# HASHING
# and
# HASHTABLES

# What is a hashtable?

- A data structure that stores key-value pairs.

- This is different from the ADTs we’ve seen so far, which only hold values.

# Hashtables

- A hashtable stores *key-value* pairs.
- The value is the element or object that you really care about (the data).
- The key allows you to find the value **quickly** in the data set.

- The main reason to use a hashtable is to find a value without having to search the entire collection.

# Examples of Keys and Values

- English language dictionary
    - key: the word
    - value: the word, pronunciation, definition, ancestry of the word, etc.
- Phone book
    - key: person’s name
    - value: person’s name, phone number, address, email address
- Social network
    - key: person’s login name
    - value: person’s login name, personal information, photos, etc.
- Voter database
    - key: phone number
    - value: voter information (name, address, phone, gender, income, registered party, etc.)

# Hashtable

- You can think about a hashtable like an array.
  - Voter[] hashtable- a Voter stored in each position

| Index | Voter Info (Including Phone) |
|---|---|
| 0 | Val Voter (555-1230, address, etc.) |
| 1 | Ursula Undecided (555-7891, address, etc.) |
| 2 | Sal StaysHome (555-1432, address, etc.) |
| 3 | Irma Independent (555-0233, address, etc.) |
| 4 | Donna Donor (555-6234, address, etc.) |
| 5 | Vicky Volunteer (555-2195, address, etc.) |

# Hashtable

| Index | Voter Info (Including Phone) |
|---|---|
| 0 | Val Voter (555-1230, address, etc.) |
| 1 | Ursula Undecided (555-7891, address, etc.) |
| 2 | Sal StaysHome (555-1432, address, etc.) |
| 3 | Irma Independent (555-0233, address, etc.) |
| 4 | Donna Donor (555-6234, address, etc.) |
| 5 | Vicky Volunteer (555-2195, address, etc.) |

Give me the voter information for the voter at 555-7891.

Let me search through all the voters to find a match.

**List**

Sure, that voter info is at index 1.

**Hashtable**

# Hashtables

- If we are storing in a list, we’d have to search the whole list until we found a voter with a matching phone number.
  - O(n) or O(log n)
- With a hashtable, use the key to figure out where the voter is. We then only have to look in often only 1 location (or at least a number not dependent on n).
  - O(1)

# Hash Functions

- So how do we go from the key to the index?
  - How do we get from the key “555-7890” to index 1?
- A hash function!
  - A hash function takes in a key and converts it to an index within the array.

| Index | Voter Info (Including Phone) |
|---|---|
| 0 | Val Voter (555-1230, address, etc.) |
| 1 | Ursula Undecided (555-7891, address, etc.) |
| 2 | Sal StaysHome (555-1432, address, etc.) |
| 3 | Irma Independent (555-0233, address, etc.) |
| 4 | Donna Donor (555-6234, address, etc.) |
| 5 | Vicky Volunteer (555-2195, address, etc.) |

# Hash Functions

- A (very bad!) hash function might be:
  - hash(phone) = last digit

| Index | Voter Info (Including Phone) |
|---|---|
| 0 | Val Voter (555-1230, address, etc.) |
| 1 | Ursula Undecided (555-7891, address, etc.) |
| 2 | Sal StaysHome (555-1432, address, etc.) |
| 3 | Irma Independent (555-0233, address, etc.) |
| 4 | Donna Donor (555-6234, address, etc.) |
| 5 | Vicky Volunteer (555-2195, address, etc.) |

# Hash Functions

- hash(phone) = last digit
    - We’ll learn more about good/bad hah functions later.
- For now, it’s important just to know that the purpose of the hash function is to convert a key to an index in the table.
- This is what allows for O(1) access to elements in the table.

# “Hash” in Java

- You’ll hear the word “hash” in a lot of different ways in Java and in computer science more generally.

- Computer science: a “hashtable” is the ADT that stores key-value pairs.
    - Also called a dictionary, map, table.
- In Java:
    - a method: hashCode()
    - a class: HashSet
    - a class: HashMap
    - others

# The hashCode() Method

- This method is inherited from Object
- It is meant to define a hash function for a given object
  - hash(myObject) = someHashCode // this is how we've been describing it
  - int someHashCode = myObject.hashCode(); // this is how it looks in Java
- The inherited version of hashCode returns a code based on the memory address of the object
- When we write our own classes, we usually override toString() and equals(…). We should also override hashCode().
  - We must follow the *contract* defined by equals and hashCode.

# The equals(…) Contract

- Your equals method must meet the following criteria.

- Reflexive: x.equals(x) is true
- Symmetric: x.equals(y ) if and only if y.equals(x)
- Transitive: if x.equals(y ) and y.equals(z) then x.equals(z)
- Consistent: x.equals(y ) returns the same value upon multiple invokations (unless you have modified the objects)
- If x is not null, then x.equals(null) is false

# The hashCode(…) Contract

- Your hashCode method must meet the following criteria.

- Consistent: x.hashCode() should return the same value upon multiple invokation within the same program (unless information used to determine the hash code is modified)
- Consistent with equals: If x.equals(y ) then x.hashCode() == y.hashCode()

# hashCode() and equals(...)

- Two objects that are equal must have the same hashCode.
- But the reverse is not necessarily (and not required to be) true.
  - Two objects that are *unequal* could still have the same hashCode.
  - This would result in a collision. It is allowed.
  - Producing distinct hash codes for unequal objects will result in improved performance, but it is not a requirement.

# How to Override hashCode()

- Option 1: Write a method that combines all of the hash codes of the instance data variables for that class and combine them with prime numbers.
- For example, consider a Movie class:

```
@Override
public int hashCode() {
   return 7 * title.hashCode() +
       11 * director.hashCode() +
       13 * new Integer.valueOf(year).hashCode();
}
```

- eclipse (and many other IDEs), will automatically generate such code for you
  - Go to Source > Generate hashCode() and equals()

# How to Override hashCode()

- Option 2: As of Java v7, we have an awesome shortcut we can use with the static Object**s**.hash method
  - (note the "s" in the class name: the Objects class
- With this method, we just send in all our instance data variables (primitive or object) and the method will calculate a hash for us:

```
@Override
public int hashCode() {
   return Objects.hash(title, director, year);
}
```

- Hashing has never been so easy!

# HashSet

- HashSet is a class that implements the Set interface.
  - A set is a collection of unordered objects that does not allow duplicates.
- Form the client view, a HashSet is just a set.
  - It allows you to add and remove elements from a group.
  - It will not allow you to add a duplicate element.
  - We take the client view when using HashSet in our code.
  - We specify the values (the objects we add to the set) and HashSet figures out the key (using the objects’ hashCode() method)
- HashSet is a very efficient class to use when you do not need elements to be ordered and you don't have any duplicates.
- Be extra certain to override the equals and hashCode method for whatever kind of objects you're putting into your HashSet!

# HashSet

- Behind the scenes, HashSet is implemented with a hash table (hence the name "Hash"Set).
- To check if the set contains an element, HashSet looks up its hash code and sees if that bucket contains the element.
  - This supports very fast lookup: essentially O(1) instead of O(n), which you'd have with a list.
- The add method also uses the hash code to first check if the element is already in the table.
  - the element is only added if it's not already there.
- Again: be extra certain to override the equals and hashCode method of your class!

# HashMap

- HashMap implements the Map interface.
  - A map is a data structure that stores key-value pairs.
  - This is essentially Java's hash table class.
- This class allows you to specify what type of object will be the key and what type of object will be the value.
  - You do not have to use an object’s hashCode() method- you get to choose what the key is.
  - You specify what type of object will be the key (K) and what type of object will be the value (V).
- Example: HashMap<String, Movie>
  - key is a uniqueMovieID String and value is a Movie object
- Example: HashMap<Integer, Employee>
  - key is employee id and value is the Employee object

# HashMap

- The put(key, value) method will replace (and return) an element if one already exists at the specified key location.
- The get(key) method will return the element associated with the specified key or null if there is no element.
- You can also access a collection of all the keys with the keySet() method.
    - This allows you to traverse the collection of keys and access each value in the map.
- Behind the scenes, HashMap is also implemented with a hashtable!
- HashMap is an efficient class if you do not need your elements to be ordered and you do not need to store multiple values at the same key location. (You *can* do this, but it’s not quite as straightforward.)

# Examples

- Review the examples of overriding hashCode(), using HashSet with the hashCode() method, and using HashMap by defining our own keys.

# Hashtable: Behind the Scenes

- Let’s look at building our own hashtable to see how things work behind the scenes.

# The Hash Function

- Hashing is the method that determines the location (index) of a value based on the key.
- It is the process of using the hash function.
    - A *hash function* has one input (the key) and one output (the index/location)

hash(key) = index

# Hash Functions

- Let’s revisit our very bad hash function:
  - hash(phone) = last digit
- Many different phone numbers will have the same last digit. So this isn’t a very good way to go from a key to an index.

| Index | Voter Info (Including Phone) |
|---|---|
| 0 | Val Voter (555-1230, address, etc.) |
| 1 | Ursula Undecided (555-7891, address, etc.) |
| 2 | Sal StaysHome (555-1432, address, etc.) |
| 3 | Irma Independent (555-0233, address, etc.) |
| 4 | Donna Donor (555-6234, address, etc.) |
| 5 | Vicky Volunteer (555-2195, address, etc.) |

# Perfect Hash Function

- A *perfect hash function* maps each key to one unique address (or index in the table).

# Example- Perfect Hash Function

- A perfect hash function might just use the whole phone number.
- hash(phoneNumber) = phoneNumber
- Each phone number will be linked to one unique position in the array (one index).
  - So each phone number can be used to locate one voter.
- This is good because each key has a unique place.
- However… we need 9,999,999 places in our table, even if we only have data on 5,000 voters!

# Sparsity

- A hashtable is *sparse* if the space to hold data is greater than the data in the table.
    - Usually: less than √n
- The voter example could create a very sparse table!
- A sparse table is not ideal.

# Revised Hash Function

- We can *compress* the range of possible index values into a size that is closer to the number of locations we actually want to use.
- One way to compress is with the modulus (remainder) operator!

location = hash(key) % tableSize

# Revised Hash Function

- Let’s say we have data on 500 voters.
- We could make our table size 1000.
- The hash function is the last four digits.

location = hash (phone) % tableSize

- location = hash(555-1234) % 1000 = 1234 % 1000 = 234
- location = hash(555-7890) % 1000 = 7890 % 1000 = 890

# Revised Hash Function

- Now we need a much smaller table.
  - That’s good!
- However…
  - location = hash(555-6234) % 1000 = 6234 % 1000 = 234
  - location = hash(555-1234) % 1000 = 1234 % 1000 = 234
- Two phone numbers get hashed to the same location!
  - Which number gets put there?
  - Where does the other one go?

# Collisions

- When two keys hash to the same index, we get a *collision*.
- Collisions can be addressed using:
    - Open addressing
    - Separate chaining

# Good Hash Functions

- Minimize collisions and clustering
- Distribute uniformly across the table
- Create a table that isn’t too sparse
- Are fast to compute

# Open Addressing

- When a collision occurs, use another location.
  - Also called *closed buckets* because each location can only store one value.
- How do you choose the other location?
  - Linear probing
  - Other ways

# Linear Probing- Adding to the Hashtable

- When we have a collision, *probing* is the process used to find another open location in the hashtable where we can store our value. Linear probing is one approach.
- if location *k* is occupied, we look to see if *k+1* is occupied. If it is, we look in *k+2*, and so on.
- We store the entry in the first available location.
- If this *probe sequence* reaches the end of the table, we circle back to the beginning of the table and keep looking.
- If we get back to where we started without finding an open space, that means the table is full and we need to create a new larger table and re-hash our keys.

# Linear Probing- Retrieving from the Hashtable

- We search in the location for our key.
- If the key is there, we are done.
- If a different key is there, we have to keep looking, so we look in the next spot.
- We keep going until we either a) find the key, b) hit an empty spot (key is not there), or c) loop back around to where we started (key is not there and the table is full).

# Example

- Voter data. The key is the last four digits of the phone number.
- The location of an object is the key % tableSize.
  - We'll use a table size of 101.

TABLE_SIZE = 101

hash(phone number key) = last four phone number

location = last four digits % TABLE_SIZE

h(555-1214) = 1214%101 = 2

2 is empty

store voter information (& key) at position 2

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| … | … |
| 100 | null |

h(555-1214) = 1214%101 = 2

2 is empty

store voter information (& key) at position 2

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| … | … |
| 100 | null |

h(555-2022) = 2022%101 = 2

2 is occupied

3 is empty

store voter information (& key) at position 3

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| … | … |
| 100 | null |

h(555-2022) = 2022%101 = 2

2 is occupied

3 is empty

store voter information (& key) at position 3

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| … | … |
| 100 | null |

h(555-3033) = 3033%101 = 3

3 is occupied

4 is empty

store voter information (& key) at position 4

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| … | … |
| 100 | null |

h(555-3033) = 3033%101 = 3

3 is occupied

4 is empty

store voter information (& key) at position 4

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | null |
| 7 | null |
| … | … |
| 100 | null |

h(555-2026) = 2026%101 = 6

6 is empty

store voter information (& key) at 6

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | null |
| 7 | null |
| … | … |
| 100 | null |

h(555-2026) = 2026%101 = 6

6 is empty

store voter information (& key) at 6

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 2026 |
| 7 | null |
| … | … |
| 100 | null |

# Example Retrieval

- What if we want to retrieve the voter information associated with phone number 555-3033?
- We find the location: hash(phone) = 3033 % 101 = 3

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 2026 |
| 7 | null |
| … | … |
| 100 | null |

# Example Retrieval

- We check location 3.
  - It is occupied, but with key 2022, not 3033.
- So we check location 4.
  - It is occupied with the key we want, so we can retrieve the voter information.
- This is why we need to store the key with the data- so that when we find a location occupied, we can check to see whether it has the data we really want.

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 2026 |
| 7 | null |
| … | … |
| 100 | null |

# Example Retrieval

- Retrieve the data for the voter with phone number 555-2023.
- We find the location: hash(phone) = 2023 % 101 = 3
- We check location 3.
  - It's occupied, but with the wrong key.
- We check location 4.
  - It's occupied, but with the wrong key.
- We now reach location 5 and it is empty. So we know that this key is not in the table.

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 2026 |
| 7 | null |
| … | … |
| 100 | null |

# Example Retrieval

- Retrieve the data for the voter with phone number 555-2127.
- We find the location: hash(phone) = 2127 % 101 = 6
- We check location 6.
    - It's occupied, but with the wrong key.
- We check location 7.
    - It’s empty so we know the key is not there.
- (Note: If we were to look in 100 and find it occupied, we would loop around and keep checking at 0.)

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 2026 |
| 7 | null |
| … | … |
| 100 | null |

# Example Deletion

- What would happen if we wanted to remove the data associated with key 1214?

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | null |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 2026 |
| 7 | null |
| … | … |
| 100 | null |

# Example Retrieval

- Retrieve the data for the voter with phone number 555-2022.
- We find the location: hash(phone) = 2022% 101 = 2
  - We look in position 2, it's empty, so we would assume our data is not in the table.
- But the data is there!

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | null |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 2026 |
| 7 | null |
| … | … |
| 100 | null |

# Empty States for Open Addressing

- If we are using open addressing with probing, we actually can't use just two states (empty/not), but we need three states:
    - occupied
    - available: used when an item was once there but has been removed
    - empty (null): used when an item has **never** been stored there

# Example Deletion

- Let’s back up to our deletion but use these three states.
- What would happen if we wanted to remove the data associated with key 1214?

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | **available** |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 2026 |
| 7 | null |
| … | … |
| 100 | null |

# Example Retrieval

- Retrieve the data for the voter with phone number 555-2022.
- We find the location: hash(phone) = 2022% 101 = 2
  - We look in position 2, it's available, so we keep looking.
- We look in position 3.
  - It is occupied with the key we want, so we can retrieve the voter information.

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | available |
| 3 | Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 4 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 2026 |
| 7 | null |
| … | … |
| 100 | null |

# Separate Chaining

- Allow multiple entries in a single location.
    - Also called *open buckets*.
- A *bucket* is a location in a hash table that can store more than one value.
- To use this method, we link values together that all belong at the same hash index using linked nodes.
- Separate chaining is a more efficient way to handle collisions, but it requires more memory space than open addressing. But most of the time, we care more about time efficiency than space efficiency, so, in general, separate chaining is used more.

# Separate Chaining

- When we add a value and there is already something in the bucket, we add the new value to the linked chain of values.
- If the elements are unsorted and might contain duplicates, additions should be made with *insert head*. This is the most efficient way to add to a chain of linked nodes.
- If the elements are unsorted but unique, additions should be made with *insert tail*. You need to traverse the chain to make sure there isn't a duplicate in the list. Once you traverse the list, you can add the new element at the end of the chain.
- If the elements are sorted, additions should be made with *insert in order*.

# Example

- Voter data. The key is the last four digits of the phone number.
- The location of an object is the key % tableSize.
  - We’ll use a table size of 101.

TABLE_SIZE = 101

hash(phone number key) = last four phone number

location = last four digits % TABLE_SIZE

h(555-1214) = 1214%101 = 2

2 is empty

store voter information (& key) at position 2

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | null |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| 7 | null |
| … | … |
| 100 | null |

h(555-1214) = 1214%101 = 2

2 is empty

store voter information (& key) at position 2

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| … | … |
| 100 | null |

h(555-2022) = 2022%101 = 2

2 is occupied

store voter information (& key) at position 2

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 → Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 3 | null |
| 4 | null |
| 5 | null |
| 6 | null |
| … | … |
| 100 | null |

h(555-3033) = 3033%101 = 3

3 is empty

store voter information (& key) at position 3

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 → Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 3 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 4 | null |
| 5 | null |
| 6 | null |
| … | … |
| 100 | null |

h(555-2026) = 2026%101 = 6

6 is empty

store voter information (& key) at 6

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 → Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 3 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 4 | null |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 555-2026 |
| … | … |
| 100 | null |

- What if we want to retrieve the voter information associated with phone number 555-3033?
- We find the location: hash(phone) = 3033 % 101 = 3
- We check there- our key is there.

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 → Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 3 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 4 | null |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 555-2026 |
| … | … |
| 100 | null |

- What if we want to retrieve the voter information associated with phone number 555-2022?
- We find the location: hash(phone) = 2022 % 101 = 2
- We check there- the first node is not a match, so we traverse the chain until we find a match.

| Index | Key and Value |
|---|---|
| 0 | null |
| 1 | null |
| 2 | Key = 1214<br>Value = Voter object w/ phone 555-1214 → Key = 2022<br>Value = Voter object w/ phone 555-2022 |
| 3 | Key = 3033<br>Value = Voter object w/ phone 555-3033 |
| 4 | null |
| 5 | null |
| 6 | Key = 2026<br>Value = Voter object w/ phone 555-2026 |
| … | … |
| 100 | null |

# Examples

- Review the examples of creating our own hashtable and using open addressing with linear probing and separate chaining.