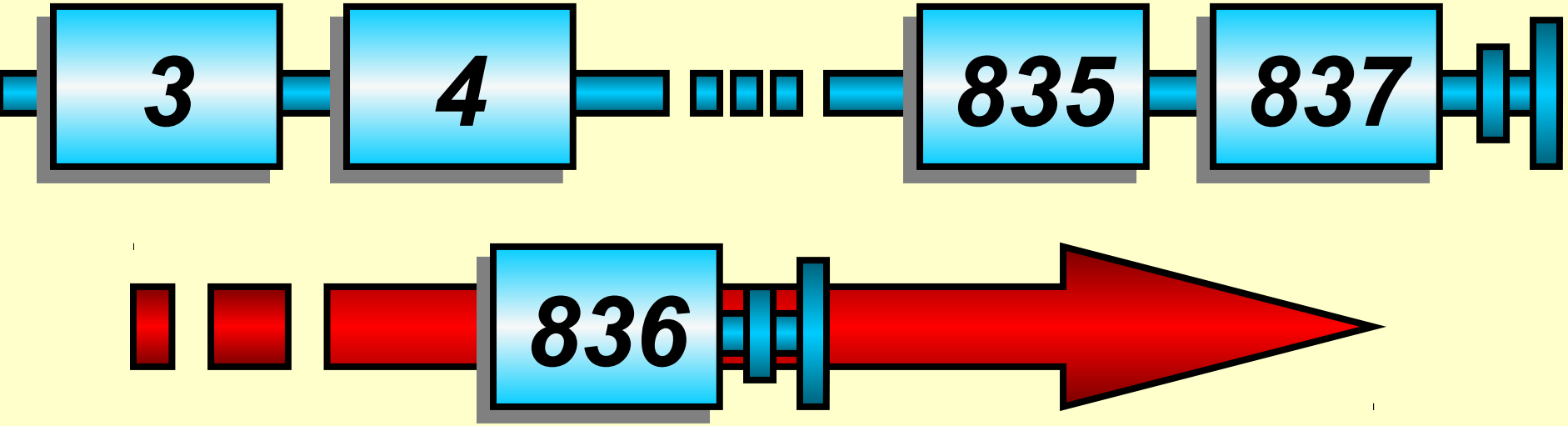
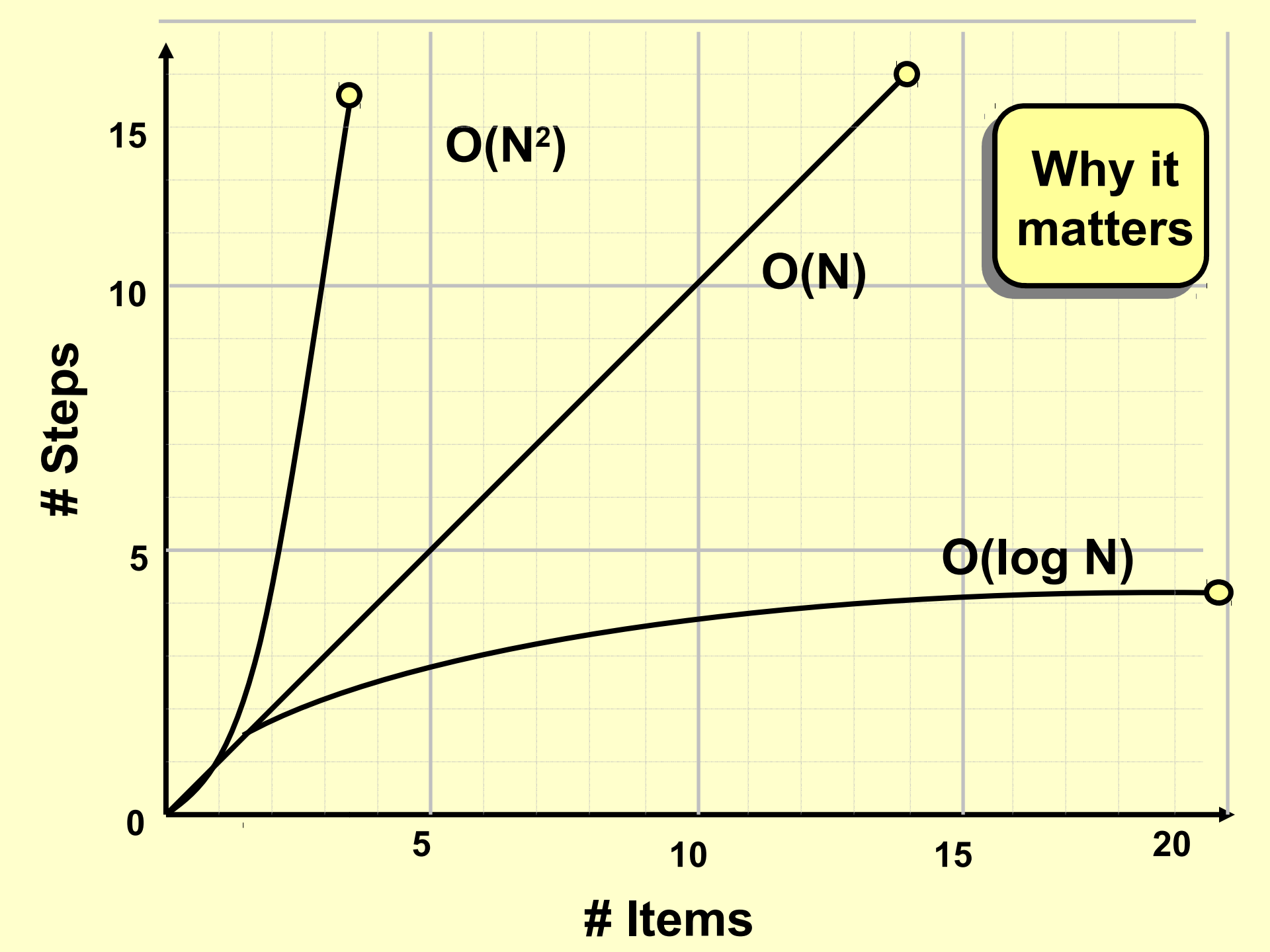
Hashing

Desire

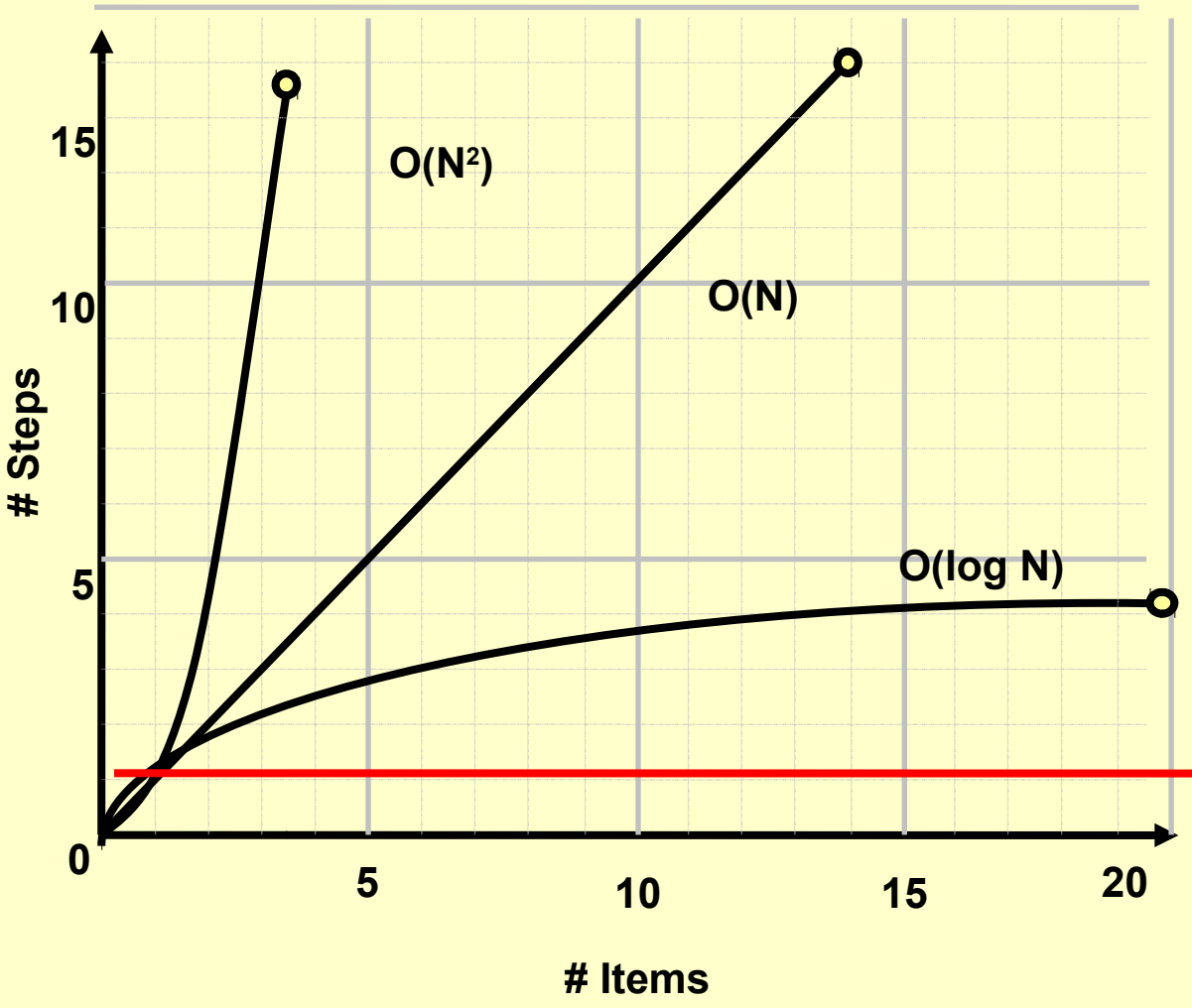
- We want to store objects in some structure and be able to retrieve them extremely quickly.
- The number of items to store might be big.

Hashing--Why?



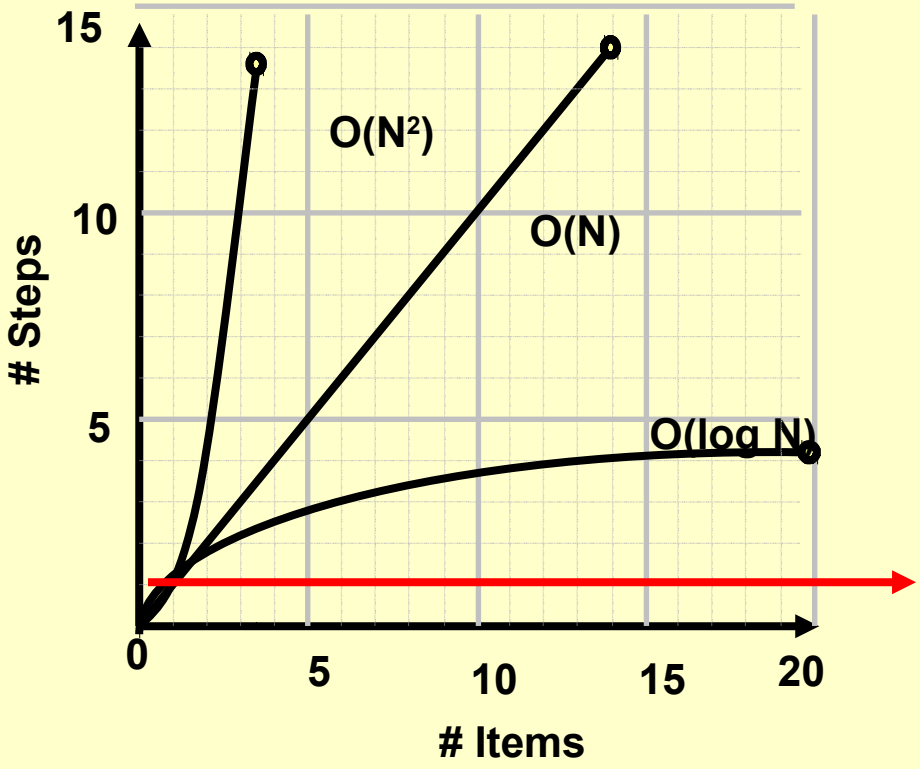

15
10
5
0
# Steps
O(N²)
O(N)
O(log N)
Why it matters
5
10
15
20
Items

Big Uh Oh



There's a way to reduce access time down to O(1), or nearly so

Sanity Check



A search time of O(1)? How is this possible?

Introducing....

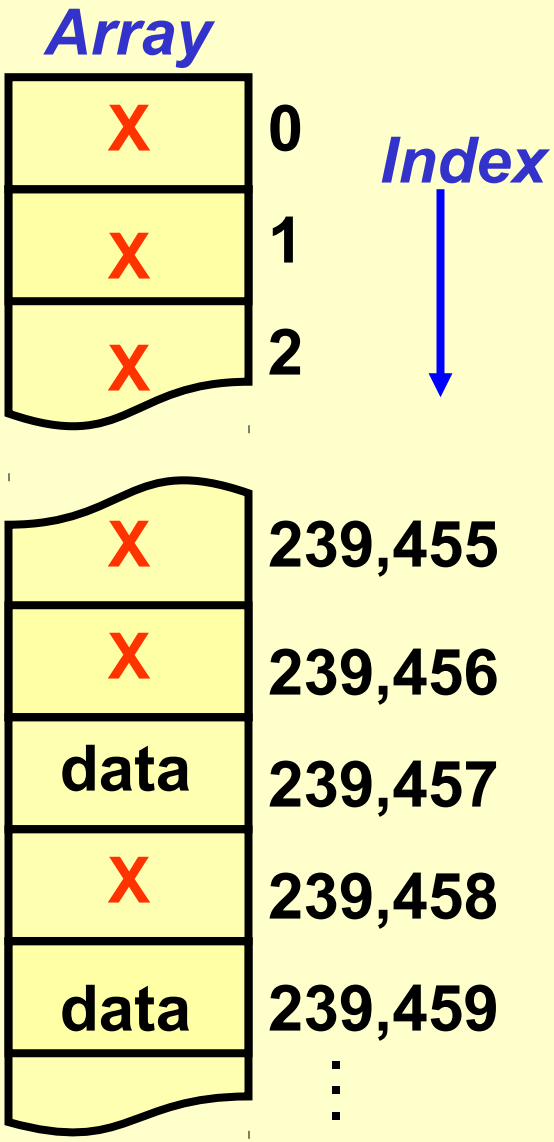
Hashing!

Naive Solution:

```
Data[ ] myRecord = new Data[ 999999999 ];

/*
 * NOTE: Here, we assume there are
 * approximately a billion social security
 * numbers
 */
```

Perhaps not the best?

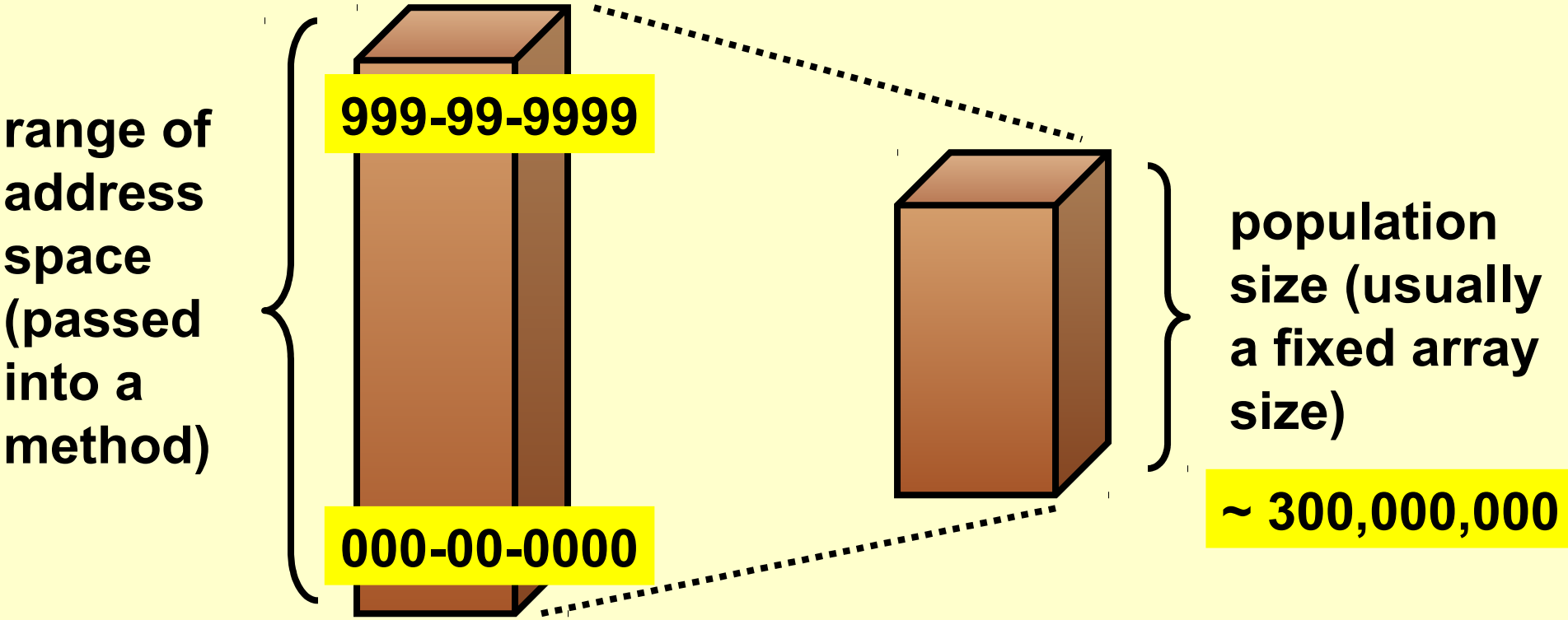
Example


Array
X
X
X
0
1
2
Index
X
X
data
X
data
239,455
239,456
239,457
239,458
239,459

Hashing

Idea:

Shrink the *address space* to fit the *population size*.



```
class AllContacts {
    // storing contacts in a hashmap from names to contacts
    HashMap<String, Contact> contactMap
              = new HashMap<String, Contact>();

    AllContacts(){}

    // return contact that goes with a given name
    // will return null if the name isn't in the map
    Contact findContactByName(String name) {
      return contactMap.get(name);
    }

    // takes the whole contact as input
    AllContacts addContact(Contact aContact) {
      Contact oldContact
              = contactMap.put(aContact.name, aContact);
      return this;
    }
}
```

```
class AllContacts {
    // storing contacts in a hashmap from names to contacts
    HashMap<String, Contact> contactMap
              = new HashMap<String, Contact>();

    AllContacts(){}

    // return contact that goes with a given name
    // will return null if the name isn't in the map
    Contact findContactByName(String name) {
      return contactMap.get(name);
    }

    // takes the whole contact as input
    AllContacts addContact(Contact aContact) {
      Contact oldContact
              = contactMap.put(aContact.name, aContact);
      return this;
    }
}
```

```
class AllContacts {
    // storing contacts in a hashmap from names to contacts
    HashMap<String, LinkedList<Contact>> contactMap
       = new HashMap<String, LinkedList<Contact>>();

    AllContacts(){}

    // return contact that goes with a given name
    // will return null if the name isn't in the map
    LinkedList<Contact> findContactByName(String name) {
      return contactMap.get(name);
    }

    // takes the whole contact as input
    AllContacts addContact(LinkedList<Contact> aContact) {
      Contact oldContact
              = contactMap.put(aContact.name, aContact);
      return this;
    }
}
```

```
AllContacts addContact(LinkedList<Contact> aContact) {
  Contact oldContact
          = contactMap.put(aContact.name, aContact);
  return this;
}
```

```
AllContacts addContact(Contact aContact) {
  LinkedList<Contact> contacts
          = this.findContactByName(aContact.name);
  contacts.add(aContact);
  Contact oldContact
          = contactMap.put(aContact.name, contacts);
  return this;
}
```

Reality Check

Everyone getting the idea?

Hash Function Example

Instead of

```
StudentFile temp = studentRecords[iSocSecNum];
```

reduce the range:

```
StudentFile temp =
      record[iSocSecNum % record.length];
```

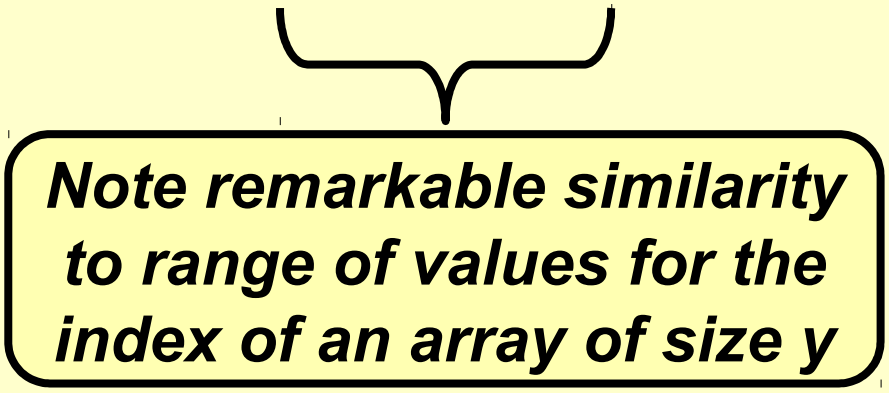
returns an index within the appropriate range

Recall

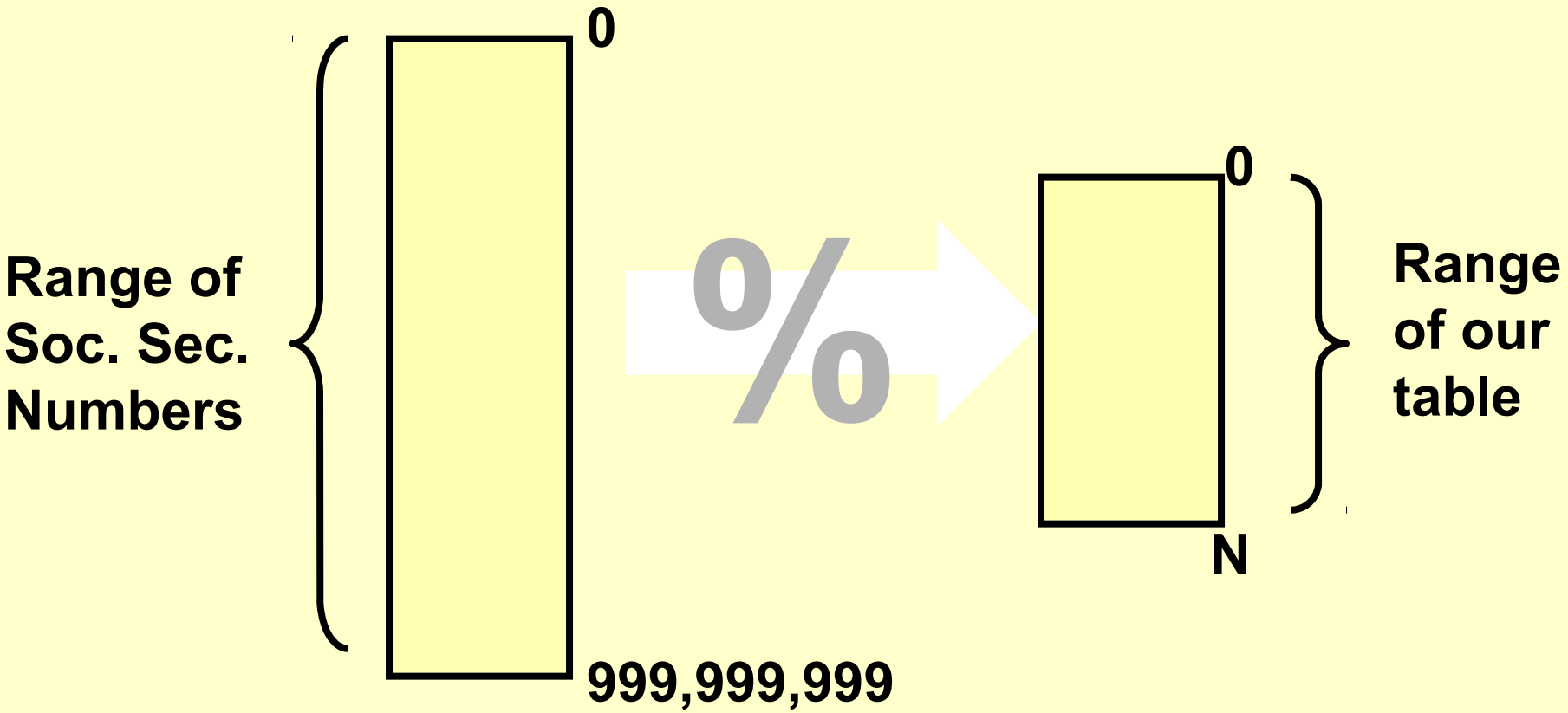
- Our friend the Mod Function %

```
x % y
```

- will yield values between 0 and y - 1



The Art of Hashing



A problem...

- We have an array of length 100
- We have about 50 students
- We hash using: **`ssn % 100`**

- George P. Burdell
  - 123-45-6789
- George W. Bush
  - 321-54-7689

Collision!

Designing Hash Functions

The Perfect Hash Function:

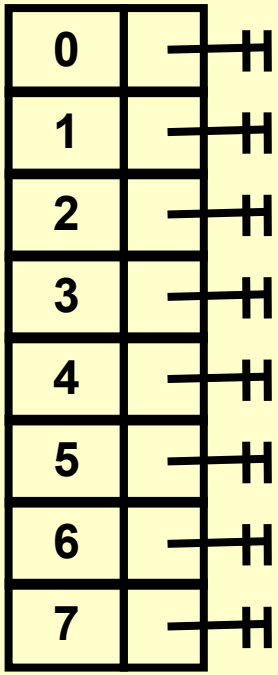
- **Fast**
- **No collisions**

***Common Hash Functions*:**

- ***Digit selection*: e.g., last 4 digits of phone number**
- ***Division*: modulo**
- ***Character keys*: use ASCII num values for chars (e.g., 'R' is 82)**

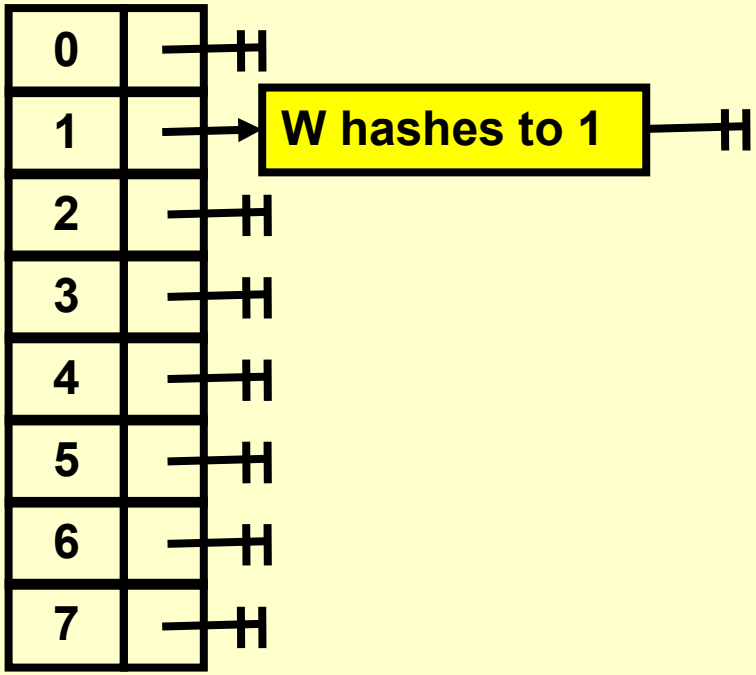
Collision Resolution

Technique: External chaining:



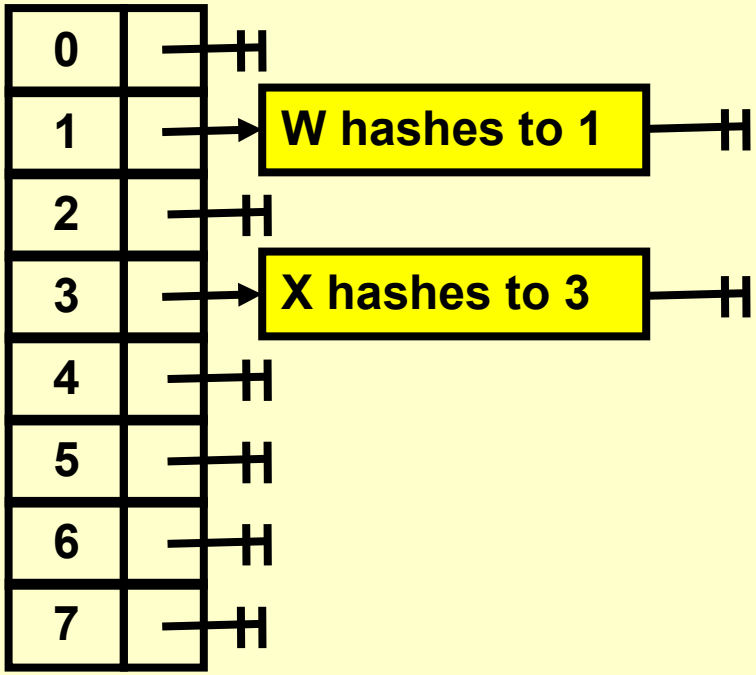
Collision Resolution

Technique: External chaining:



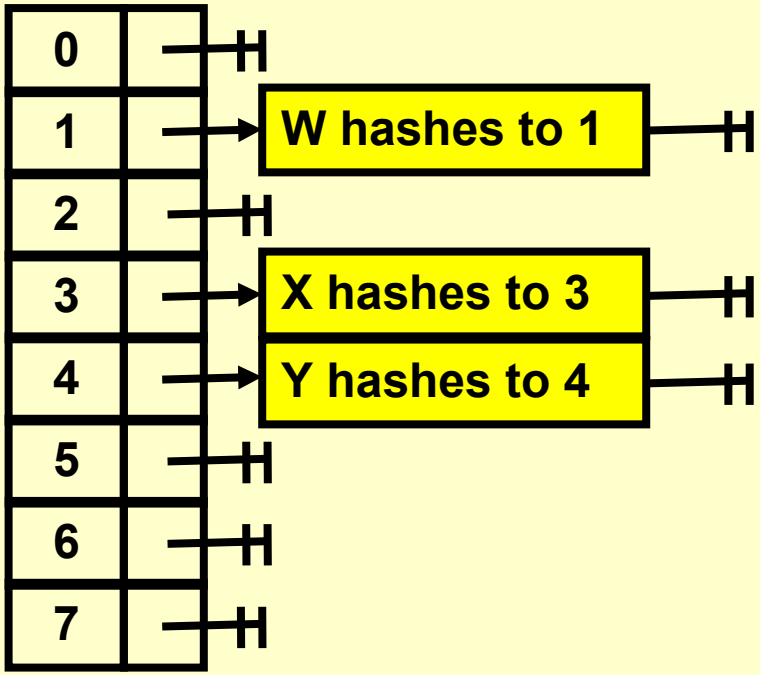
Collision Resolution

Technique: External chaining:



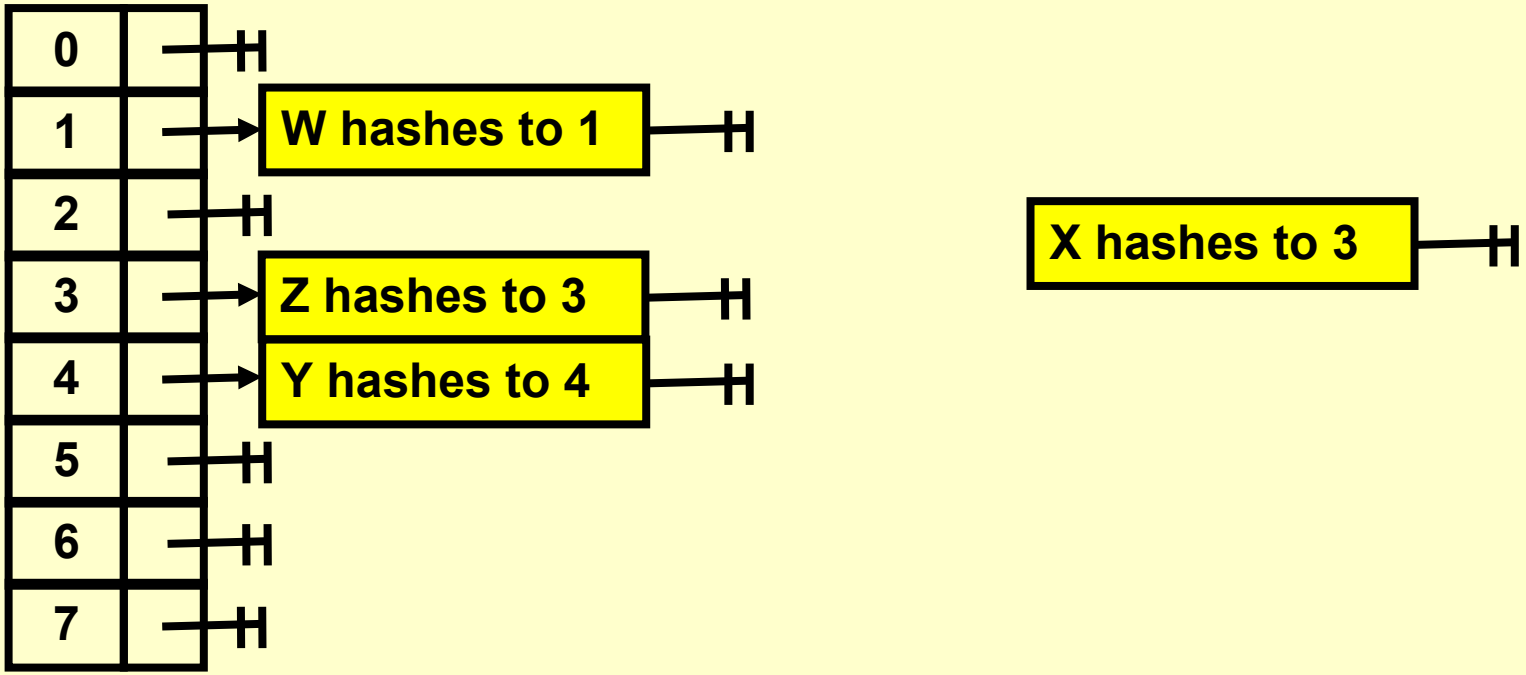
Collision Resolution

Technique: External chaining:



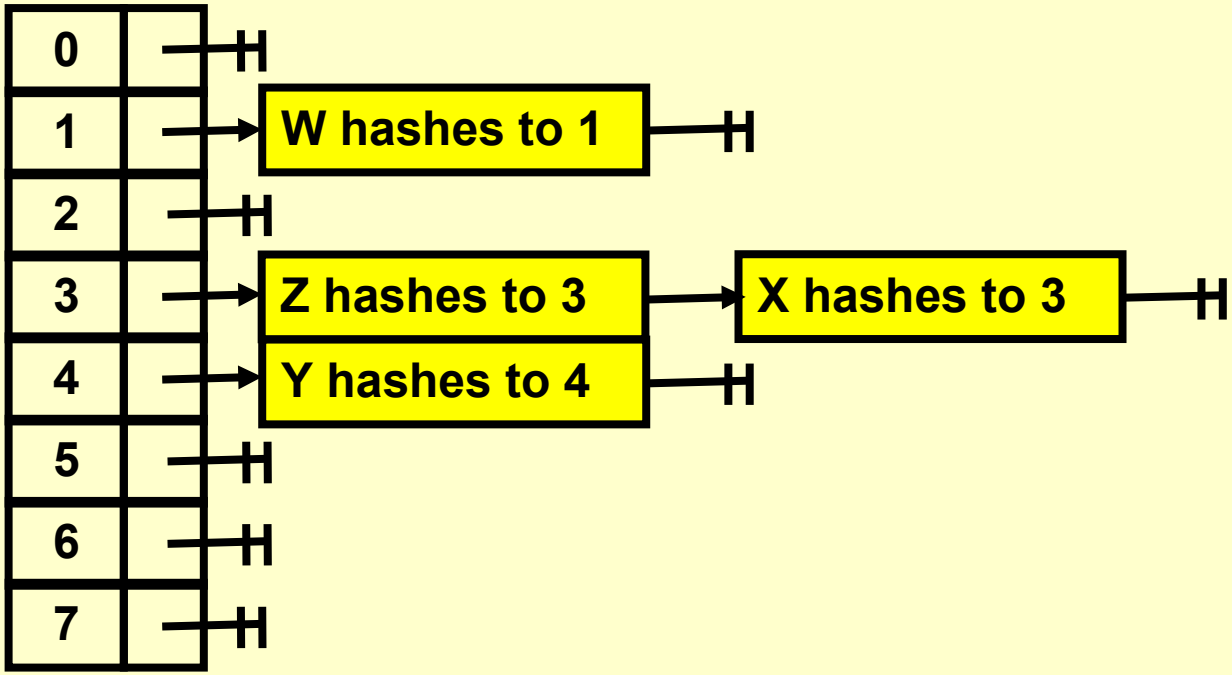
Collision Resolution

Technique: External chaining:



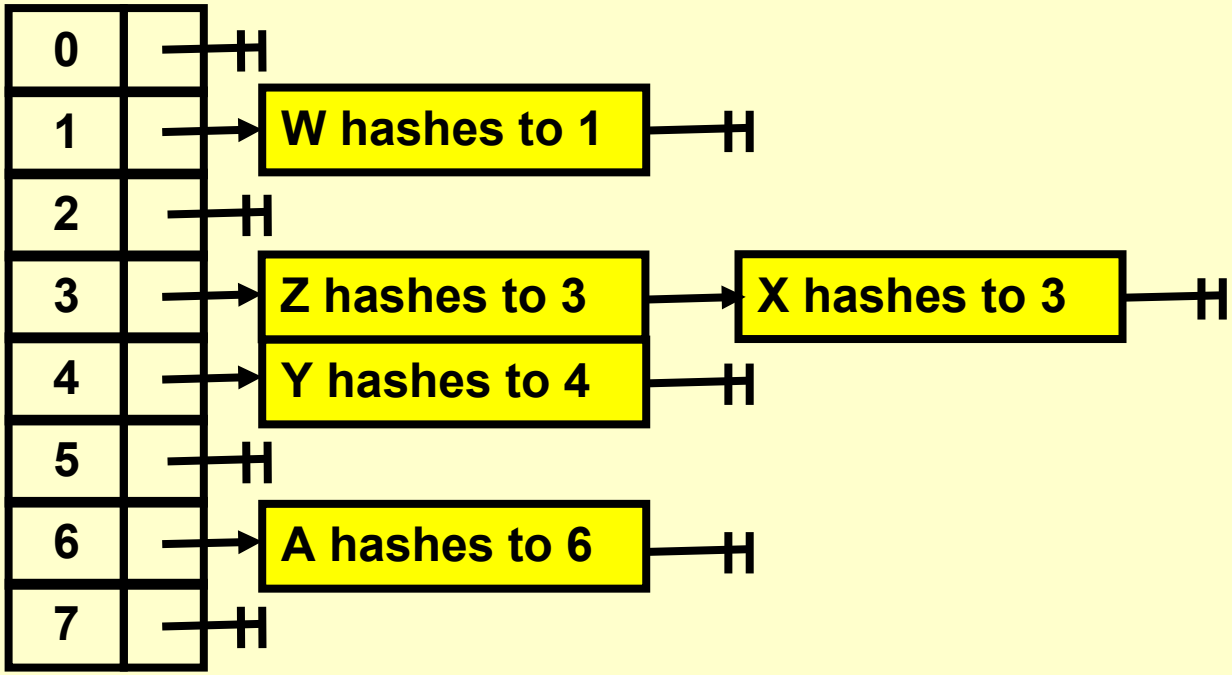
Collision Resolution

Technique: External chaining:



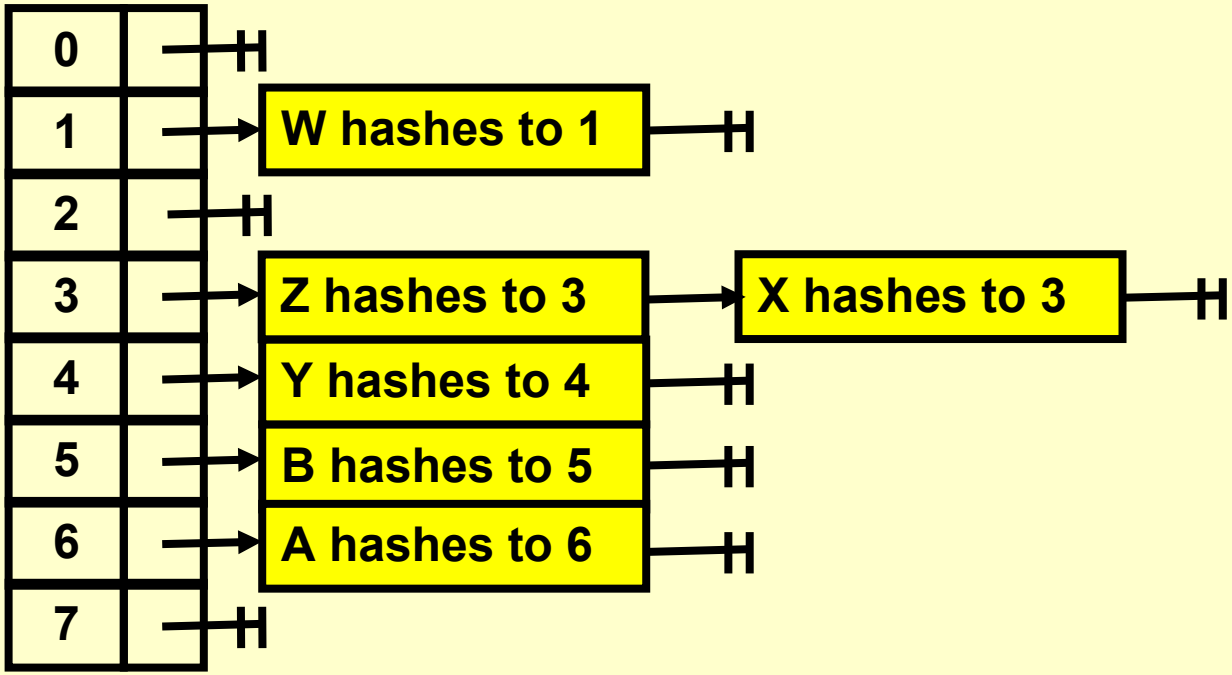
Collision Resolution

Technique: External chaining:



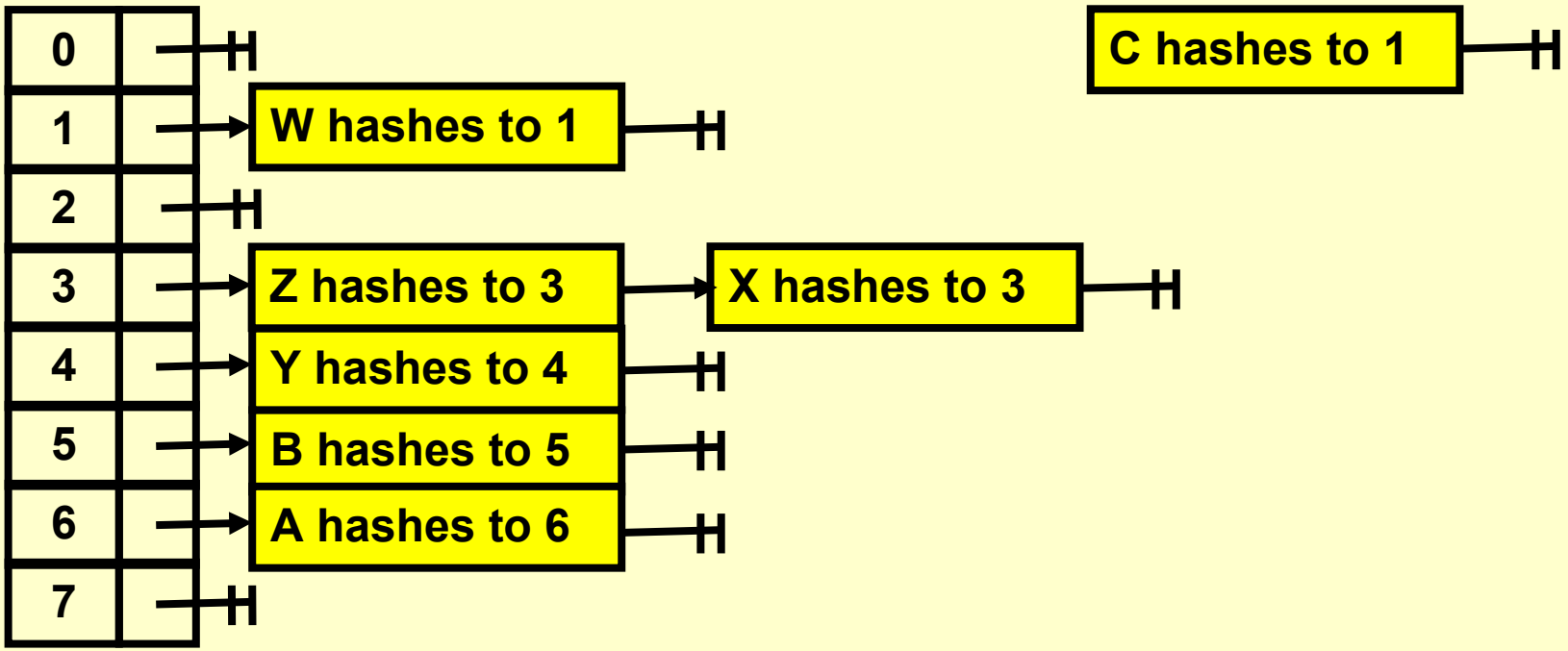
Collision Resolution

Technique: External chaining:



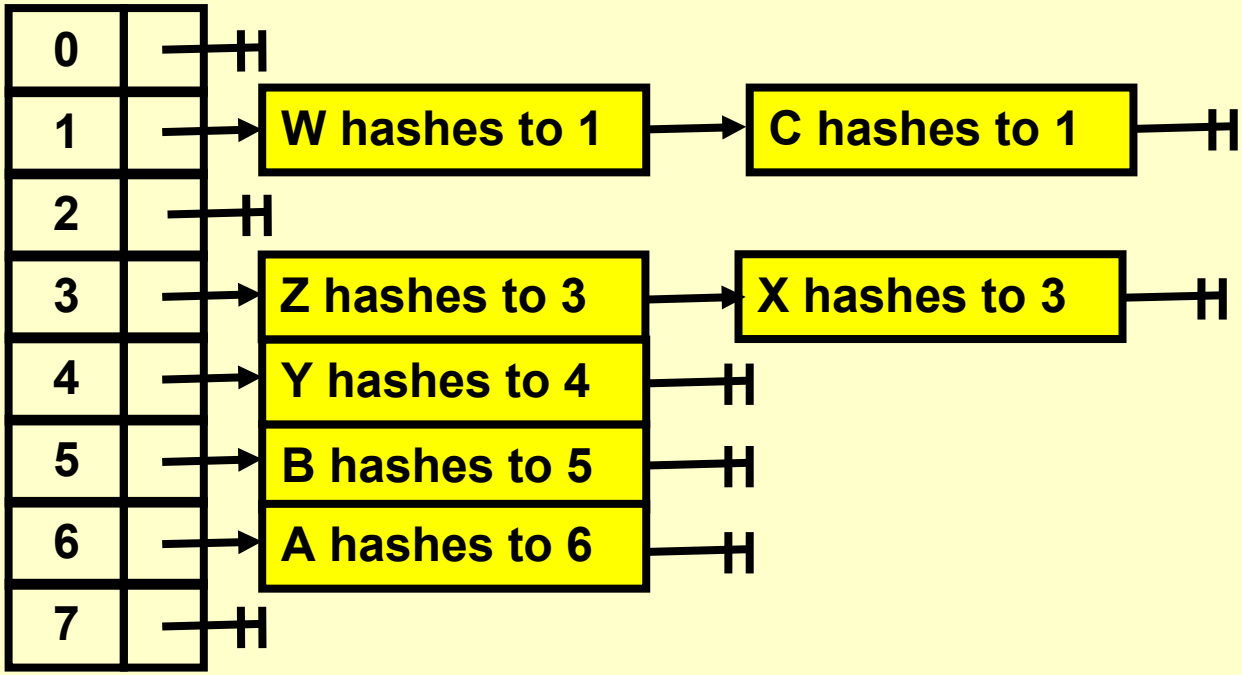
Collision Resolution

Technique: External chaining:



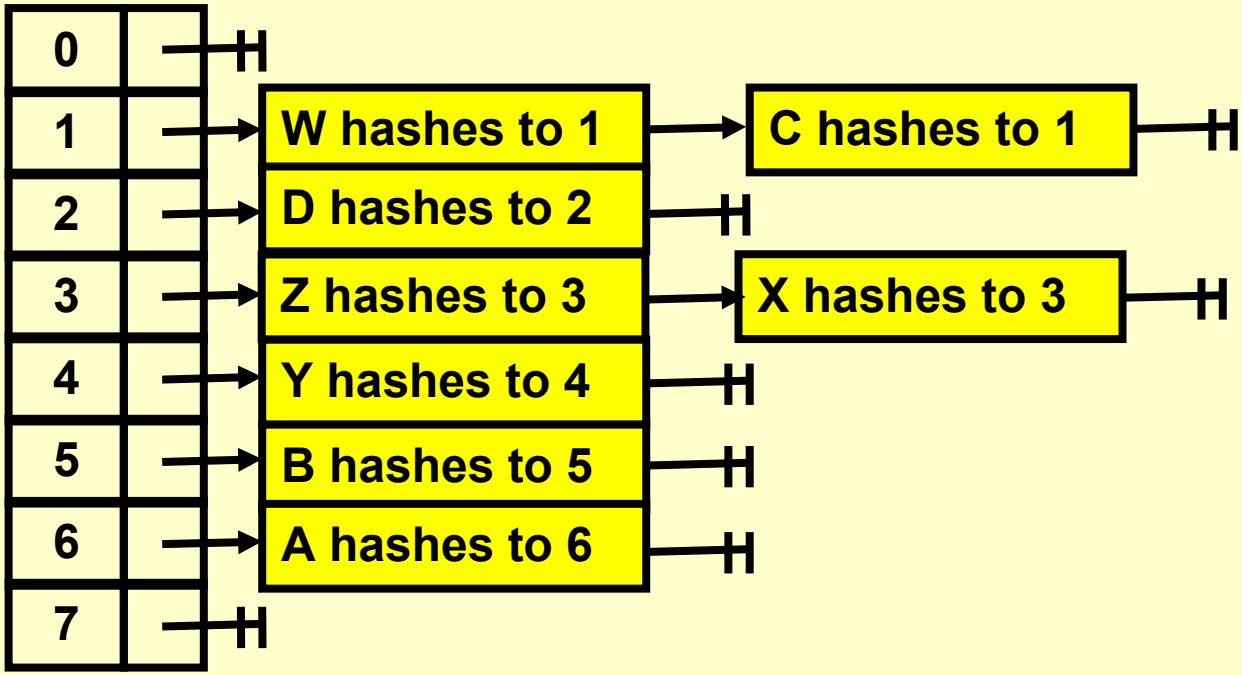
Collision Resolution

Technique: External chaining:



Collision Resolution

Technique: External chaining:



Better Hashing

The key to efficient hashing is the hash function.

How can we *efficiently* convert a name into a key number?

```
public int getHash(String strName);
```

Hashing Names

Version # 1

```
public int getHash (String strName) {

     int hash = 0;

     for (int i =0; i < strName.length(); i++) {
          hash +=  (int) strName.charAt(i);
     }

    hash %= tableSize;

    return hash;
}
```

This works, but what are its limitations?

Hashing Names

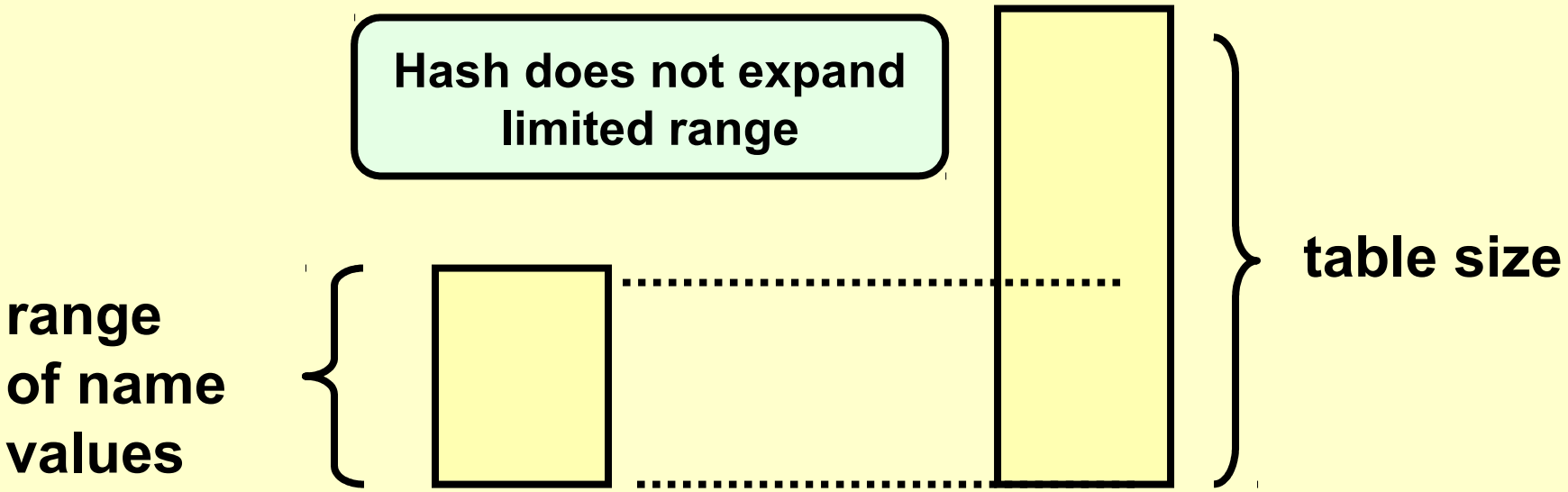
```
public int getHash (String strName) {

  int hash = 0;

  hash = (int) strName.charAt(0) +
    128 * (byte) strName.charAt(1) +
    16384 * (byte) strName.charAt(2);

  hash %= tableSize;

  return hash;
}
```

Version # 2

Strategy: only examine first three characters

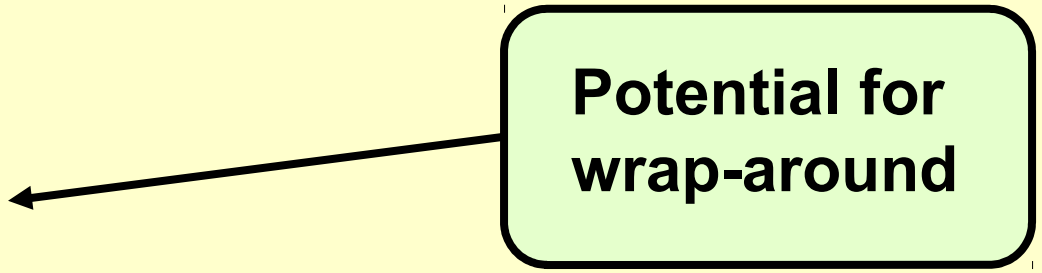
This works, but what are its limitations?

Inductive Analysis



Improved Hash Function

```
public int getHash (String strName) {

    int hash = 0;

    for (int i=0; i< strName.length(); i++)
        hash = 128 * hash + (int) strName.charAt(i);

   hash %= tableSize;

   if (hash < 0 )
         hash += tableSize;

   return hash;
}
```



```
import java.util.HashMap;

class Addr {
  int num;
  String street;

  //... code snipped

  // a simple hashCode function multiplies the hashCode of each
  // field that matters by a unique prime,
  // and sums the results
  public int hashCode() {
    return (this.num * 3) + (this.street.hashCode() * 11);
  }
}
```

```
import java.util.HashMap;

class Addr {
  int num;
  String street;

  //... code snipped

  // a simple hashCode function multiplies the hashCode of each
  // field that matters by a unique prime,
  // and sums the results
  public int hashCode() {
    return (this.num * 3) + (this.street.hashCode() * 11);
  }
}
```

Hard Lessons about Hashing

Your hash function must be *carefully* selected.

It varies with your data. You have to study your input, and base your hash on the properties of the input data.

Your range of input should be larger than your table size (else your hashing will under utilize the table).

Experience (and theory beyond this course) show that prime numbers not close to a power of two will lead to the best distribution of keys

Summary of Hash Tables

- Purpose: Allows extremely fast lookup given a key value. Reduce the address space of the information to the table space of the hash table.
- Hash function: the reduction function
- Collision: hash(a) == hash(b), but a!=b
- Chaining is most general solution to collision