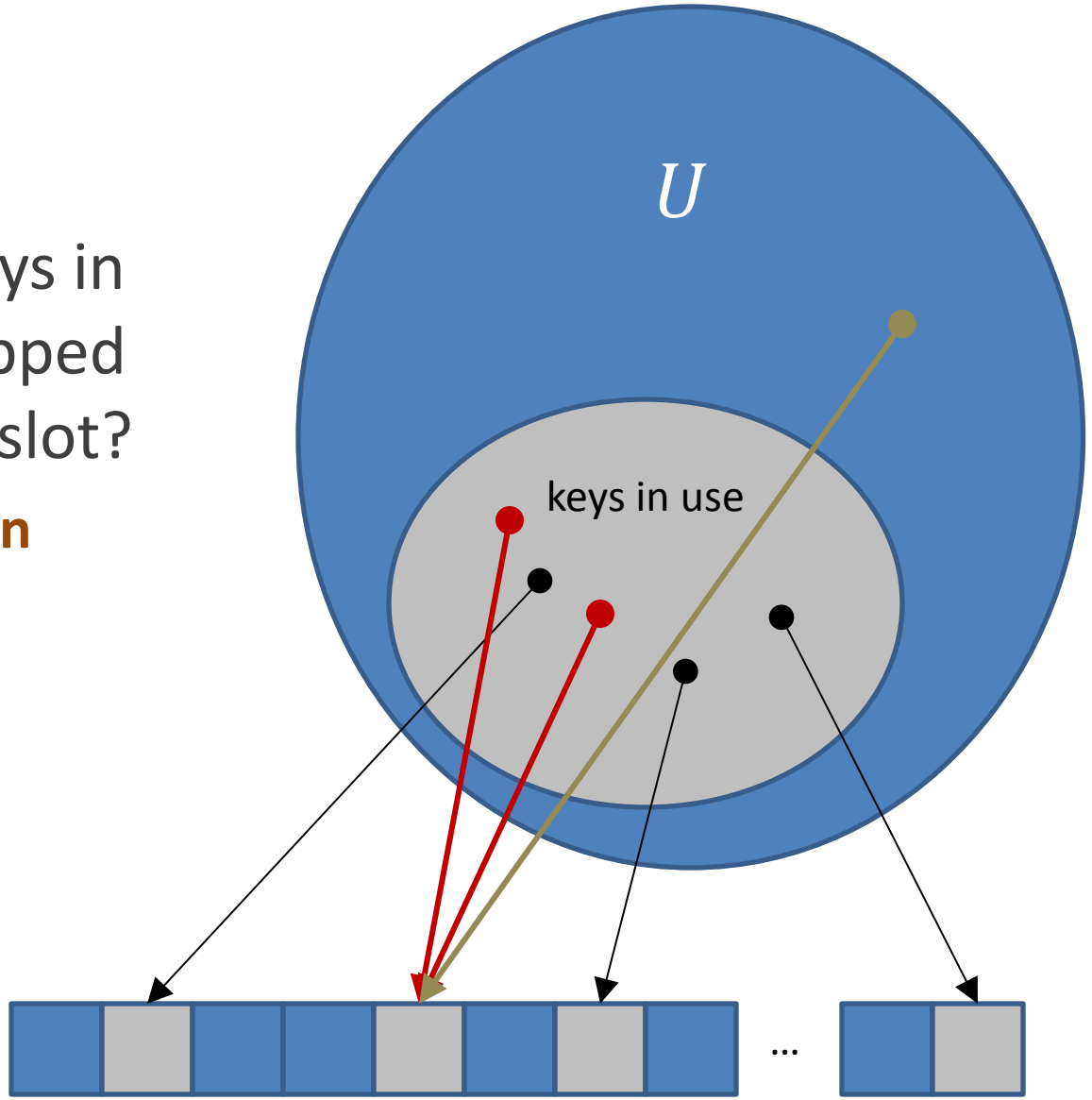
Note: in practice addresses are usually **multiples of 4** (for 32 bit machines) or 8 (for 64 bit machines)

- So in real life, you'd use the address divided by 4 or 8
  - Or right-shift by 2 or 3 bits, respectively
- So hash functions often look like:

```
( ((unsigned int)x)>>2)%tableSize
( ((unsigned int)x)>>3)%tableSize
```

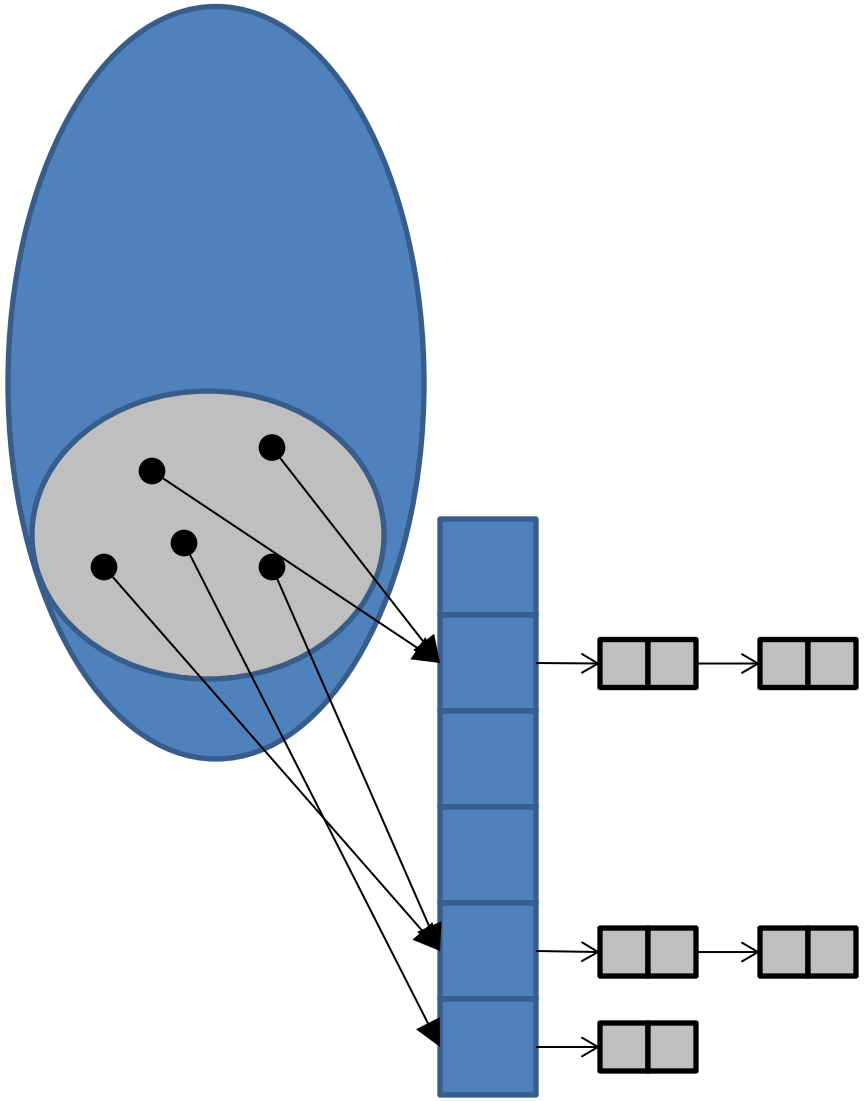
Collisions

- Great!
- But what if two keys in actual use get mapped to the same array slot?
  - We have a **collision**



Chaining

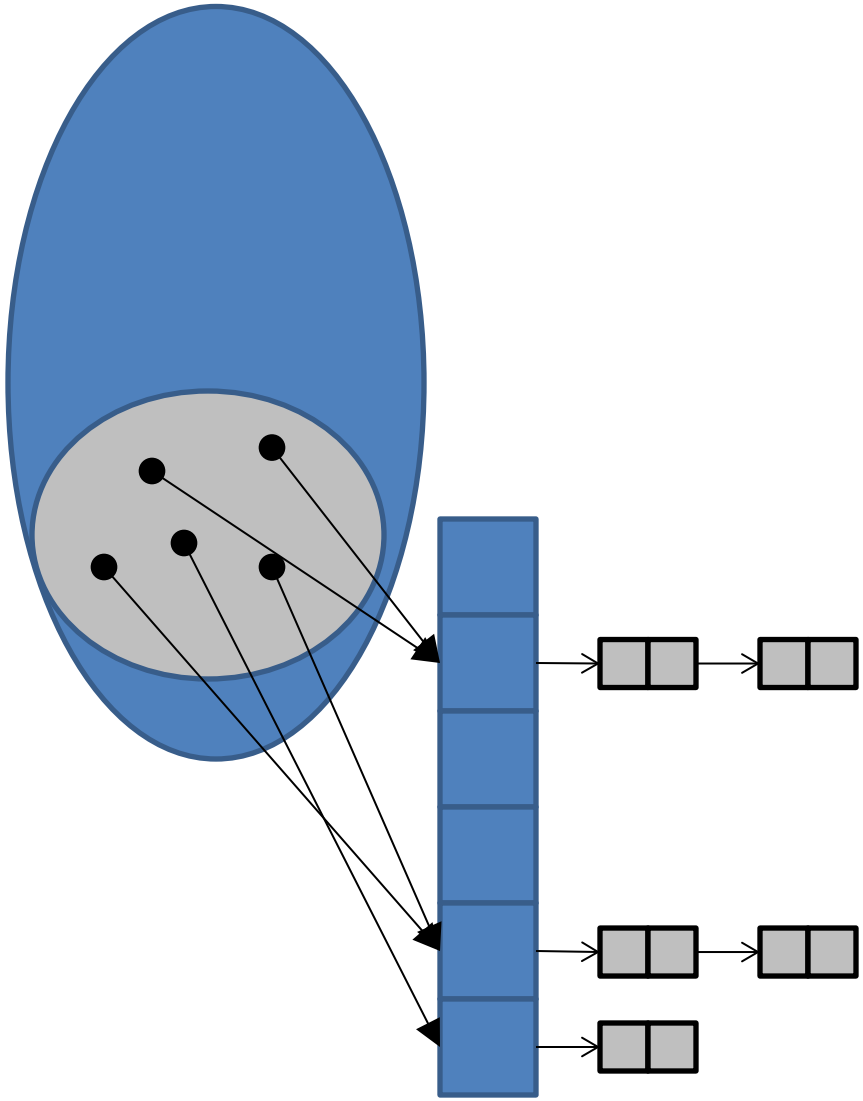
- The easy solution to this is to store a small **linked list dictionary** in each array cell
  - Stores just the key/value pairs for those keys in use that map to that particular array cell



Is this $O(1)$?

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

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

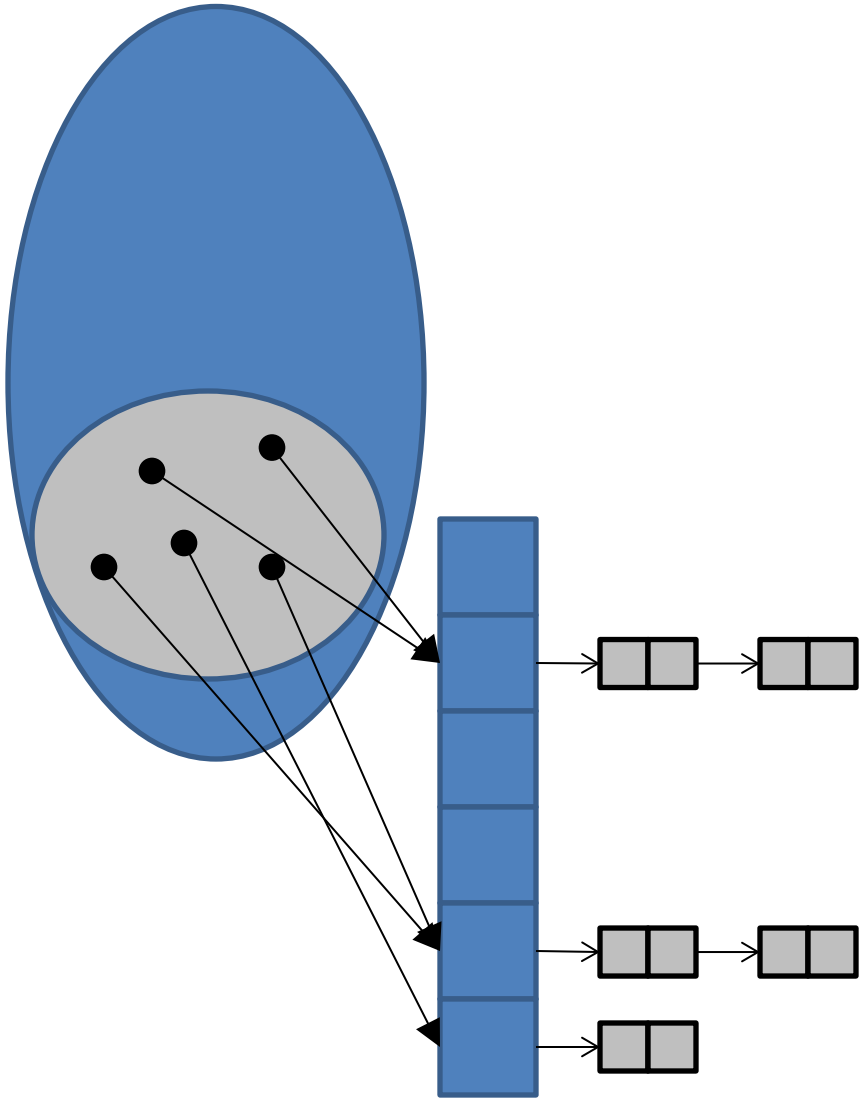


Is this $O(1)$?

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

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

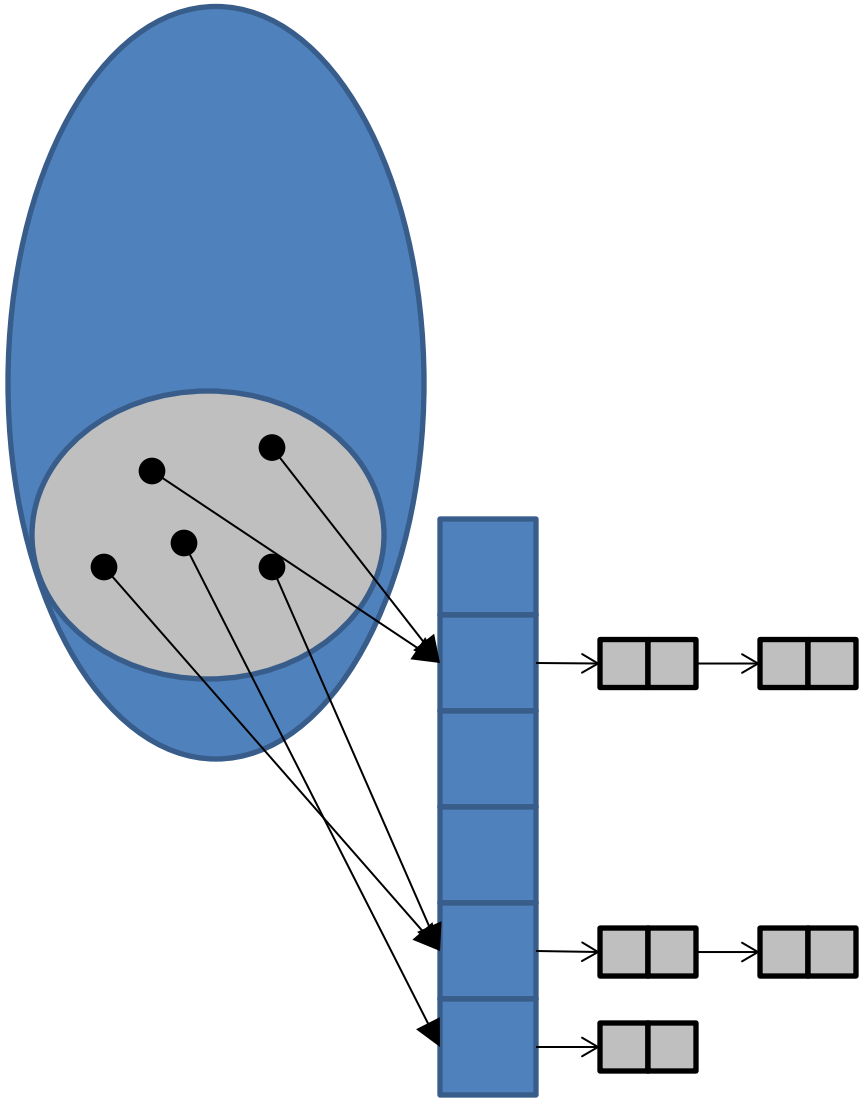
- No!



Is this $O(1)$?

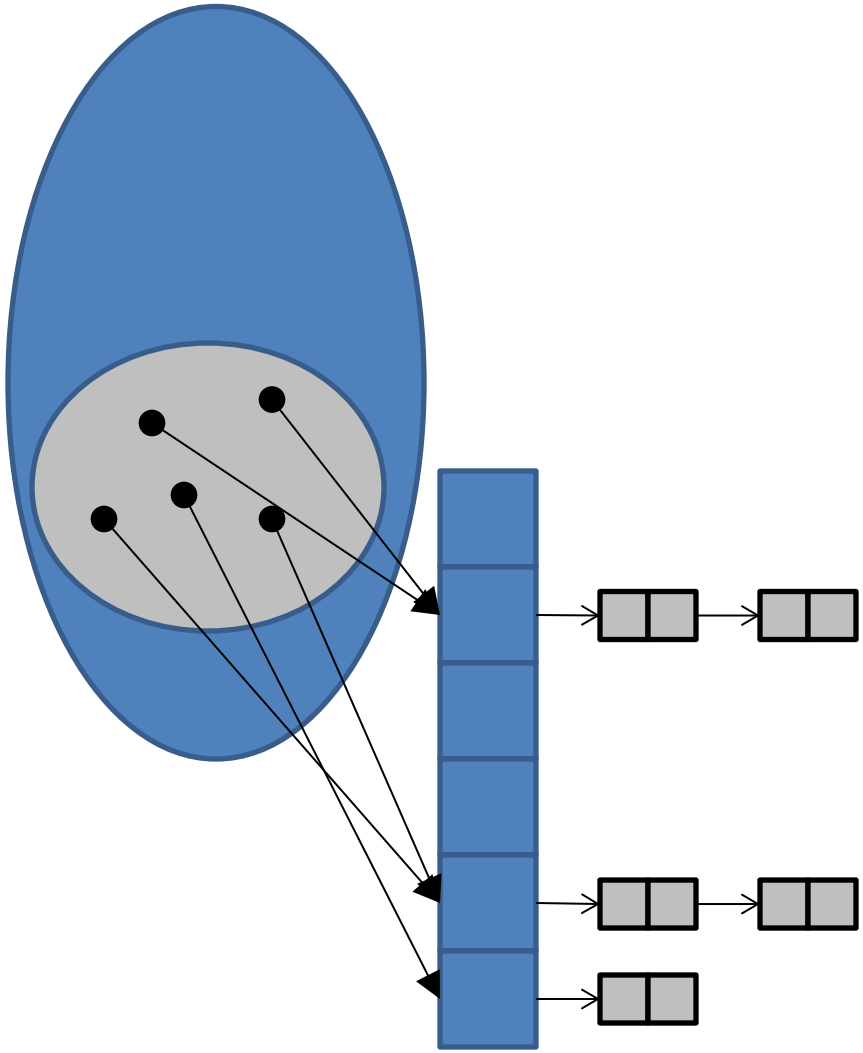
- No!
- The linked list operations are linear in the number of list elements
- How many list elements will there be?
  - Best case: 1
  - Worst case: all the elements in the dictionary!

- **Worst-case** performance of a hash table with n keys stored in it is $\boldsymbol{O(n)}$
- Crap!



Average-case performance

- In practice, hash tables **generally perform well**
- So how do we explain that theoretically?

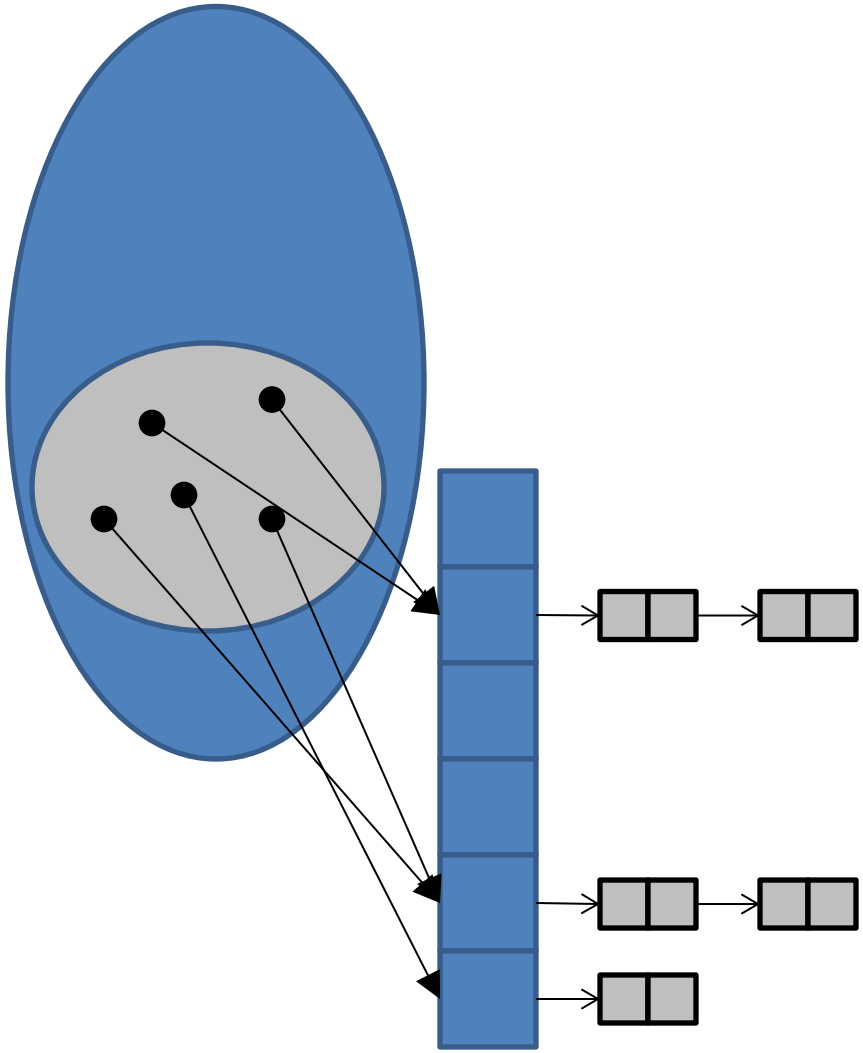


Average-case performance

First, let's make some assumptions

- Simple, **uniform** hashing
  - Hash function **uniformly distributes** keys through array

- **Efficient** hash function
  - We'll assume h can be computed in $\boldsymbol{O(1)}$ time
    - Or at least that it doesn't depend on the number of elements n in the hash table

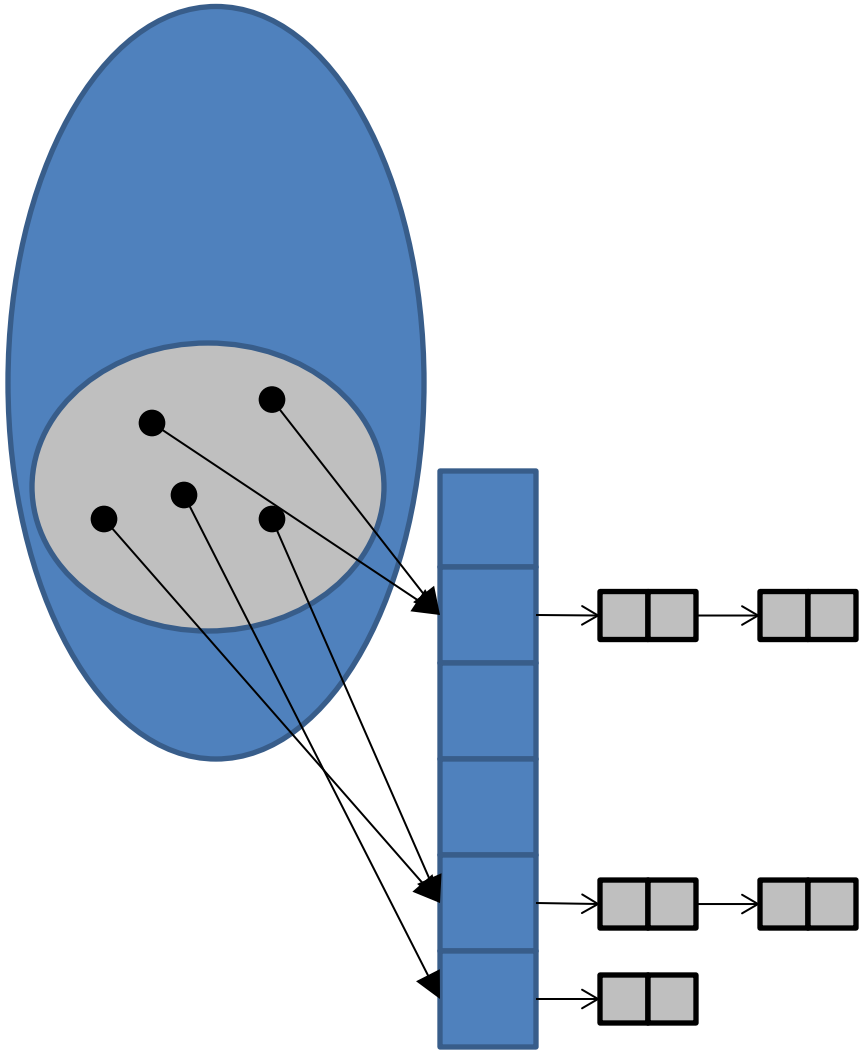


Average-case performance

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**



Average-case performance

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

Remember that $\Theta(n)$ is like $O(n)$, except it's both an upper and lower bound

So it's a stronger assertion

… and if something is $\Theta(1 + \alpha)$, it's also $O(\alpha)$

Average-case performance

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

Proof:

- The hash table has $\boldsymbol{n}$ **keys**
  - and so n linked list cells
- But $\boldsymbol{m}$ **lists** (one for each array cell)
- So the **average list length** is $\frac{n}{m} = \boldsymbol{\alpha}$
- Time to search is
  - Time to **compute hash** + time to **search list**
  - $\Theta(1) + \Theta(\alpha) = \Theta(1+\alpha)$

Average-case performance

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

Proof:

Average-case performance

Lemma: The total number of linked list elements searched while inserting n items into an empty hash table is on average $(n-1)n/2m$

Proof:

- When we **insert the ith key**, we perform an unsuccessful search of the list for its hash bucket
- At that point, there are $\boldsymbol{i-1}$ **keys**
- So the load factor is $\boldsymbol{\alpha = (i-1)/m}$
- But the **expected number** of searches is $\boldsymbol{\alpha}$
- So the **total number** of search operations is:

$$\sum_{i=1}^{n} \frac{i-1}{m} = \frac{1}{m}\sum_{i=1}^{n} i-1$$

$$= \left(\frac{1}{m}\right)\left(\frac{(n-1)n}{2}\right)$$

$$= \frac{(n-1)n}{2m}$$

Average-case performance

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

Proof:

- The number of list elements searched for a successful search is
  - The number of elements searched **when the key was originally inserted**
  - Plus 1
- How many elements were searched when the key was inserted? (on average)
  - The total number of elements searched while **inserting all the elements in the table**
  - **Divided by** the number of elements in the table

$$= \left(\frac{1}{n}\right)\left(\frac{(n-1)n}{2m}\right) = \frac{n-1}{2m} = \frac{n}{2m} - \frac{1}{2m} = \frac{\alpha}{2} - \frac{1}{2m}$$

Average-case performance

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

Proof:

- So the total number of elements searched (on average) is

$$1+\frac{\alpha}{2}-\frac{1}{2m}$$

- And so the total execution time is

$$\Theta(1)+\Theta\left(1+\frac{\alpha}{2}-\frac{1}{2m}\right)=\Theta(1+\alpha)$$

But wait…

- If
  - It's $\Theta(1 + \alpha)$
  - and $\alpha = \frac{n}{m}$
  - doesn't that sort of make it $O(n)$?

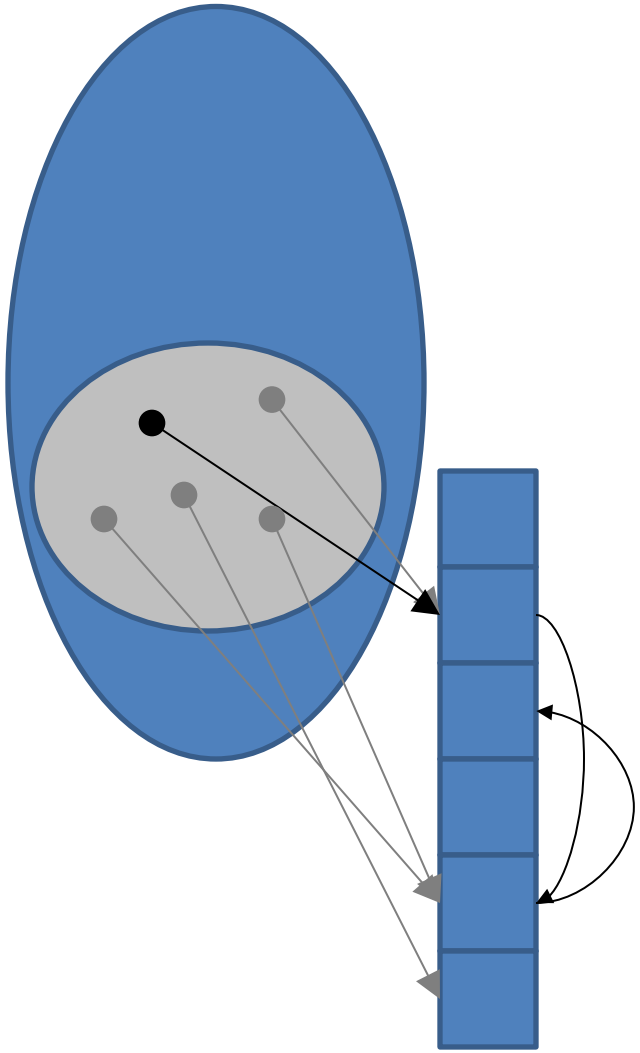
- Yes, but…
  - While time **depends on n**
  - it also **depends on m**
  - So you can **compensate** for larger values of n, by **increasing m** to bring the load factor down again

But wait…

- So really, it just means that you need to make sure the table is **big enough** for the number of entries you're going to have in it

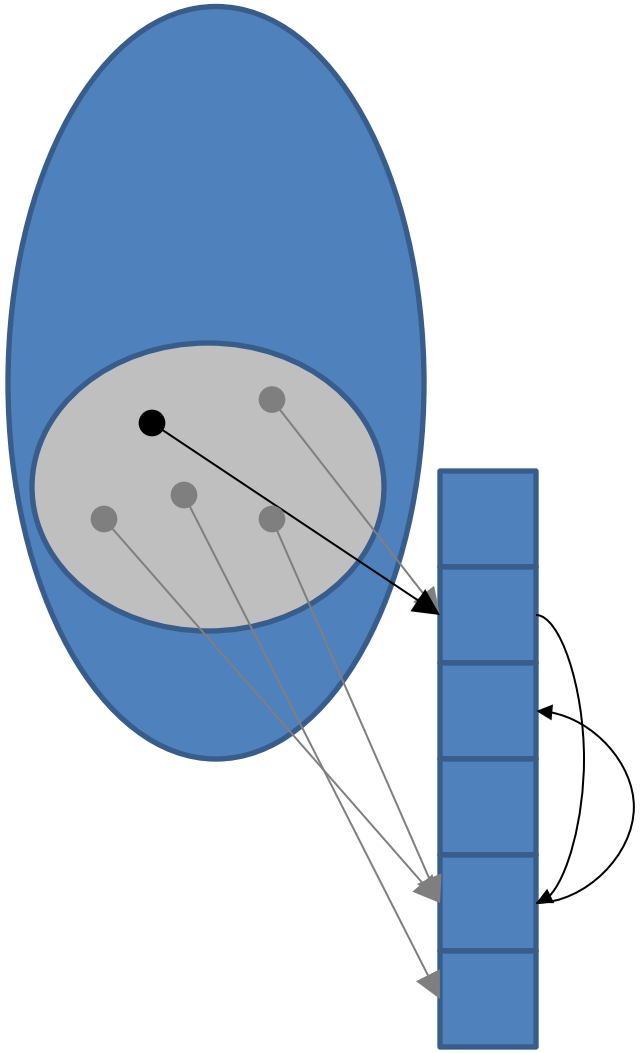
- In fact, it's possible to make tables that **grow automatically** to keep the load factor down
  - We'll talk about this later

Open addressing



- Instead of using a linked list
- We can resolve collisions by **trying successive locations** in the hash table
- Note: we'll assume now that the array is an array of key/value pairs

Open addressing



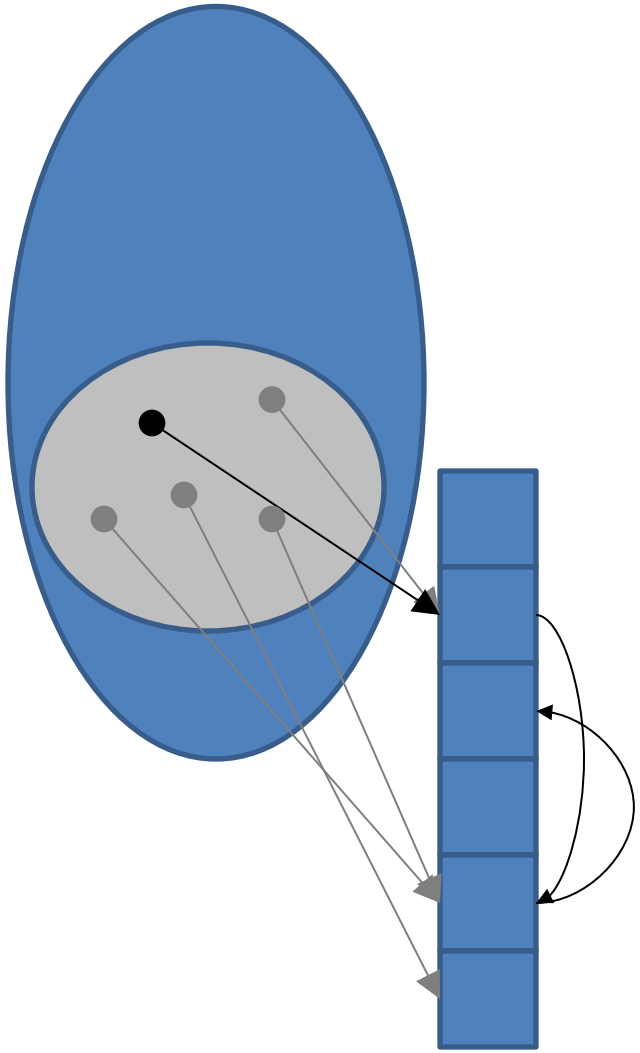
- We redefine the hash function h to take a **second argument** between 0 and $m - 1$

$$h : U \times M \rightarrow M$$

- where $M = \{0, 1, \ldots m - 1\}$

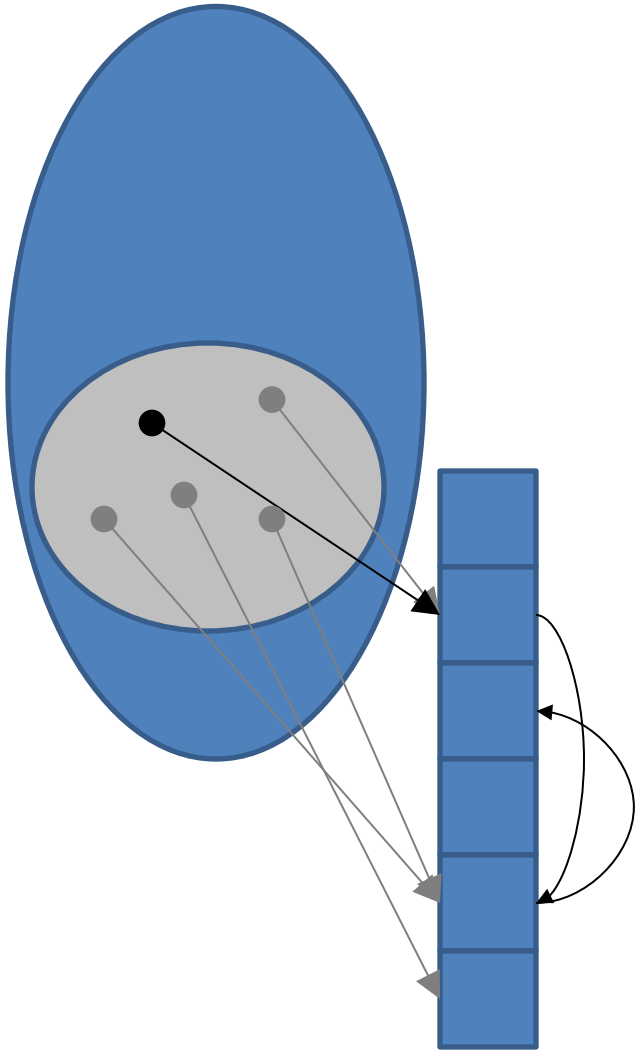
- The second argument specifies **which attempt** we're on (first, second, etc.)

Open addressing



```
void Store(key, value) {
  for (int i = 0; i<m; i++) {
    int j = h(key, i);
    if (array[j].key == key)
      array[j].value = value;
    else if (array[j].key==null) {
      array[j].key = key;
      array[j].value = value;
      return;
    }
  }
  throw new Exception("table full");
}
```

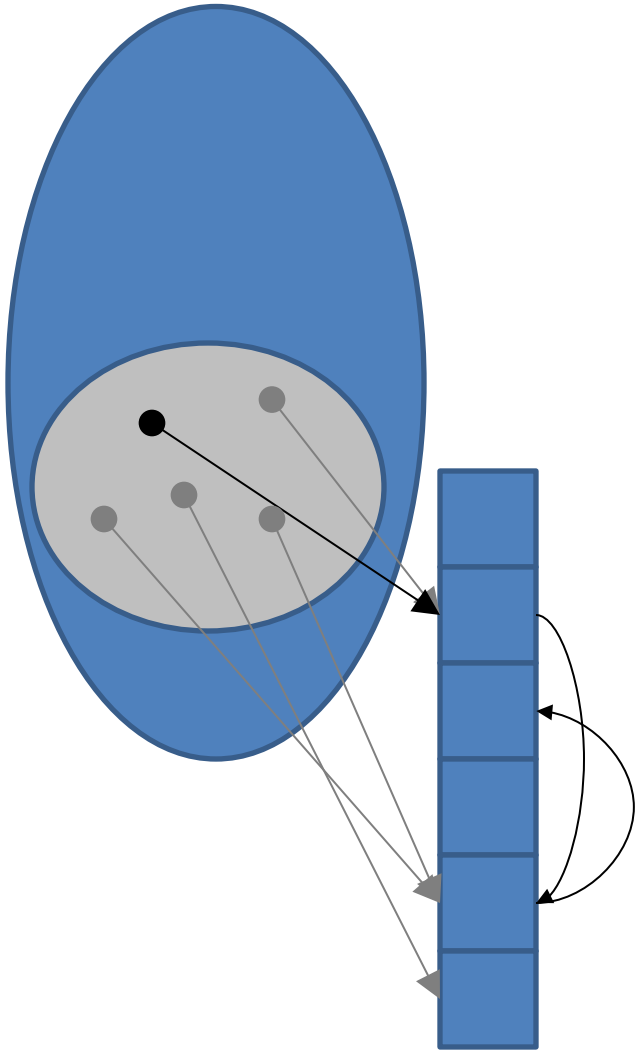
Open addressing



```
void Store(key, value) {
  for (int i = 0; i<m; i++) {
    int j = h(key, i);
    if (array[j].key == key)
      array[j].value = value;
    else if (array[j].key==null) {
      array[j].key = key;
      array[j].value = value;
      return;
    }
  }
  throw new Exception("table full");
}
```

Question: why do we **give up after m iterations**?

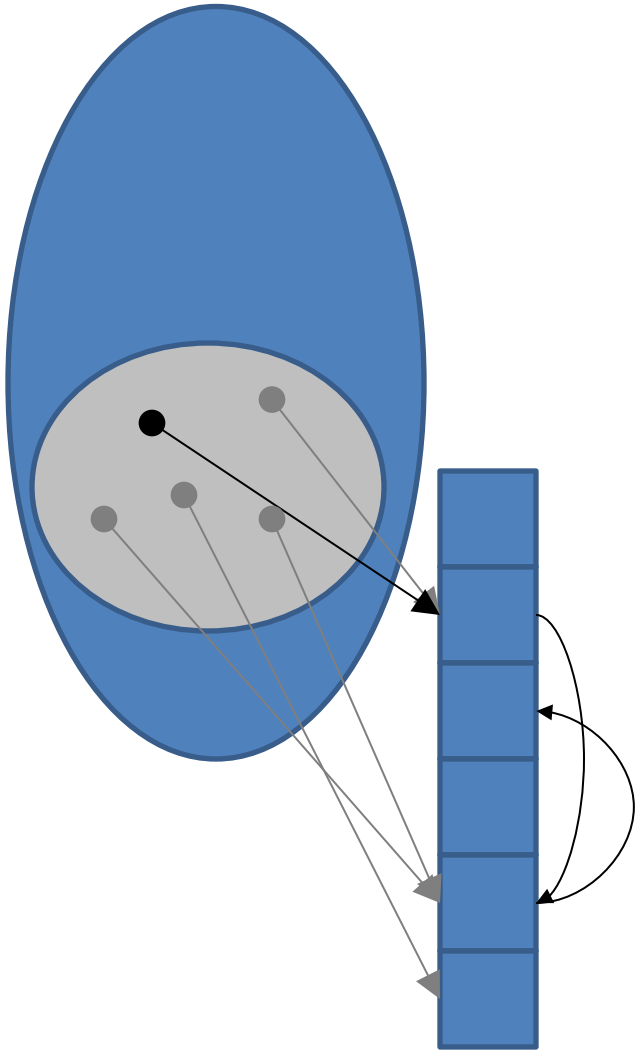
Open addressing



```
void Store(key, value) {
  for (int i = 0; i<m; i++) {
    int j = h(key, i);
    if (array[j].key == key)
      array[j].value = value;
    else if (array[j].key==null) {
      array[j].key = key;
      array[j].value = value;
      return;
    }
  }
  throw new Exception("table full");
}
```

Because the array only has **m elements**

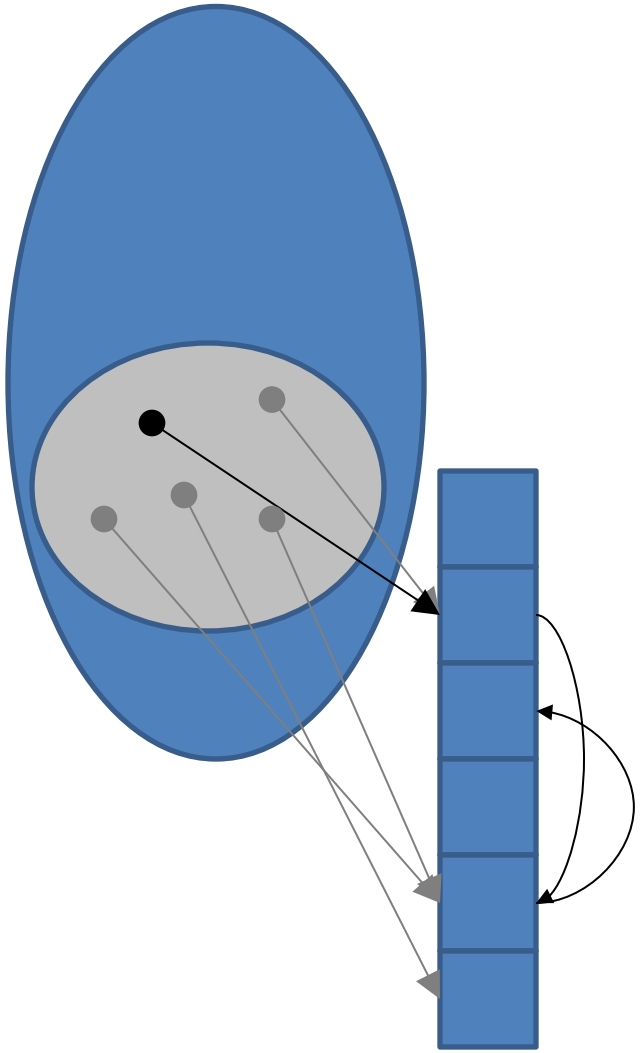
Open addressing



```
void Store(key, value) {
  for (int i = 0; i<m; i++) {
    int j = h(key, i);
    if (array[j].key == key)
      array[j].value = value;
    else if (array[j].key==null) {
      array[j].key = key;
      array[j].value = value;
      return;
    }
  }
  throw new Exception("table full");
}
```

So we've **tried them all**

Open addressing



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

Probe sequences

- The two-argument hash function defines a whole sequence of locations to check
- Called the **probe sequence**
- In practice, the two-argument hash function is built from
  - A **one-argument hash** function
  - And some **extra magic**

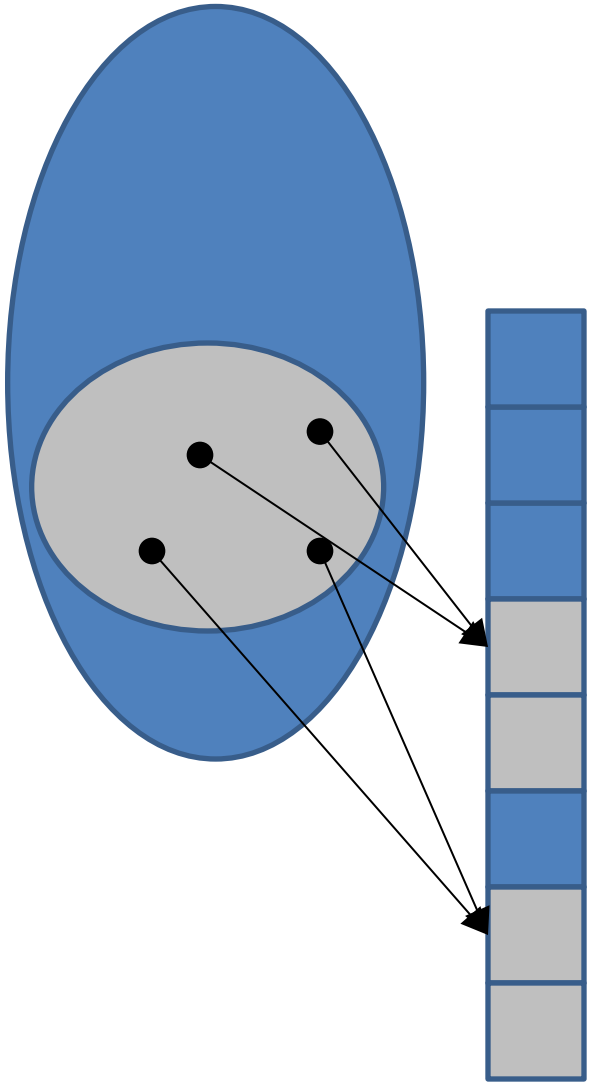
Linear probing

$$h(k,i) = (h'(k) + i) \bmod m$$

- We start at the location defined by a normal hash function h'
- And then we
  - Search **consecutive cells** (adding i does this)
  - **Wrapping around** the end to the beginning if we run off the end of the array (mod'ing by m does this)

Clustering

- Linear probing produces **long runs** of occupied cells
- Any **new key that hashes to one of those runs** has to go all the way at the **end of it**
- So performance isn't very good



Quadratic probing

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

- We can hair it up by adding a **quadratic polynomial** in i rather than just i
- Makes probe sequences more **complicated**
  - And **less prone to clustering**
- But still, two keys that **hash to the same h' value** will probe the same sequence

Double hashing

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

- We can solve a lot of the problems of linear and quadratic hashing by taking a linear combination of **two different hash functions**, h_1 and h_2

- Now two keys that **hash to the same h_1** value will still have **different probe sequences**
  - So long as their h_2 values are different

Performance of open addressing

Theorem: assuming uniform hashing (no clustering), the **average number of probes** in an **unsuccessful** search is at most

$$\frac{1}{1-\alpha}$$

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

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}+\frac{1}{\alpha}$$

Proofs:
Outside the scope of this class, but in the CLR book.

Variations on the interface

- Sometimes we think of a hash table **as just holding objects**
  - **Not key/value pairs**
- Then the hash table just stores objects based on some hash we can compute from the object
  - C# calls these "**HashSets**"
- The CLR book assumes this interface

Example: canonicalizing strings

- **String comparisons** are expensive
  - Have to compare them character by character
  - $O(n)$ time
- What if we only had **one copy** of each possible string?
  - Then we could just ask if two string variables pointed to the **same object**
  - $\boldsymbol{O(1)}$ time
- This technique is used in a number of systems
  - Including .NET

```
string MakeString(char[] chars) {
    string probe =
      StringTable.Lookup(chars);
    if (probe != null)
      return probe;
    else {
      string newString =
        new string(chars);
      StringTable.Add(newString);
      return newString;
    }
}
```

Designing hash functions

Remember a hash function

- Maps a **key**
- To an **integer between 0 and $m-1$**

$$h : U \to \{0, 1, \ldots, m-1\}$$

We want a hash function that:

- Is easy (read: **fast**) to compute
- Has the statistical property that
  - For a randomly chosen key
  - **Each possible hash** value is **equally likely**

Designing hash functions

Break problem up into **two parts**:

1. Map the key to an **integer** (often a large one)
   - E.g. add all the characters of a string together
   - This gives us an integer, but usually with the **wrong range** and **wrong statistical properties**
2. **Remap the integer** to a hash value in the right range with the right statistical properties

Mapping to a large integer

- Easy for small objects
  - Integers, characters, Booleans, pointers, etc.
  - Just take the bits of the representation and **cast them to an int**
    - int i = (int)whatever;
- For compound objects (hashing on **object identity**)
  - Use **address** of object in memory (C, C++)
  - Assign **serial number** in some hidden field of the object (Java, C#)
- For compound objects (hashing on **object contents**)
  - **Convert individual fields** to integers
  - **Combine integers** into a bigger, or at least different integer

Radix notation

Basic **idea**:

- Suppose we have an **array of numbers** from 0 to 9
  - E.g. { 1, 7, 3, 2, 8 }
- **Write them all down** as if they were **one number**
  - E.g. 17328
  - Or:

$$1 \times 10^4 + 7 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 8 \times 10^0$$

More **formally**:

- Given an array a of integers, each in the range 0 to $r - 1$
- We map a to the integer:

$$\sum_i a[i] r^i$$

Radix notation

- A **string** is essentially an **array of characters**
- ASCII characters are essentially integers in the range 0-127

- So we can map any **string s to the integer**

$$\sum_i s[i]128^i$$

- That makes for a very **large integer**
  - Overflows 32 bits after 4 characters
- But sometimes it's practical

- We're mostly just mentioning radix notation because it's the method the CLR book mentions

Summing

- **Radix notation** has the possibly useful property of being **1:1**
  - There's a unique integer for every string (or array, or whatever)
  - And vice-versa
- But **who cares** if we're hashing?
  - We're just going to run it through the hash function, which isn't 1:1 anyway
- If we don't care about reversability, we can just **add all the elements together**:

$$\sum_i s[i]$$

- Which gives us a much more manageable sized integer

Exclusive-or'ing

- Another common option is to use bitwise exclusive or (often written **xor** or $\bigoplus$) instead of addition
  - In C/C++/C#, it's the "^" operator

$$\bigoplus_{i} s[i]$$

```
int XorString(string s) {
  int result = 0;
  foreach (char c in s)
    result = result ^ c;
  return result;
}
```

- Bitwise xor has a lot of the **nice properties of addition**, but guarantees the result is the **same number of bits** as the input
  - Which is sometimes useful

XOR (1-bit case)

- The XOR of two bits is:
  - 1, if one or the other of the inputs is 1
    - But not both
  - 0, otherwise

- Ways of thinking about XOR
  - 1 if the inputs are different, 0 otherwise
  - Add the two bits (in base 2) and throw away the carry
  - Flip the first bit if the second is 1
  - Flip the second bit if the first is 1

input	0	1
0	0	1
1	1	0

XOR (n-bit case)

- Break inputs up into bits
- Result is the "bitwise" XOR of the two groups of bits
  - ith bit of output is the XOR of the ith bits of the respective inputs

Inputs	00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

Shifting

- Summing and xoring have two **potential problems**
  - The integer might be ***too small***
  - If you **swap two characters** (or array elements, or whatever), you get the same integer back
    - Potentially more hash collisions
- Basically a variant on radix notation
- We can fix that by doing a **left shift** (equivalent of multiplying by 2) each time we add in another element

```
int StringToInt(string s) {
  int result = 0;
  foreach (char c in s)
    result = (result<<1) ^ c;
  return result;
}
```

Hashing to a small integer

- Now we want to **hash the big integer** down to
  - A small integer in the range: 0 … m
  - Such that each output is equally common
    - So that keys are equally distributed in buckets

Division method

- The simplest method is to **divide by m** and take the **remainder**

$$h(i) = i \bmod m$$

- Causes problems if **multiples of m are common** in the keys
  - For example if keys are prices (so that numbers like $3.99, $4.99 are disproportionately common)
  - then having m be a multiple of 10 might be bad

- Powers of 2 can also be problematic
  - All numbers with the same low-order bits map to the same hash

- Conventional wisdom is to choose m to be a **prime number**

Multiplication method

- **Multiply** key by some A between 0 and 1
  - Knuth suggests $(\sqrt{5} - 1)/2$
- Throw away the integer portion and **take just the fraction**
- **Multiply by m**
- Take the "**floor**" of the result
  - Fancy way of saying "**round down** to the nearest integer"

Formally:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

- There's a clever way of computing this using integer arithmetic and bit masking operations
- But we're running short on time
  - And you can look it up in the CLR book if you ever need to implement it

Universal hashing

- Any hash function will have collisions
- So no matter what hash function you choose, there's some set of keys that will make it perform badly

- You can mitigate this by choosing your hash function **randomly** when you make the hash table

- **Universal hashing** is hashing with a random function

- Take your key k
  - Assume it's a 32 bit (4 byte integer)
  - Let x_0, x_1, x_2, x_3 be the individual bytes of k (as smaller integers)
- Pick 4 random integers a_0, a_1, a_2, a_3 between 0 and m-1

- Now define:

$$h(k) = \sum_{i=0}^{3} a_i x_i \bmod m$$

Perfect hashing

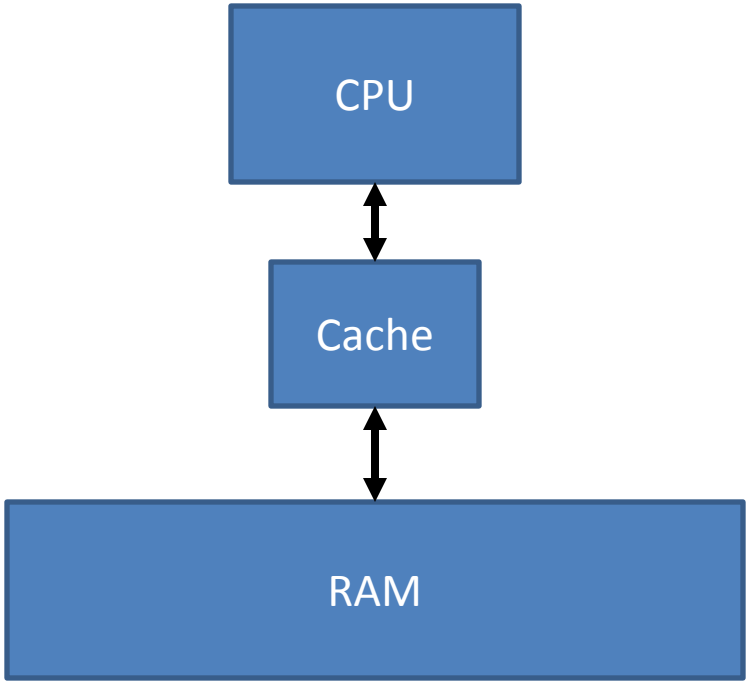
- Sometimes, there's a **small, fixed, set** of objects you want to hash
  - E.g. a compiler keeping track of keywords of its language (like if, else, switch)
- **Perfect hashing** is when you choose a hash function in advance that you know will be **collision-free** for your keys
- How do we find a perfect hash function?
- See the discussion in the book, but in the end, it will amount to
  - Writing an algorithm
  - To **try different hash functions**
  - Until it finds one that's perfect

Hashing in C#

- The **standard C# libraries** provide built-in hash table implementations
  - System.Collections:
    - **HashTable**
  - System.Collections.Generic:
    - **Dictionary<KeyT, VT>**
- These both use the **GetHashCode()** method virtual function, which is defined in the Object class
  - Should return a 32 bit hash
  - Hash tables themselves will compress it down to a small integer
- So if you want to use the built-in hash tables you need to **override** the **GetHashCode()** method for you classes
- A common thing to do is to just **call GetHashCode()** on all the fields of your object, and **XOR** all the results together

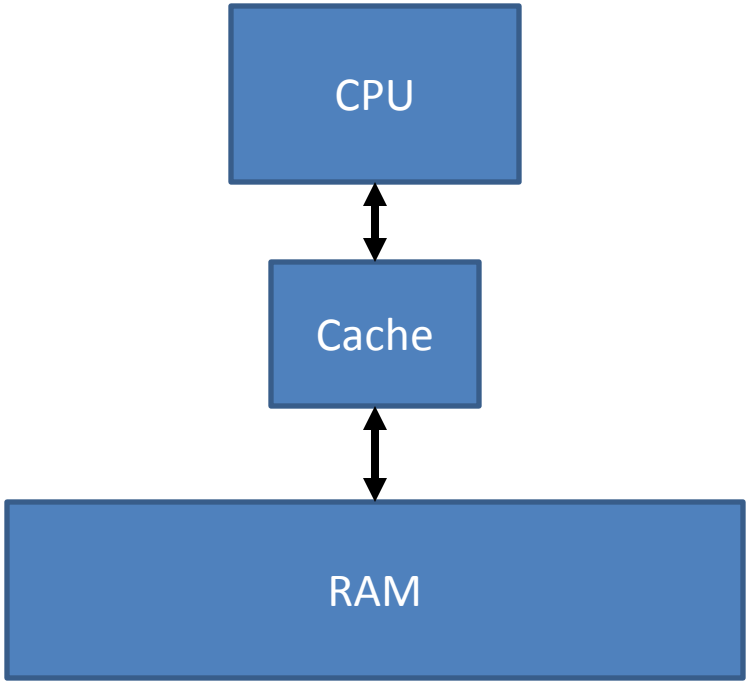
Simple example: cache memory

- Your CPU is processing an instruction every **nanosecond** or so
- However your **RAM takes 50+ ns** to do a read or write operation
- What to do?



Simple example: cache memory

- Caching!
  - Buy a small amount of very fast RAM
  - Use it to build a **hardware hash table**
    - Using low-order bits of memory address as the hash code
  - Remember recently access locations in RAM
  - **Throw away** old data when we have a **collision**
- First memory reference is very expensive
  - Subsequent references to the same location run fast



Other applications of “hash-like” functions

- **Error detecting** codes
  - Used to detect (or even correct) recording or transmission errors
- **Cryptographic hash** functions
  - A compact “**signature**” for a piece of data
    - Much smaller than original document
    - Very unlikely to have the same signature as another document

Error detecting codes

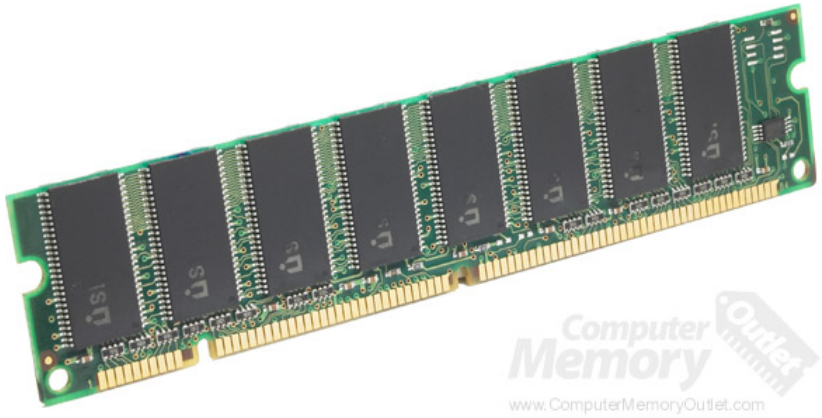
- **Checksums**
  - Add all the bytes of data together
  - Store the sum along with the data
- When reading data back from disk
  - **Recompute** the checksum
  - **Compare** it to the stored checksum
  - If they don't match, there was a recording error

Error detecting codes

- There are many sophisticated error detection codes in use
  - **Cyclic Redundancy Check** (CRC)
    - Used in many disk drives
  - **Reed-Solomon** codes
    - Used in CD-ROMs

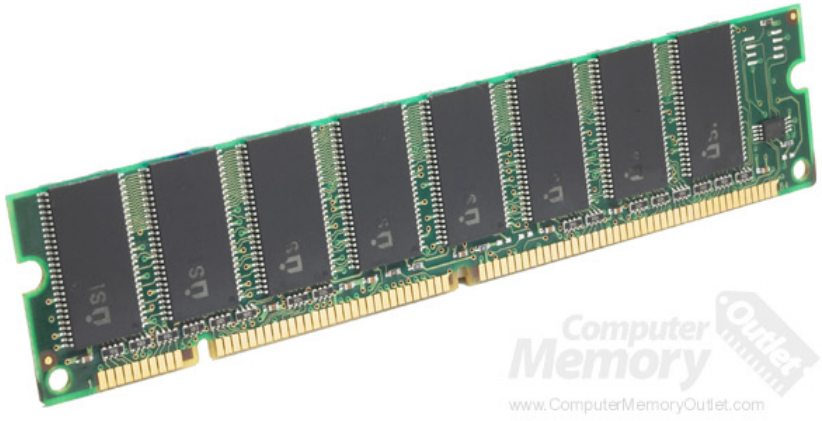
Parity

- Parity is a simpler, faster algorithm, used for checking data in RAM
- One additional bit is stored: the **XOR of all** the other bits
  - 1 if there are an odd number of 1 bits in the byte
  - 0 if there are an even number of 1-bits in the byte
  - Can be thought of as a 1-bit checksum
- Lets you **detect any single-bit error**



Error-Correcting Codes

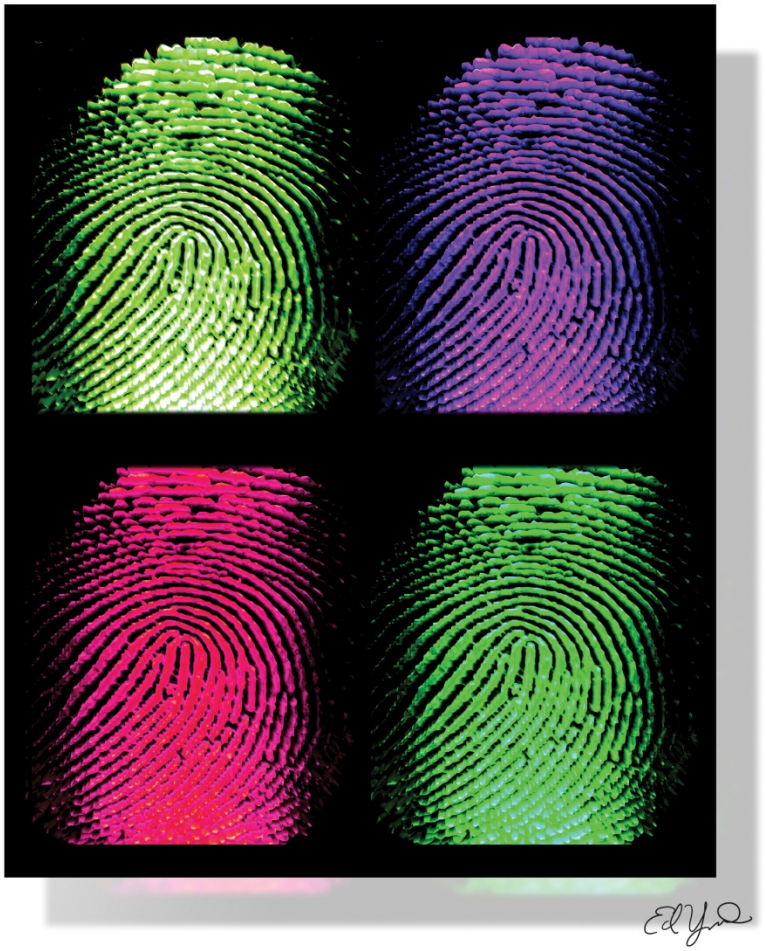
- High-end computers such as serves store **multiple parity bits** with each byte
  - The parity of all bits in the byte
  - The parity of just the first 4 bits
  - The parity of bits 1 and 2, and 5 and 6



- Examing which of these parity checks fails lets you
  - Detect double-bit errors
  - **Correct** single-bit errors

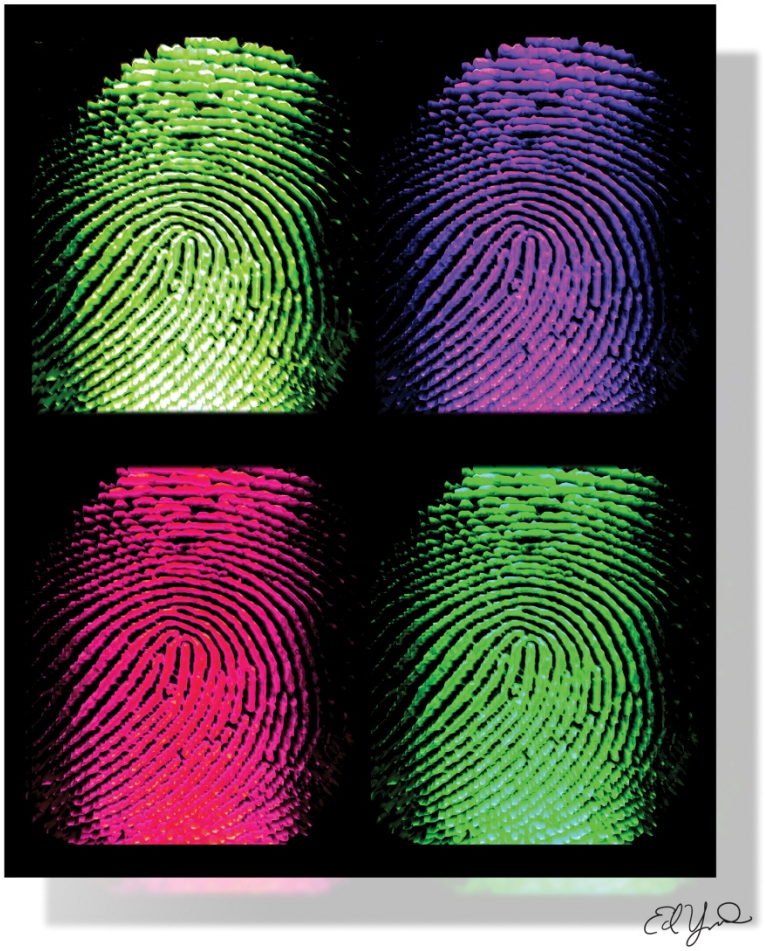
Cryptographic hashes

- Cryptographic hashes are designed to make it **very hard** to deliberately make a data object that has the same hash code as another object
- Used as compact "**fingerprints**" for larger documents
- Popular algorithms
  - MD4/MD5
  - SHA-1, SHA-256, SHA-512



Uses

- **Tamper** detection
  - Used for digitally signing messages
  - Used e.g. in bittorrent to ensure data received matches the torrent
- Identifiers for documents
- **Saving** transmission **bandwidth**
  - Send server the hash of a file before sending the file
  - Server tells you if it already has the file



Single-instance stores

- Companies are rolling out **cloud services** for music
  - Store you music “in the cloud” (i.e. on servers)
  - Stream it from any machine you like
- Amazon, google, Apple, etc.

- That’s a lot of data!
  - How do they make that work?

- Key idea: many people, few songs
  - Two people who bought the same song from the same provider have **exactly the same file**
  - Before storing in the cloud,
  - Check to see if the server already has a copy of the same file

- **Only one copy** of each song is stored in the cloud

- Saves both **memory** and **bandwidth**

Basic algorithm

To add a file:

- **Client sends hash of file** to server
- **Server checks hash table**
  - Hash **already present**?
    - Server adds pointer to **existing file** to user's directory
    - No data stransmitted
  - **Not present**?
    - **New** file
    - Client **transmits full file**
    - Server **adds** file to **hash table**

- But what about **collisions**?
- Let's say we use the SHA-1 secure hash function
  - Produces a 160 bit hash from a file
  - So there are 2^{160} possible hashes
  - So the chance of two songs having the same hash is **1 in 1461501637330902918203684832716283019655932542976**
  - Or $> 10^{49}$
  - More than the **square of Avogadro's number**

Dropbox's privacy policy

- A complaint against Dropbox for **deceptive trade practices** was filed with the FTC in 2011
- It's was filed by a Ph.D. student who
- He claimed
  - Dropbox's claims that its **employees couldn't read your files** were **false**
  - Because they were using a **single-instance store**
- But dropbox **didn't tell people** that they were using a single-instance store
  - So **how did he know**?
- Any **why would it matter** to a user?

Reading

- CLR chapter on Hash Tables