

# Hashing

Way Back When...

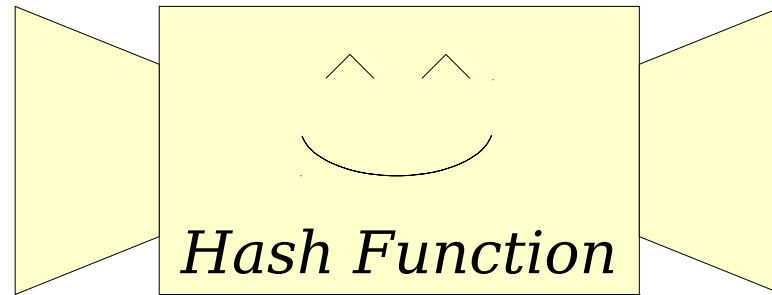
```

int nameHash(string first, string last){
    /* This hashing scheme needs two prime numbers, a large prime and a small
    * prime. These numbers were chosen because their product is less than
    * 2^31 - kLargePrime - 1.
    */
    static const int kLargePrime = 16908799;
    static const int kSmallPrime = 127;

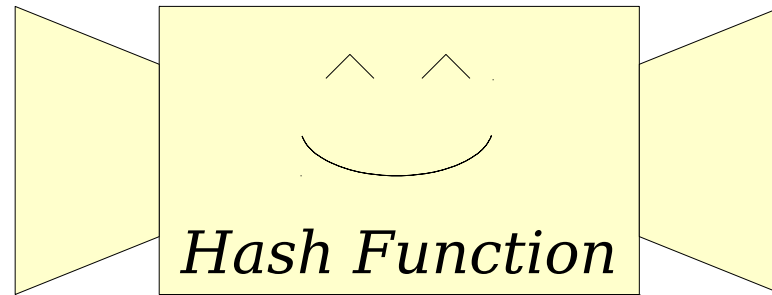
    int hashVal = 0;

    /* Iterate across all the characters in the first name, then the last
    * name, updating the hash at each step.
    */
    for (char ch: first + last) {
        /* Convert the input character to lower case. The numeric values of
        * lower-case letters are always less than 127.
        */
        ch = tolower(ch);
        hashVal = (kSmallPrime * hashVal + ch) % kLargePrime;
    }
    return hashVal;
}

```

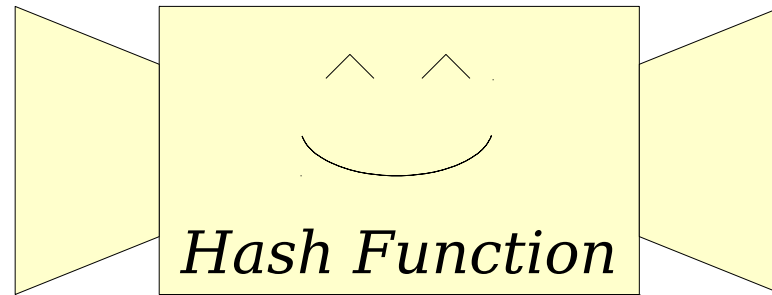


This is a ***hash function***. It's a type of function some smart math and CS people came up with.



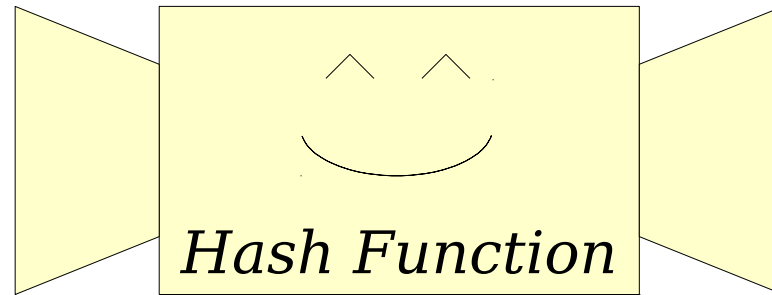
```
int hashCode( );
```

Most hash functions return a number.  
In CS106B, we'll use the **int** type.



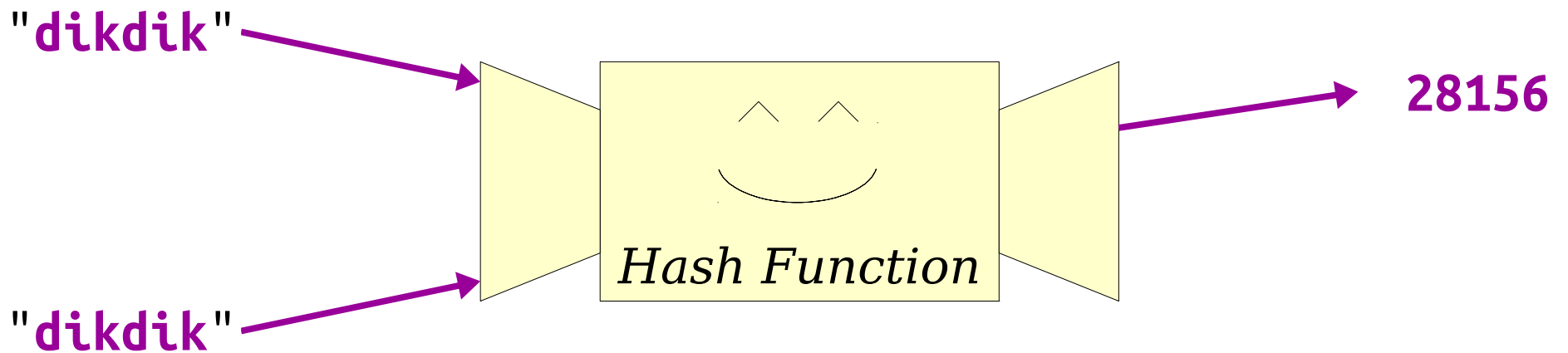
```
int hashCode(const string& input);
```

Different hash functions take inputs of different types.  
For simplicity, we'll assume this takes in a string.



```
int hashCode(const string& input);
```

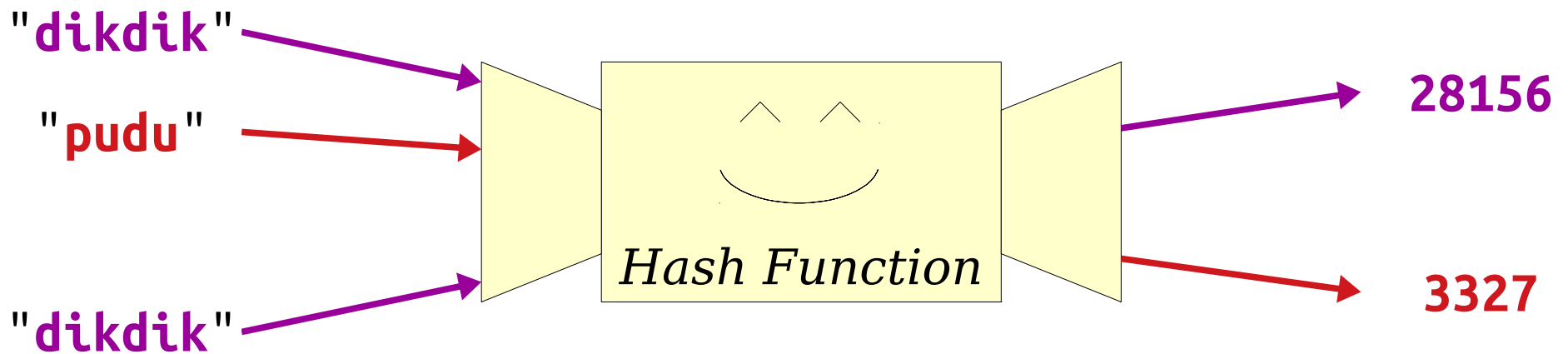
What makes this function so special?



```
int hashCode(const string& input);
```

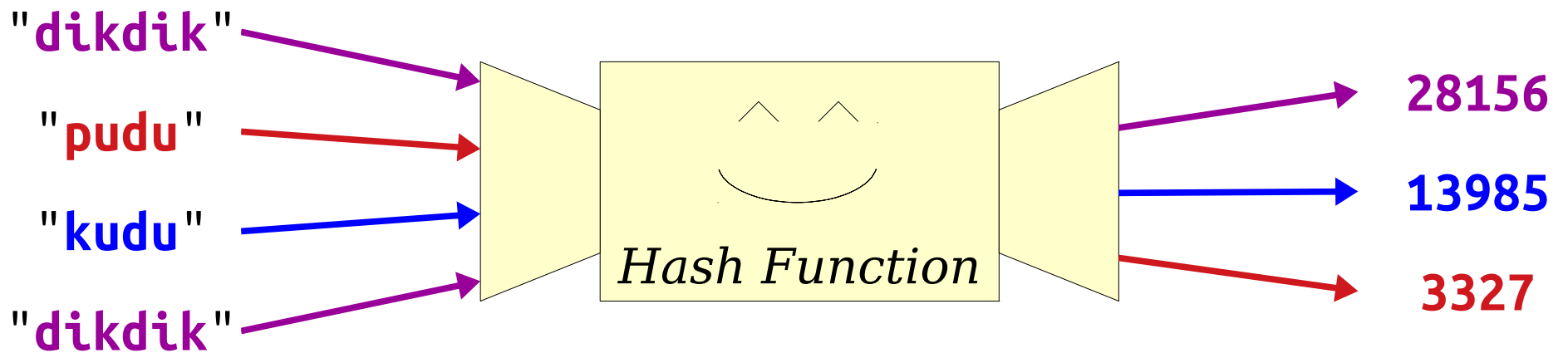
First, if you compute hashCode of the same string many times, you always get the same value.





```
int hashCode(const string& input);
```

Second, the hash codes of different inputs are (usually) very different from one another.



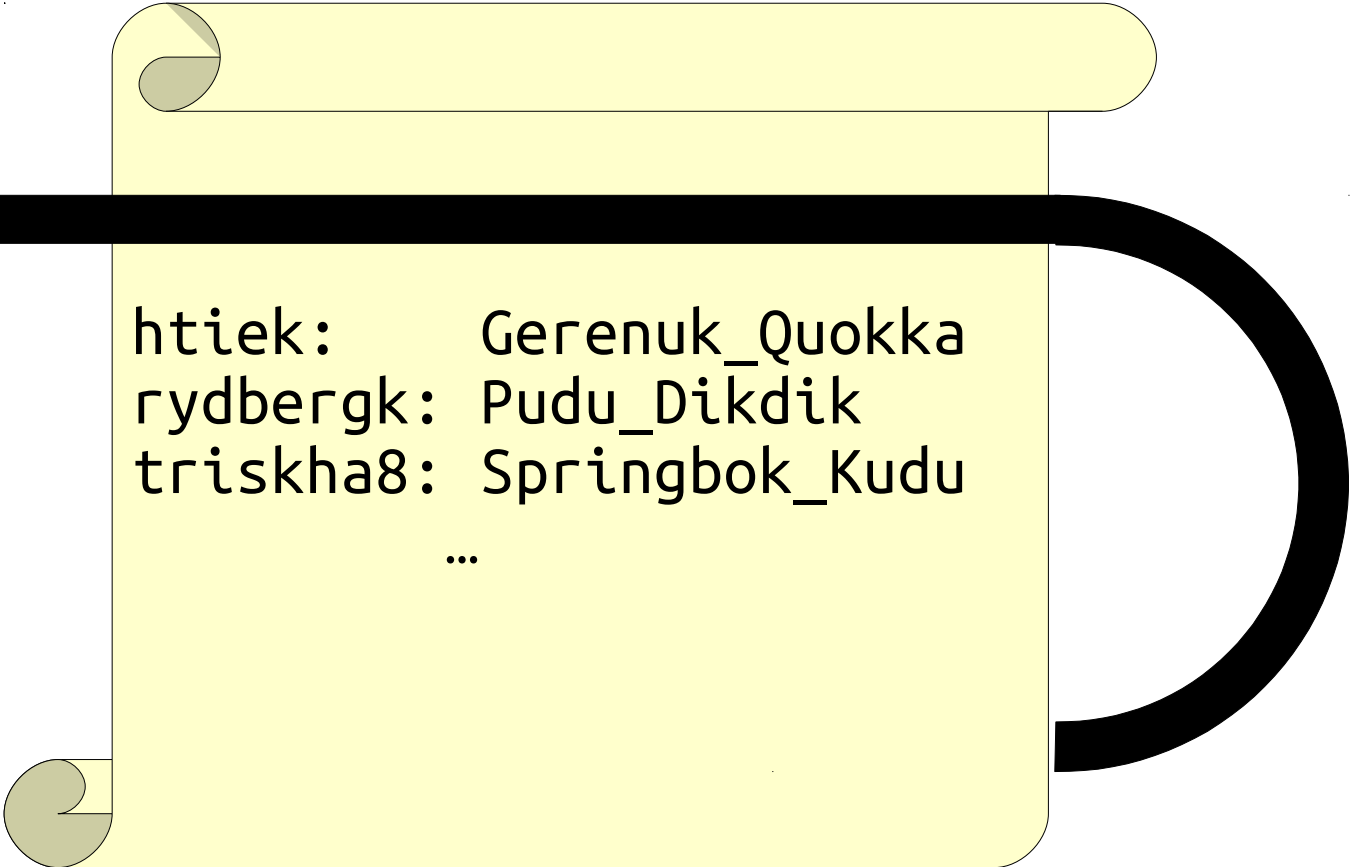
```
int hashCode(const string& input);
```

Even very similar inputs give  
very different outputs!

## ***To Recap:***

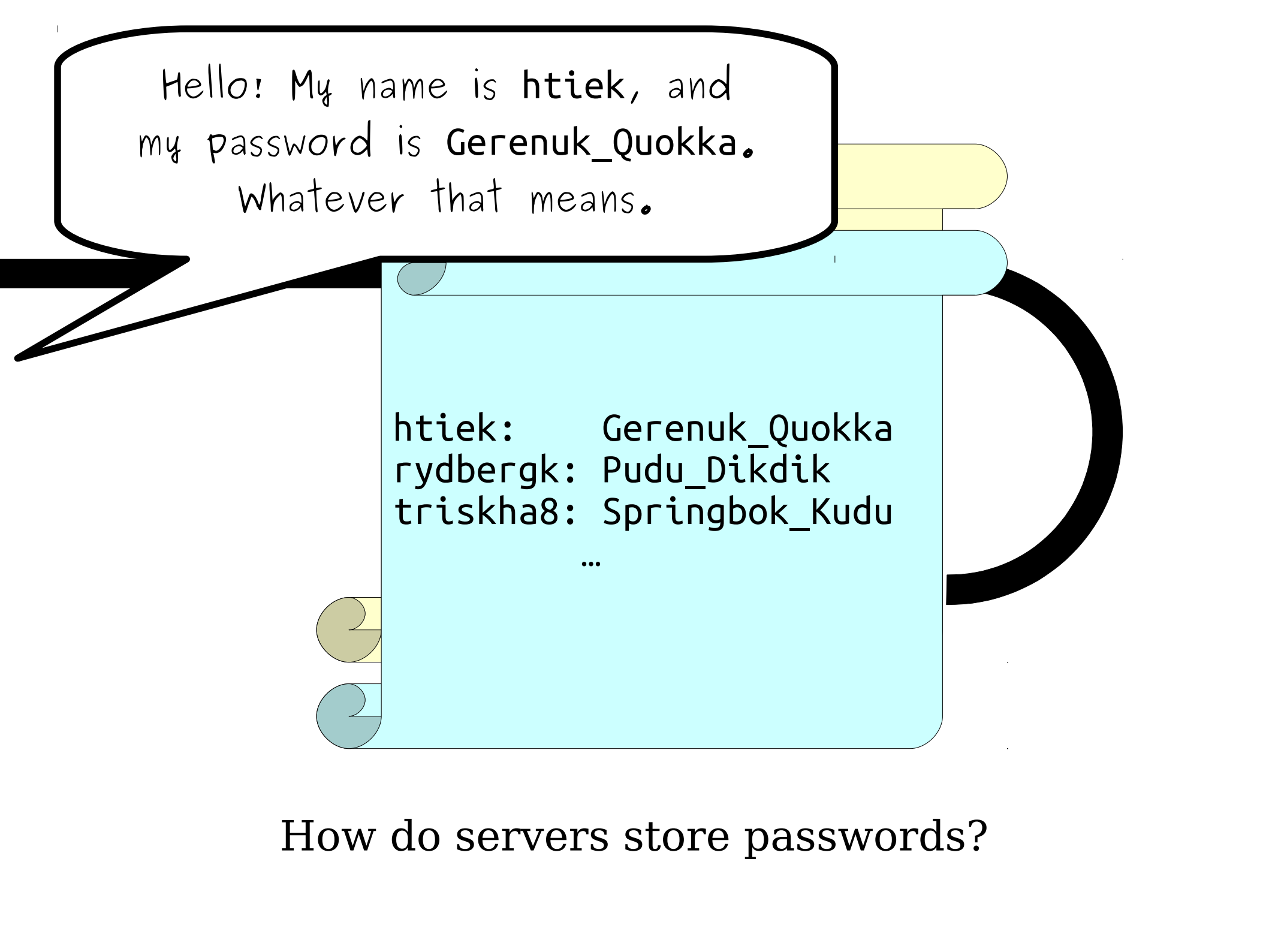
Equal inputs give equal outputs.

Unequal inputs (usually) give  
very different outputs.



```
htiek:      Gerenuk_Quokka  
rydbergk:  Pudu_Dikdik  
triskha8:  Springbok_Kudu  
...
```

How do servers store passwords?



Hello! My name is hti`ek`, and  
my password is `Gerenuk_Quokka`.  
Whatever that means.

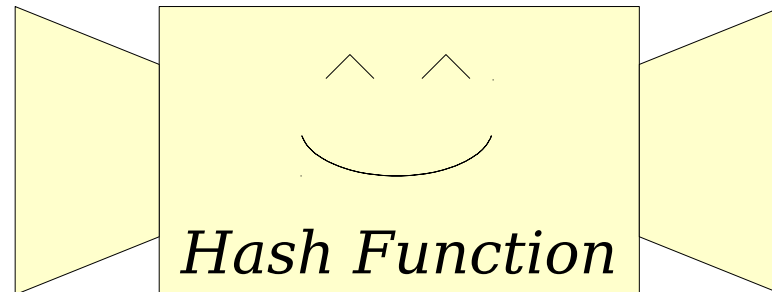
```
htiek:      Gerenuk_Quokka  
rydbergk:  Pudu_Dikdik  
triskha8:  Springbok_Kudu  
...
```

How do servers store passwords?

My name is htiek,  
and my password is,  
um, hold on...

```
htiek:      29157389323963039  
rydbergk:  54162041201524803  
triskha8:  30965171063527336
```

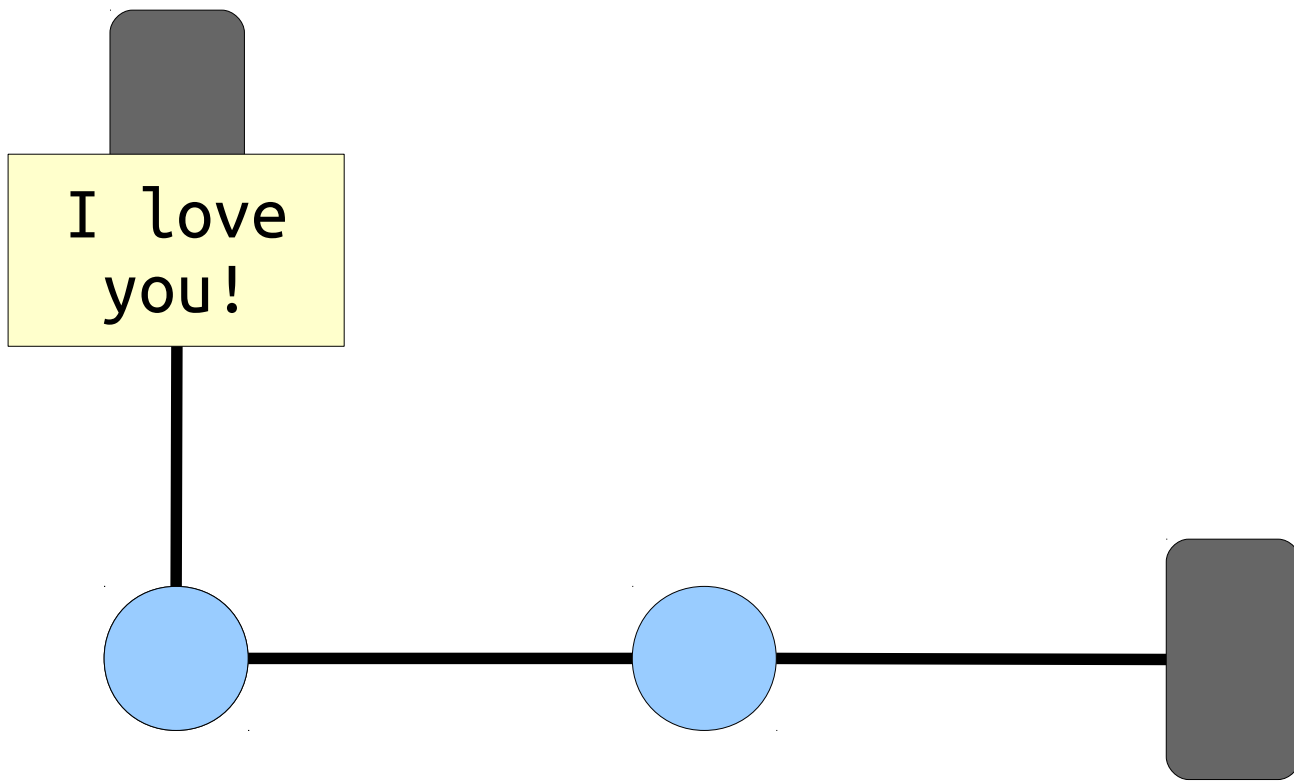
...



How do servers store passwords?

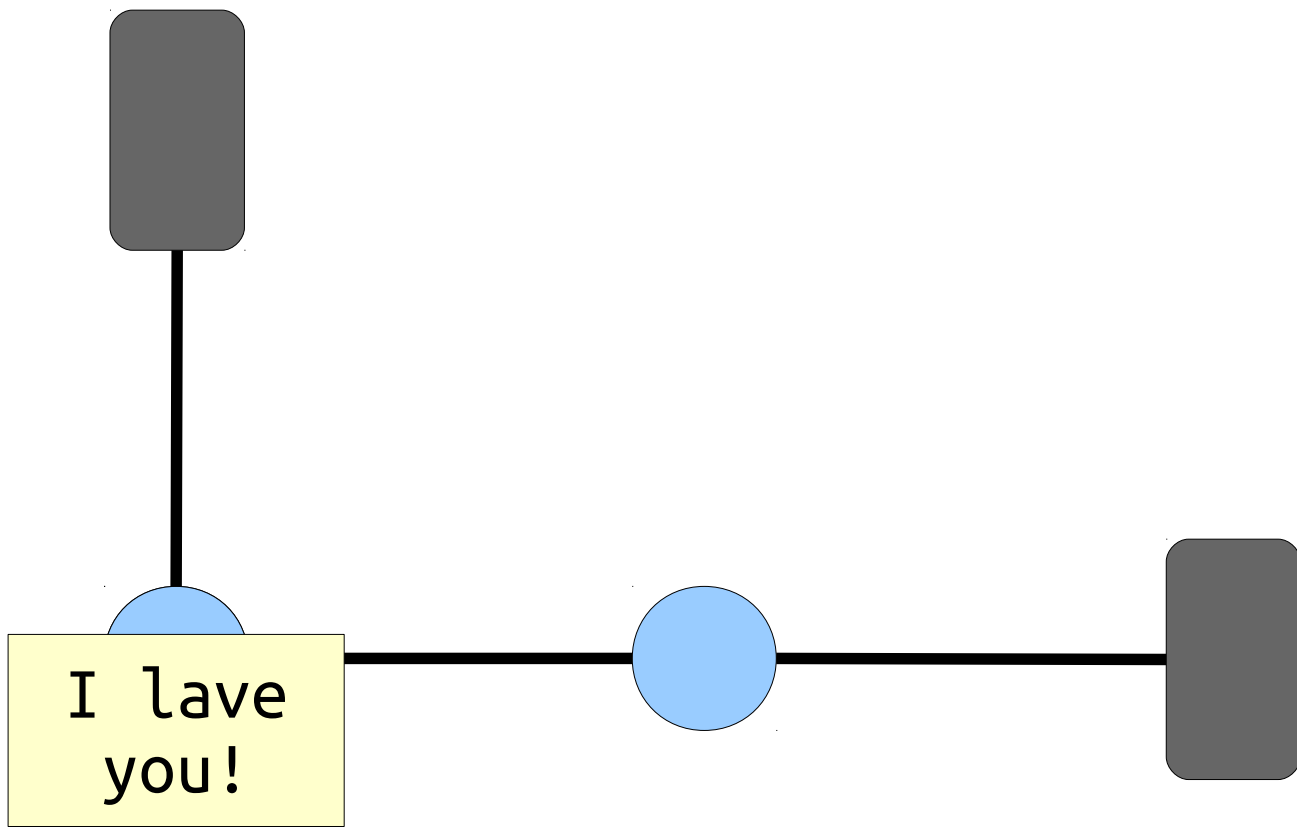
This is how passwords are typically stored.  
Look up ***salting and hashing*** for details!

And look up ***commitment schemes*** if you  
want to see some even cooler things!

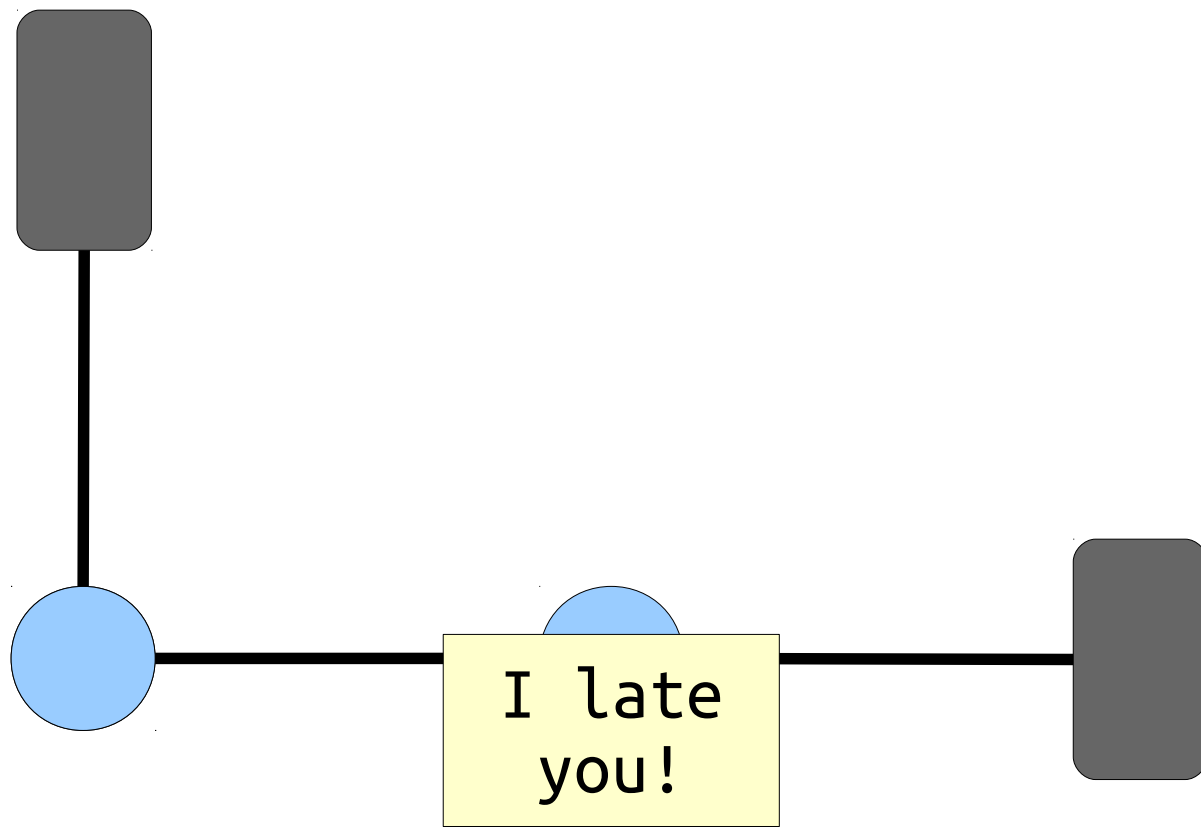


Did my data make it through the network?

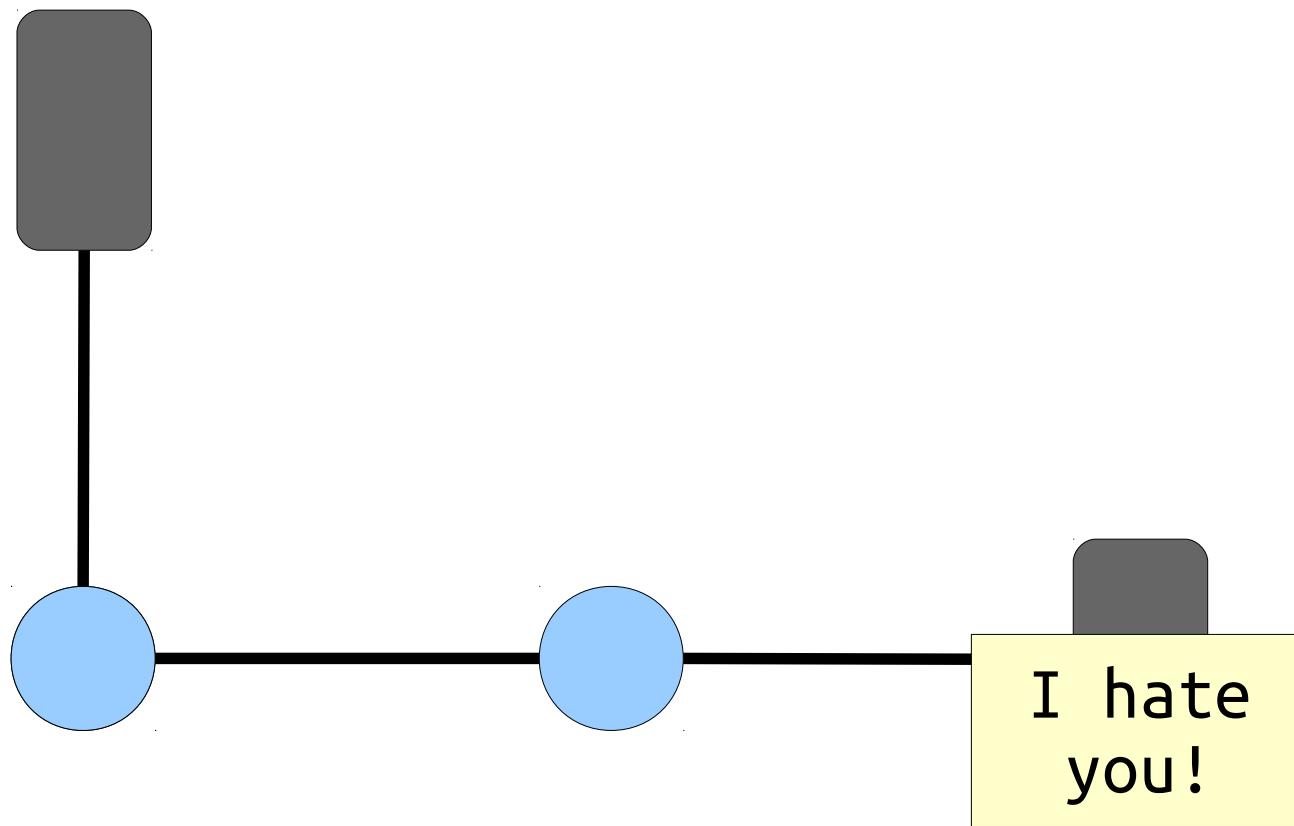




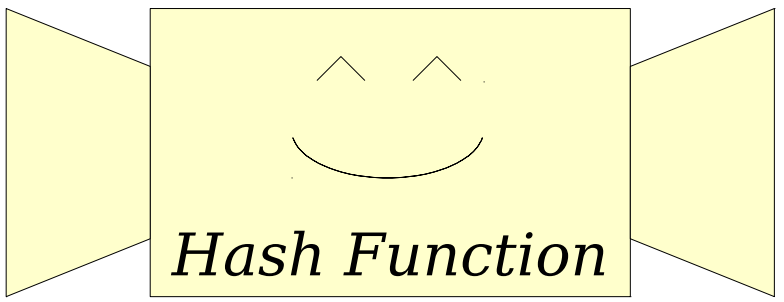
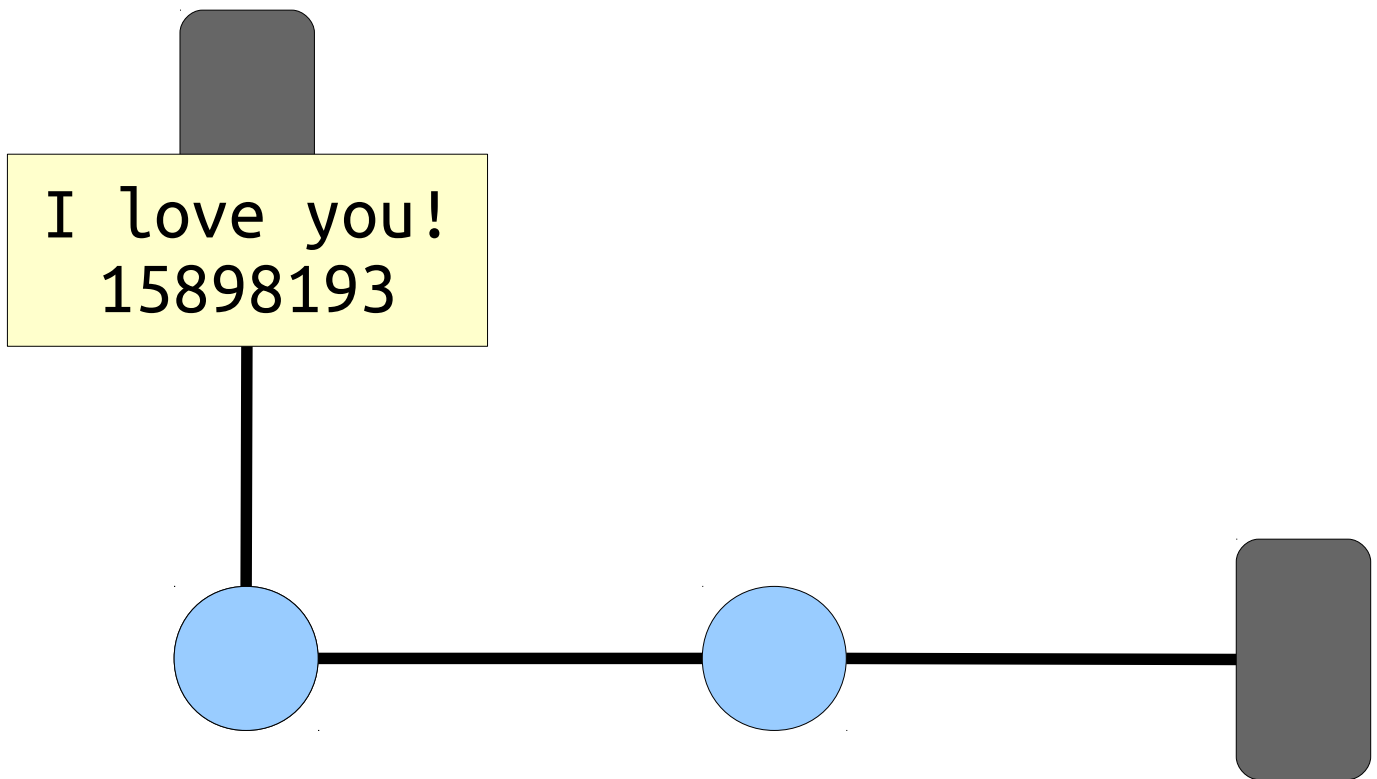
Did my data make it through the network?



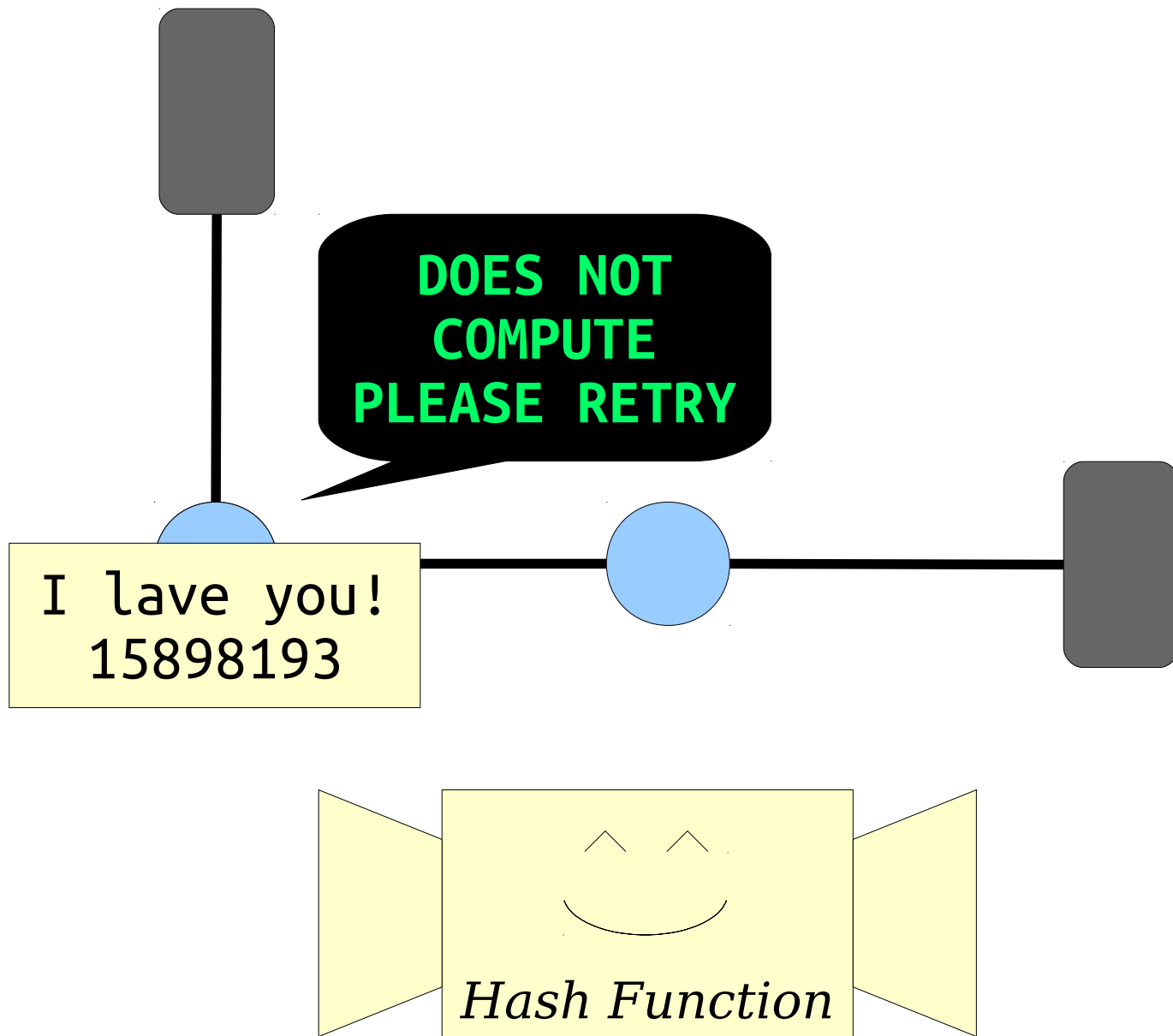
Did my data make it through the network?



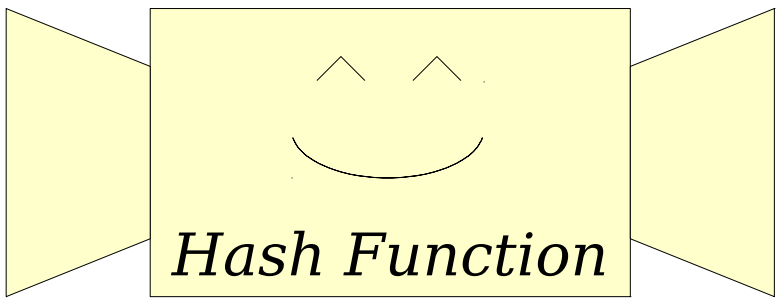
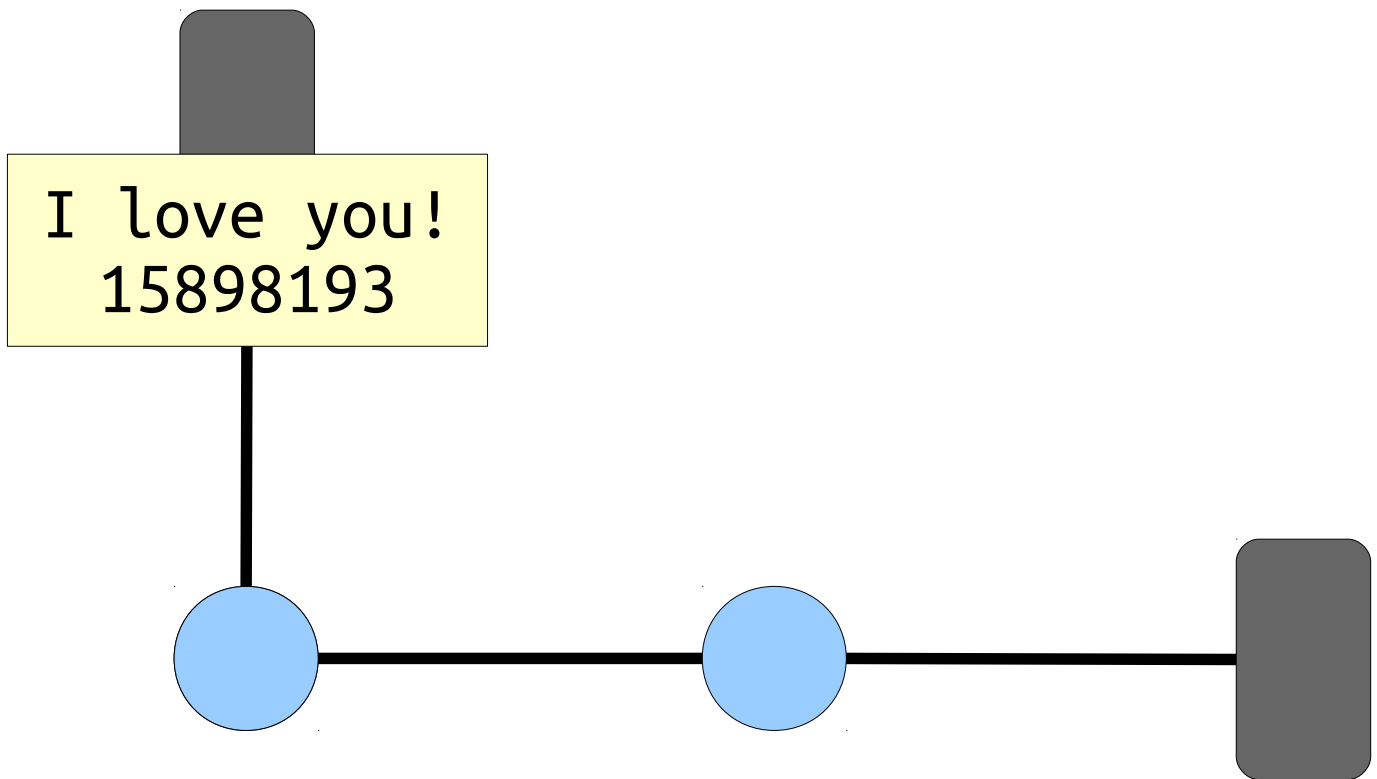
Did my data make it through the network?



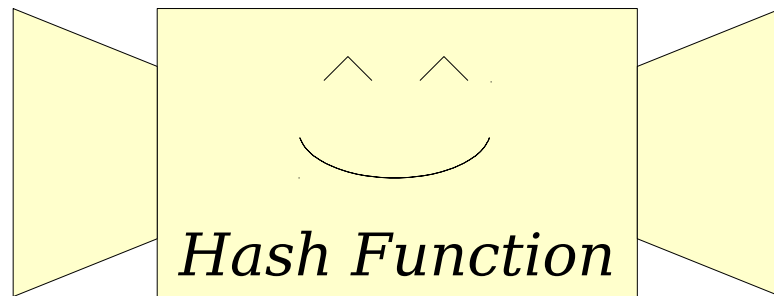
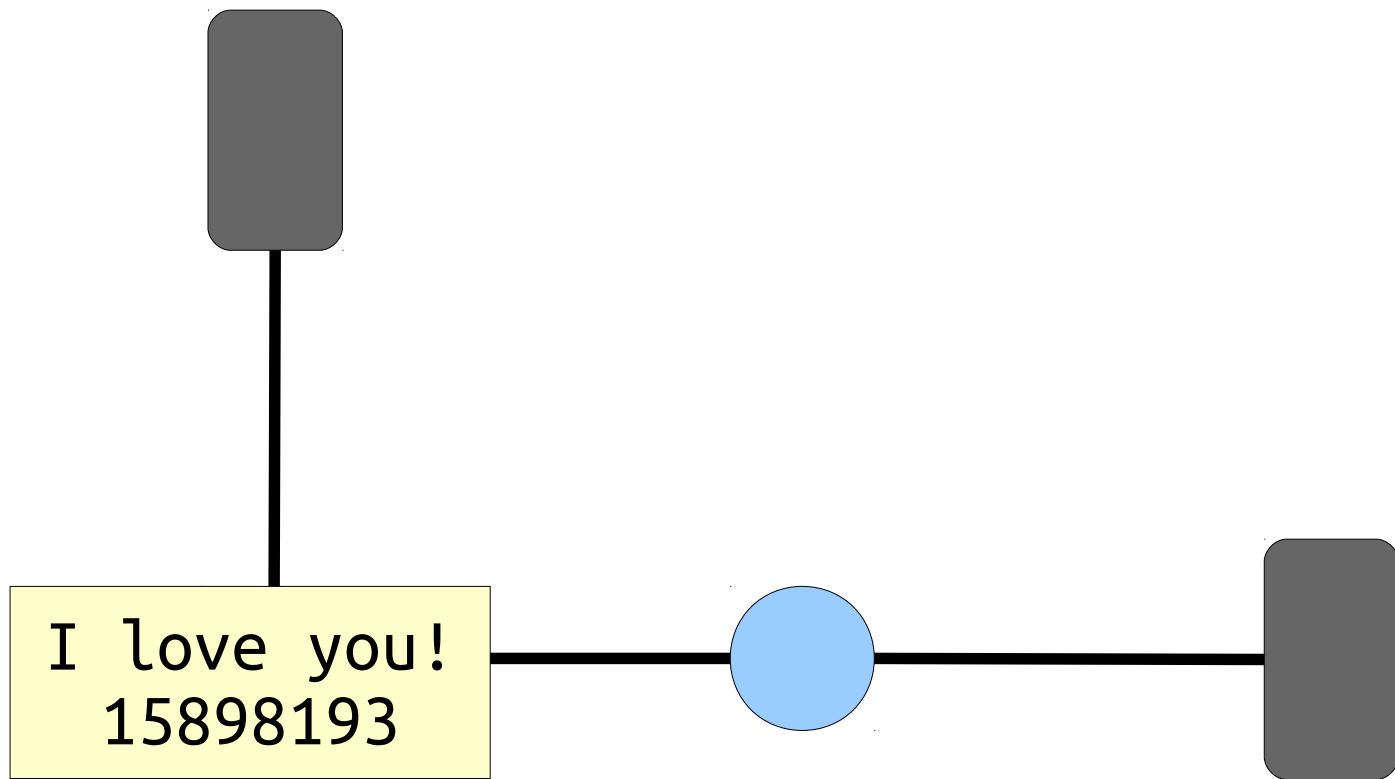
Did my data make it through the network?



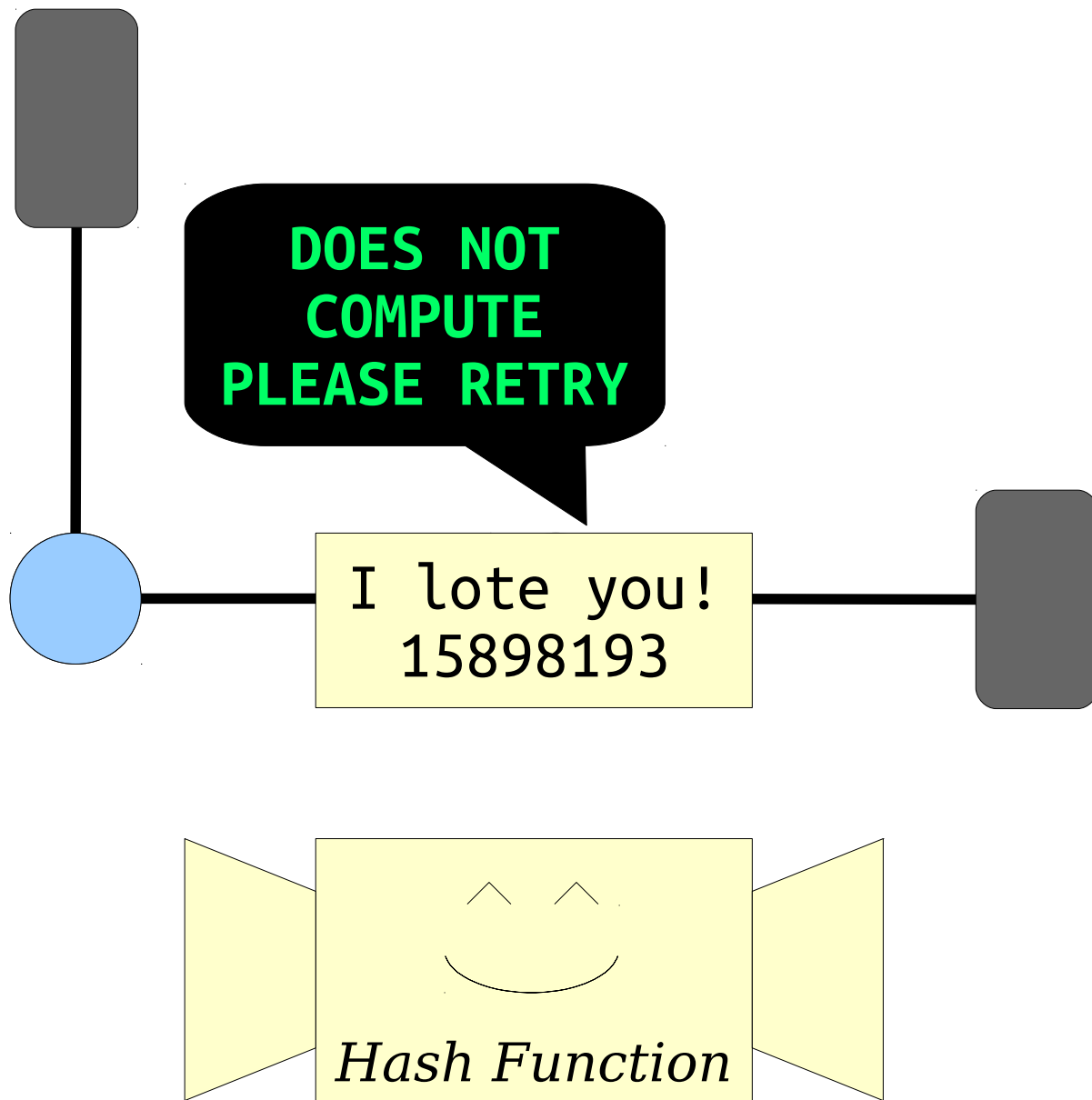
Did my data make it through the network?



Did my data make it through the network?

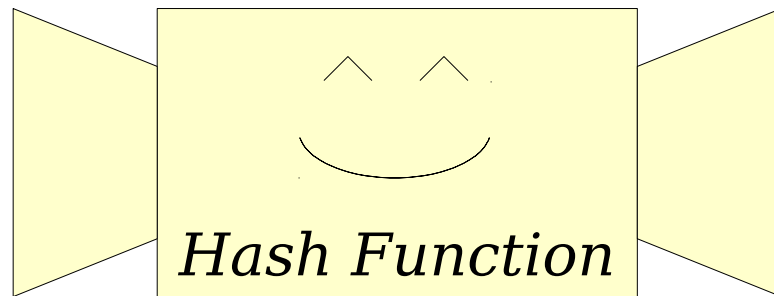
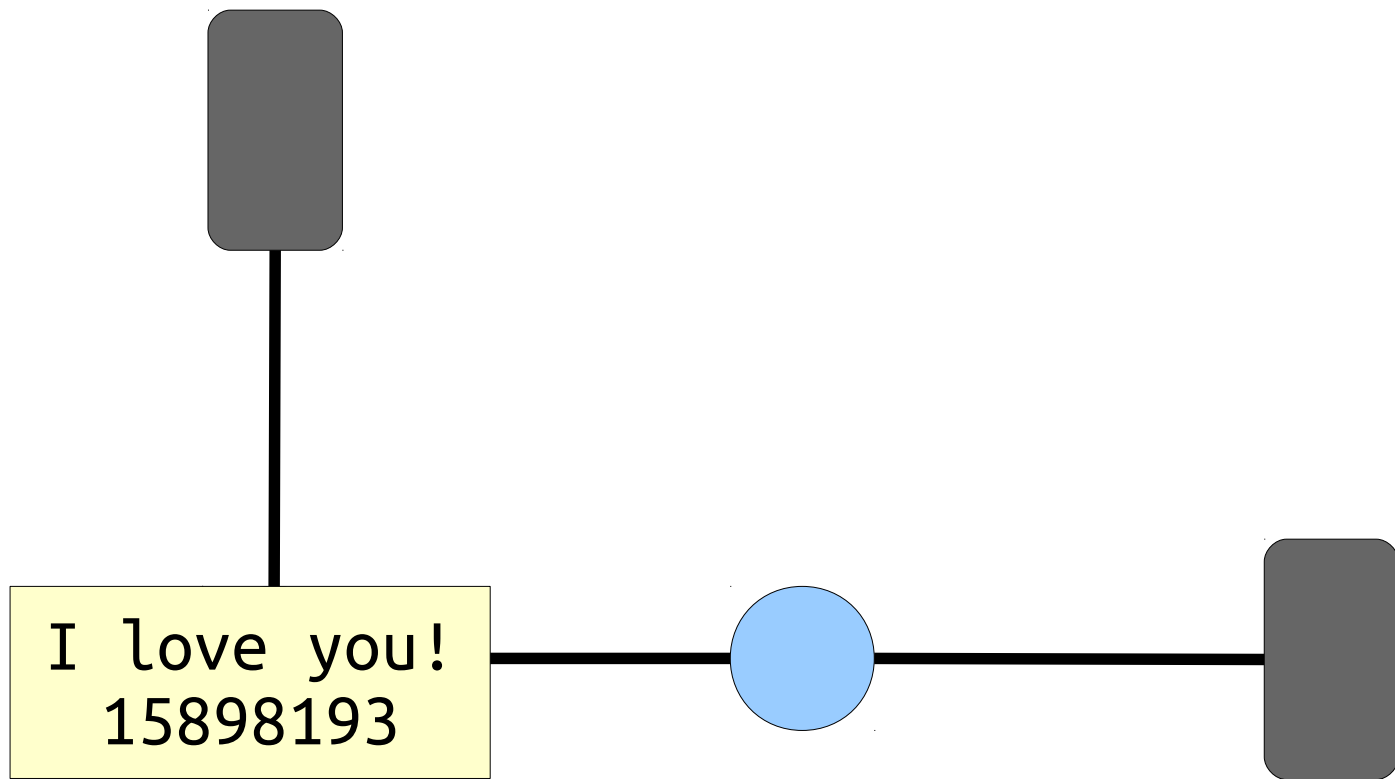


Did my data make it through the network?

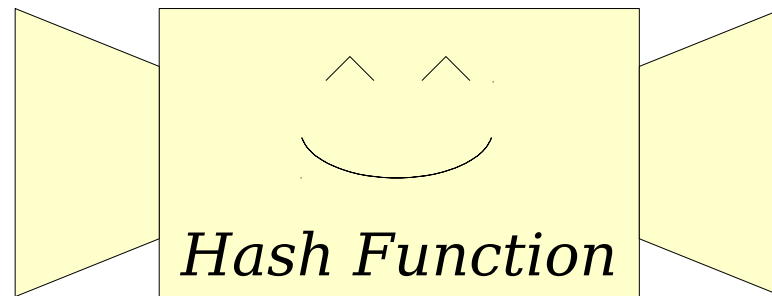
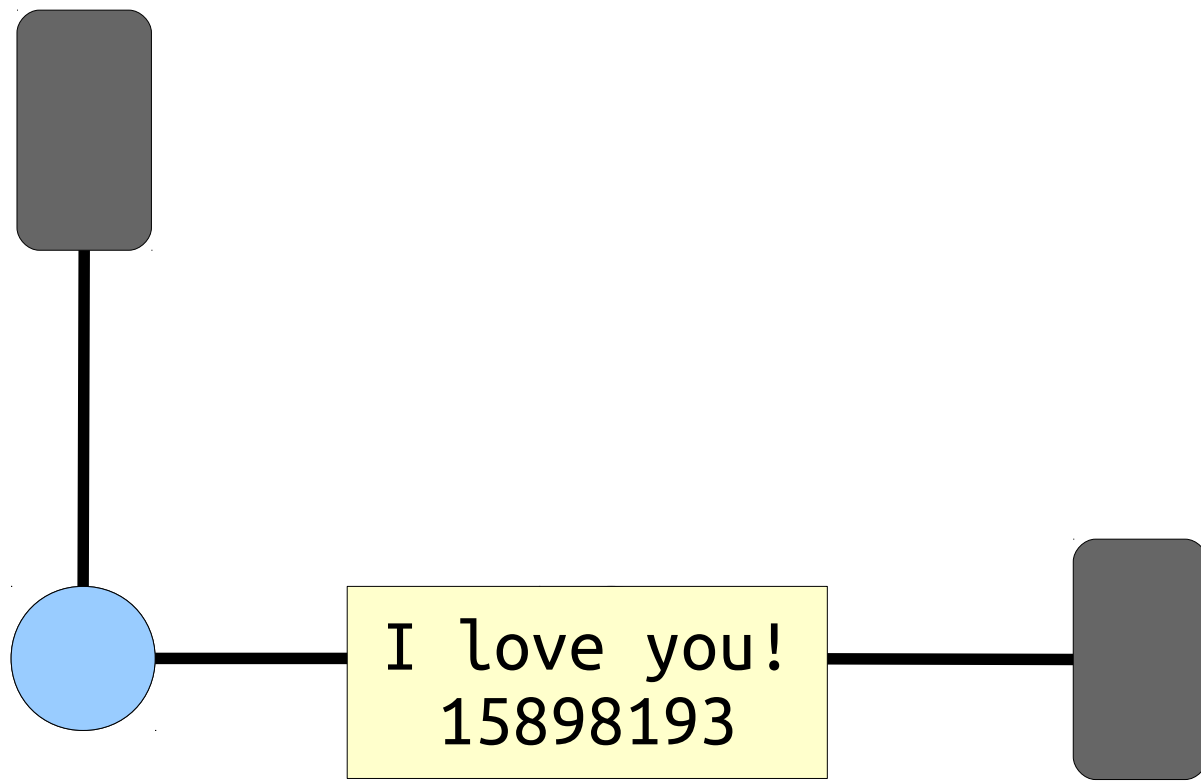


Did my data make it through the network?

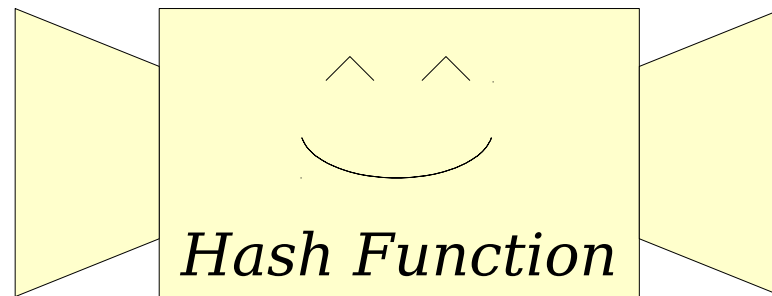
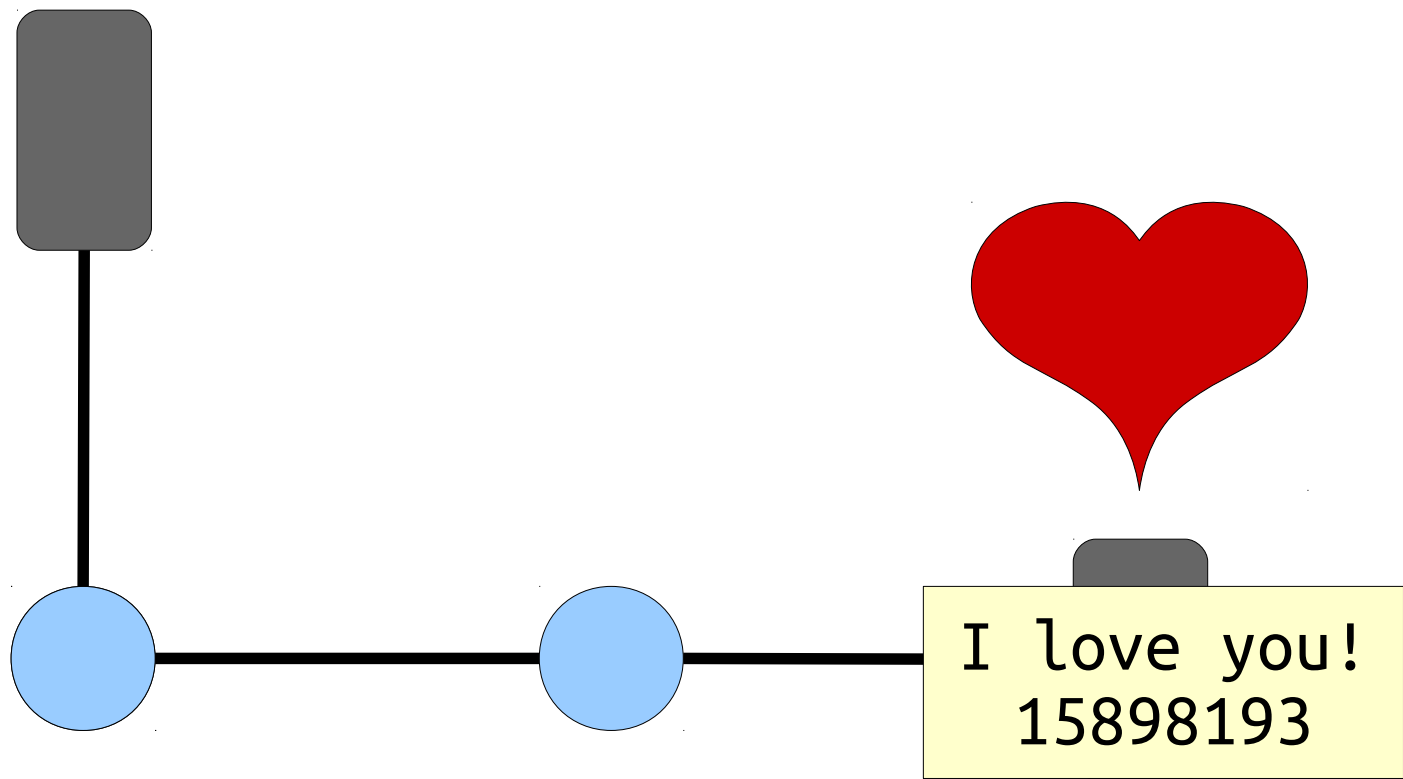




Did my data make it through the network?



Did my data make it through the network?



Did my data make it through the network?

This is done in practice!

Look up ***SHA-256***, the ***Luhn algorithm***,  
and ***CRC32*** for some examples!

And, of course, something to  
do with data structures.

# HashMap and HashSet

# An Example: Clothes



For Large Values of  $n$





# Our Strategy

- Maintain a large number of small collections called **buckets** (think drawers).
- Find a **rule** that lets us tell where each object should go (think knowing which drawer is which.)
- To find something, only look in the bucket assigned to it (think looking for socks.)

# Our Strategy

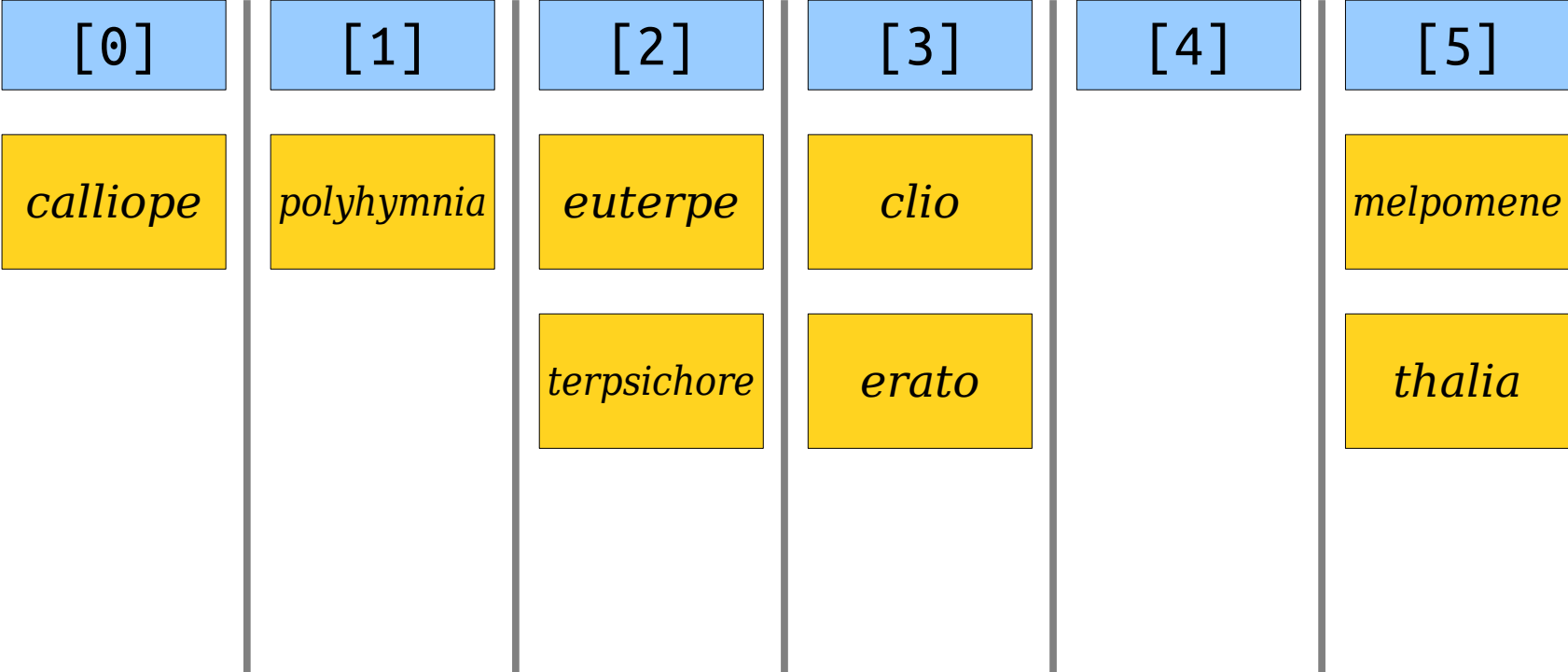
Maintain a large number of small collections called *buckets* (think drawers).

- Find a *rule* that lets us tell where each object should go (think knowing which drawer is which.)

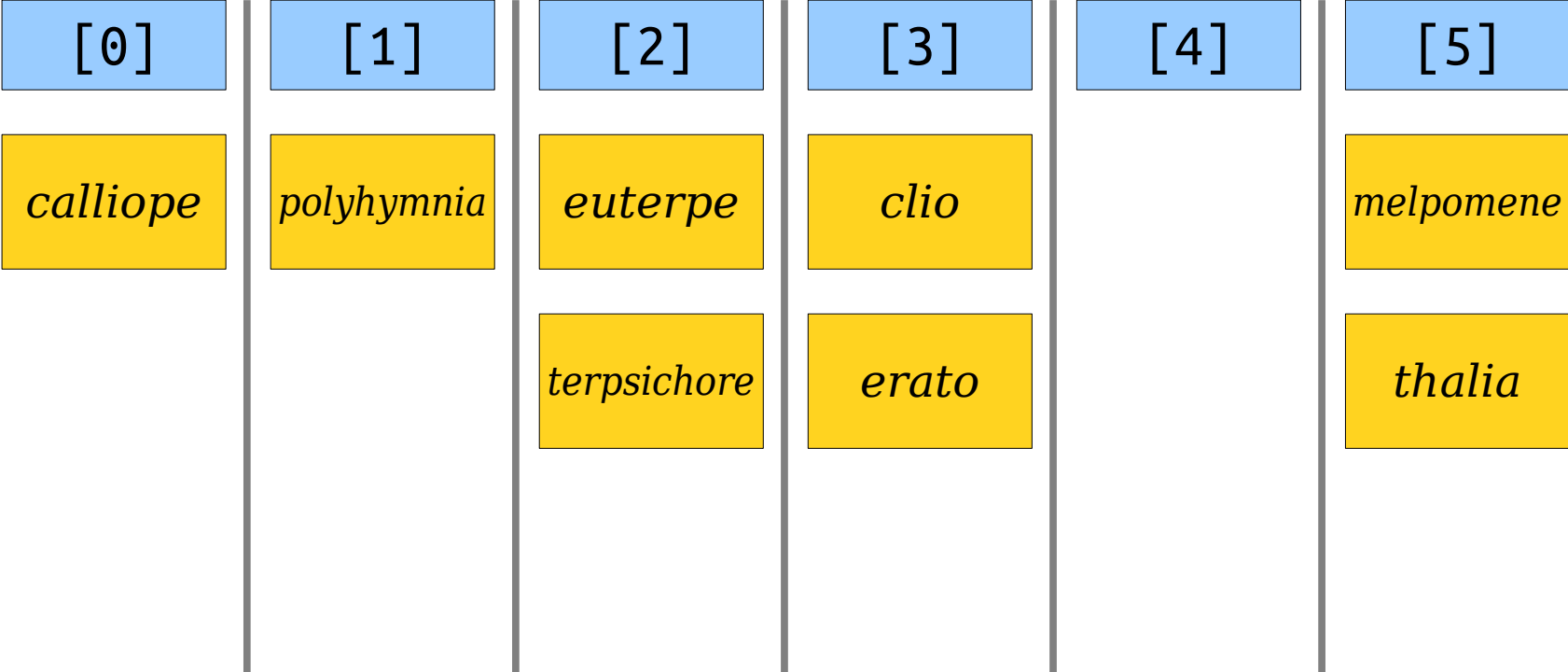
Use a hash  
function!

To find something, only the bucket assigned to it (think looking for socks.)

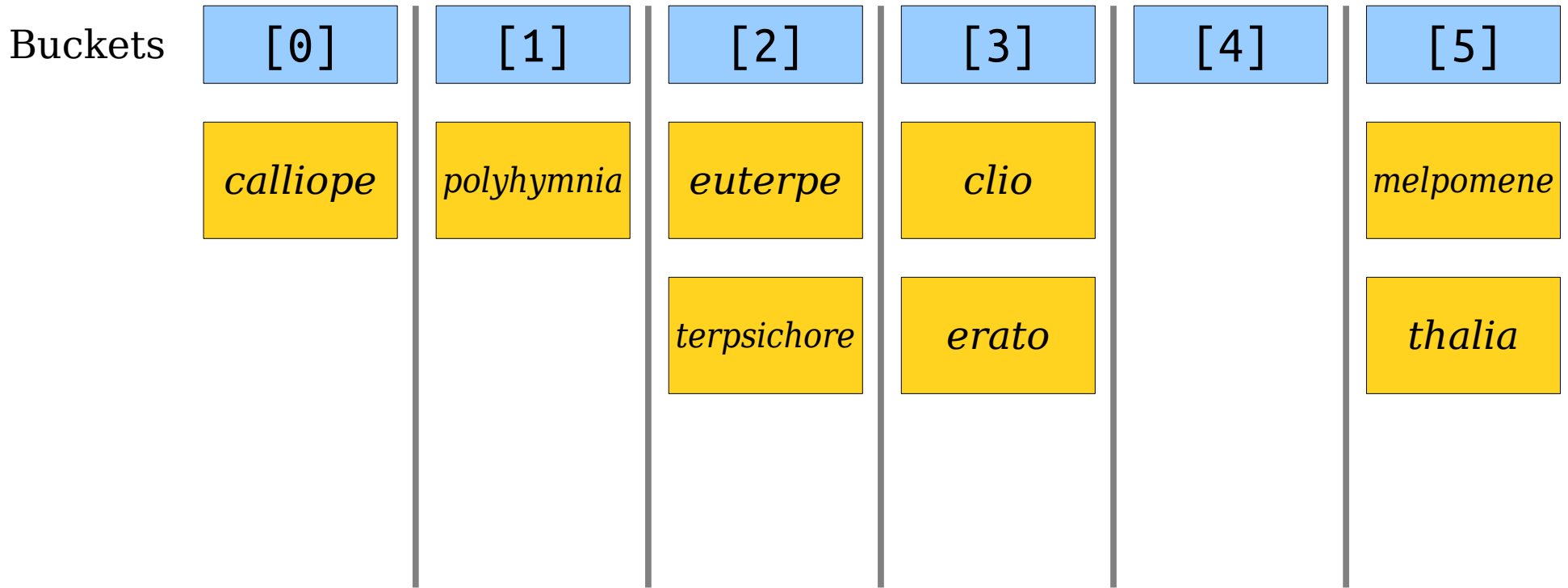
Buckets



Buckets



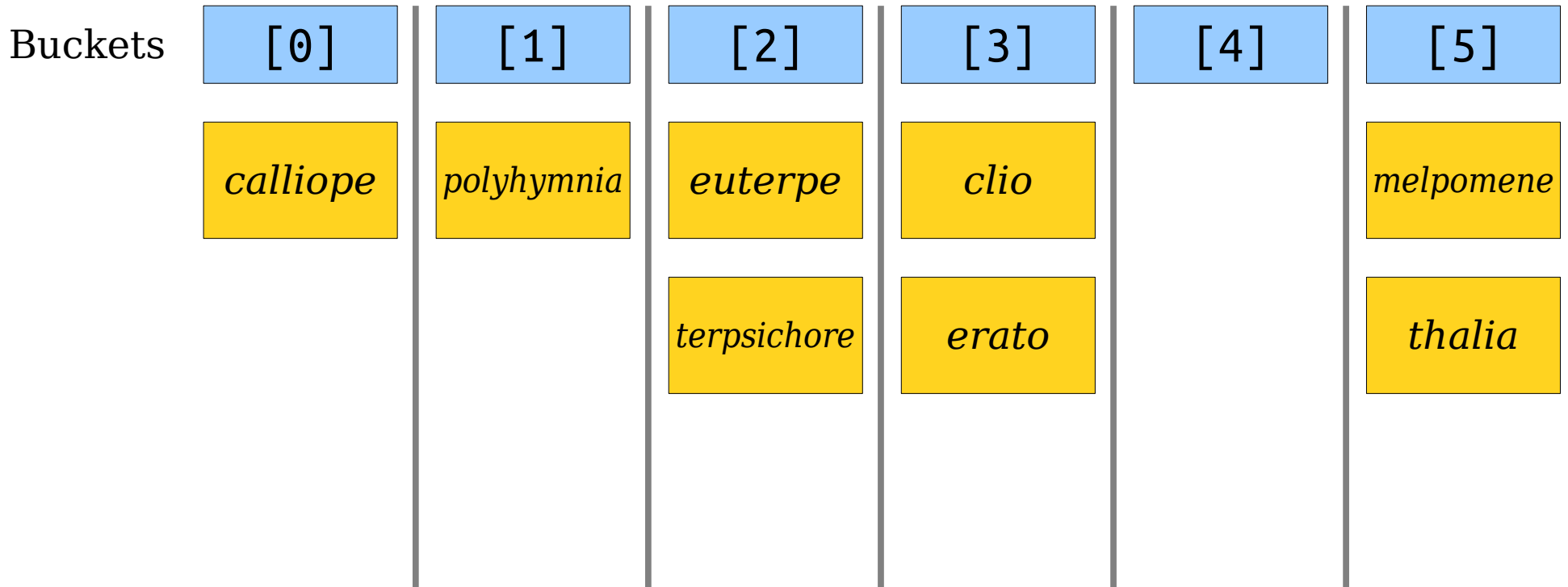
*erato*



```
bool OurHashSet::contains(const string& value) const {
```

```
}
```

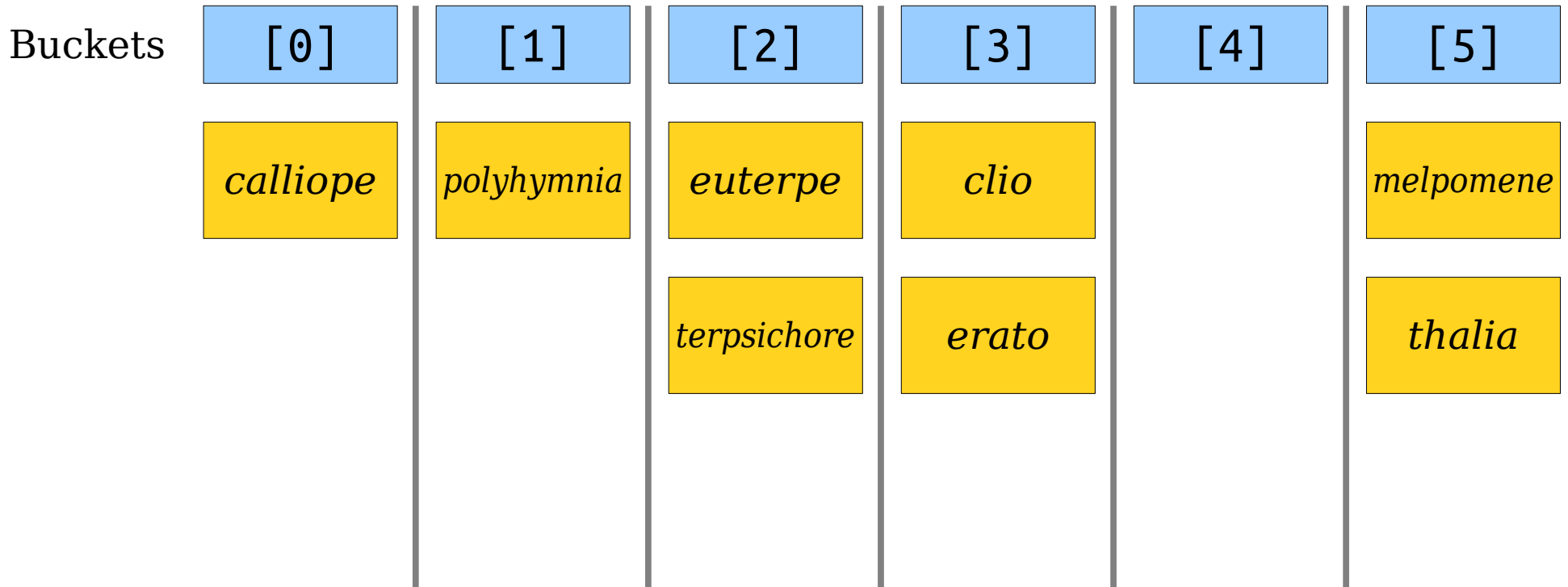
*erato*



```
bool OurHashSet::contains(const string& value) const {  
    int bucket = hashCode(value) % buckets.size();
```

```
}
```

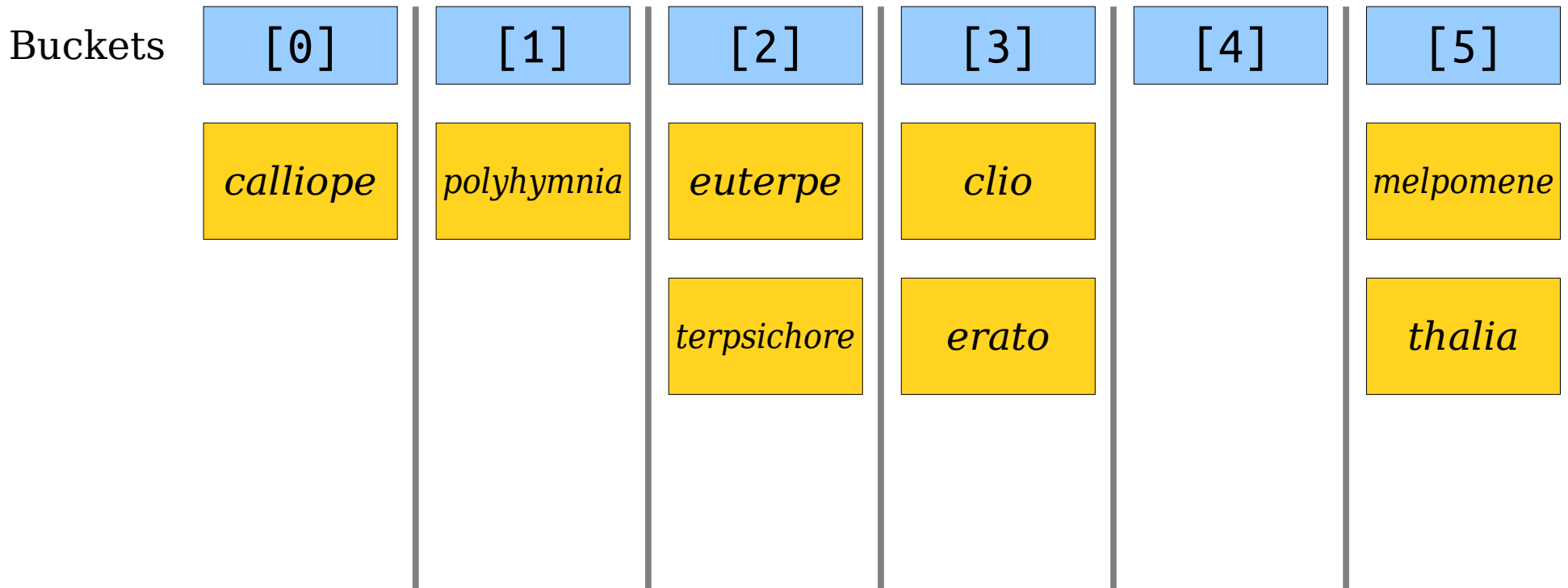
*erato*



```
bool OurHashSet::contains(const string& value) const {  
    int bucket = hashCode(value) % buckets.size();
```

```
}
```

*erato*  
(bucket 3)

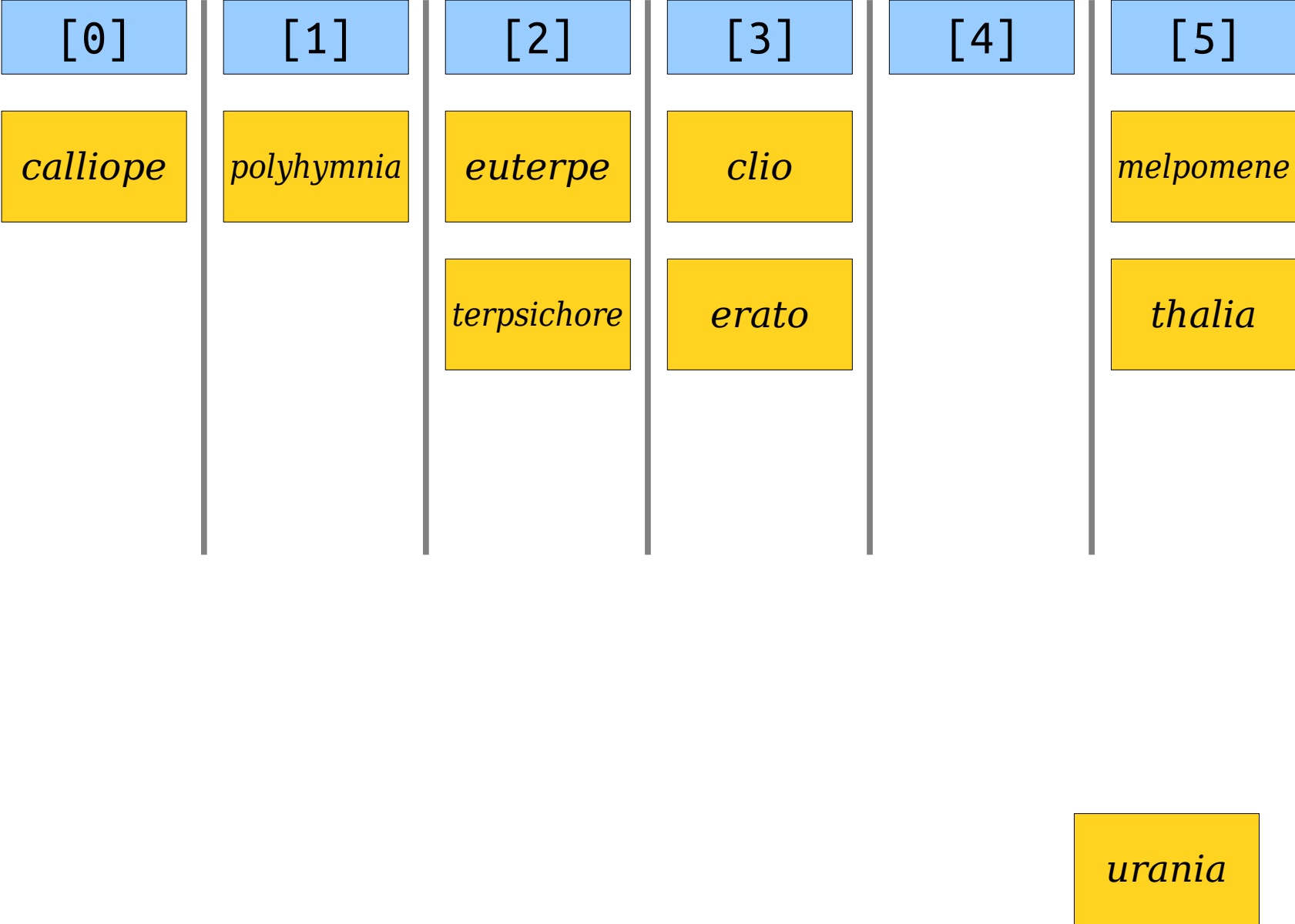


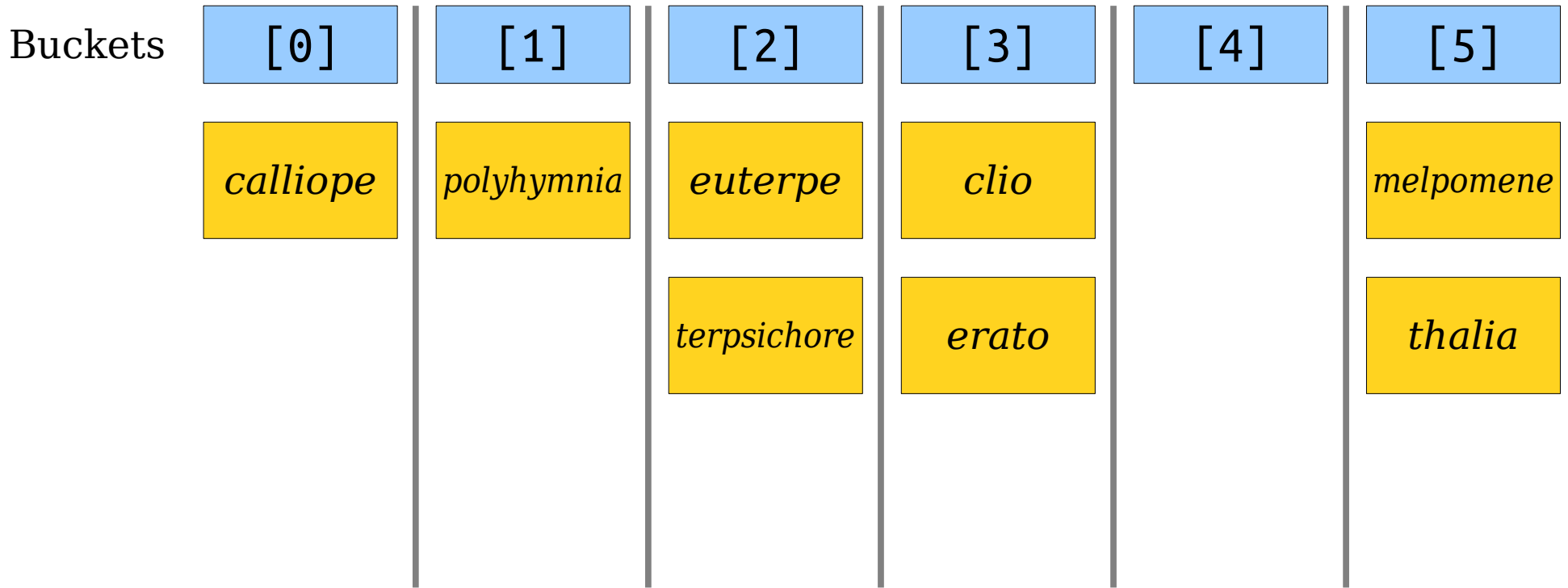
```
bool OurHashSet::contains(const string& value) const {  
    int bucket = hashCode(value) % buckets.size();  
  
    for (string elem: buckets[bucket]) {  
        if (elem == value) return true;  
    }  
  
    return false;  
}
```

*erato*  
(bucket 3)



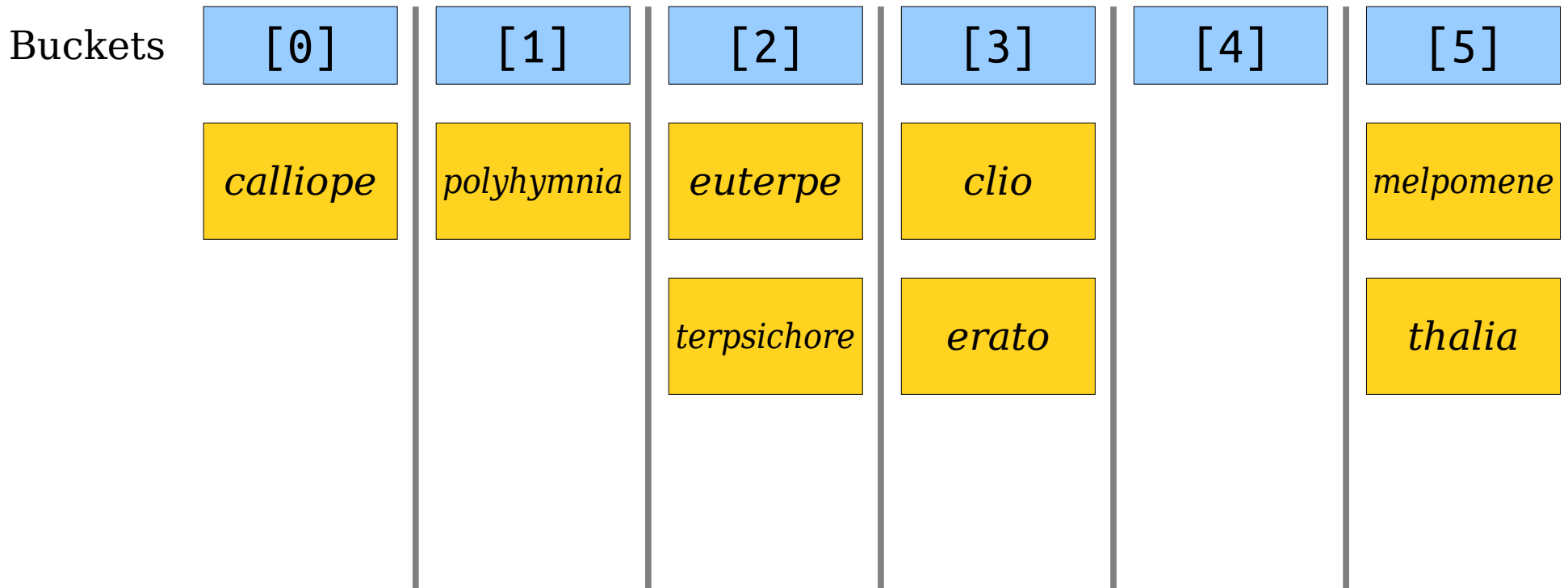
Buckets





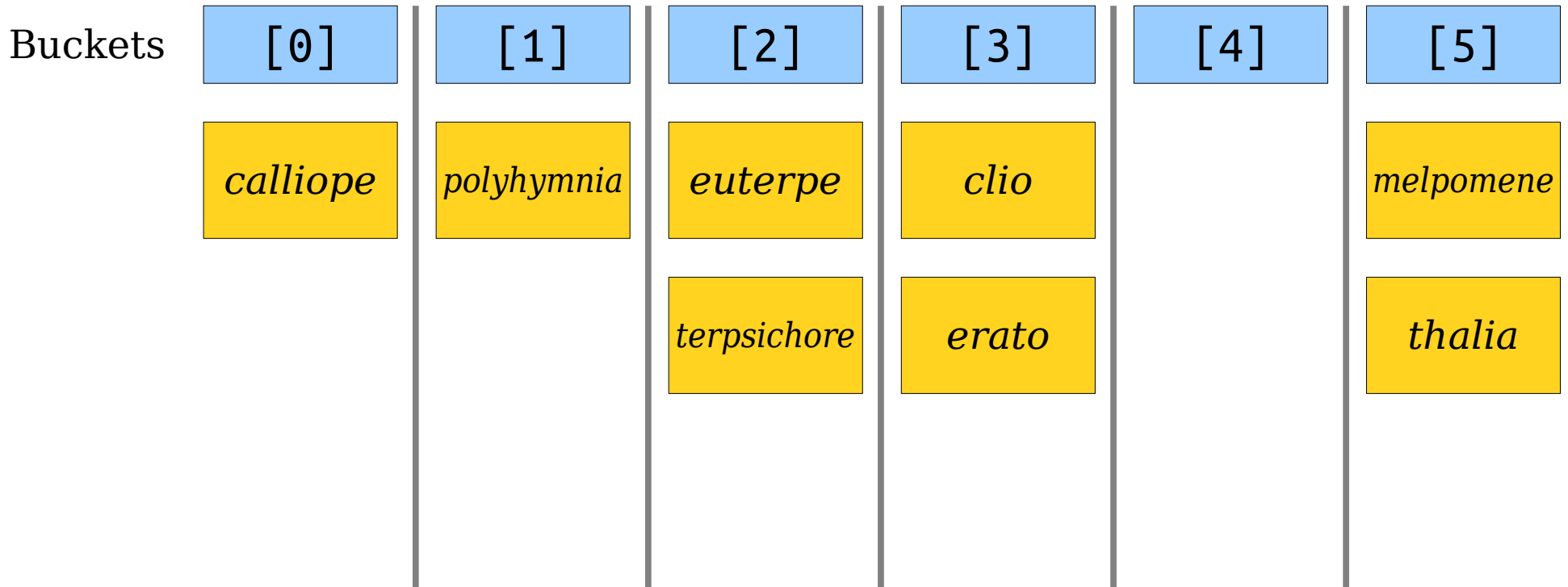
```
void OurHashSet::add(const string& value) {  
  
}
```

*urania*



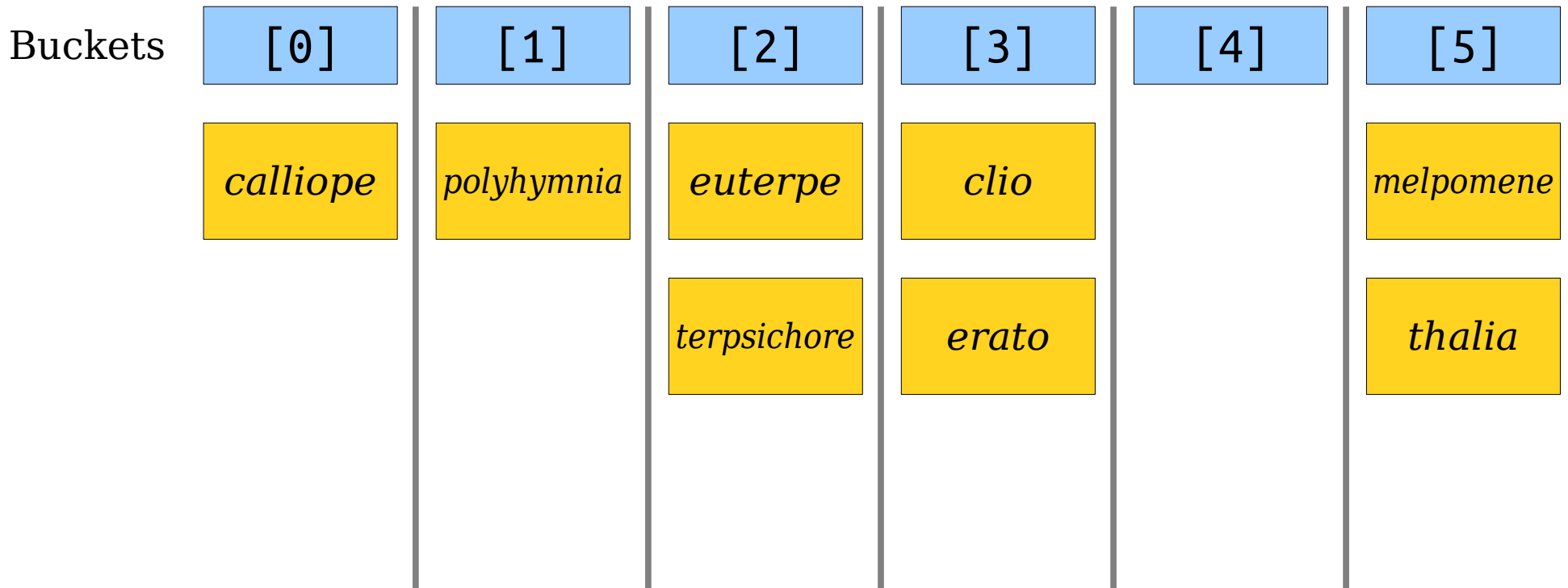
```
void OurHashSet::add(const string& value) {  
    int bucket = hashCode(value) % buckets.size();  
}
```

*urania*



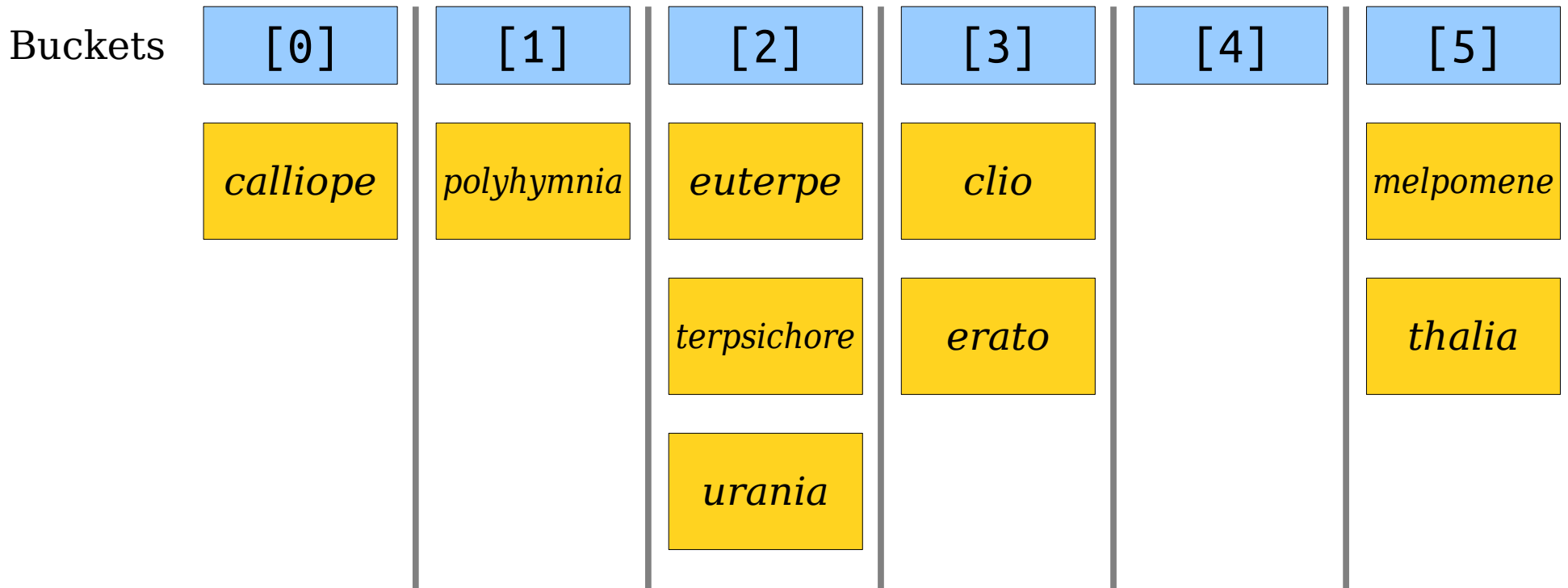
```
void OurHashSet::add(const string& value) {  
    int bucket = hashCode(value) % buckets.size();  
}
```

*urania*  
(bucket 2)



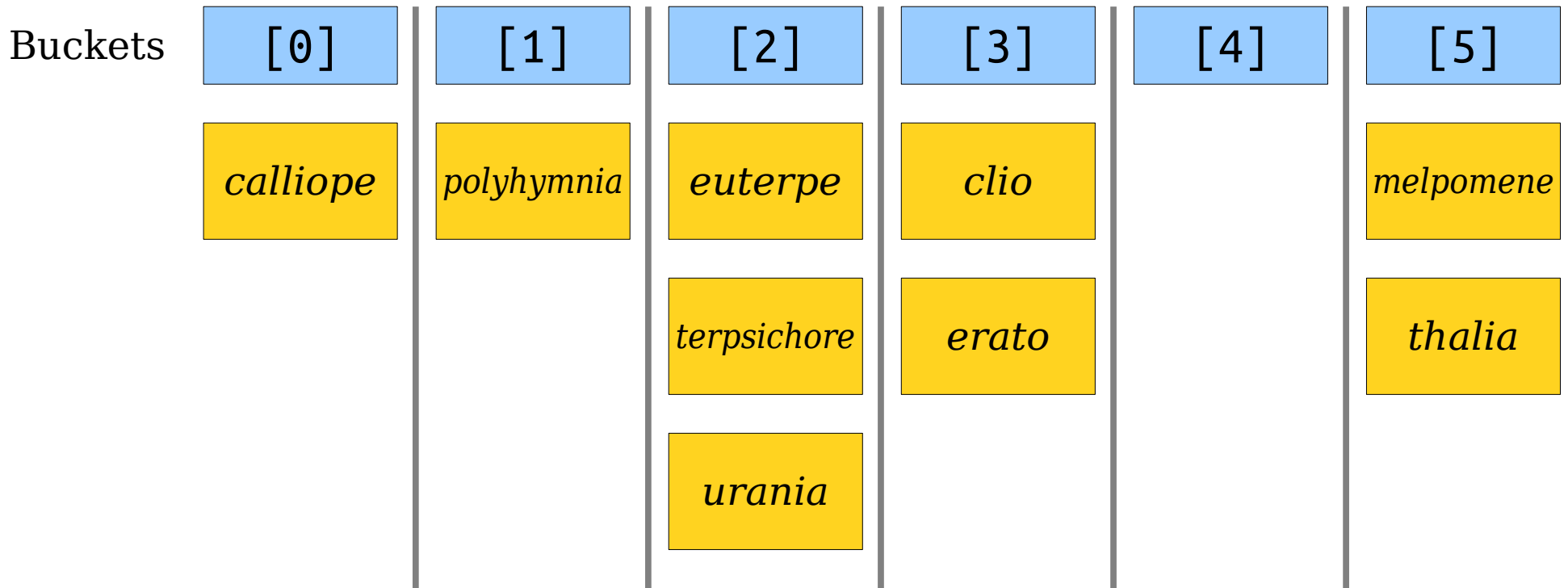
```
void OurHashSet::add(const string& value) {  
    int bucket = hashCode(value) % buckets.size();  
    buckets[bucket] += value;  
}
```

*urania*  
(bucket 2)



```
void OurHashSet::add(const string& value) {  
    int bucket = hashCode(value) % buckets.size();  
    buckets[bucket] += value;  
}
```

*urania*  
(bucket 2)



```
void OurHashSet::add(const string& value) {  
    int bucket = hashCode(value) % buckets.size();  
  
    for (string elem: buckets[bucket]) {  
        if (elem == value) return;  
    }  
  
    buckets[bucket] += value;  
}
```

*urania*  
(bucket 2)

**Time-Out for Announcements!**



# Assignment 6

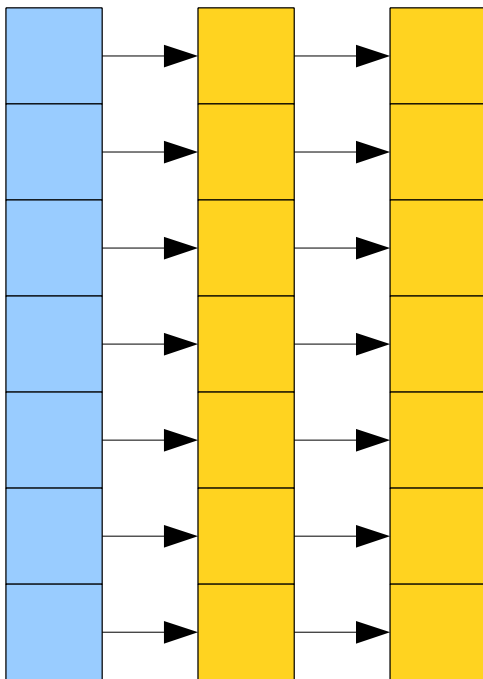
- Assignment 6 (*MiniBrowser*) is due one week from today.
  - Make slow and steady progress on this one. That will give you more time to think things over and process the ideas.
- Need help? Feel free to
  - ask conceptual questions on Piazza,
  - stop by the LaIR for coding help,
  - stop by the CLaIR for conceptual advice,
  - email your section leader for their input,
  - visit Kate in her office hours, or
  - visit Keith in his office hours!

**Back to CS106B!**

How efficient is this?

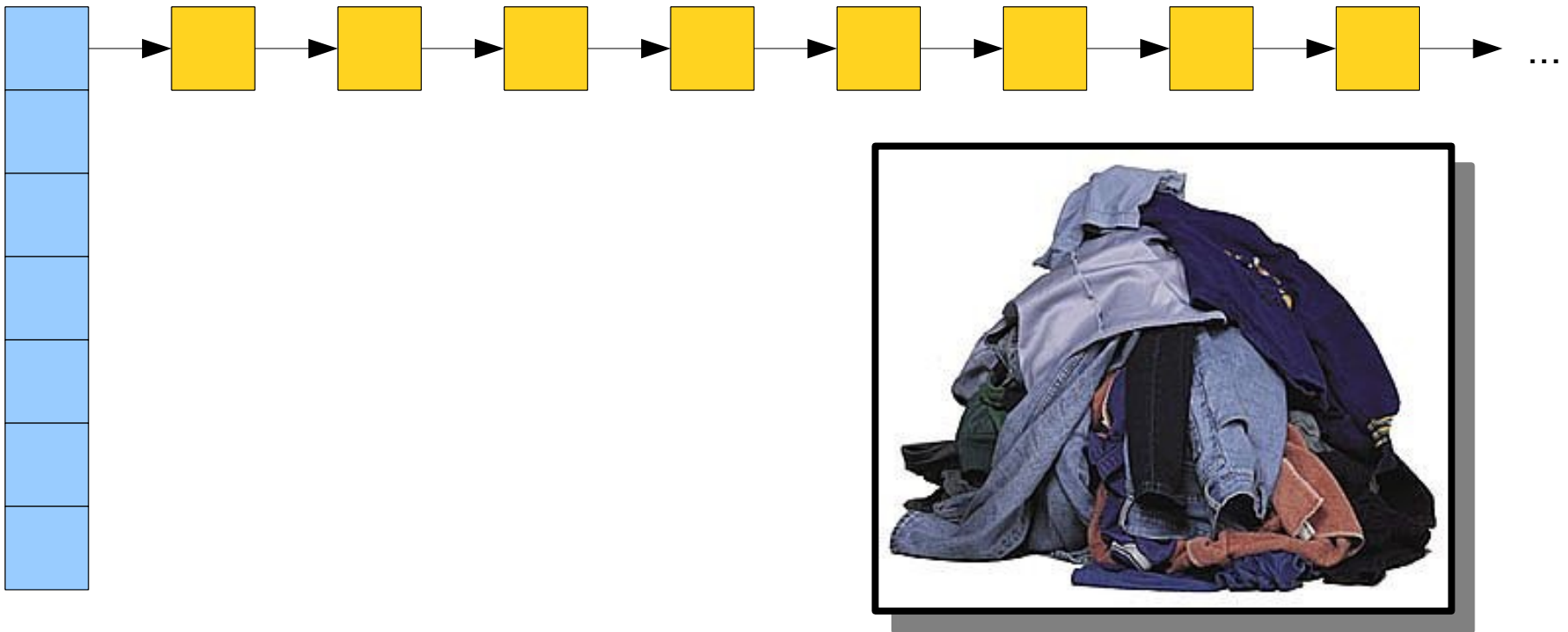
# Efficiency Concerns

- Each hash table operation
  - chooses a bucket and jumps there, then
  - potentially scans everything in the bucket.
- **Claim:** The efficiency of our hash table depends on how well-spread the elements are.



# Efficiency Concerns

- Each hash table operation
  - chooses a bucket and jumps there, then
  - potentially scans everything in the bucket.
- **Claim:** The efficiency of our hash table depends on how well-spread the elements are.



# Efficiency Concerns

- For a hash table to be fast, we need a hash function that spreads things around nicely.
- Lots of smart people have worked on this problem. If you need a hash function, use one that someone else developed.
  - Curious to see how? Take CS161, CS166, or CS255!
- Most programming languages come with some built-in hash functions for basic types. There are standard techniques for combining them together.

# Analyzing our Efficiency

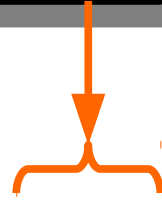
- Let's suppose we have a “strong” hash function that distributes elements fairly evenly.
- Imagine we have  $b$  buckets and  $n$  elements in our table.
- On average, how many elements will be in a bucket?

Answer:  $n / b$

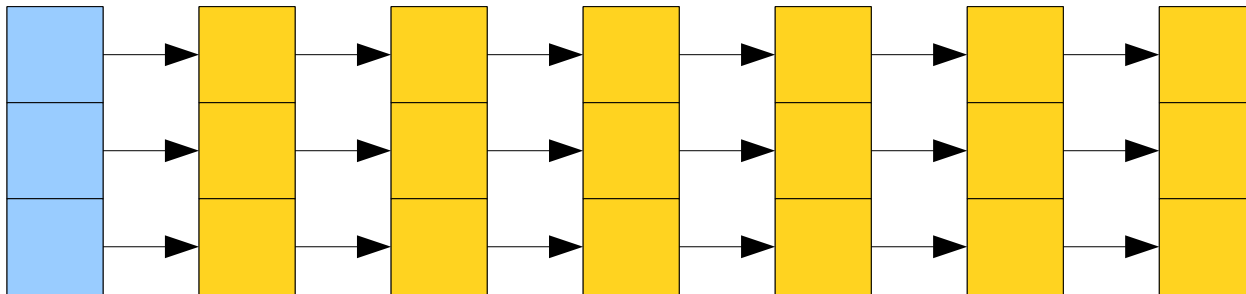
- The cost of an insertion, deletion, or lookup is therefore

$$O(1 + n / b).$$

The more elements we have, the more work we have to do.



$$O(1 + \mathbf{n} / \mathbf{b})$$

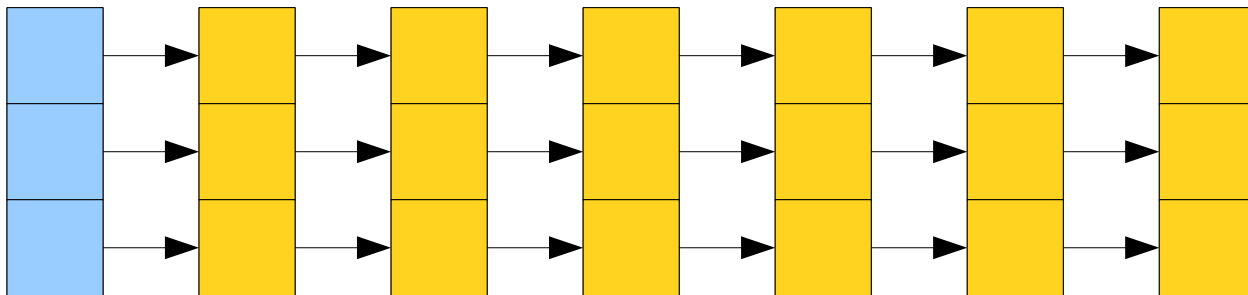




The more elements we have, the more work we have to do.

$$O(1 + n / b)$$

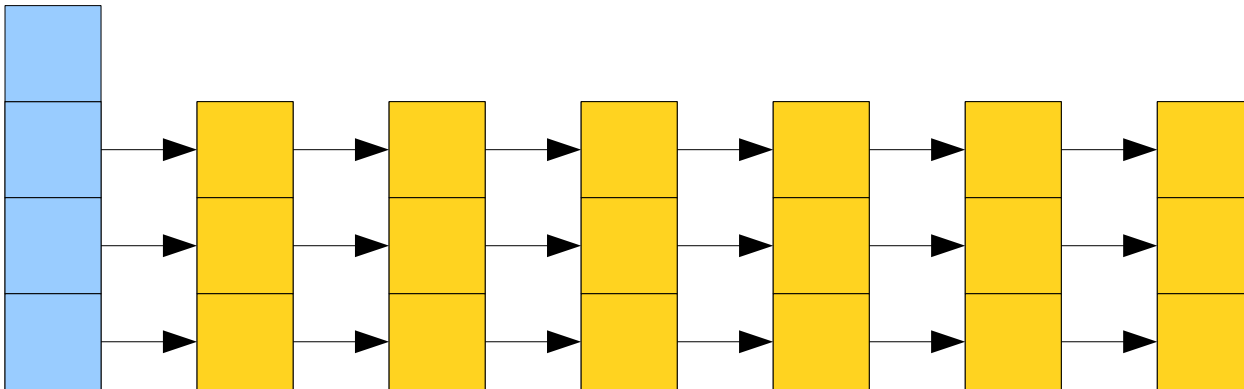
The more buckets we have, the less work we have to do.



The more elements we have, the more work we have to do.

$$O(1 + n / b)$$

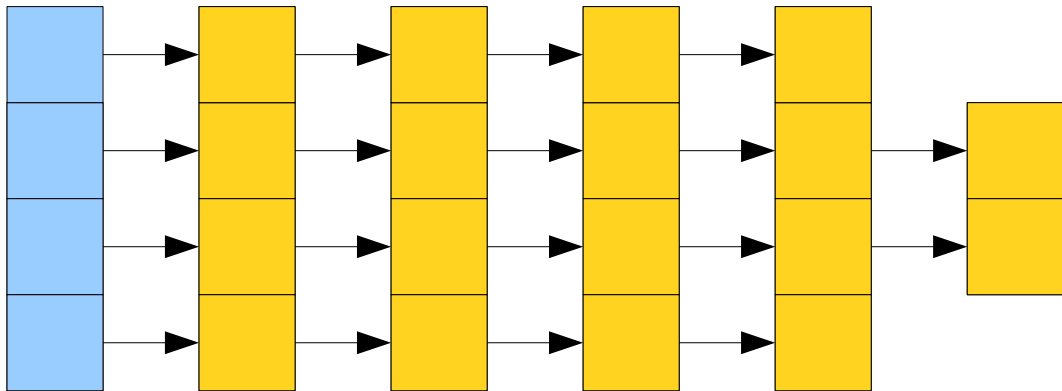
The more buckets we have, the less work we have to do.



The more elements we have, the more work we have to do.

$$O(1 + n / b)$$

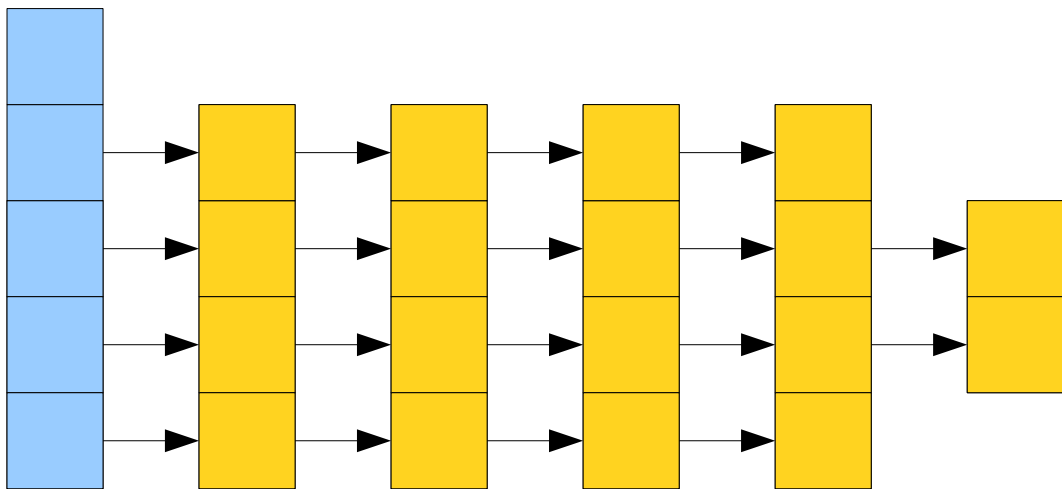
The more buckets we have, the less work we have to do.



The more elements we have, the more work we have to do.

$$O(1 + n / b)$$

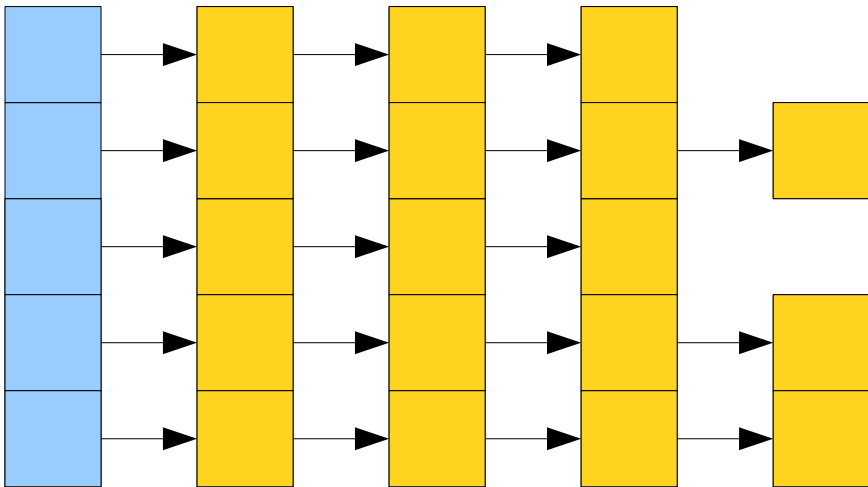
The more buckets we have, the less work we have to do.



The more elements we have, the more work we have to do.

$$O(1 + n / b)$$

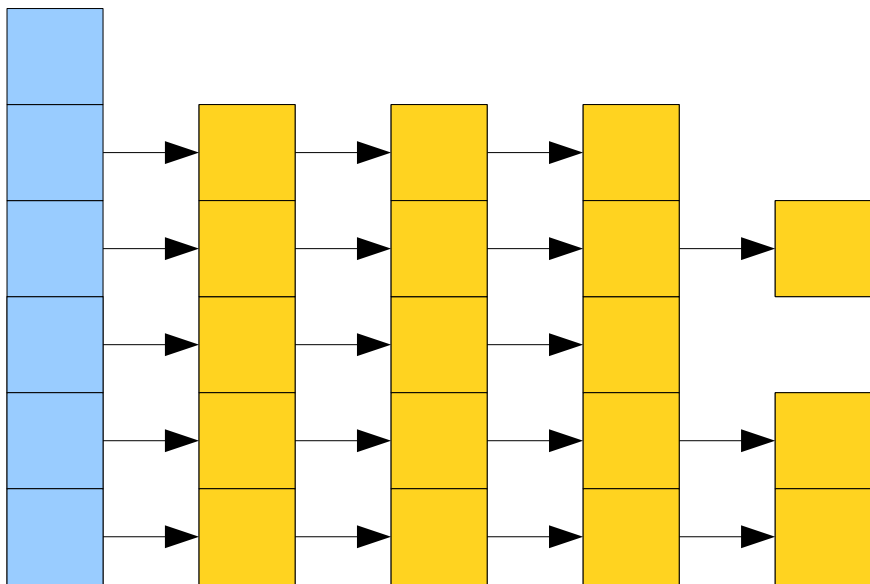
The more buckets we have, the less work we have to do.



The more elements we have, the more work we have to do.

$$O(1 + n / b)$$

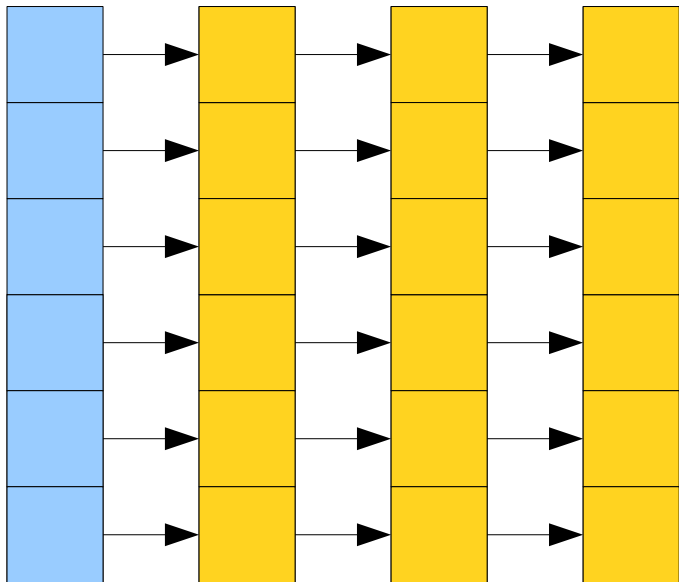
The more buckets we have, the less work we have to do.



The more elements we have, the more work we have to do.

$$O(1 + n / b)$$

The more buckets we have, the less work we have to do.



If we have way more elements than buckets, we waste time.

$$O(1 + n / b)$$

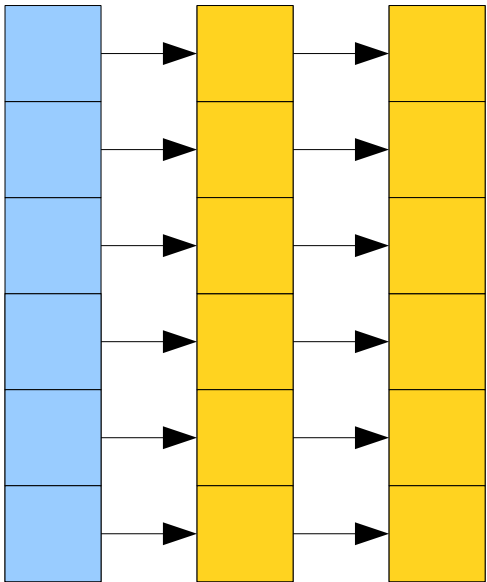
If we have way more buckets than elements, we waste space.



If we have way more elements than buckets, we waste time.

$$O(1 + n / b)$$

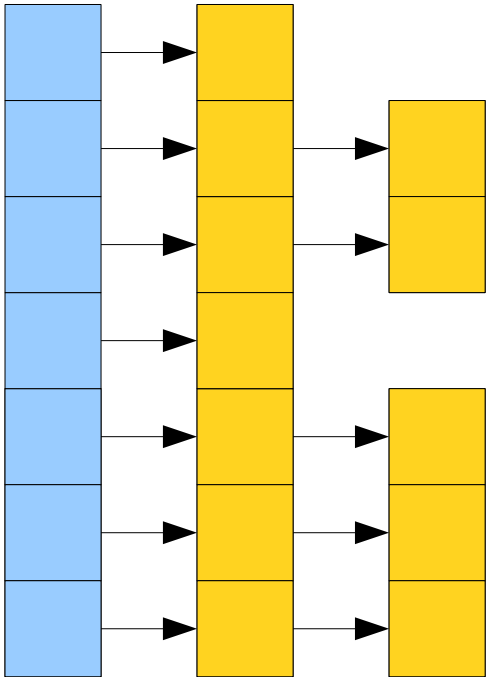
If we have way more buckets than elements, we waste space.



If we have way more elements than buckets, we waste time.

$$O(1 + n / b)$$

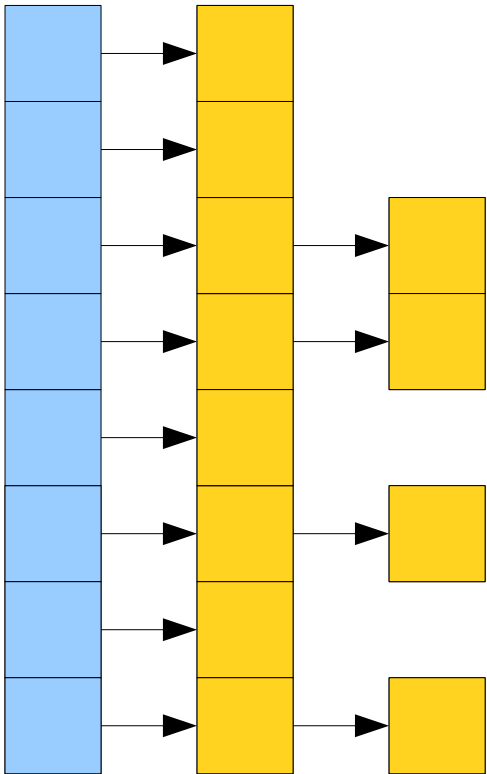
If we have way more buckets than elements, we waste space.



If we have way more elements than buckets, we waste time.

$$O(1 + n / b)$$

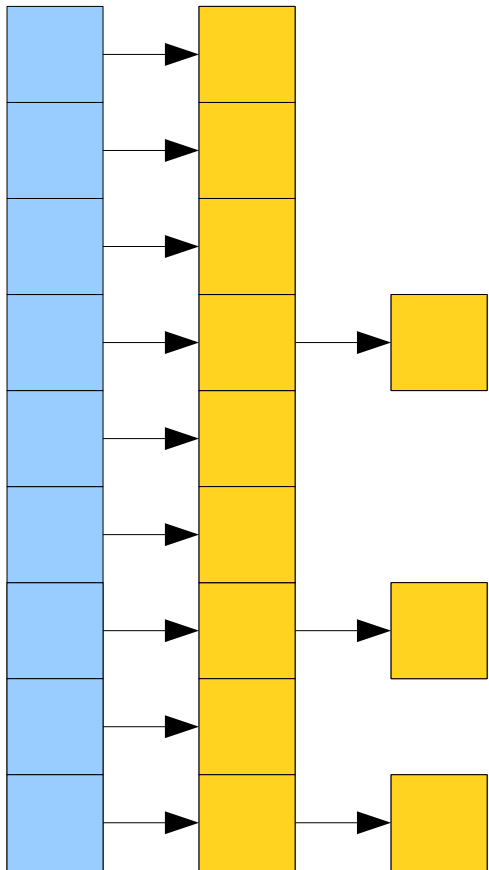
If we have way more buckets than elements, we waste space.



If we have way more elements than buckets, we waste time.

$$O(1 + n / b)$$

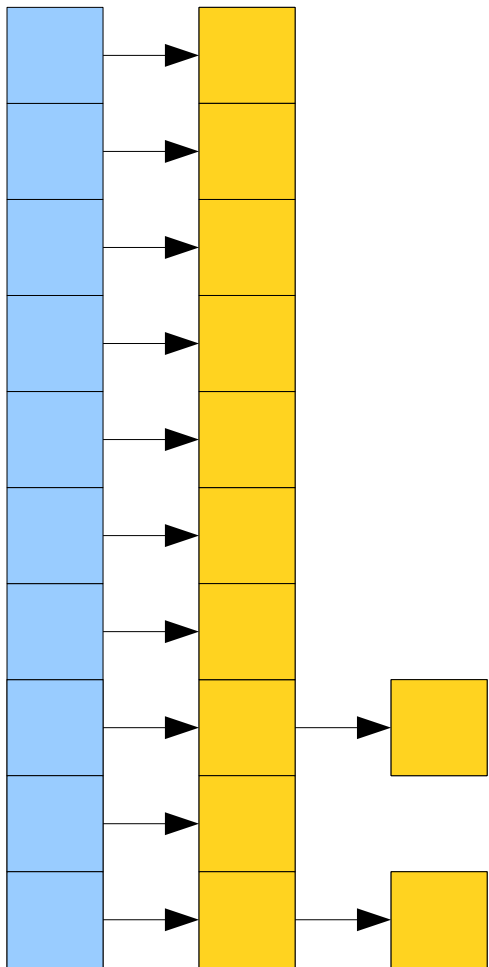
If we have way more buckets than elements, we waste space.



If we have way more elements than buckets, we waste time.

$$O(1 + n / b)$$

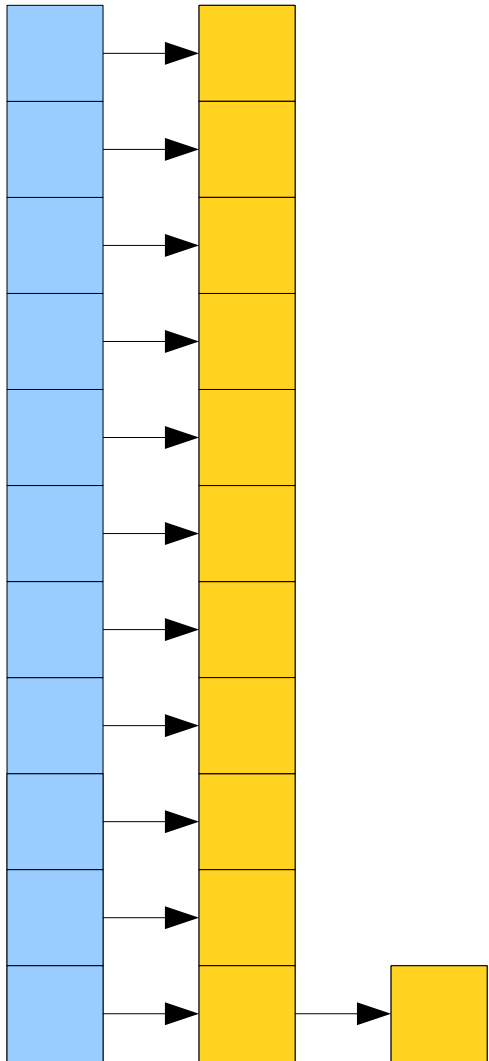
If we have way more buckets than elements, we waste space.



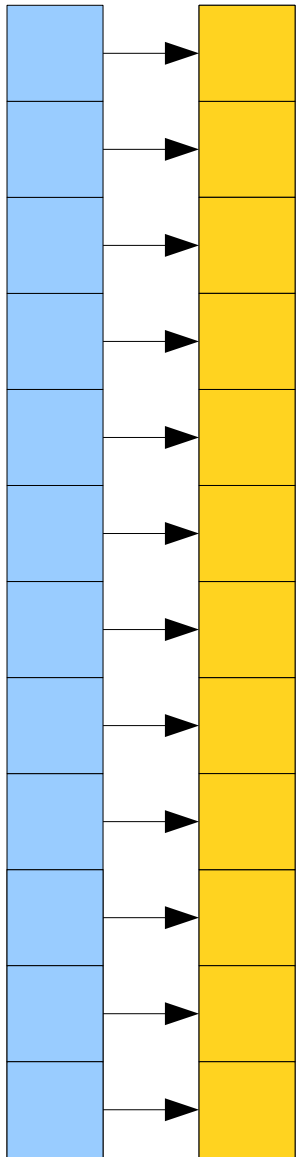
If we have way more elements than buckets, we waste time.

$$O(1 + n / b)$$

If we have way more buckets than elements, we waste space.



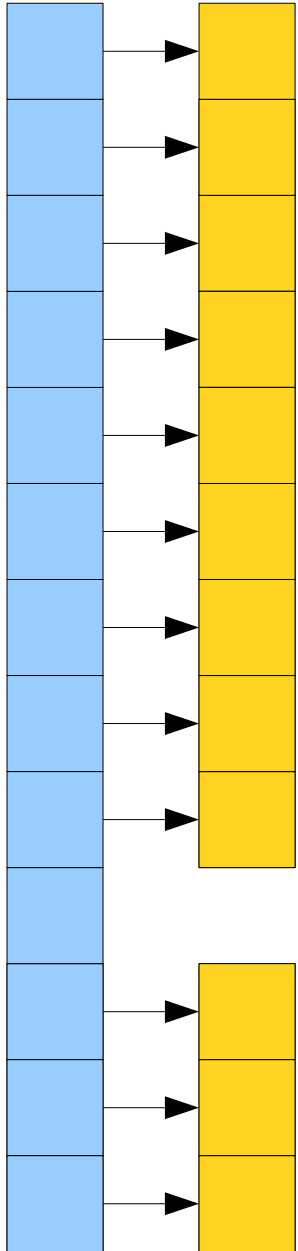
If we have way more elements than buckets, we waste time.



$$O(1 + n / b)$$

If we have way more buckets than elements, we waste space.

If we have way more elements than buckets, we waste time.



$$O(1 + n / b)$$

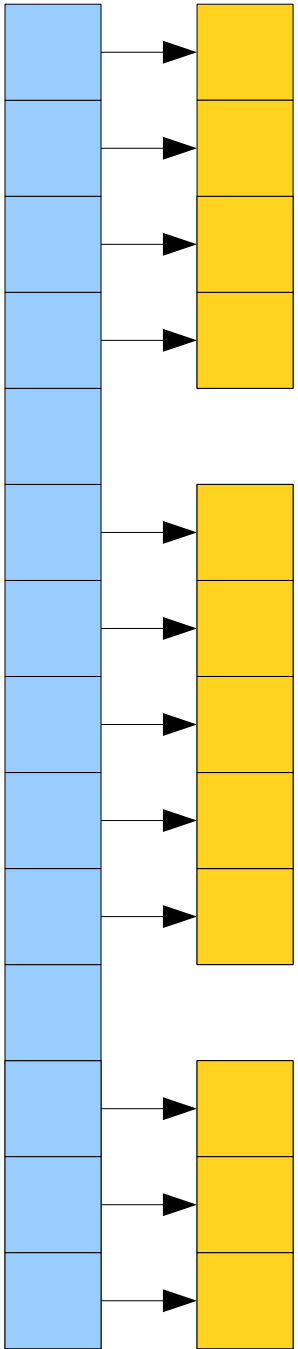
If we have way more buckets than elements, we waste space.



If we have way more elements than buckets, we waste time.

$$O(1 + n / b)$$

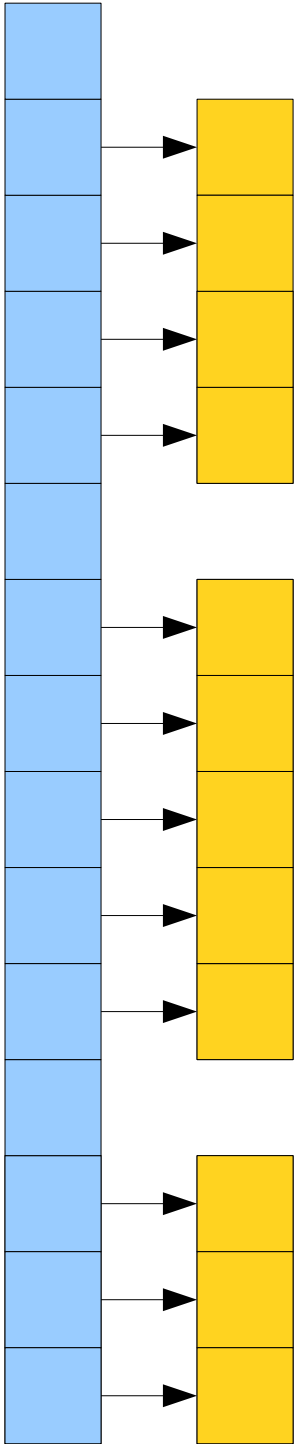
If we have way more buckets than elements, we waste space.



If we have way more elements than buckets, we waste time.

$$O(1 + n / b)$$

If we have way more buckets than elements, we waste space.



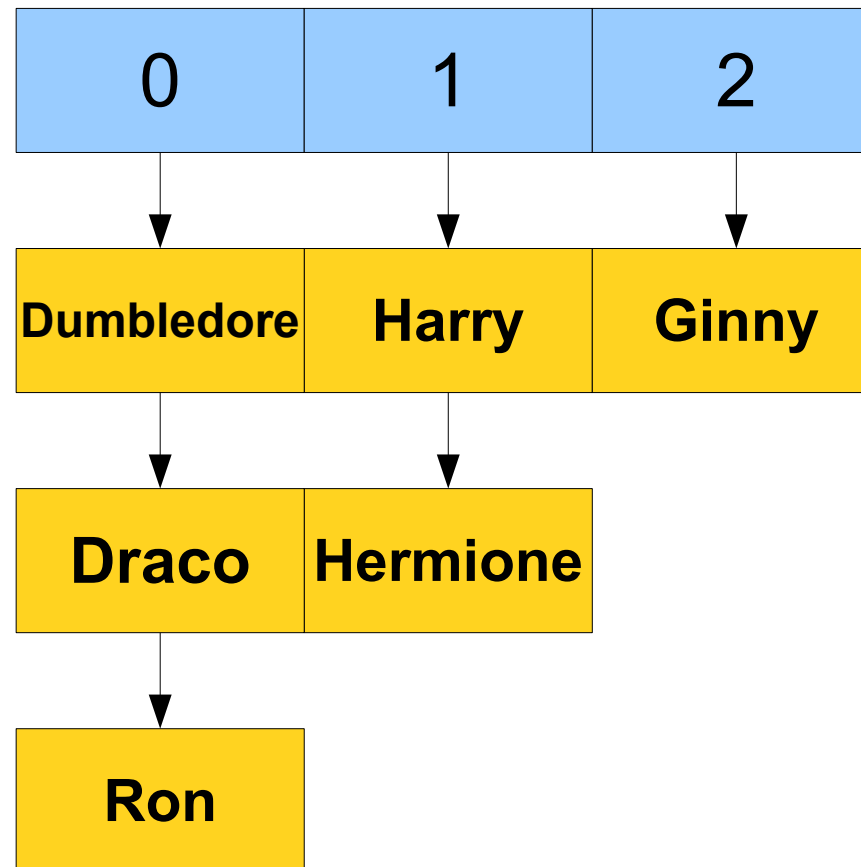
Balancing these Factors

The ***load factor*** of a hash table is the ratio of the number of elements to the number of buckets.

That is, it's the value of  $n / b$ .

***Idea:*** Keep the load factor below some constant (say, two), and increase the number of buckets when it gets too big.

# Hashing and Rehashing



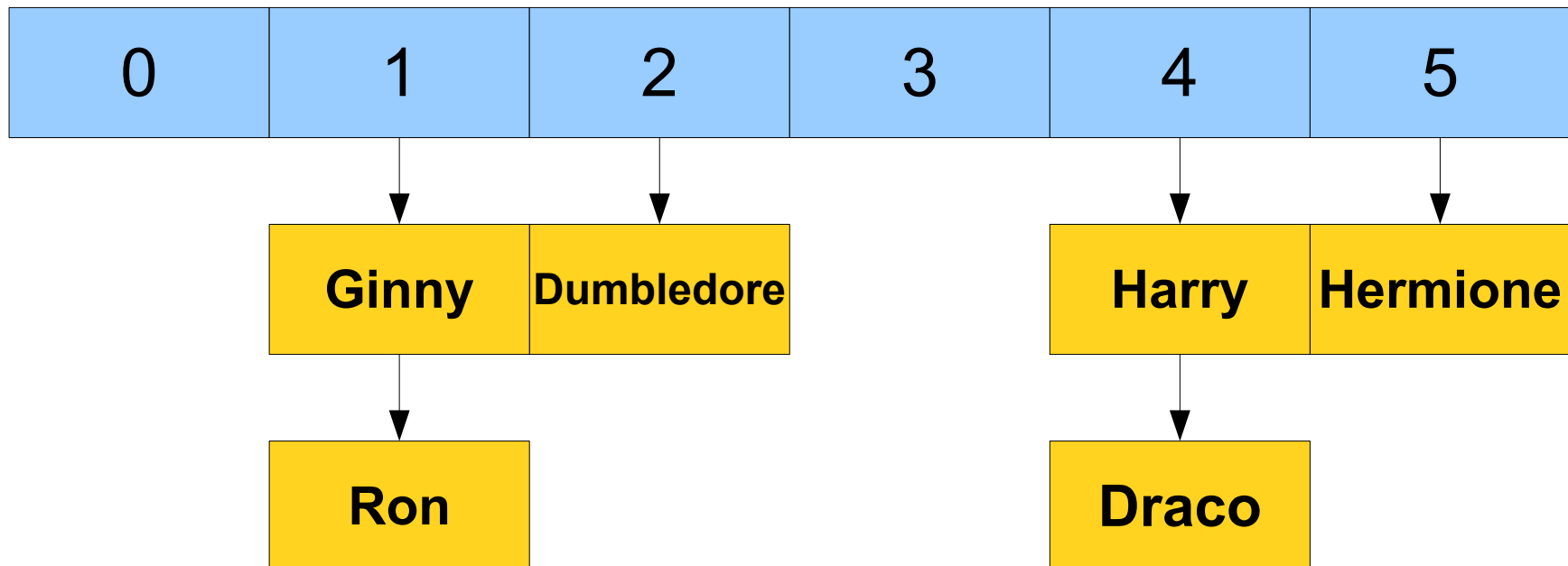
# Hashing and Rehashing

**Voldemort**

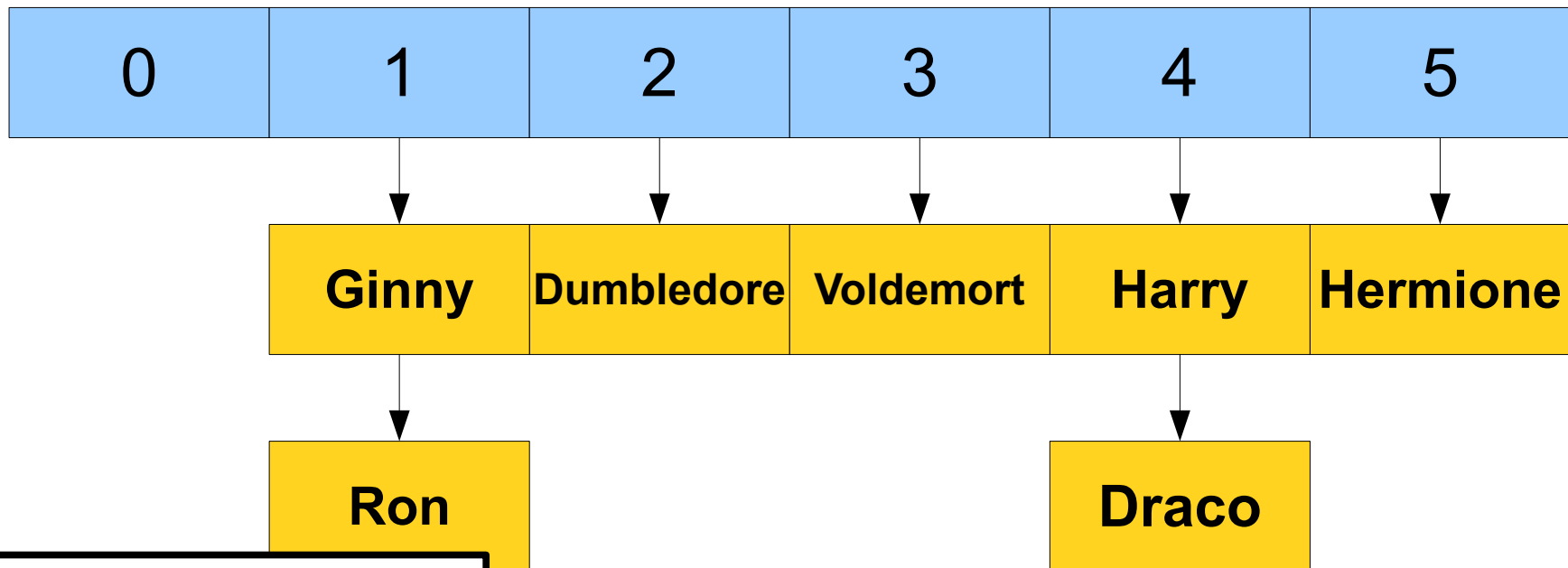


# Hashing and Rehashing

**Voldemort**



# Hashing and Rehashing



Totally unrelated: look up the term "Voldemort Type."



# Hashing and Rehashing

- When inserting, if  $n / b$  exceeds some small constant (say, 2), double the number of buckets and redistribute the elements into the new table.
  - As with Stack, this rehashing happens so infrequently that it's extremely fast on average.
- This makes  $n / b \leq 2$ , so the expected lookup time in a hash table is  **$O(1)$** .
- On average, the lookup time is *independent* of the total number of elements in the table!

# Hashing vs. BSTs

- The cost of an insertion, lookup, or deletion in a hash table is, on average,  **$O(1)$** .
  - This assumes you have a good choice of hash function. Unless you have a background in abstract algebra, just follow a template. ☺
- This contrasts with the  $O(\log n)$  operations on Map or Set.
- ***Don't pay for what you don't use***. If you need things in sorted order, or want to perform range searches, go with Map or Set. If you don't need things sorted, opt for HashMap or HashSet.

# More to Explore

- Hash functions can be used to approximate the sizes of data sets and to find frequent elements in a data stream (***cardinality estimators, count-min sketches***).
  - Curious? Take CS166, CS168, or CS263!
- They can also be used to find objects that are similar to one another (***locality-sensitive hashing***).
  - Curious? Take CS246 or CS265!
- They're one of the key steps in blockchain technology (***SHA-256***).
  - Curious? Take CS251!
- They're used to build tamperproofing for online information and physical medicines (***HMAC***)
  - Curious? Take CS255!

# Next Time

