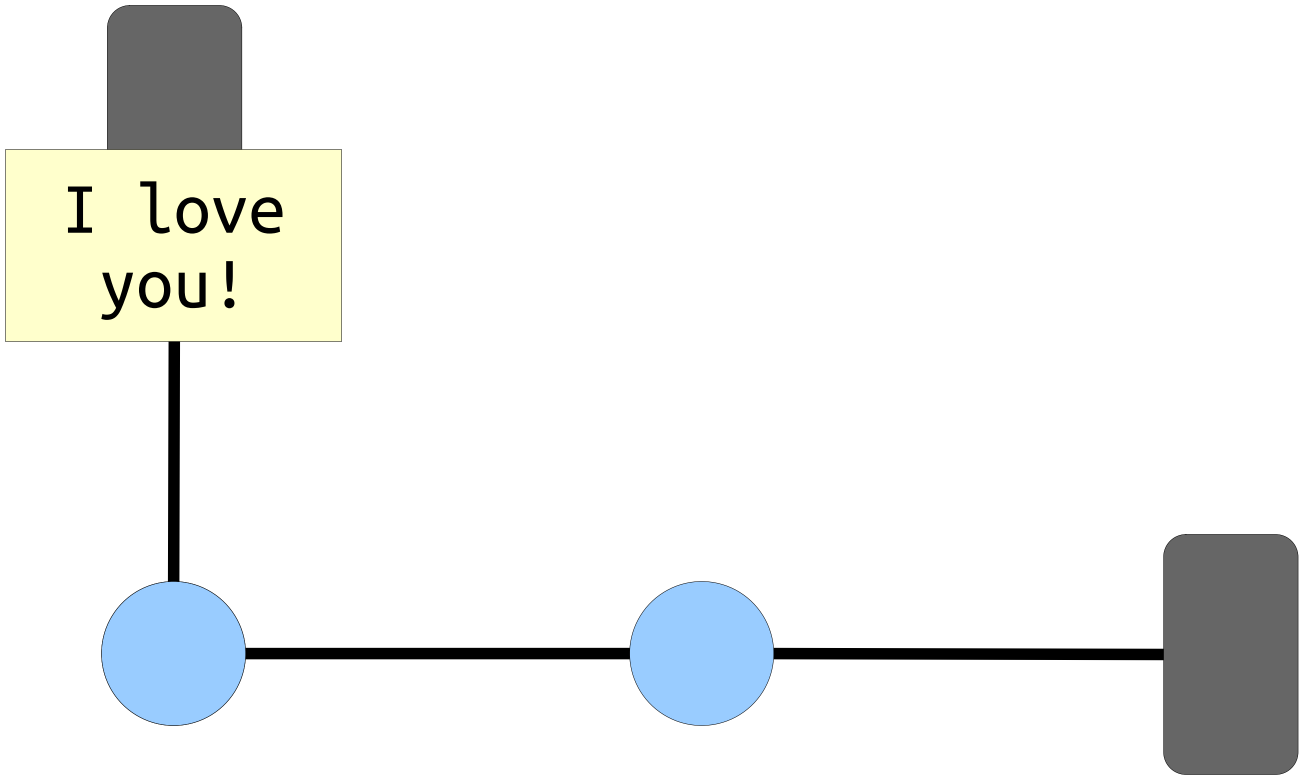
Hashing

Part One

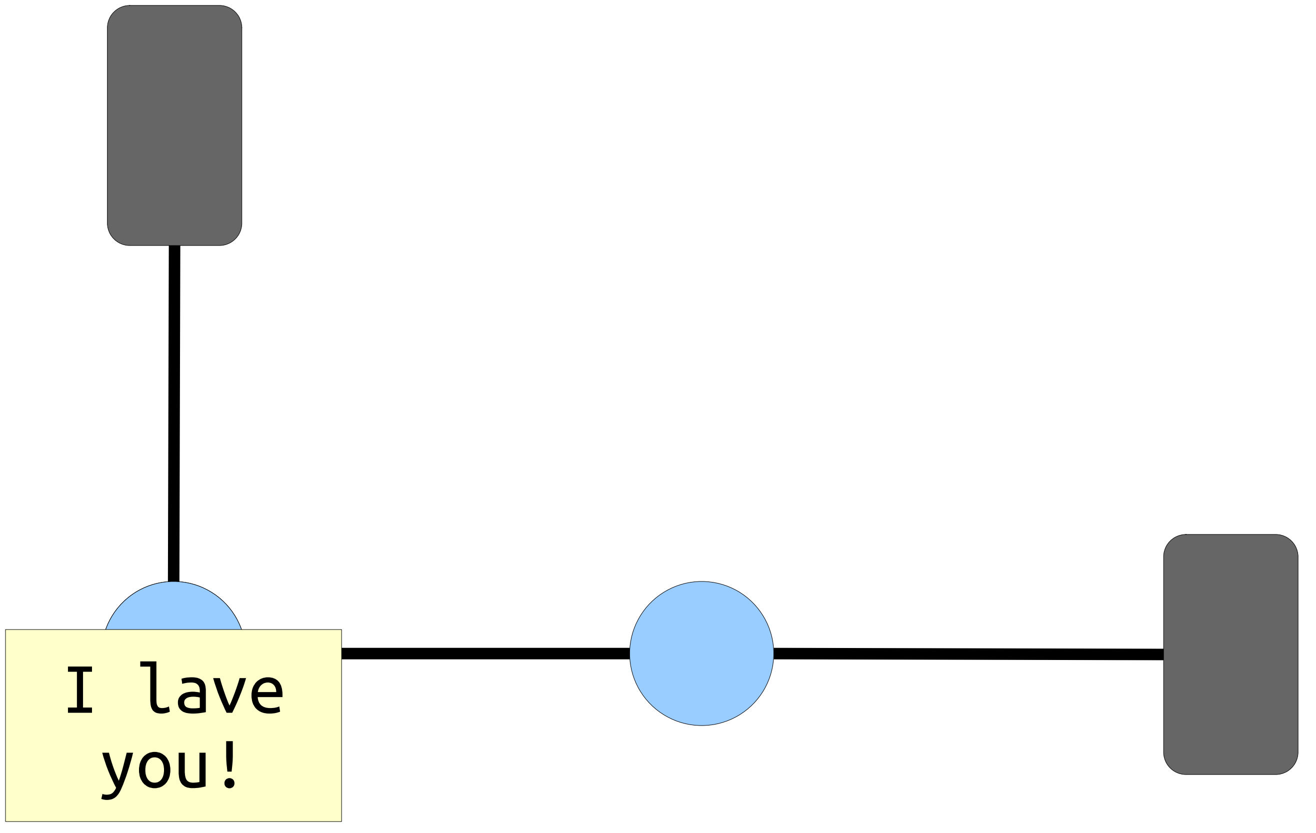
Outline for Today

- ***Hash Functions***
  - An amazingly versatile tool.
- ***Hash Tables***
  - Implementing a very fast `Map`.

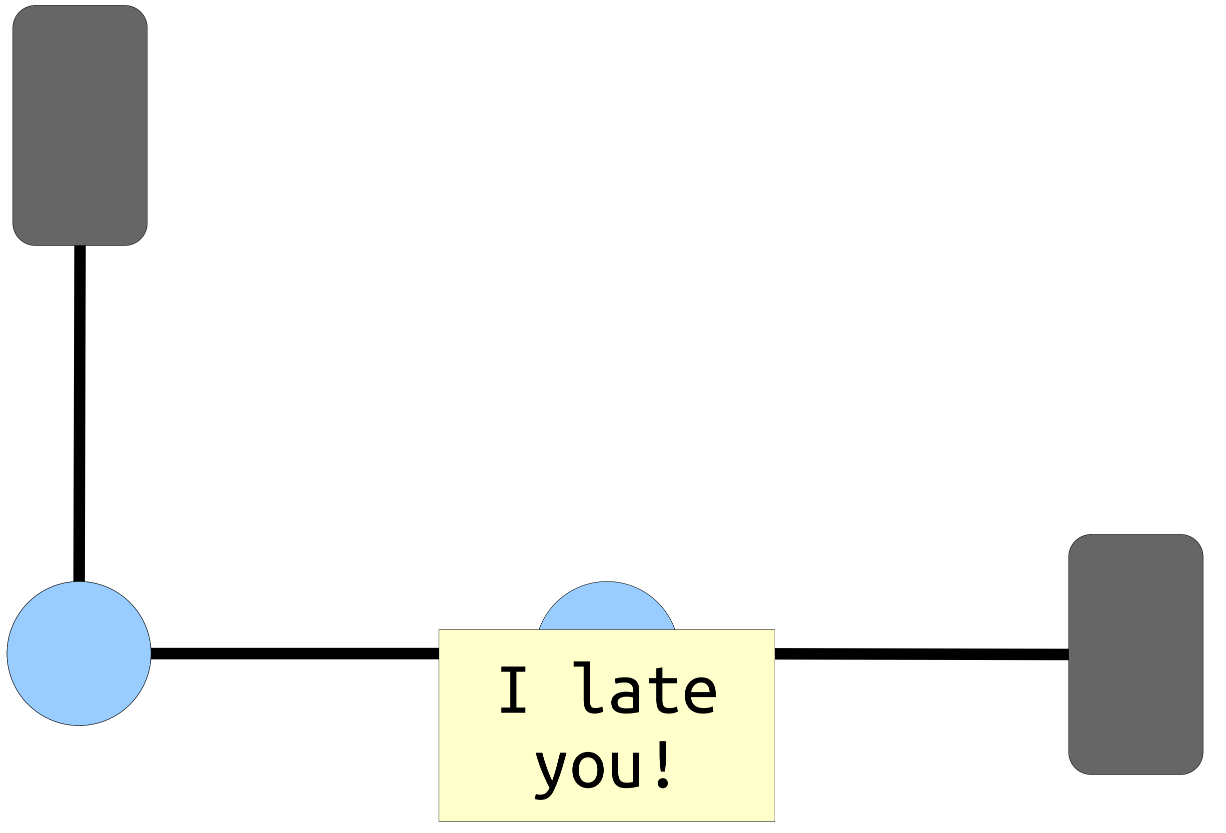
Two Motivating Problems



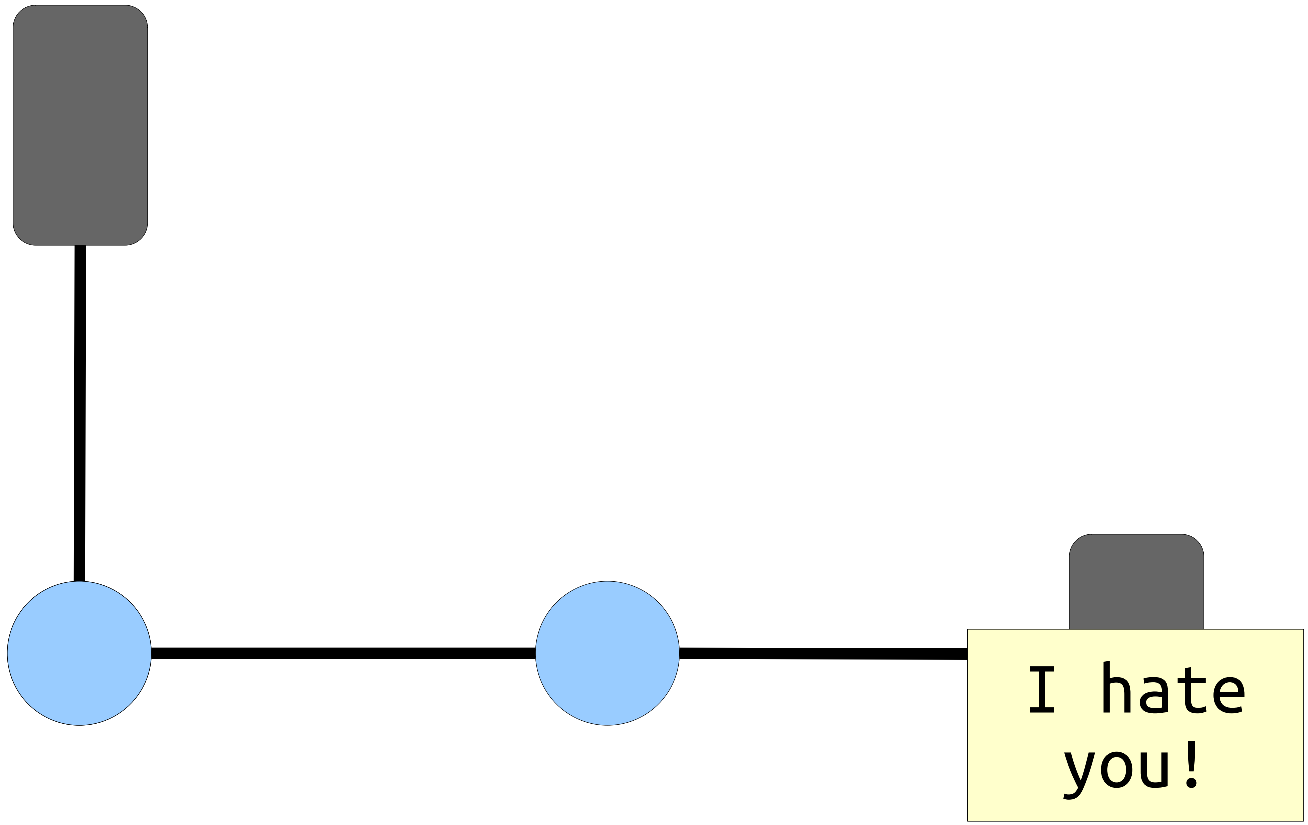
Did my data make it through the network?



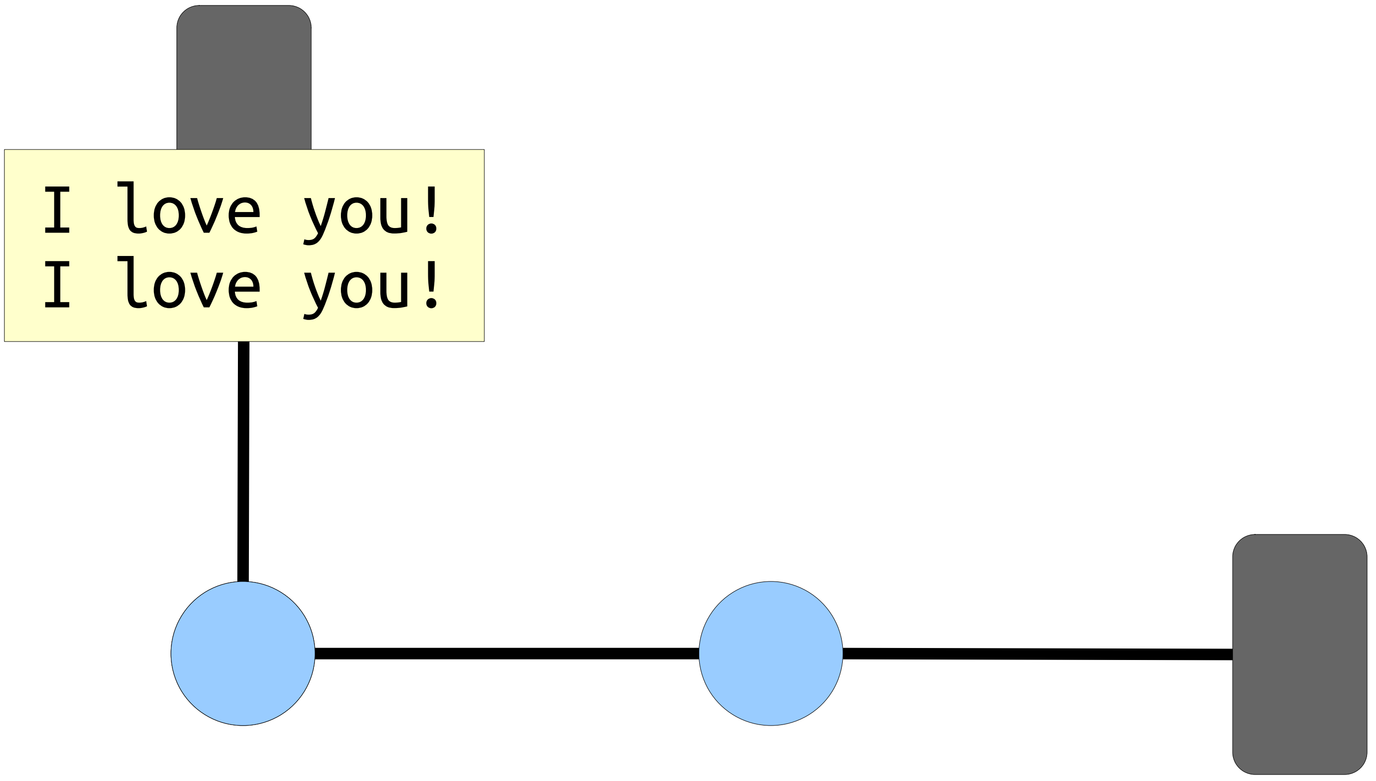
Did my data make it through the network?



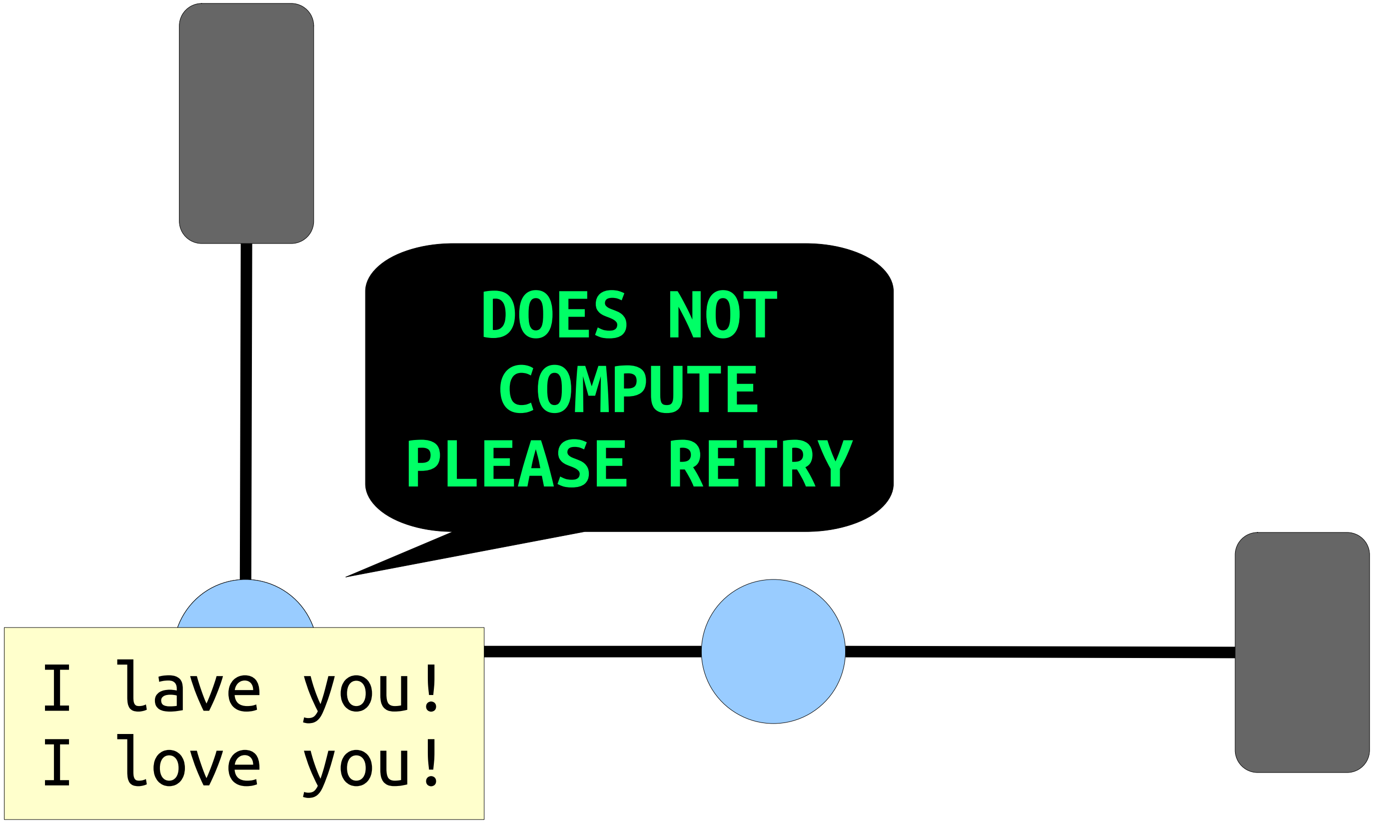
Did my data make it through the network?



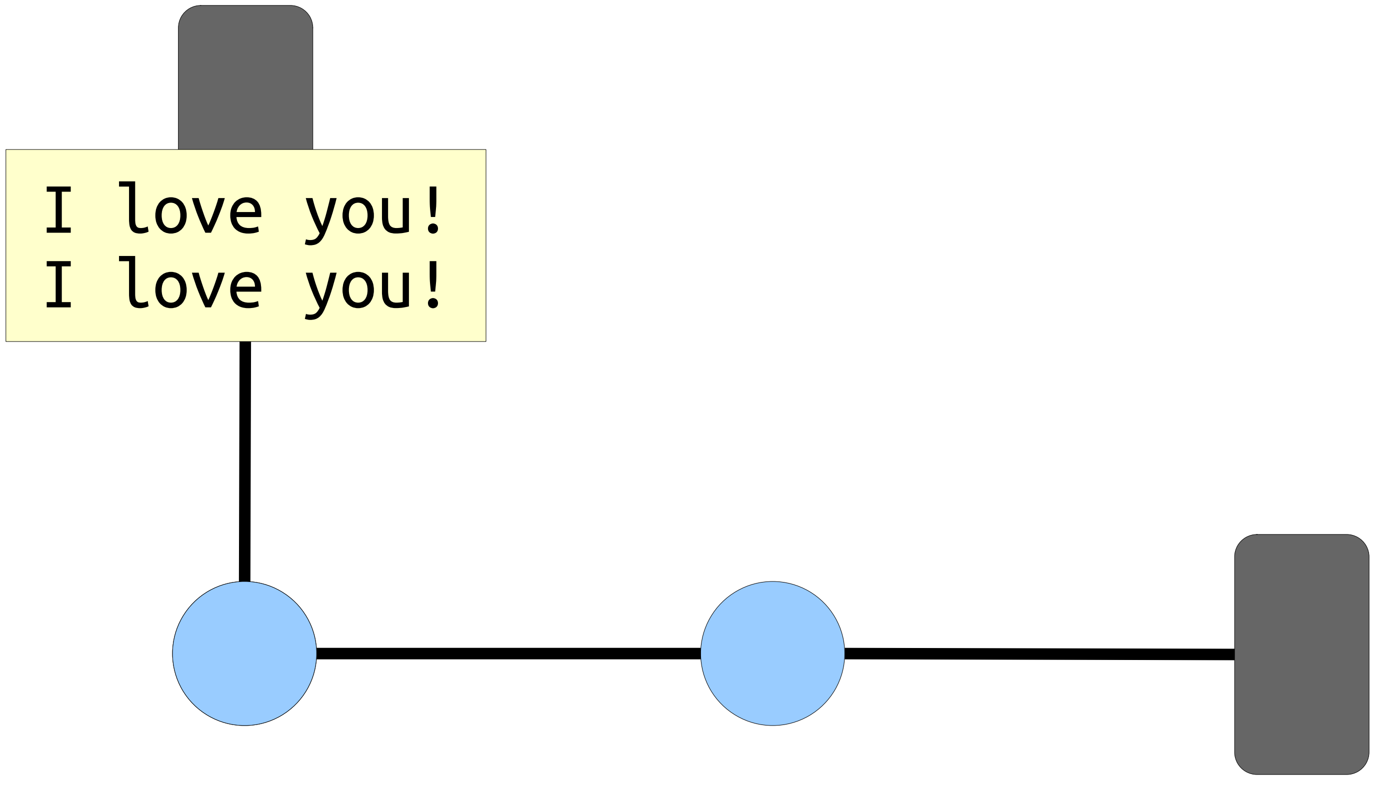
Did my data make it through the network?



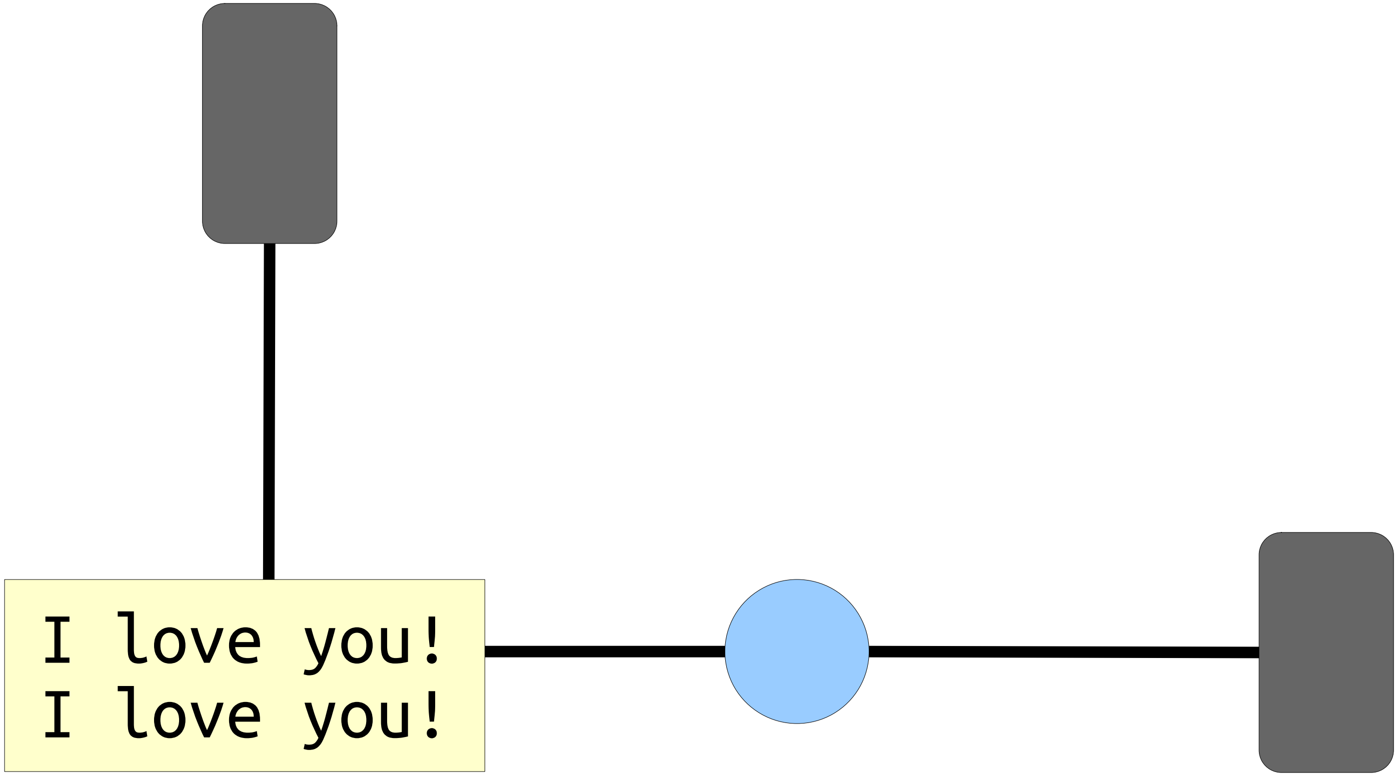
Did my data make it through the network?



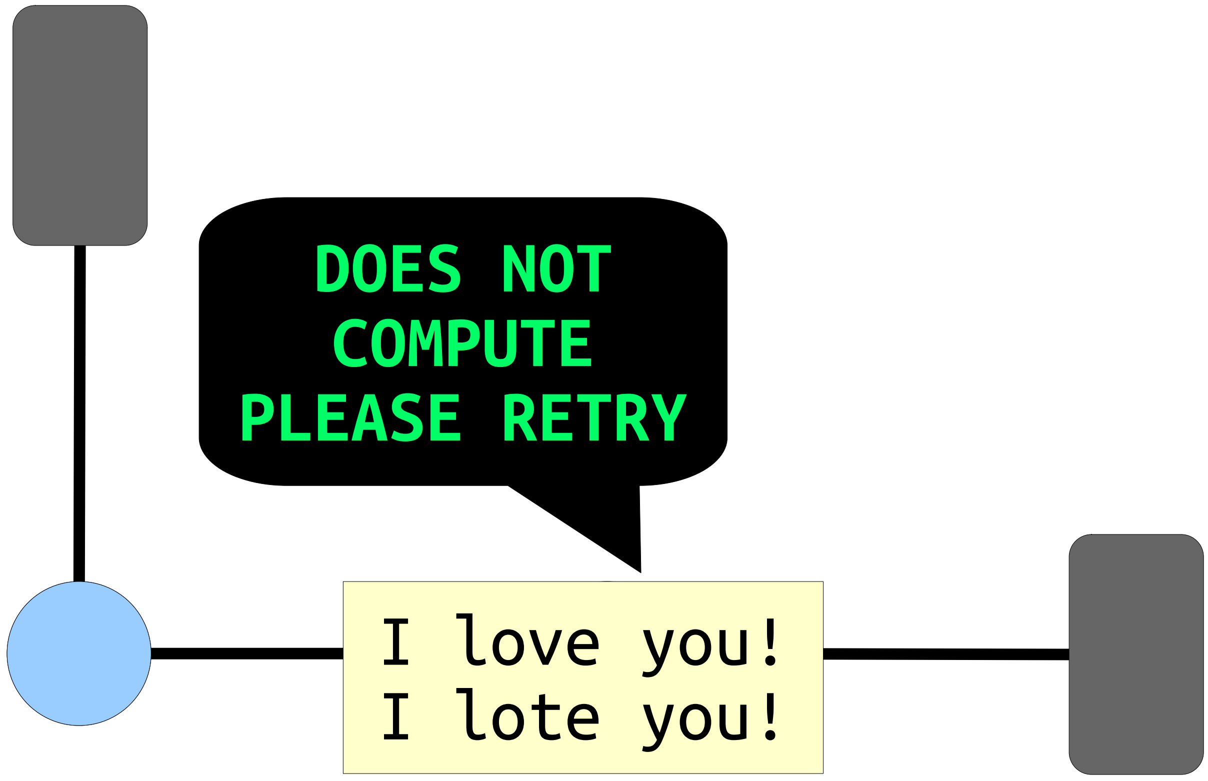
Did my data make it through the network?



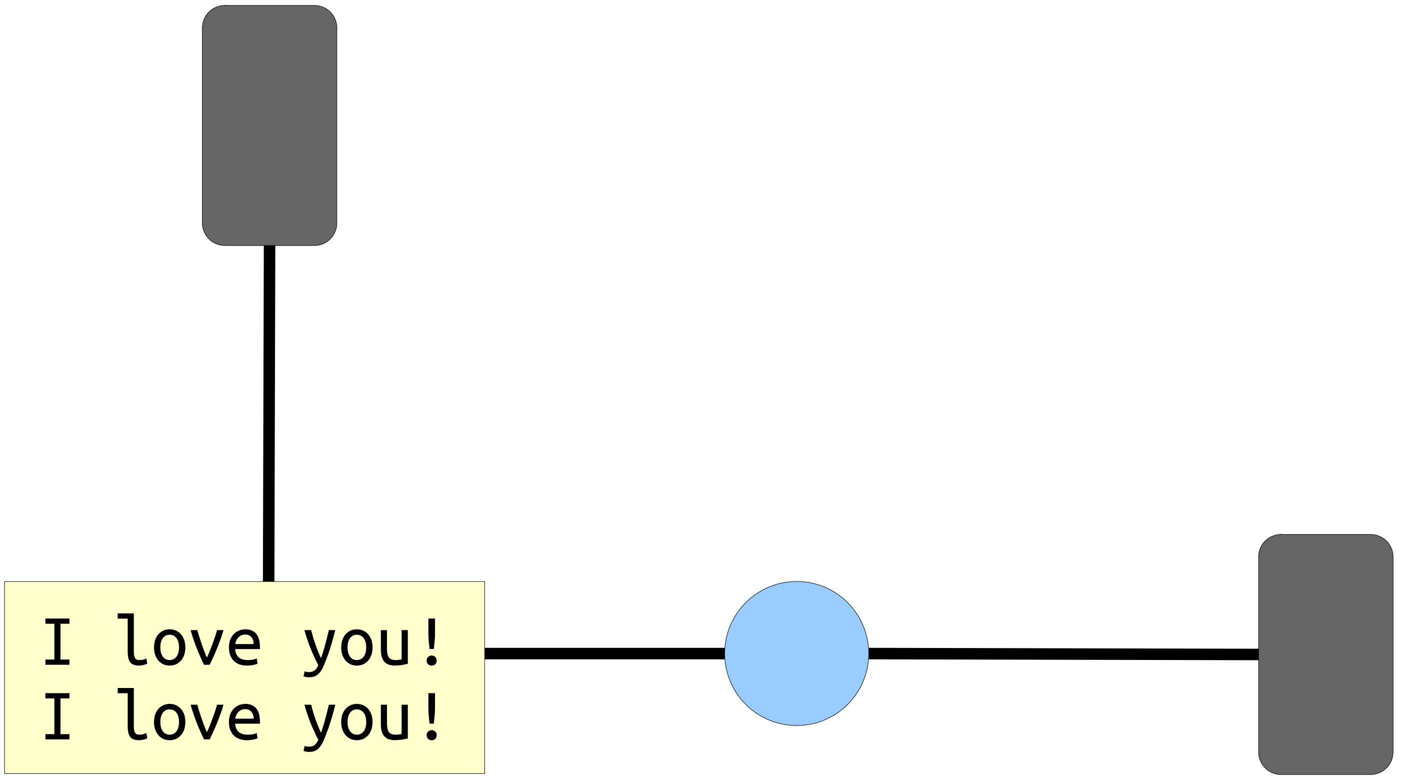
Did my data make it through the network?



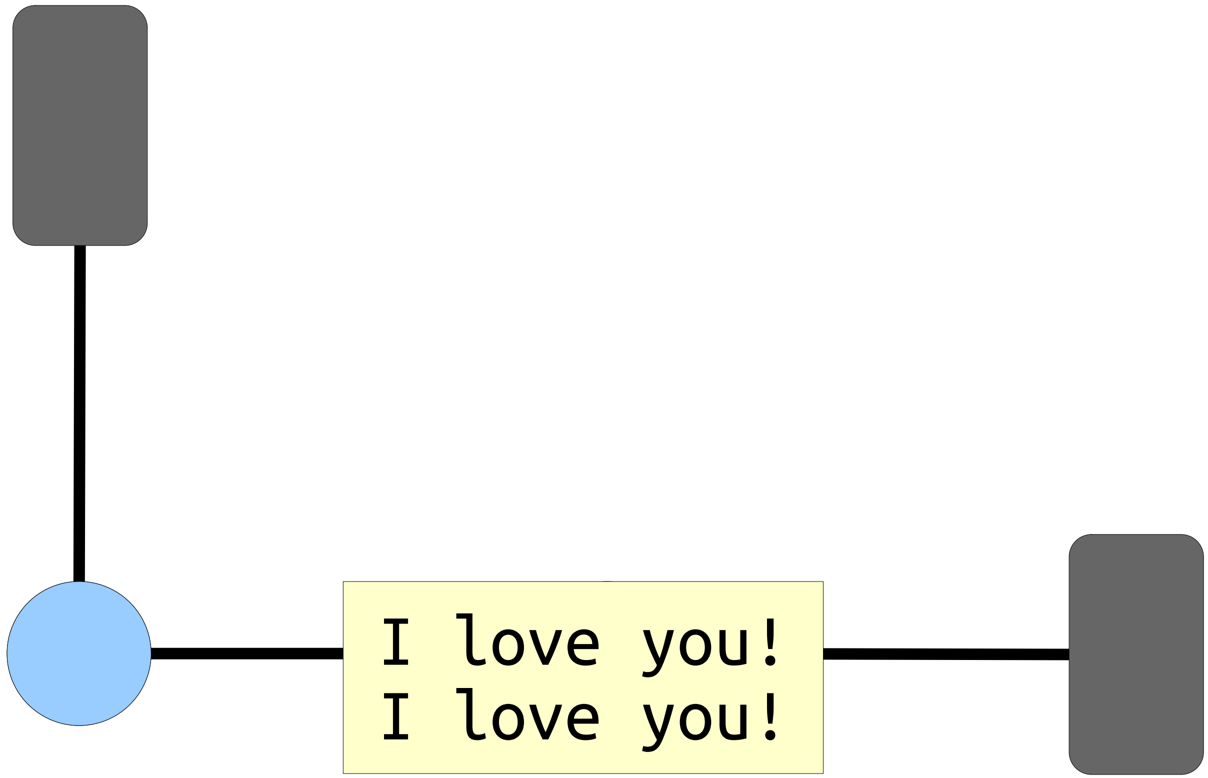
Did my data make it through the network?



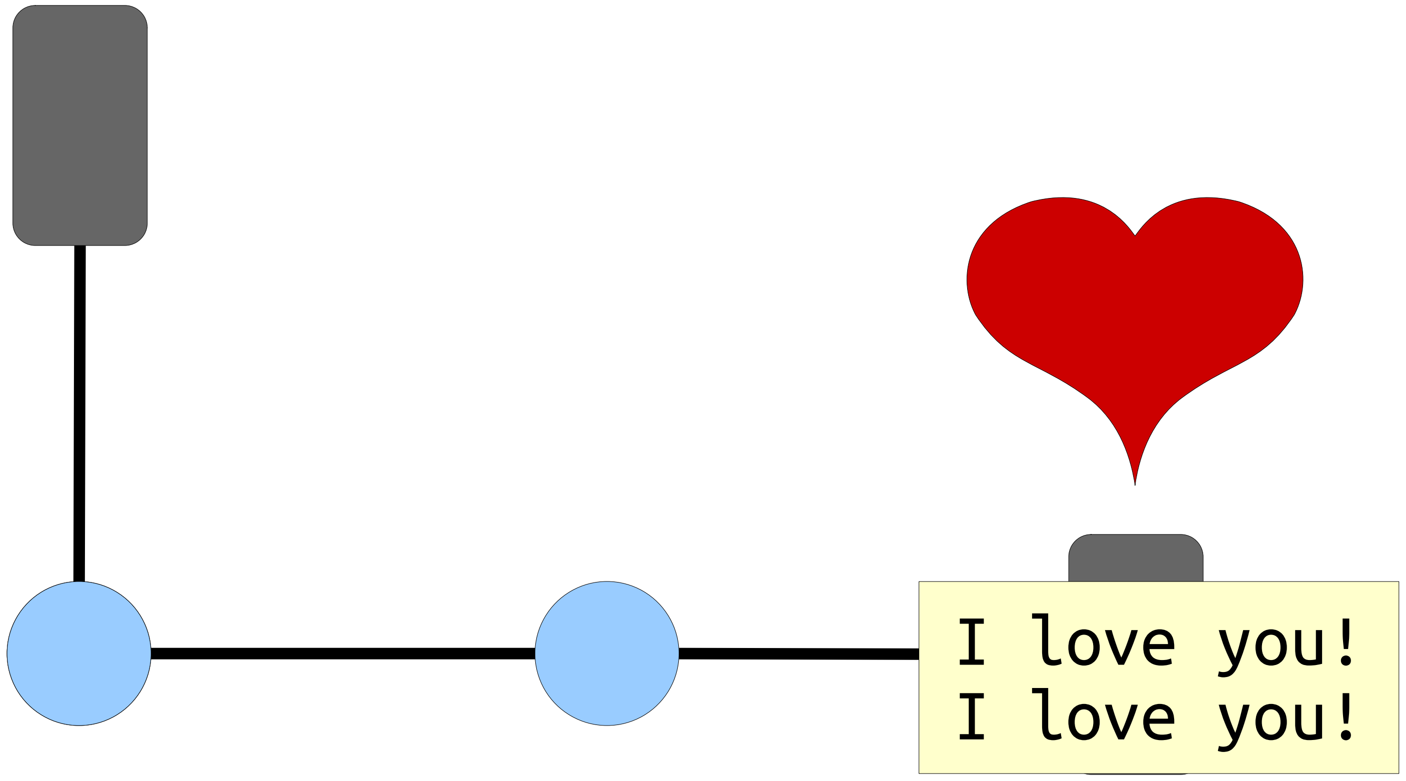
Did my data make it through the network?



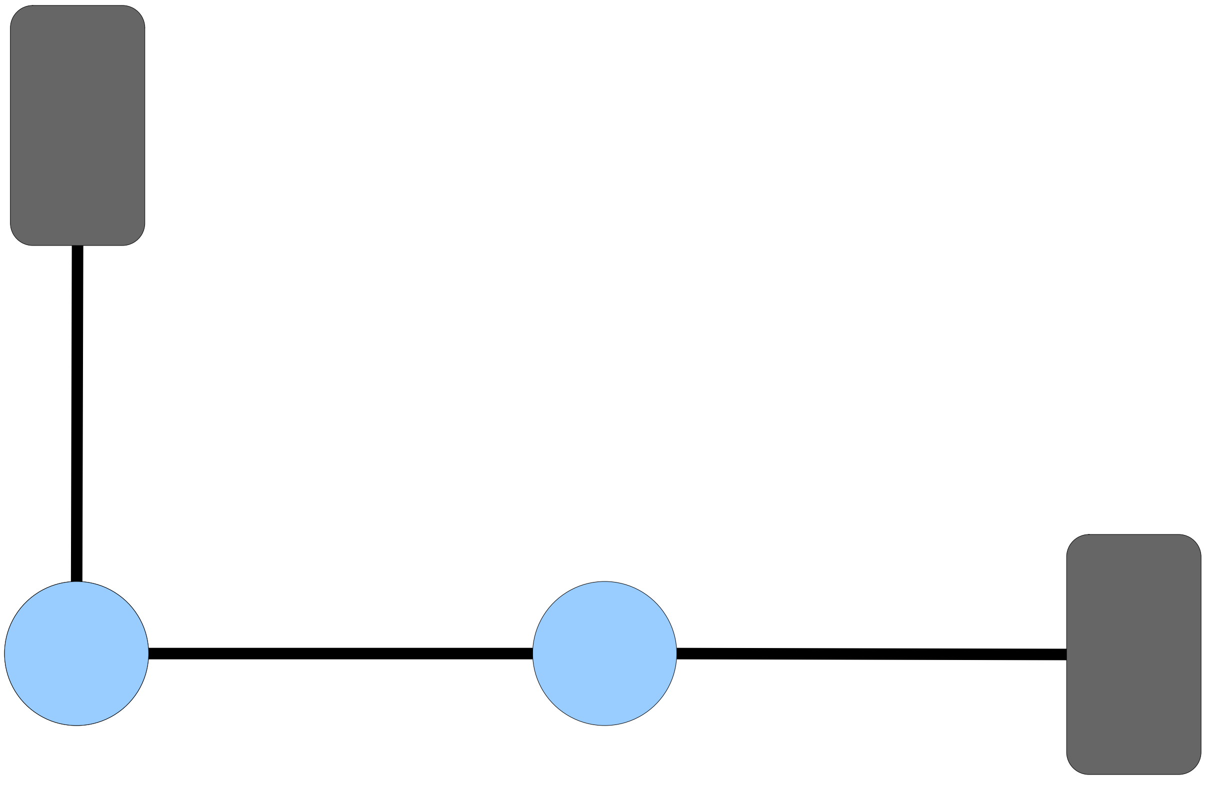
Did my data make it through the network?



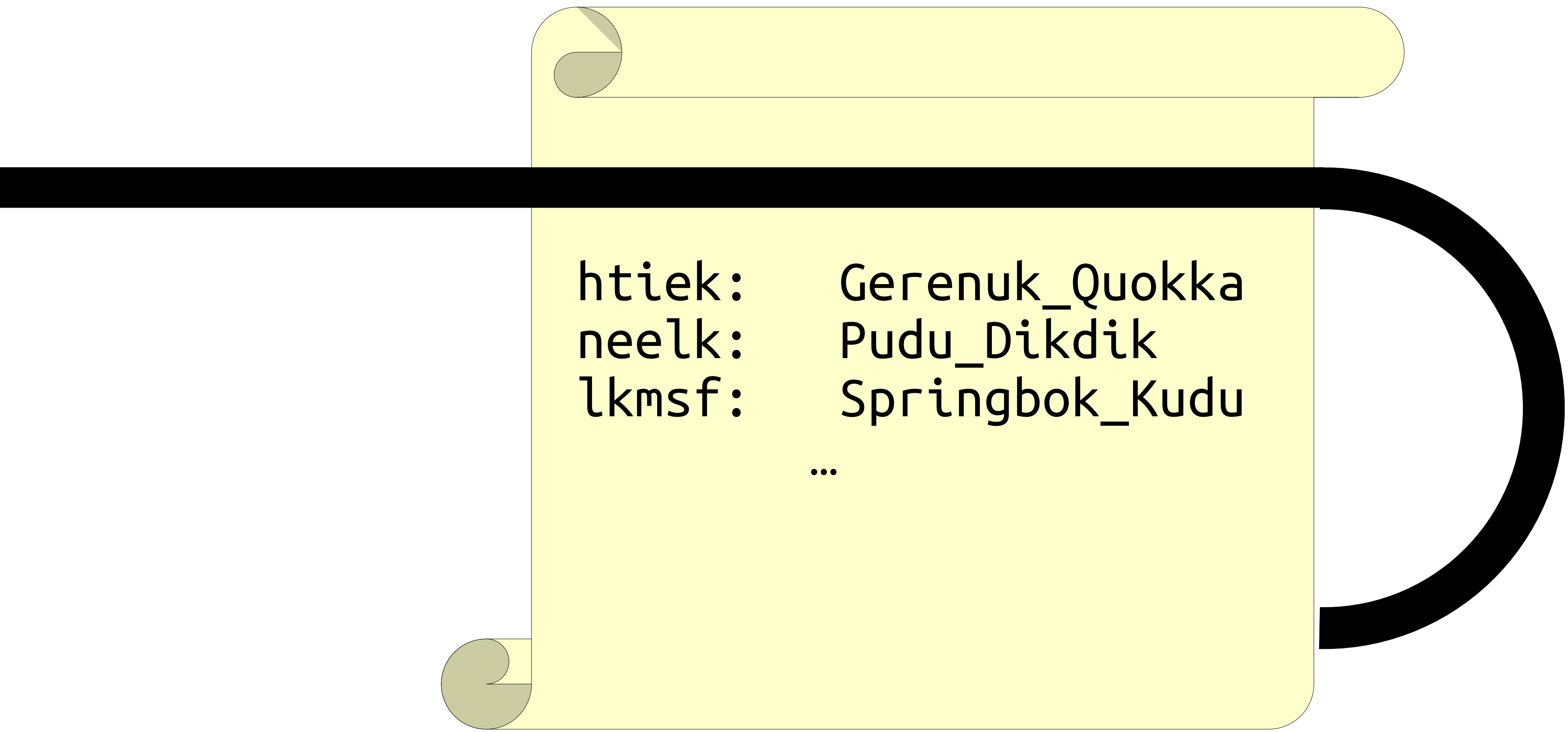
Did my data make it through the network?



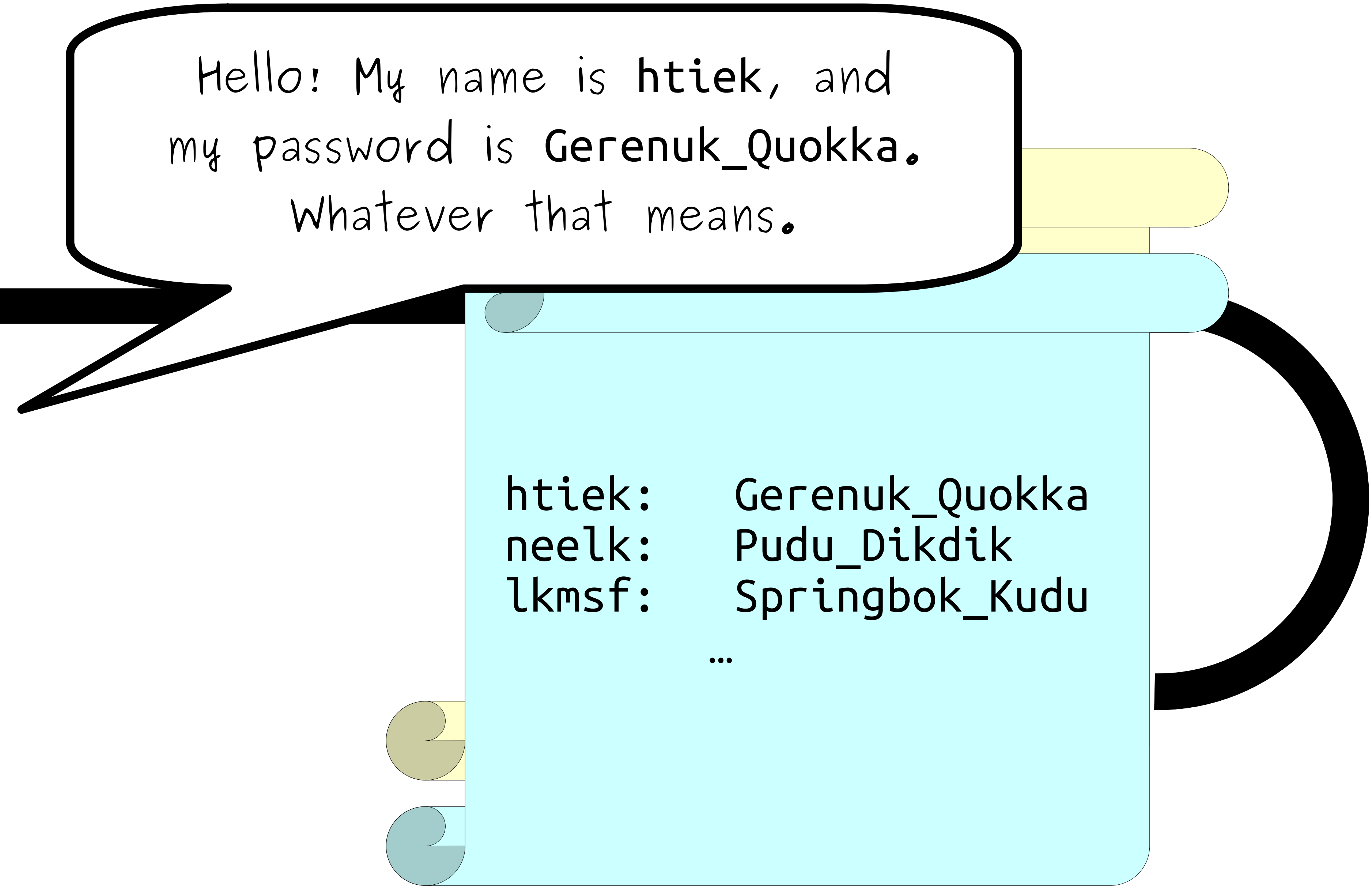
Did my data make it through the network?



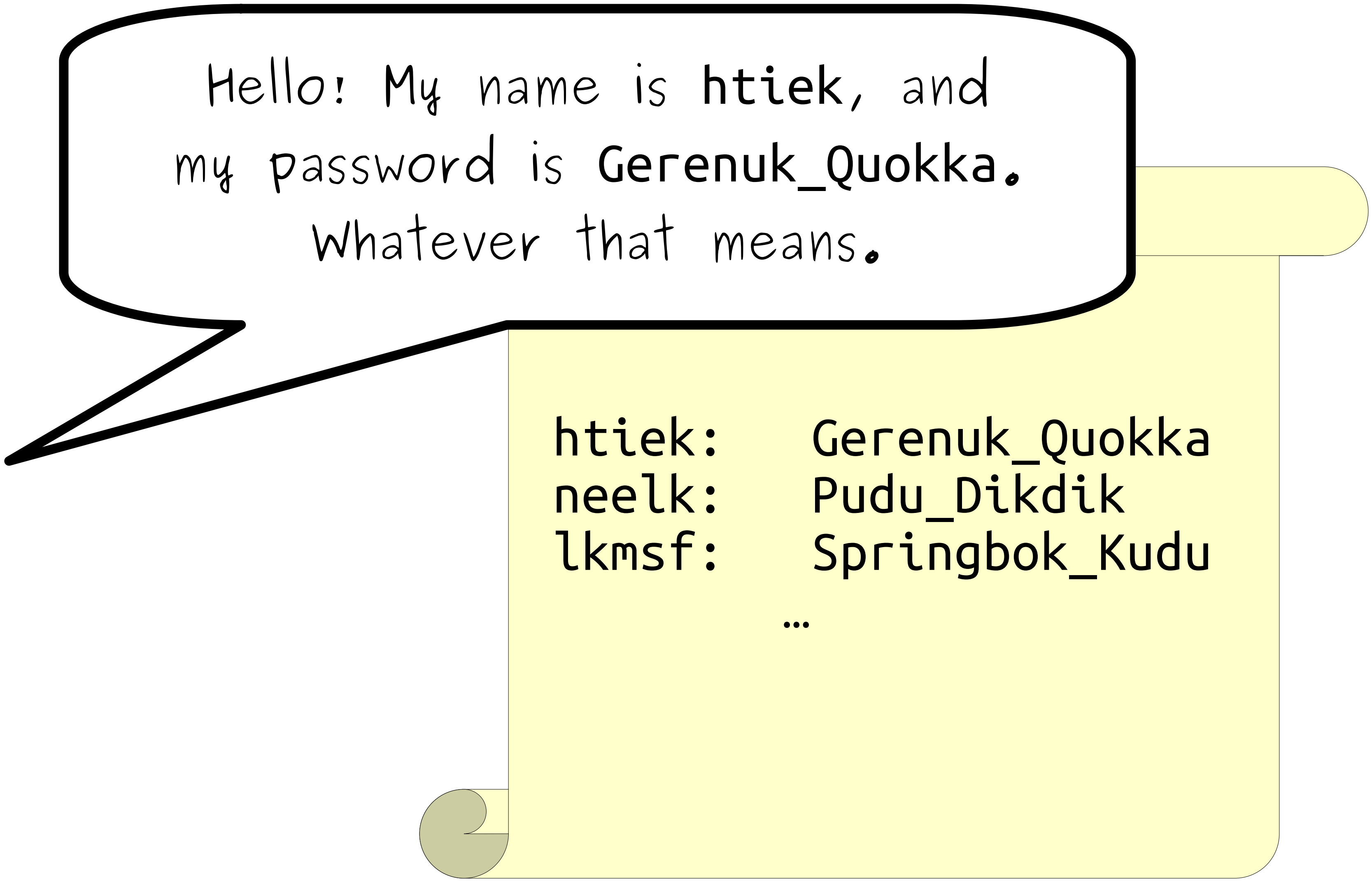
Can we do this without doubling the amount of data transmitted over the network?



How do servers store passwords?



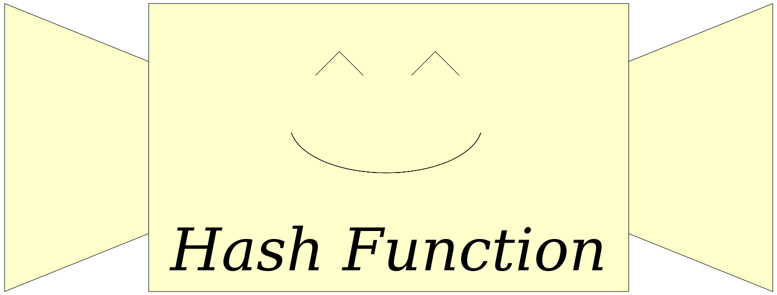
How do servers store passwords?



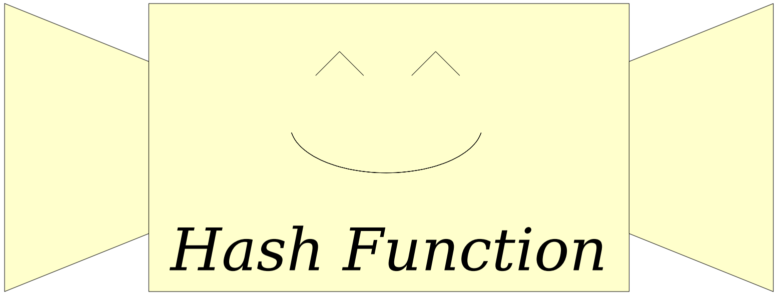
How can we store passwords safely
even if the password file is stolen?

Way Back When…

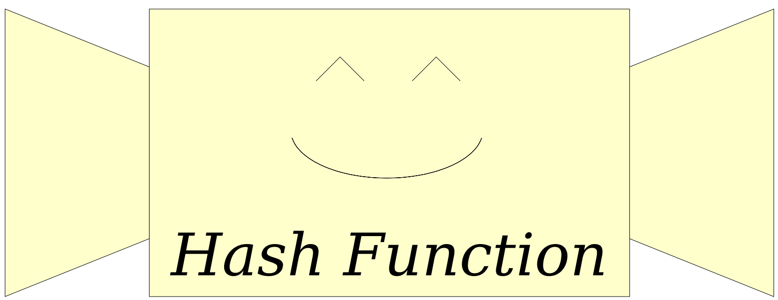
```cpp
int nameHash(string first, string last){
    /* This hashing scheme needs two prime numbers, a large prime and a small
     * prime. These numbers were chosen because their product is less than
     * 2^31 - kLargePrime - 1.
     */
    static const int kLargePrime = 16908799;
    static const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



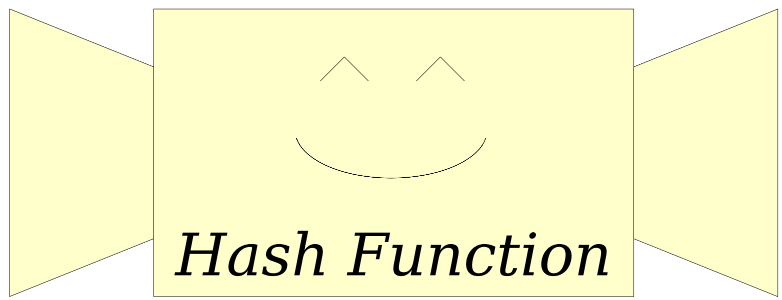
This is a ***hash function***. It's a type of function some smart math and CS people came up with.



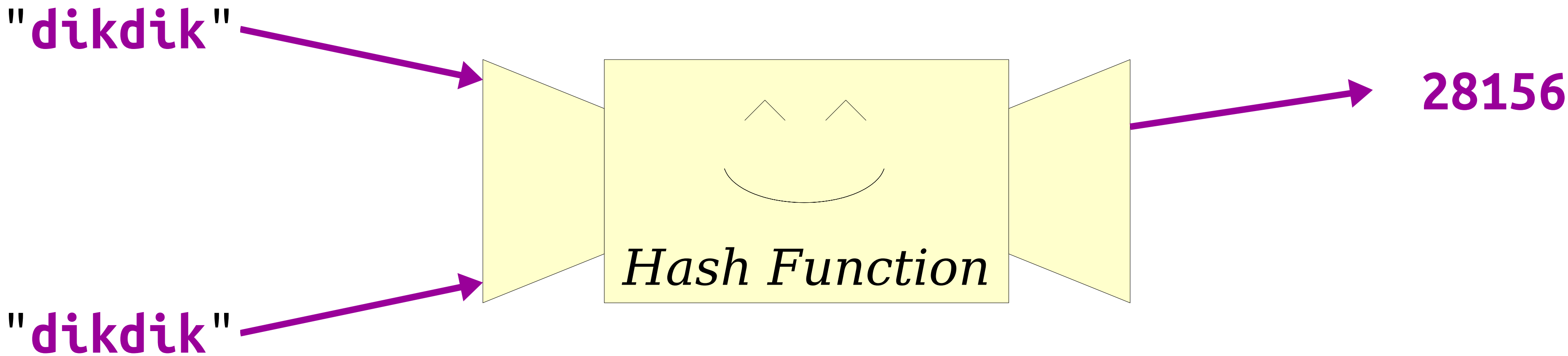
Most hash functions return a number.
In CS106B, we'll use the `int` type.



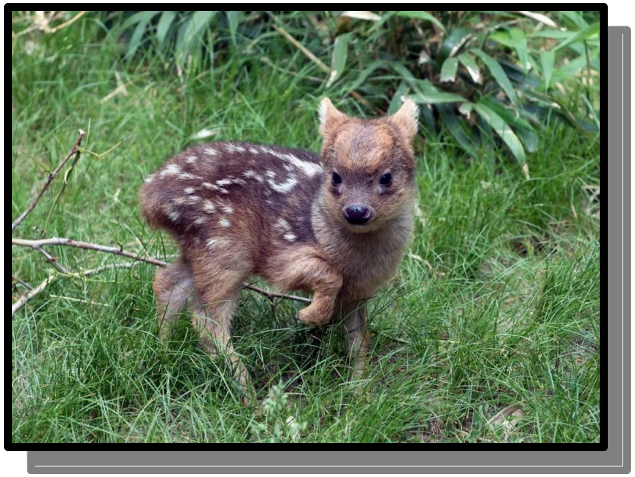
Different hash functions take inputs of different types.
In this example, we'll assume it takes `string` inputs.

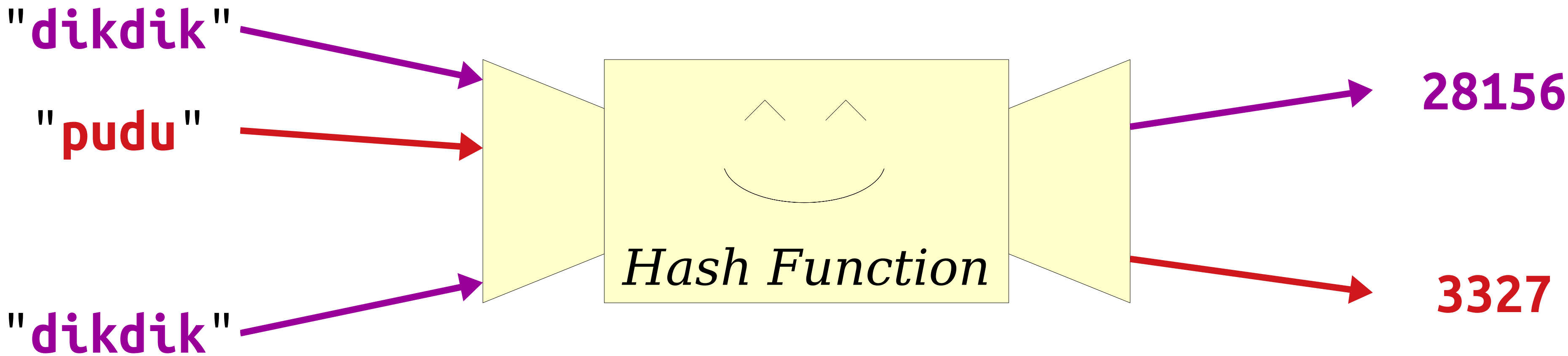


What makes this type of function so special?

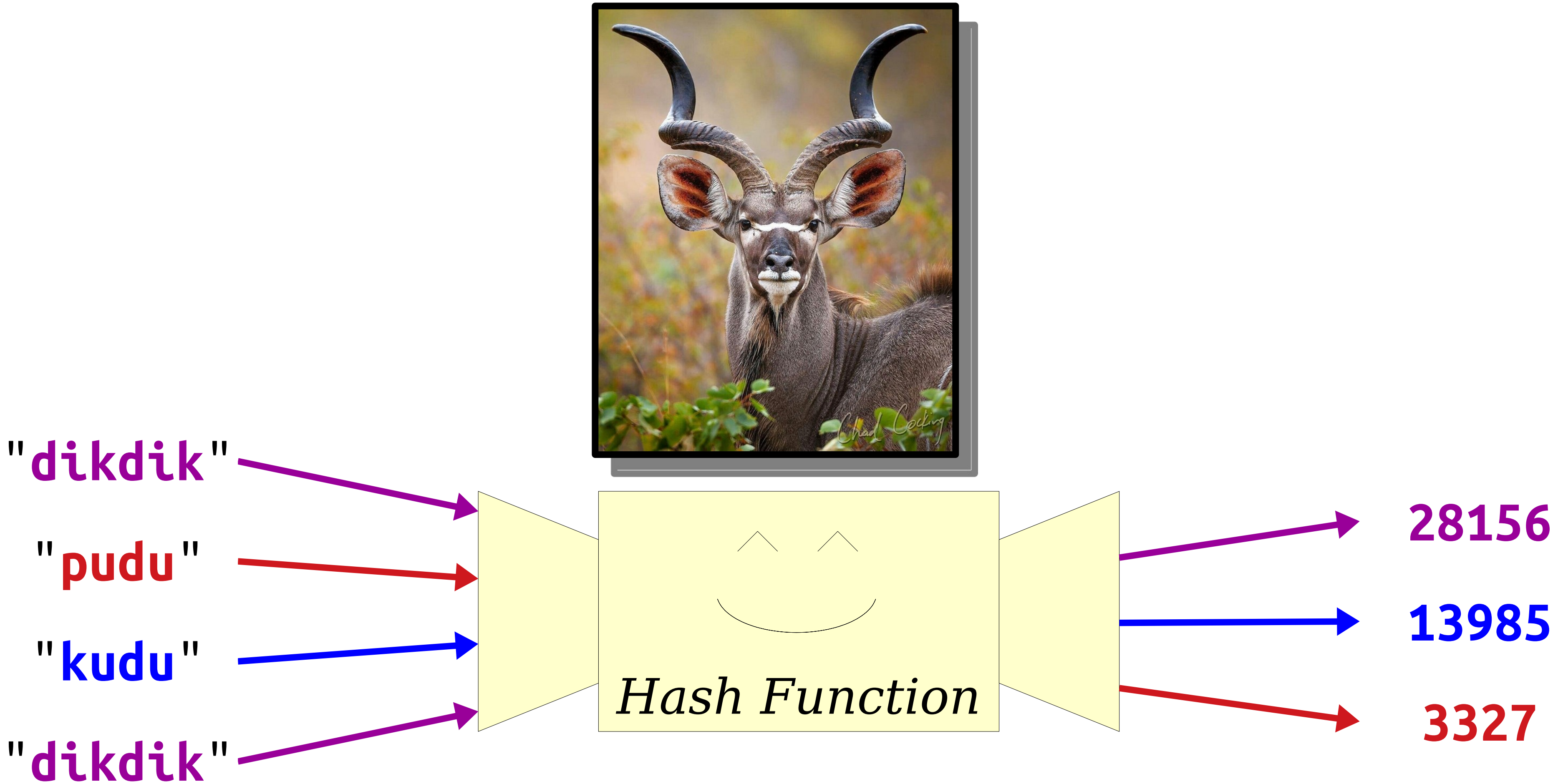


First, if you compute the hash code of the same string many times, you always get the same value.





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

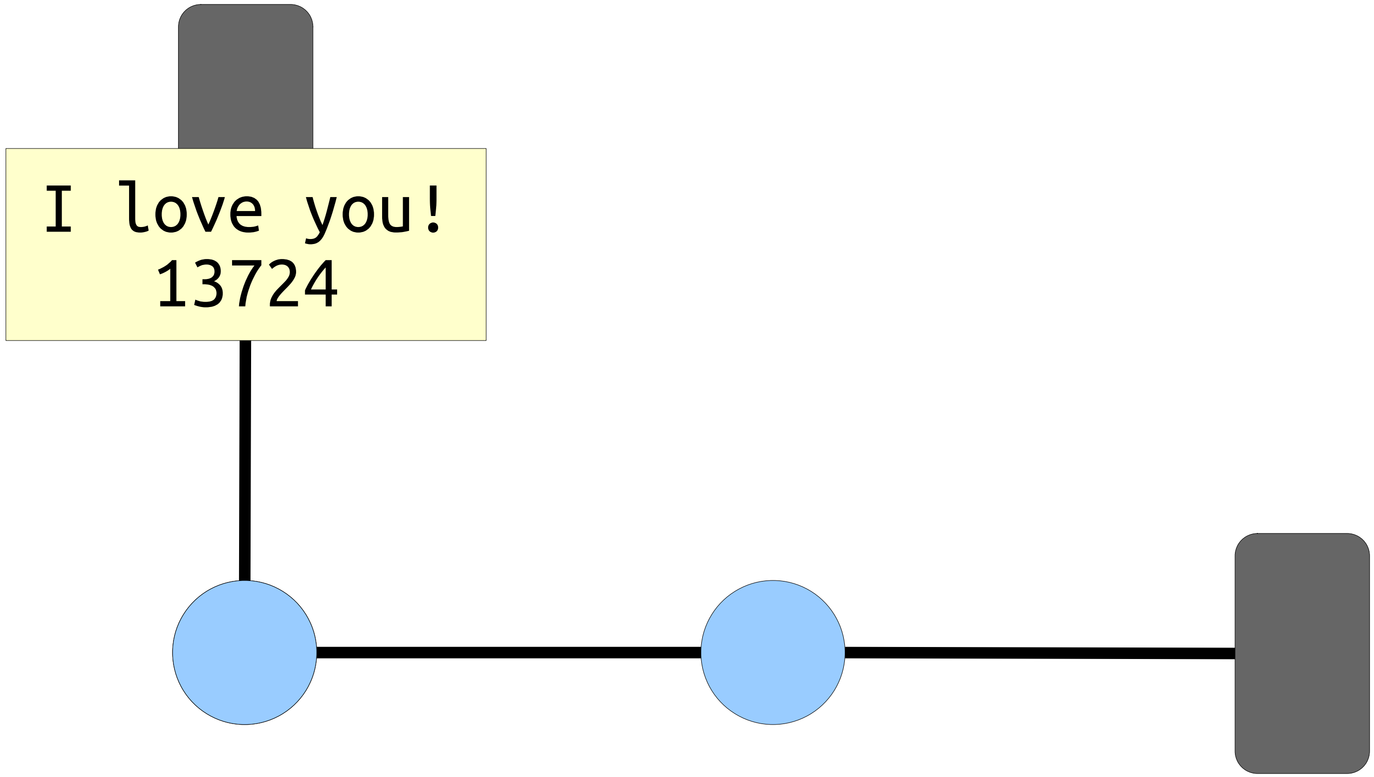


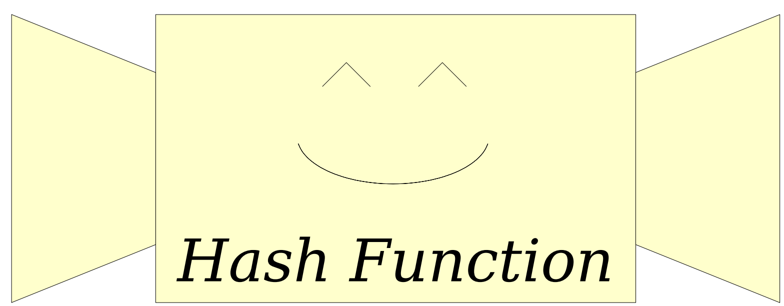
Even very similar inputs give
very different outputs!

To Recap:

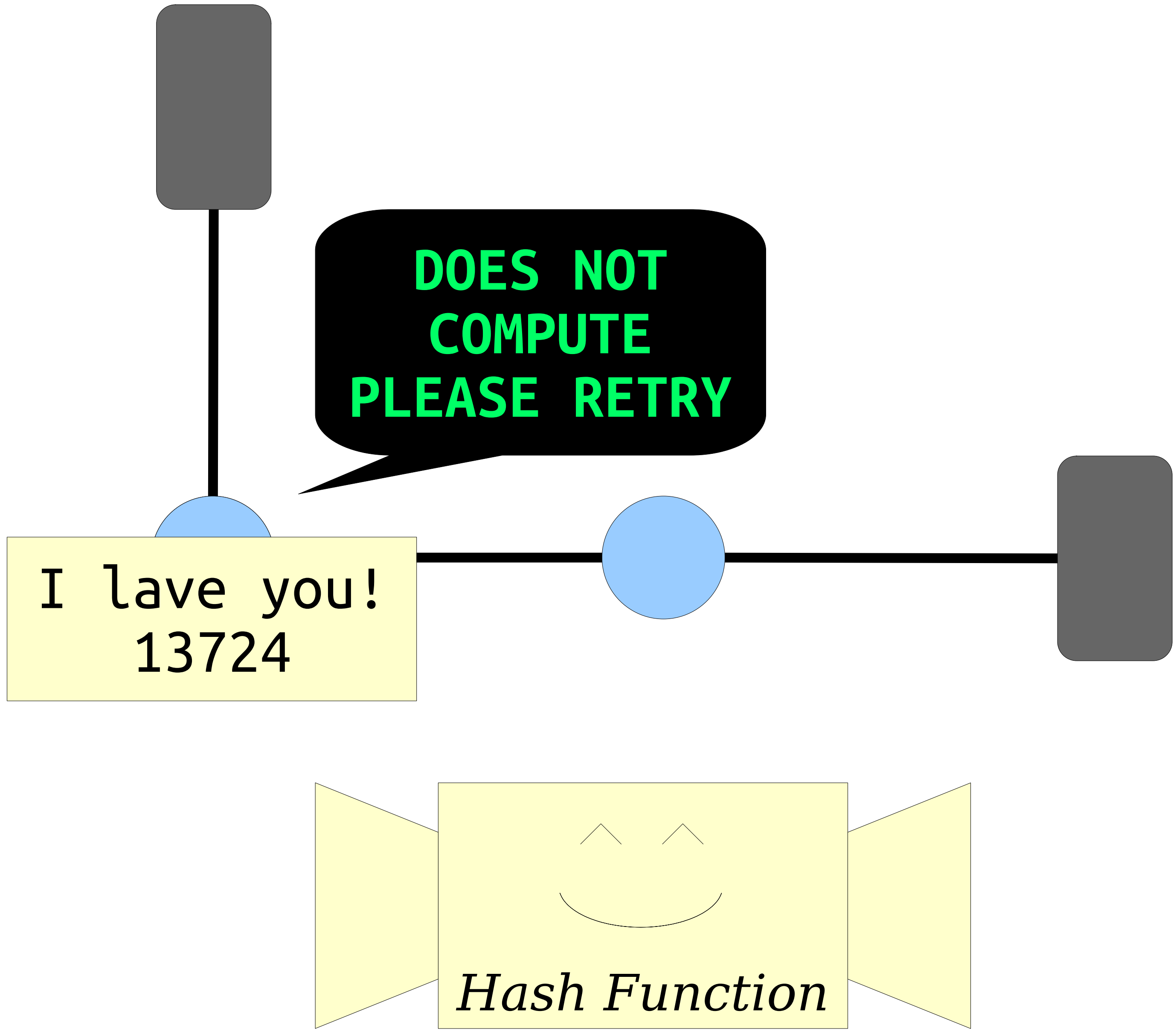
Equal inputs give equal outputs.

Unequal inputs (usually) give very different outputs.

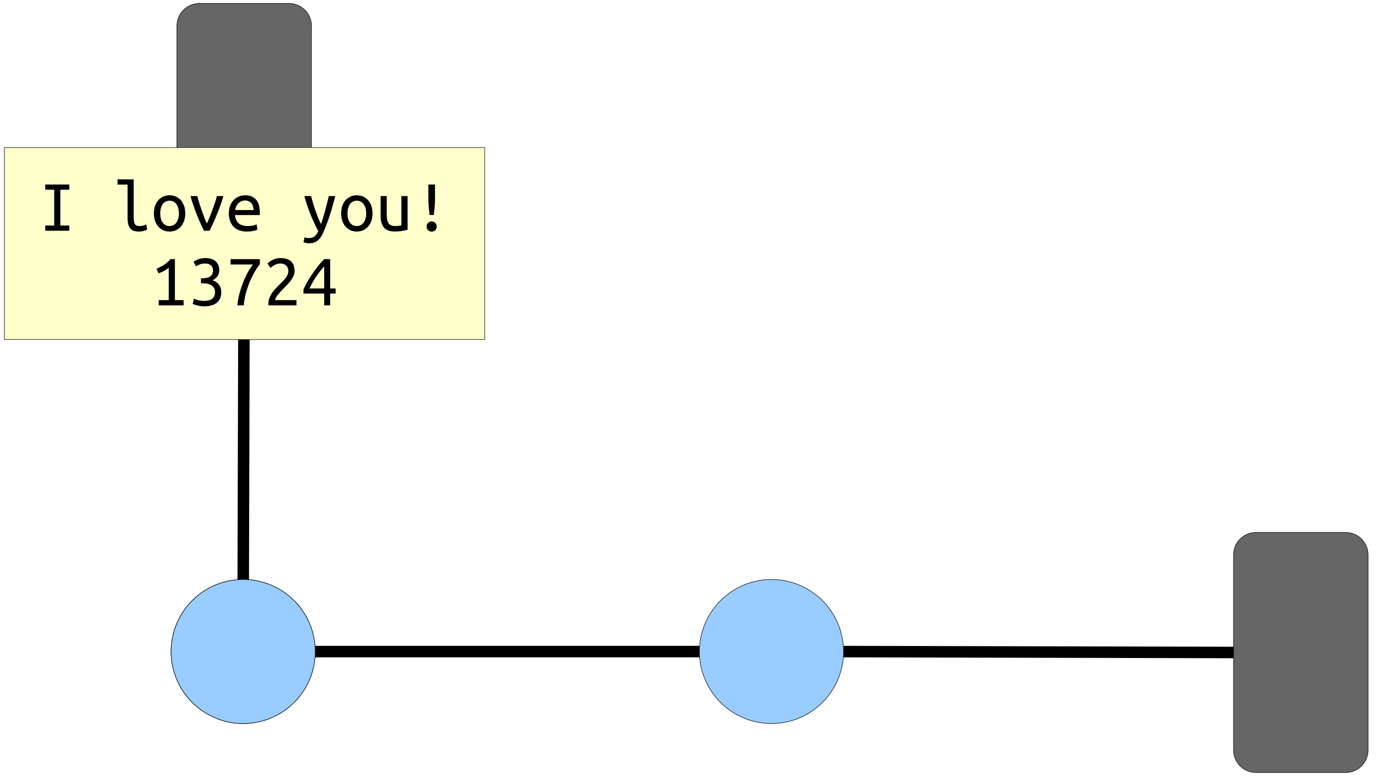


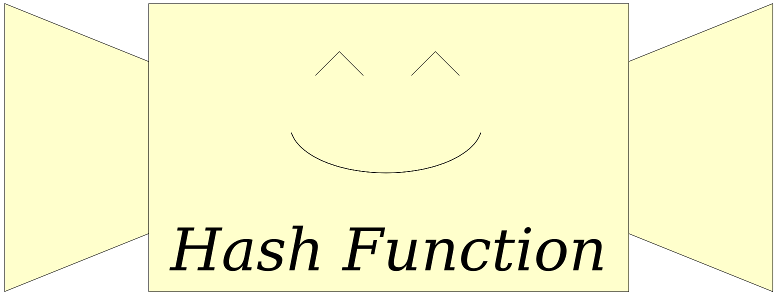


Did my data make it through the network?

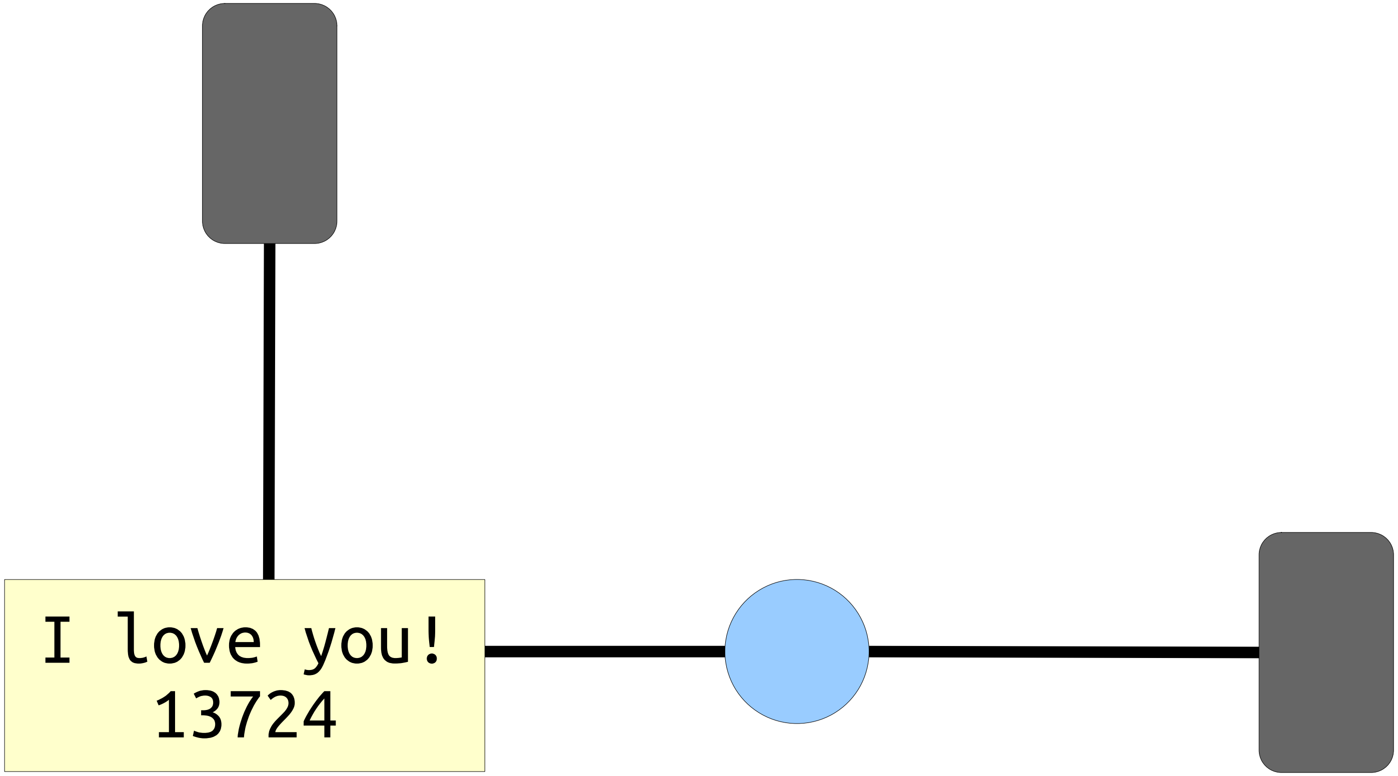


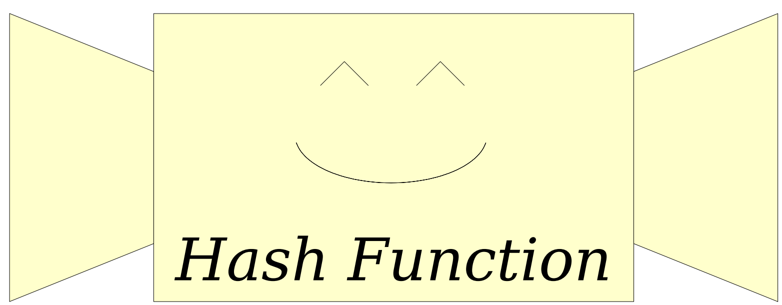
Did my data make it through the network?



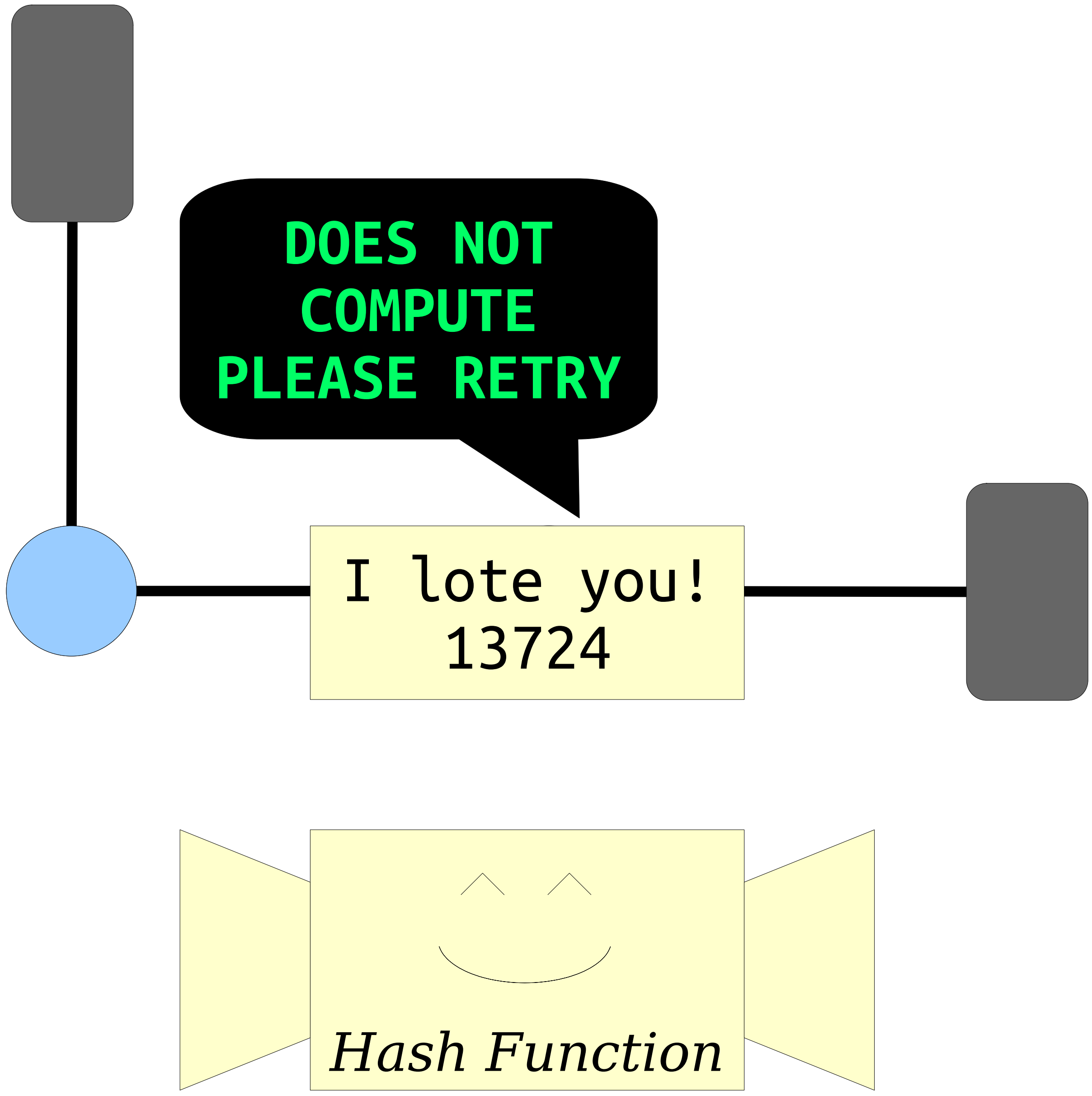


Did my data make it through the network?

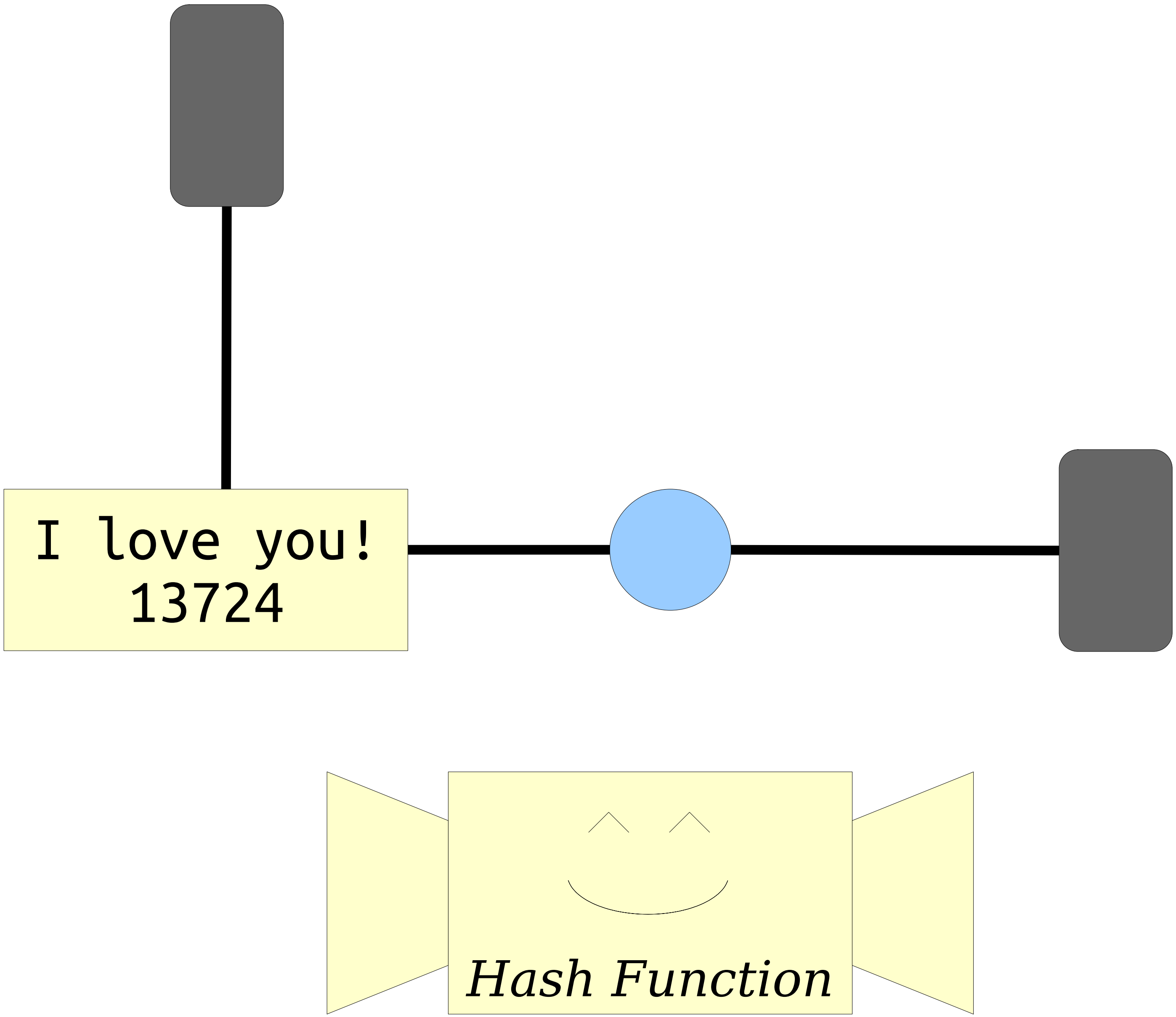




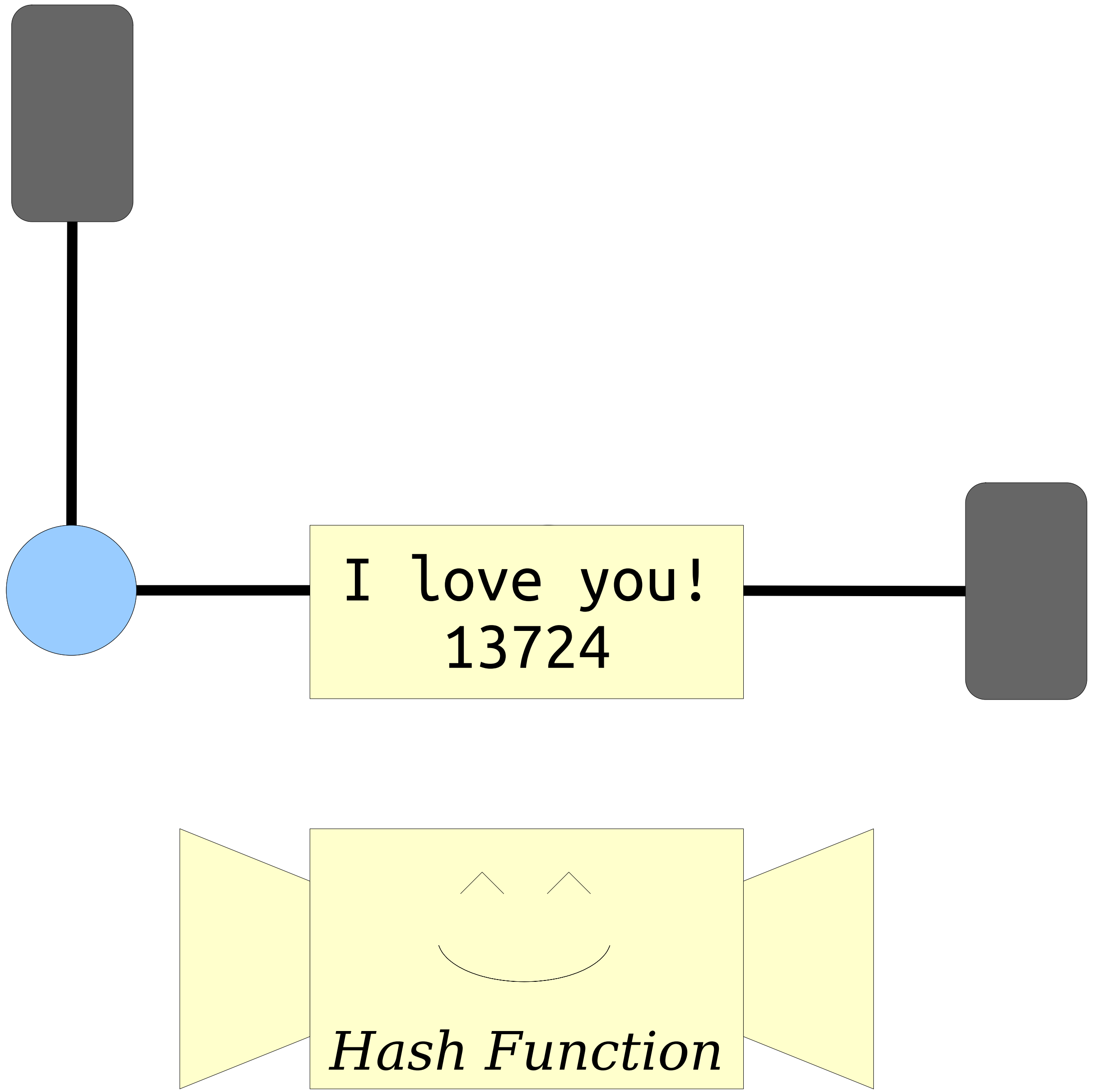
Did my data make it through the network?



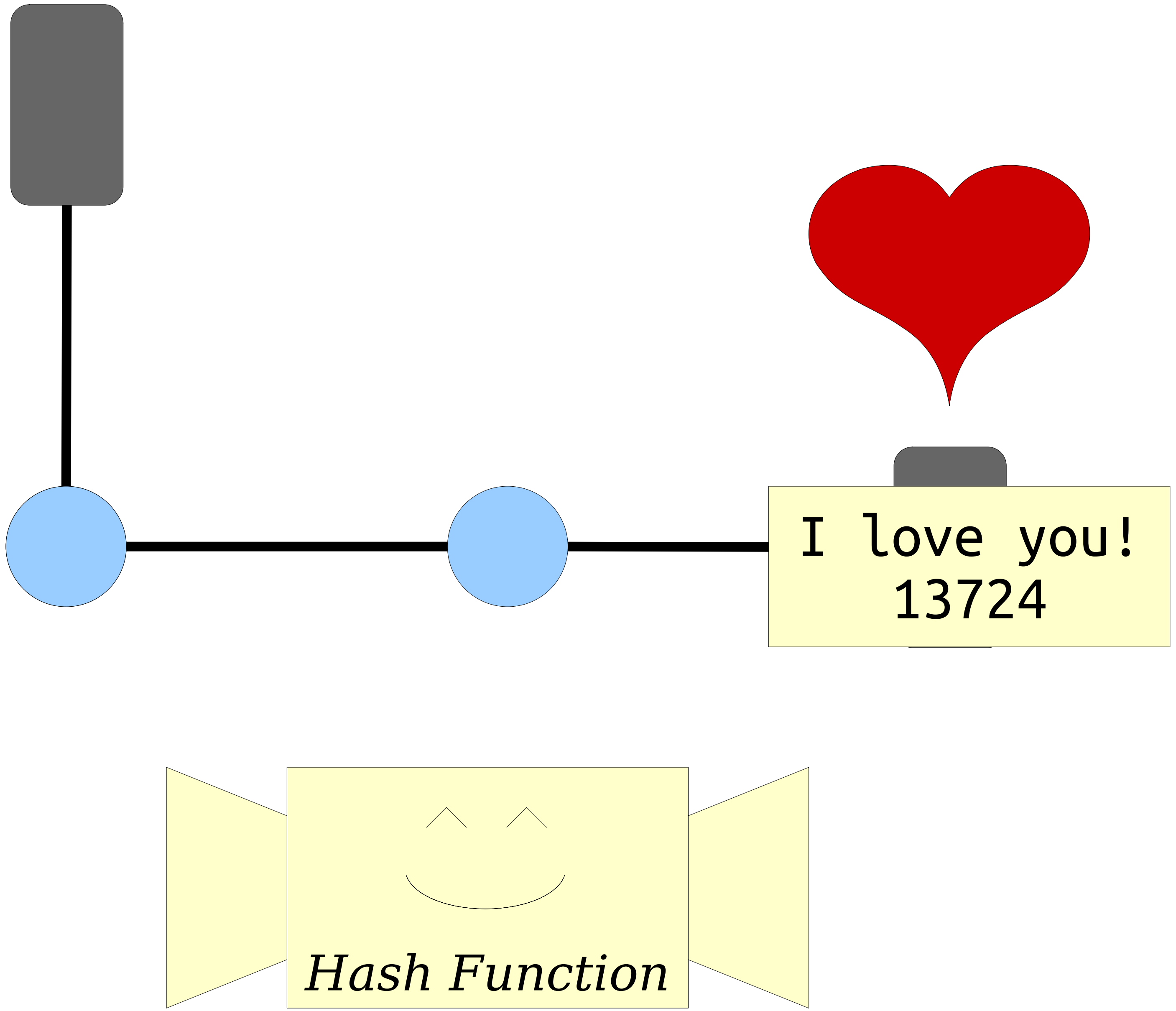
Did my data make it through the network?



Did my data make it through the network?



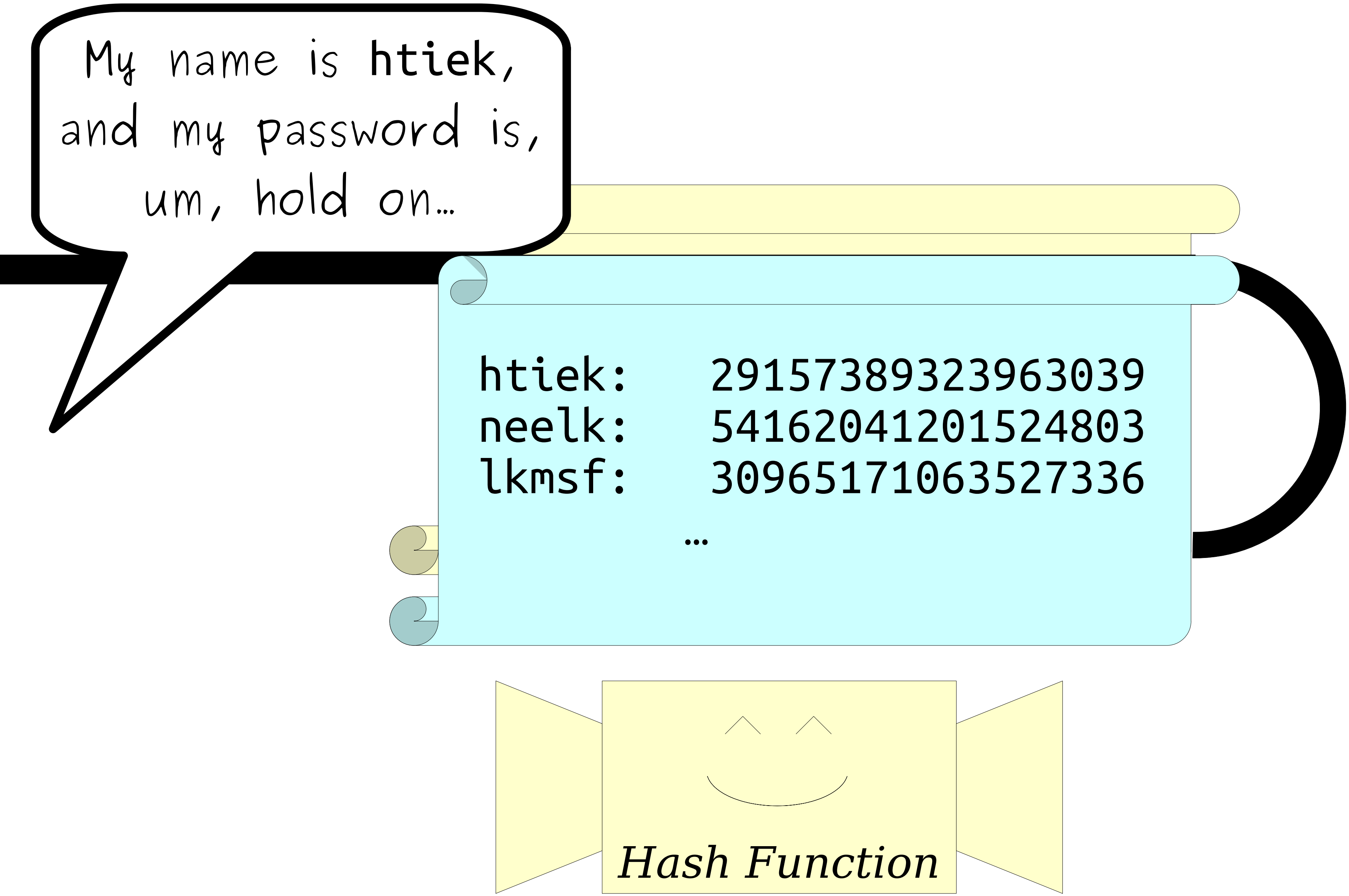
Did my data make it through the network?



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!



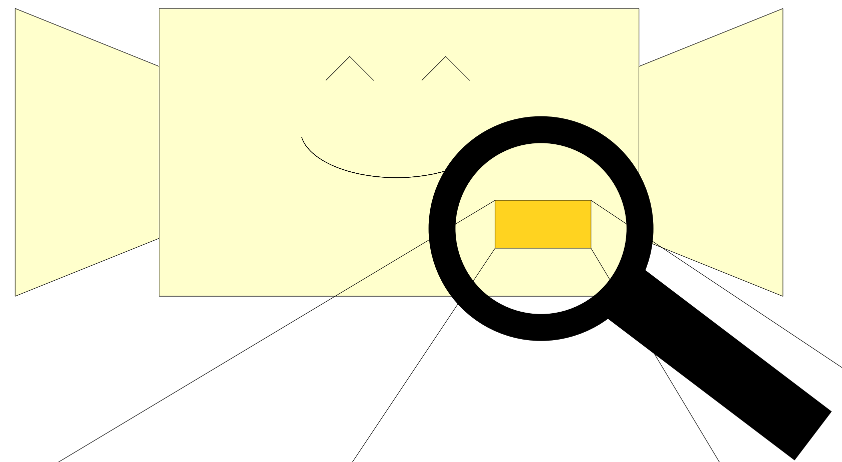
How do servers store passwords?

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!

Designing Hash Functions

- Designing good hash functions is challenging, and it's beyond the scope of what we'll explore in CS106B.
- Interested in things like independent random variables, finite fields, and the like? Come talk to me after class and I'll give the rundown.



$$\Pr_{h \in \mathscr{H}} [h(x) = s \wedge h(y) = t] = \frac{1}{m^2}$$

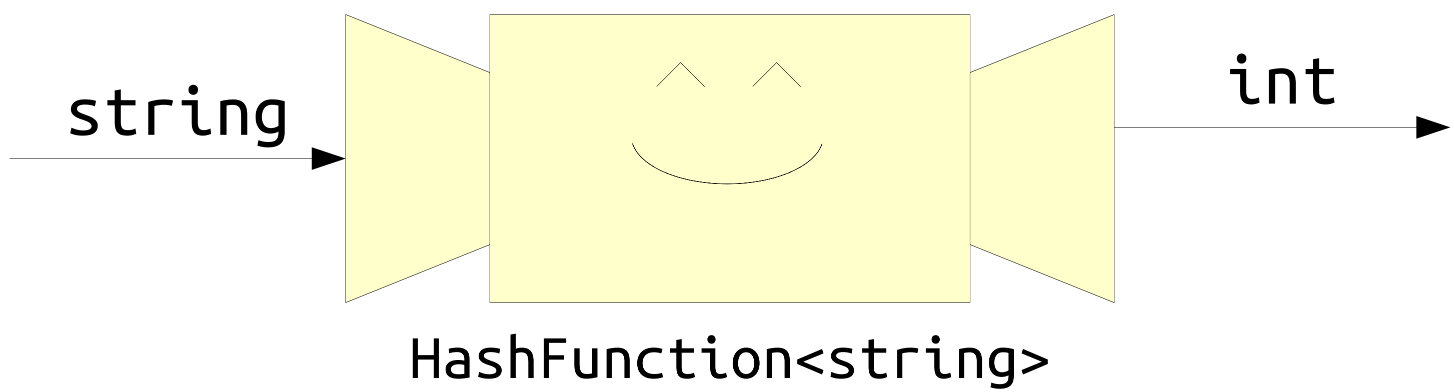
$$h(x_2 x_1 x_0) = T_0[x_0] \oplus T_1[x_1] \oplus T_2[x_2]$$

$$h(x) = \sum_{i=0}^{2} a_i x^i$$

Working with Hash Functions

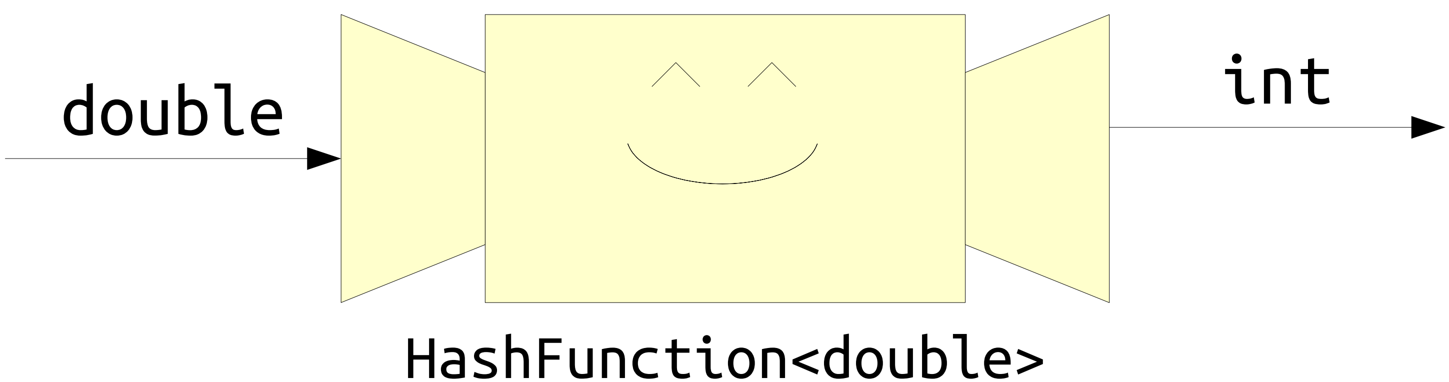
Working with Hash Functions

- Every programming language has a different way for programmers to work with hash functions.
- In CS106B, we'll represent hash functions using the type `HashFunction<T>`.



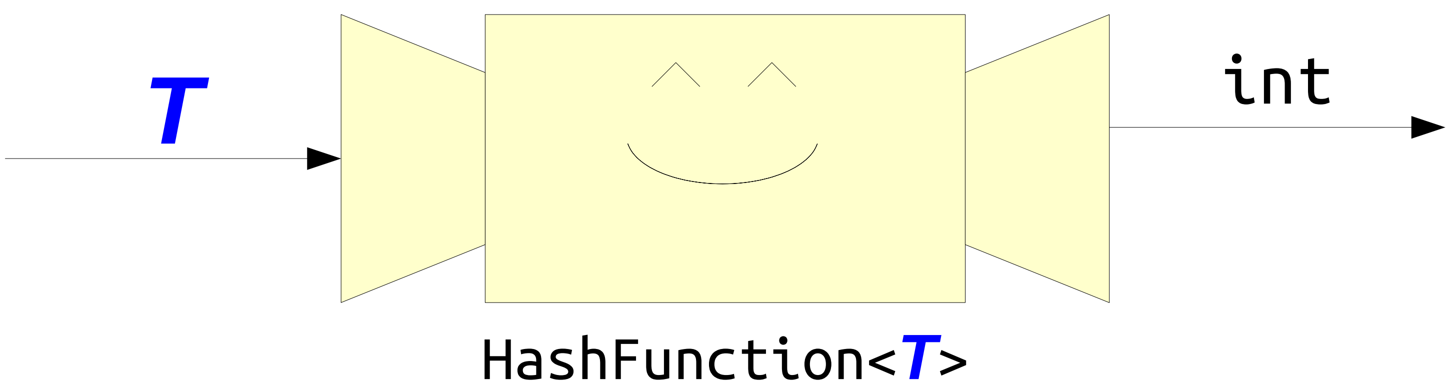
Working with Hash Functions

- Every programming language has a different way for programmers to work with hash functions.
- In CS106B, we'll represent hash functions using the type `HashFunction<T>`.



Working with Hash Functions

- Every programming language has a different way for programmers to work with hash functions.
- In CS106B, we'll represent hash functions using the type `HashFunction<T>`.



Working with Hash Functions

- Sometimes, you want a hash function that outputs values in a wide range.
  - For example, when storing hashes of passwords. *(Why?)*
- Sometimes, you want a hash function that outputs values in a small range.
  - For example, assigning tasks to volunteers.
- Our `HashFunction<T>` returns a value in the range 0, 1, 2, …, n – 1, where n is some number you provide to the constructor.

An Application:

`Map` and `Set`

```
class OurSet {
public:
    OurSet();

    void add(const std::string& str);
    bool contains(const std::string& str) const;

    int  size() const;
    bool isEmpty() const;

private:
    /* What goes here? */

};
```

In header files, we refer to the `string` type as `std::string`. It's an Endearing C++ Quirk. Feel free to ask me about this after class if you're curious why.

```cpp
class OurSet {
public:
    OurSet();

    void add(const std::string& str);
    bool contains(const std::string& str) const;

    int  size() const;
    bool isEmpty() const;

private:
    /* What goes here? */

};
```

```cpp
class OurSet {
public:
    OurSet();

    void add(const std::string& str);
    bool contains(const std::string& str) const;

    int  size() const;
    bool isEmpty() const;

private:
    /* What goes here? */

};
```

An Example: Clothes

For Large Values of *n*

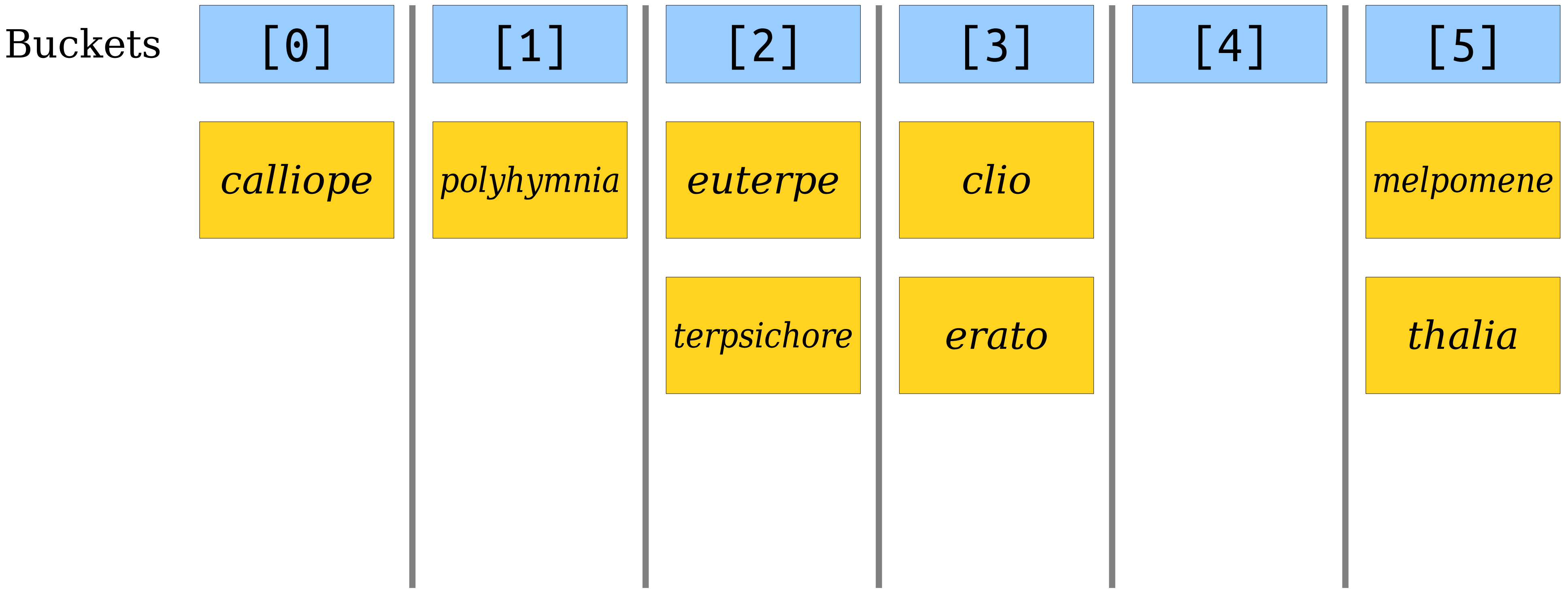
Our Strategy

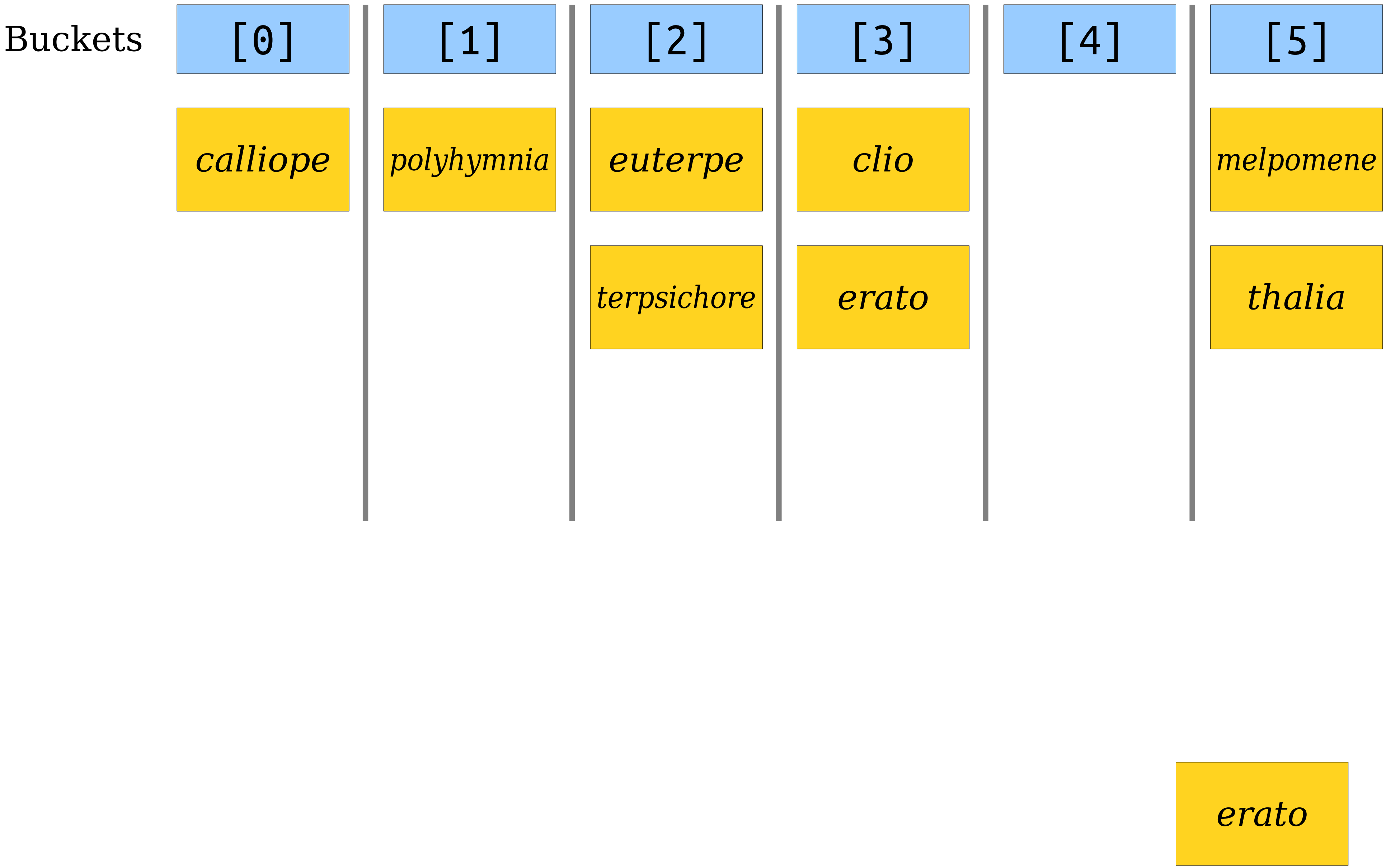
- Maintain a large number of small collections called ***buckets*** (think drawers).
- Find a ***rule*** that lets us tell where each object should go (think knowing which drawer is which).
- To find something, only look in the bucket assigned to it (think looking for socks).

Our Strategy

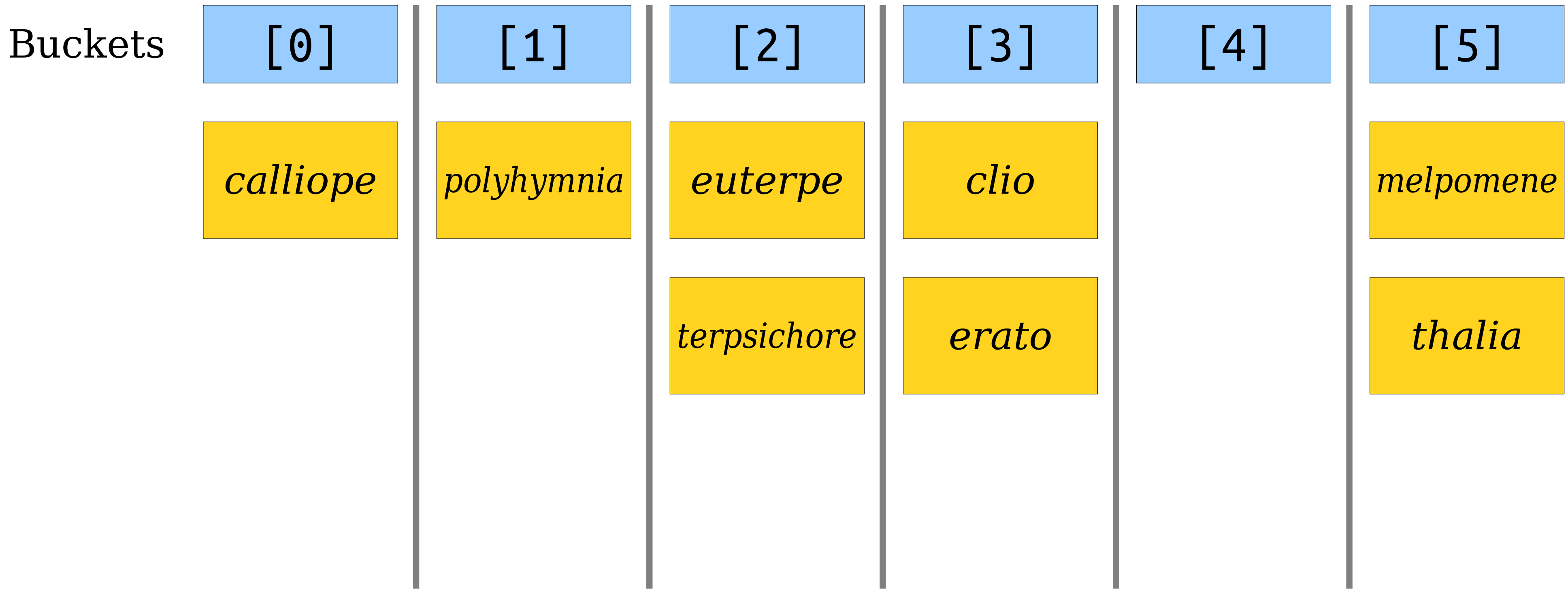
- Maintain a large number of small collections called ***buckets*** (think drawers).
- Find a ***rule*** that lets us tell where each object should go (think knowing which drawer is which).
- To find something, onl… e bucket assigned to it (think looking for socks).

Use a hash function!

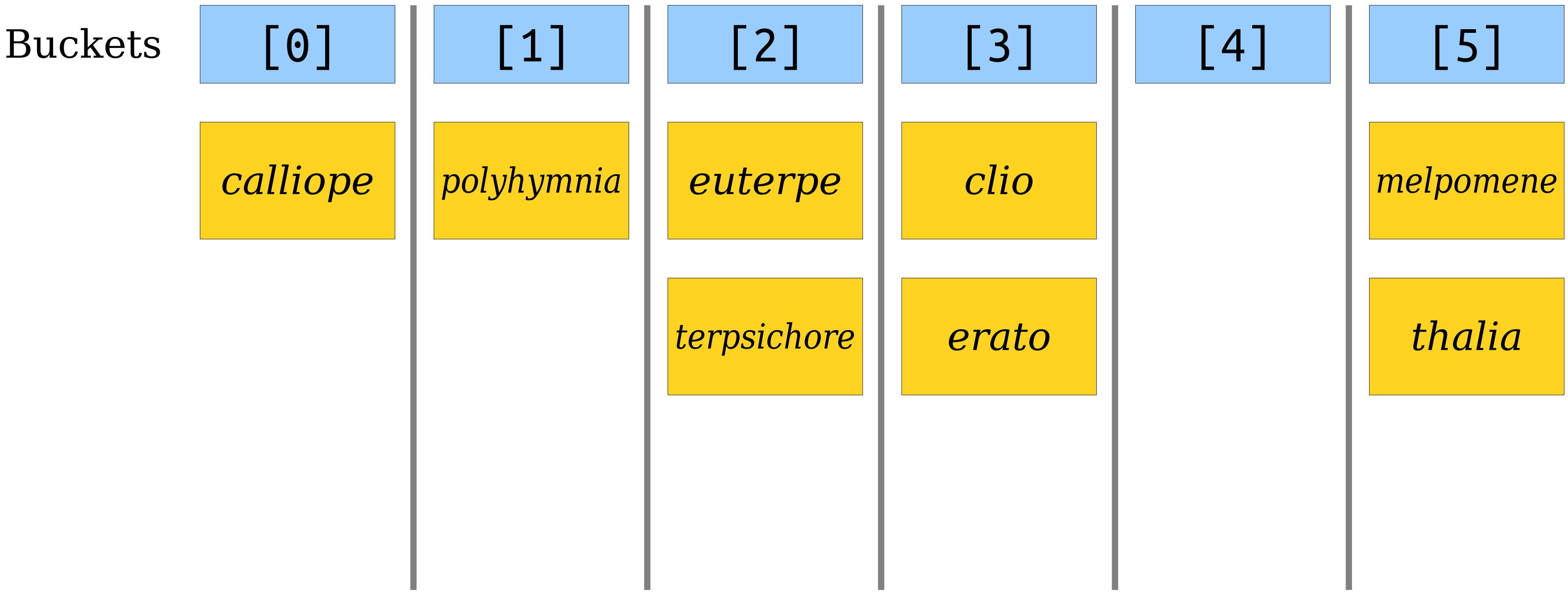

Buckets
[0]
[1]
[2]
[3]
[4]
[5]
calliope
polyhymnia
euterpe
terpsichore
clio
erato
melpomene
thalia



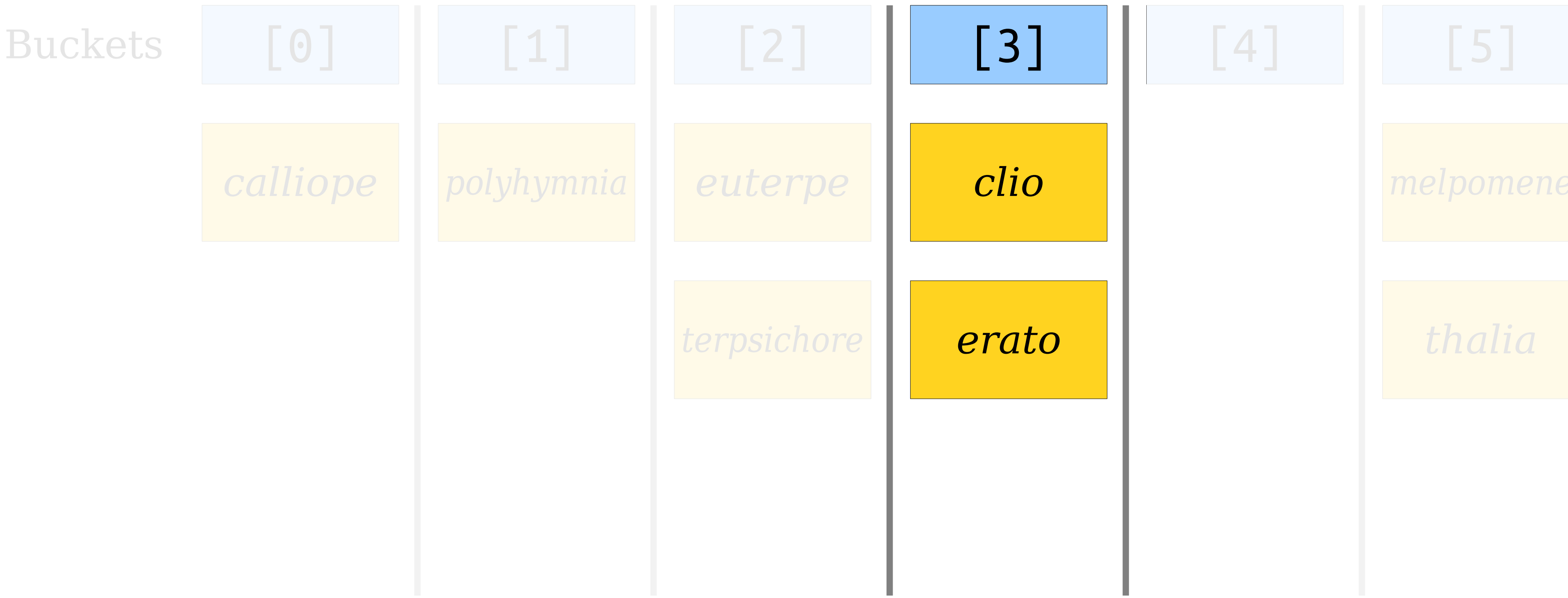
Buckets
[0]
[1]
[2]
[3]
[4]
[5]
calliope
polyhymnia
euterpe
terpsichore
clio
erato
melpomene
thalia
erato




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


}
```

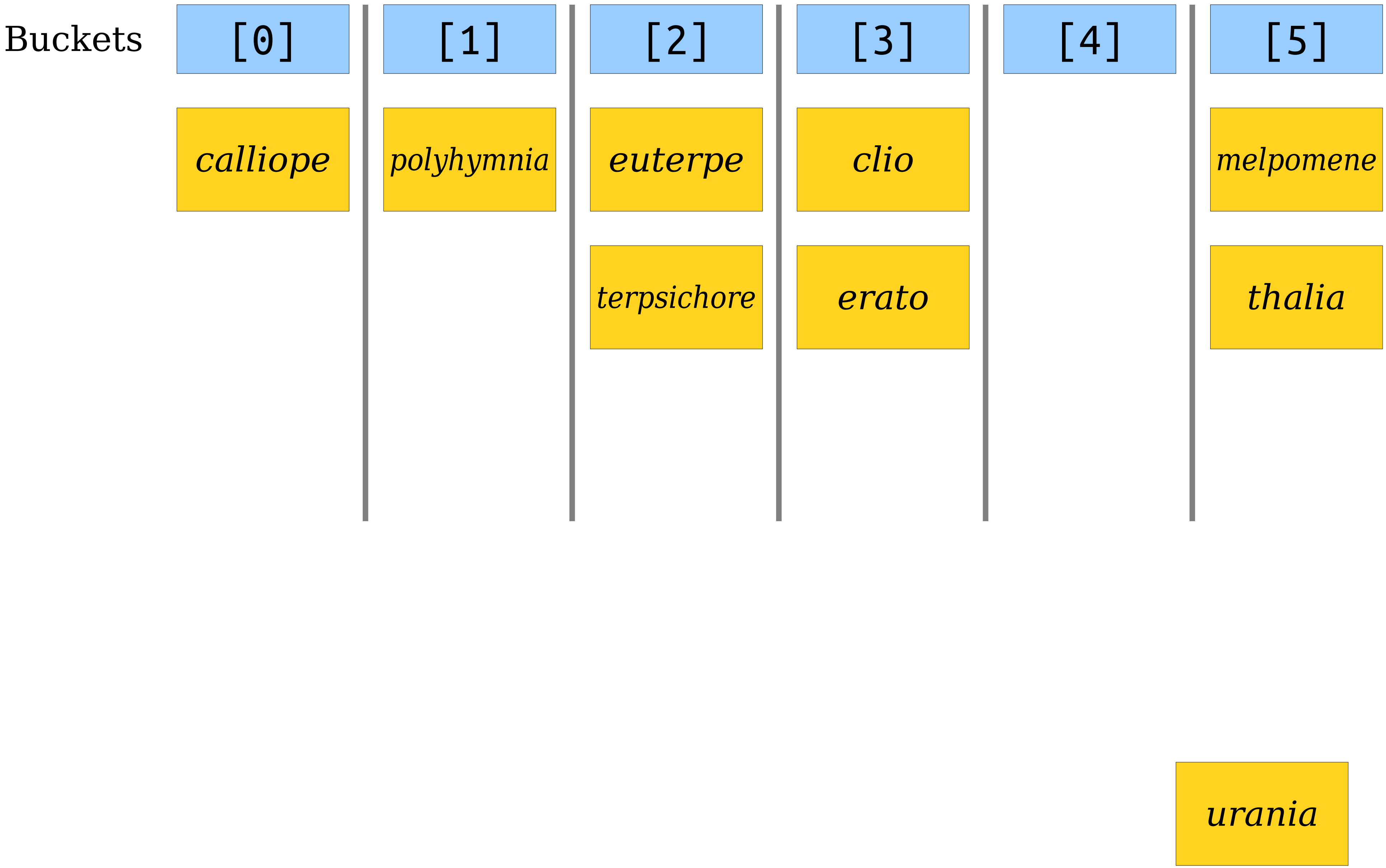


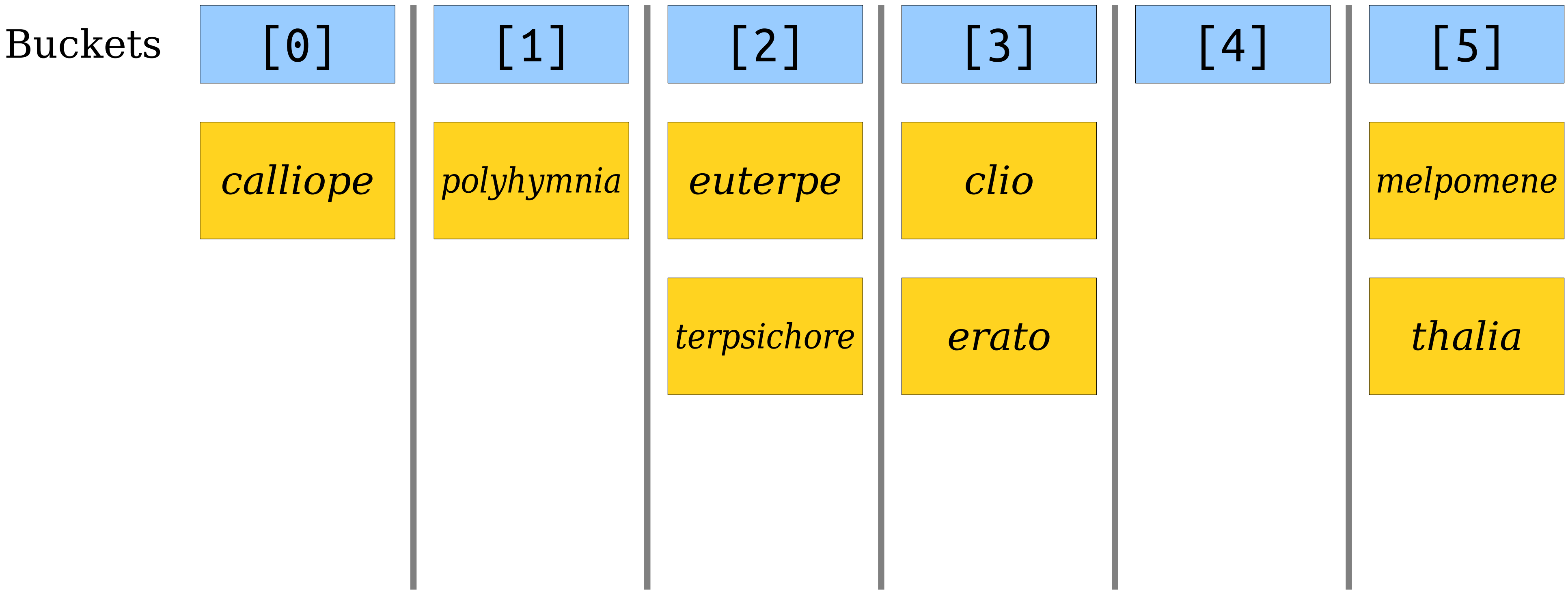

```
bool OurSet::contains(const string& value) const {
    int bucket = hashFn(value);

}
```



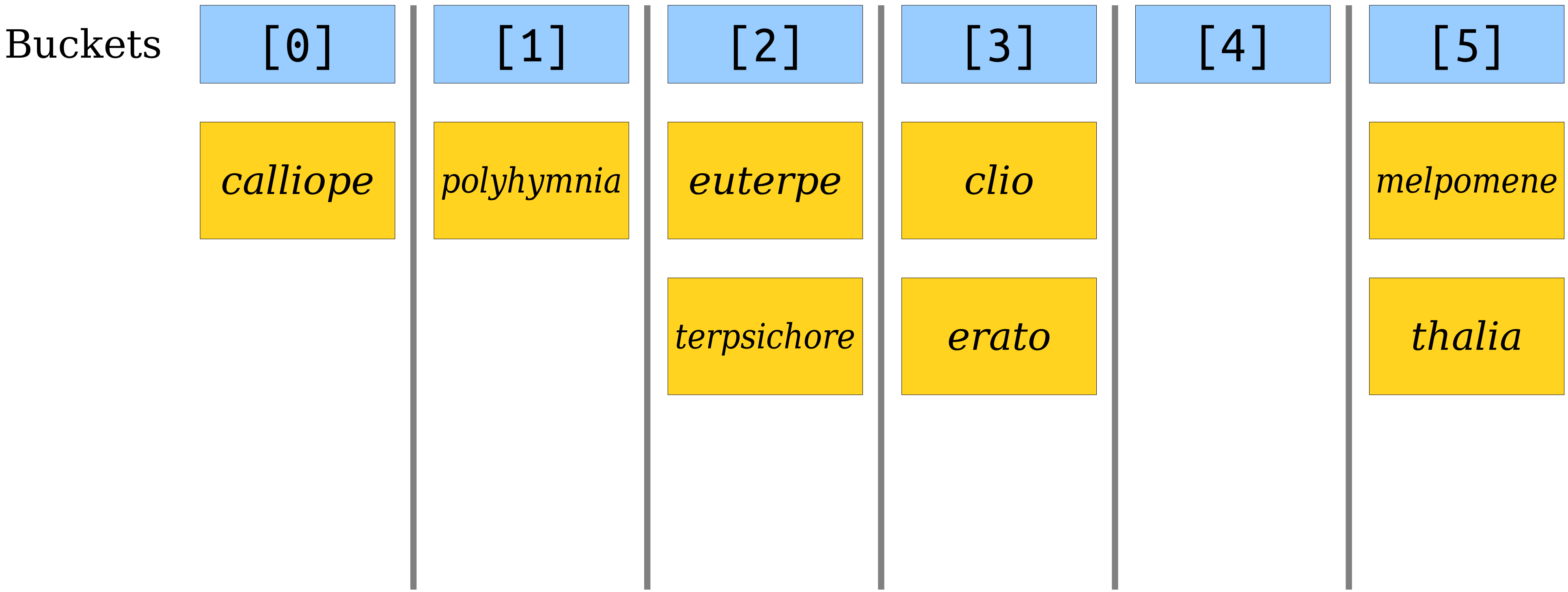

```
bool OurSet::contains(const string& value) const {
    int bucket = hashFn(value);

}
```


```
bool OurSet::contains(const string& value) const {
    int bucket = hashFn(value);
    for (string elem: buckets[bucket]) {
        if (elem == value) return true;
    }
    return false;
}
```

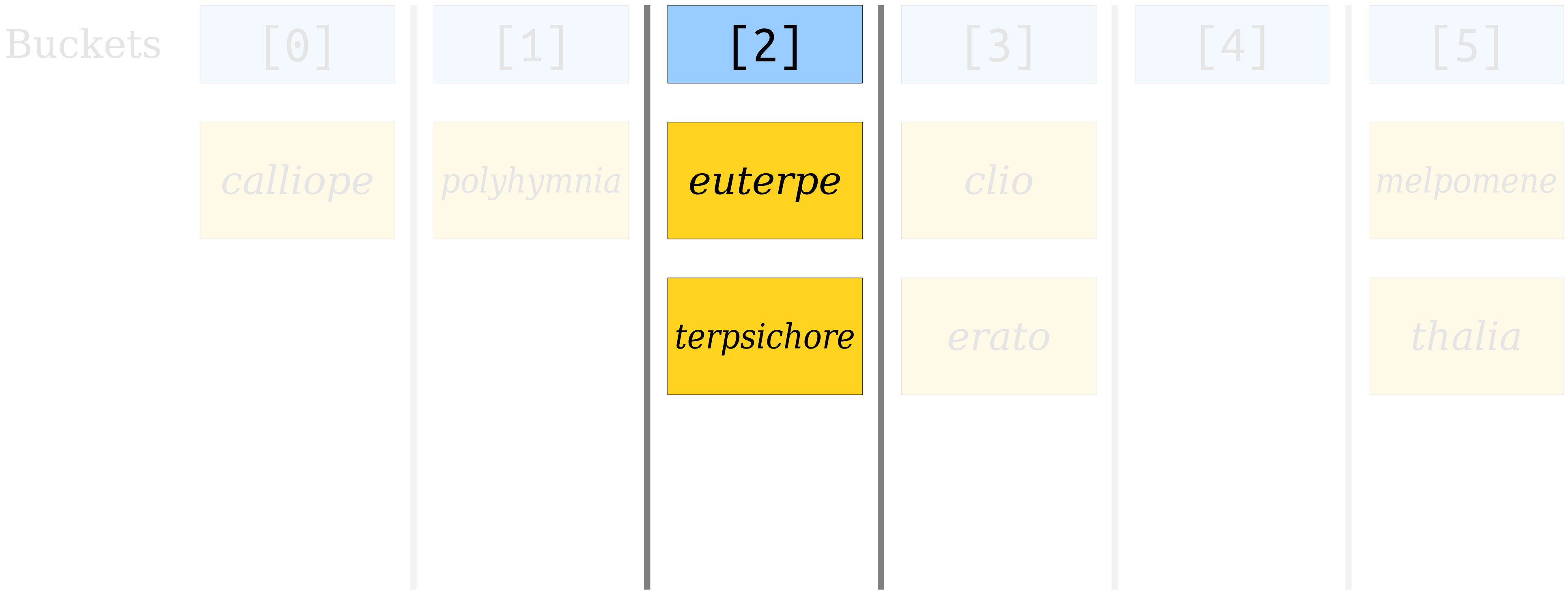

Buckets
[0]
[1]
[2]
[3]
[4]
[5]
calliope
polyhymnia
euterpe
terpsichore
clio
erato
melpomene
thalia
urania



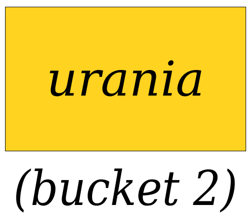

```
void OurSet::add(const string& value) {


}
```

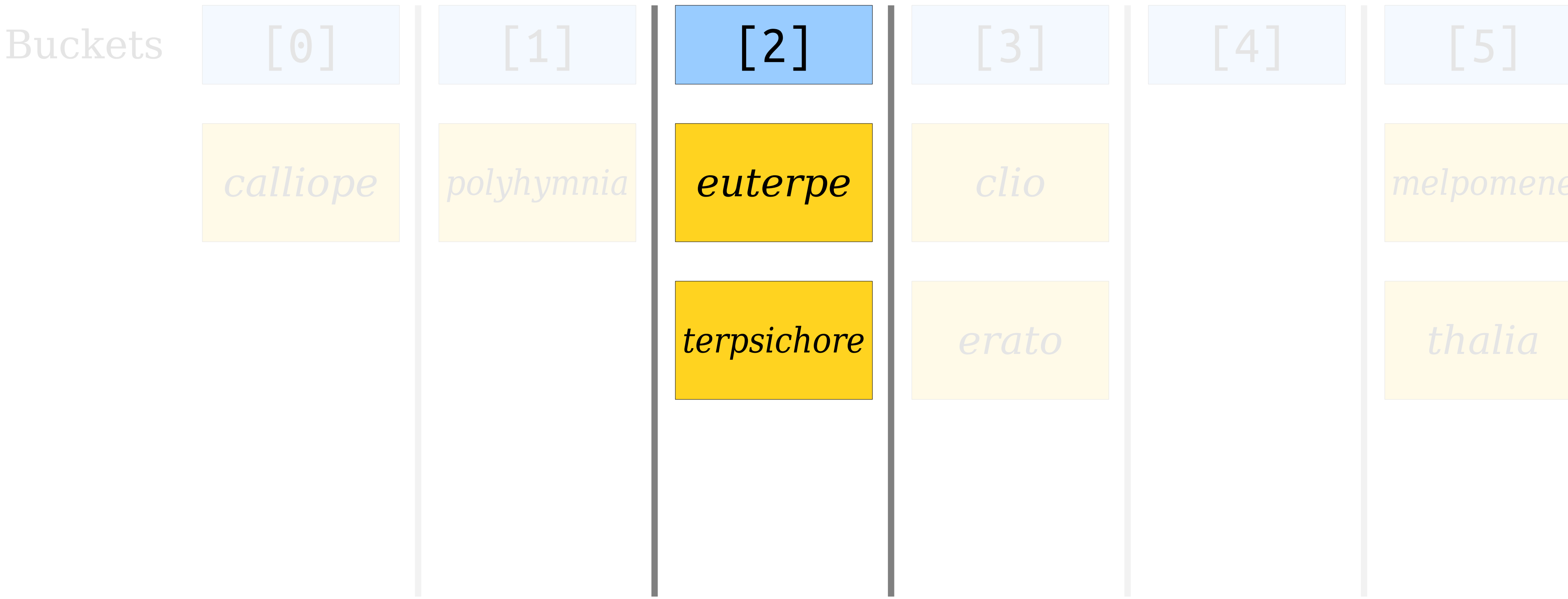



```
void OurSet::add(const string& value) {
    int bucket = hashFn(value);


}
```




```
void OurSet::add(const string& value) {
    int bucket = hashFn(value);


}
```

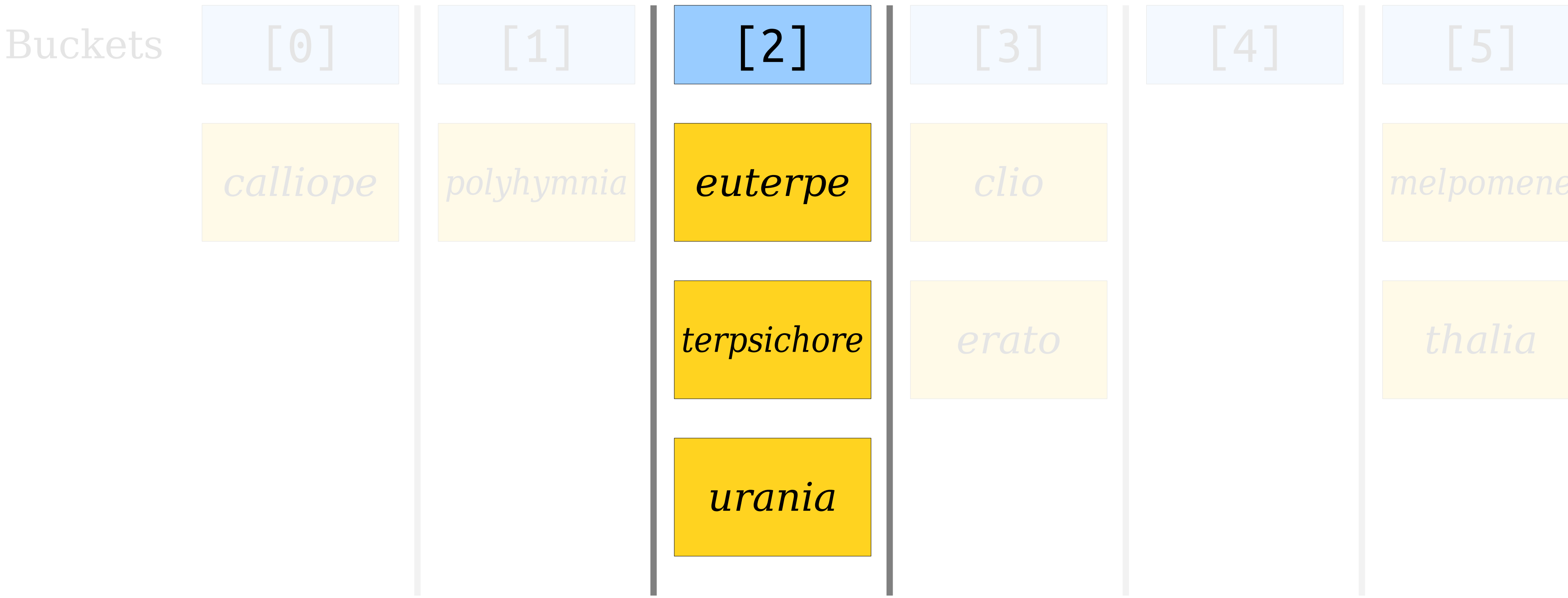




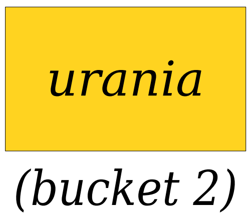

```cpp
void OurSet::add(const string& value) {
    int bucket = hashFn(value);
    buckets[bucket] += value;

}
```

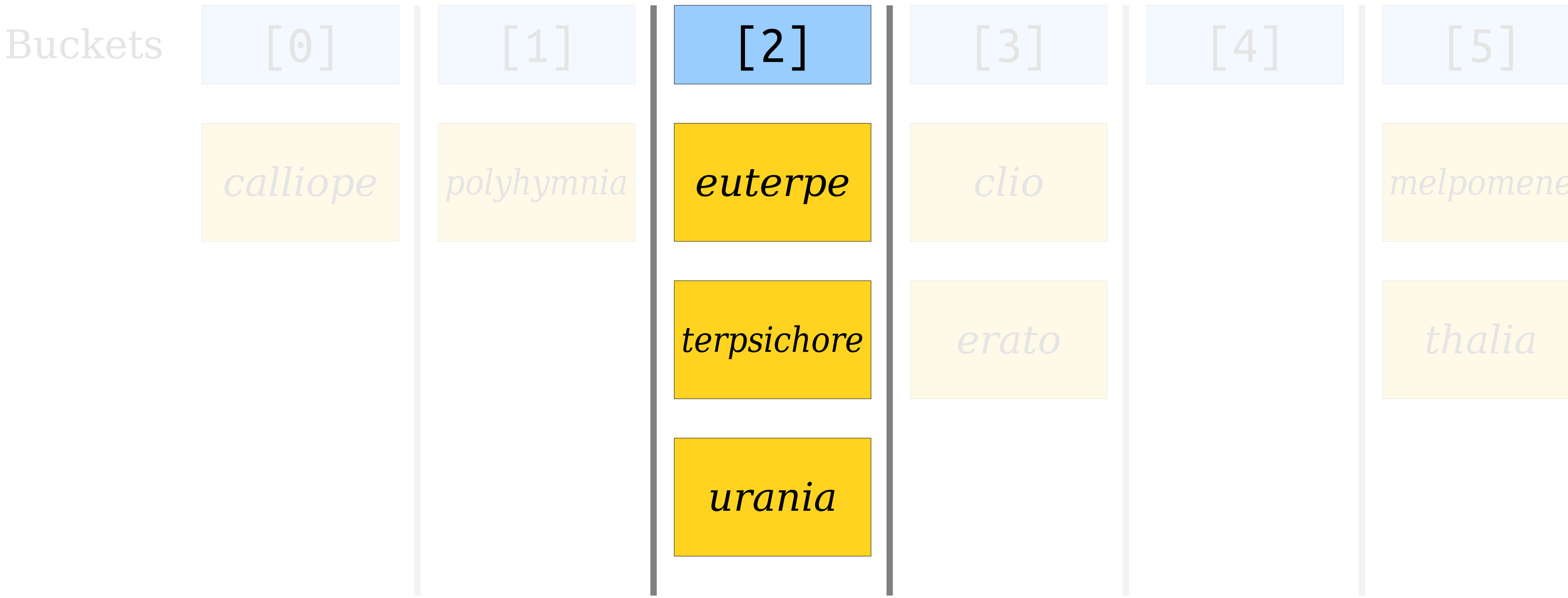
urania

(bucket 2)



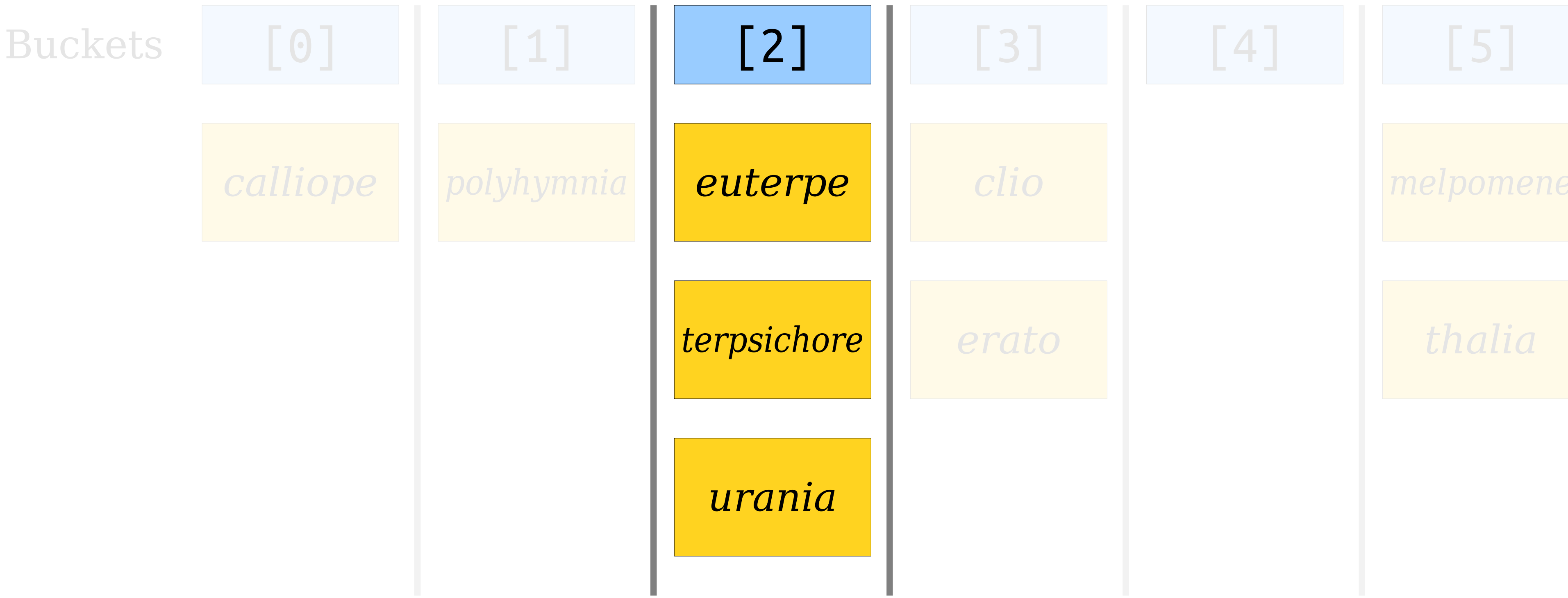

```
void OurSet::add(const string& value) {
    int bucket = hashFn(value);
    buckets[bucket] += value;

}
```






```
void OurSet::add(const string& value) {
    int bucket = hashFn(value);
    buckets[bucket] += value;
    numElems++;
}
```

(bucket 2)



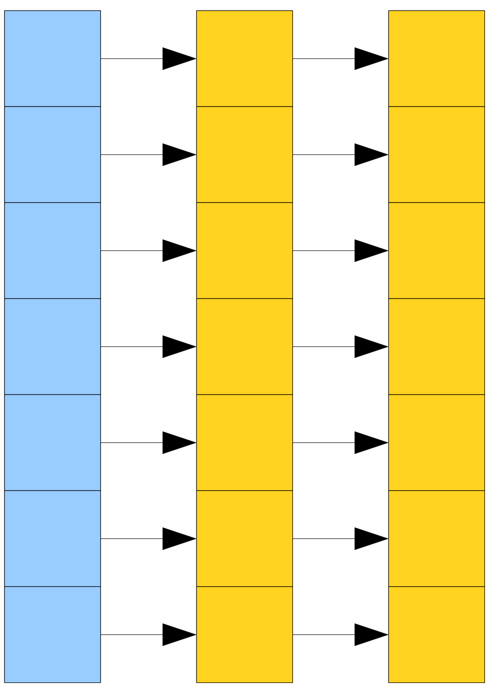

```
void OurSet::add(const string& value) {
    if (contains(value)) return;

    int bucket = hashFn(value);
    buckets[bucket] += value;
    numElems++;
}
```

(bucket 2)

How efficient is this?

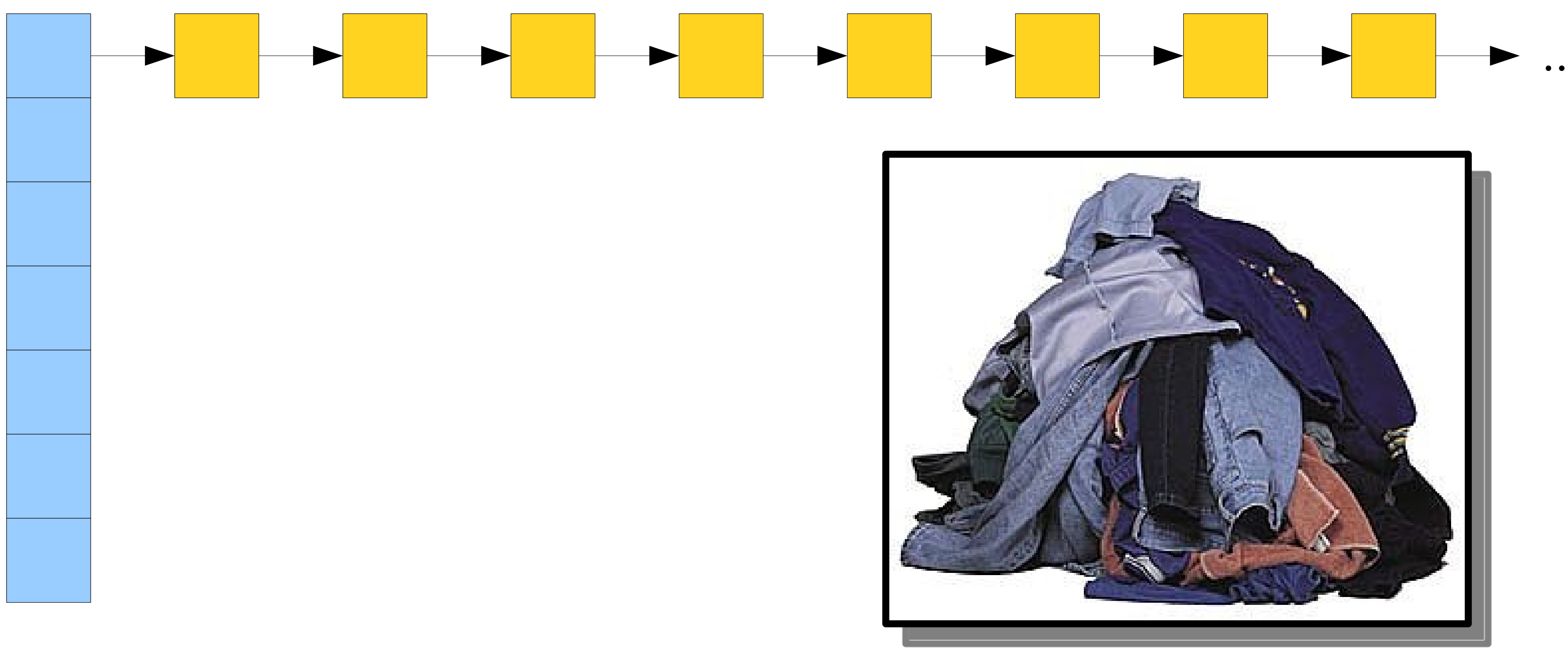
Efficiency Concerns

- 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.
- We'll assume our `HashFunction<T>` type distributes elements more or less randomly.
- Writing good hash functions – or quantifying how good they are – is the domain of courses like CS161, CS166, and CS265. Come talk to me after class if you're curious!

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 *expected* cost of an insertion, deletion, or lookup is therefore

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

Load Factors

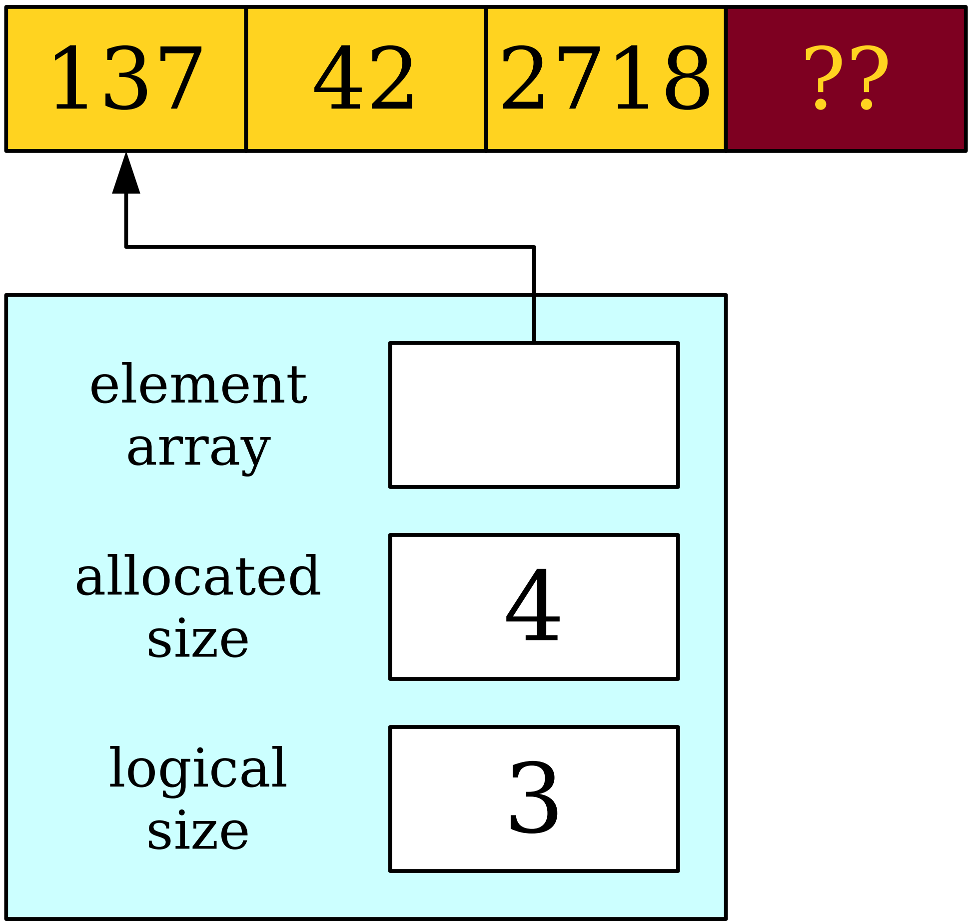
- The ***load factor*** of a hash table with n elements and b buckets is denoted **α** and given by the expression

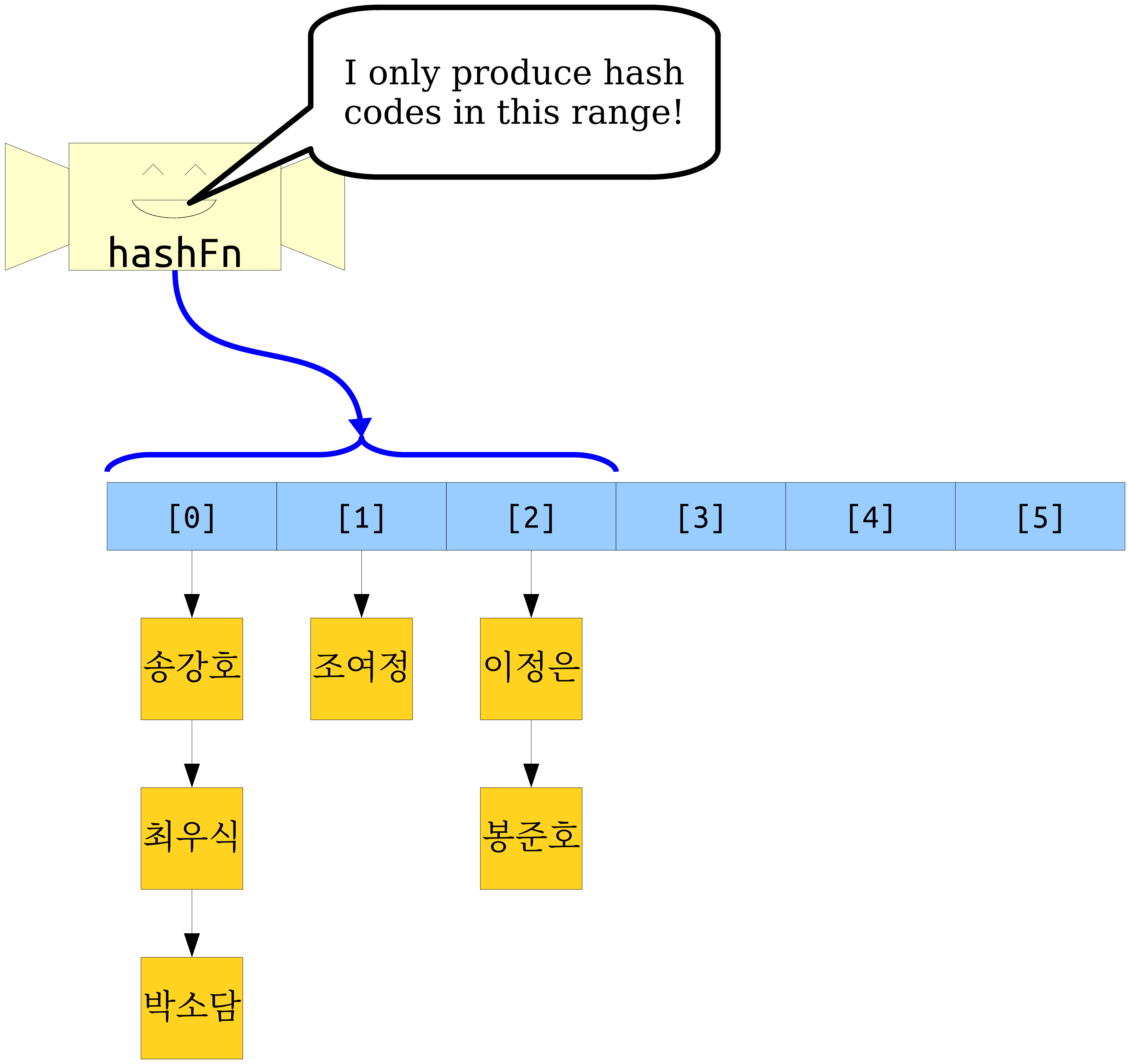
$$\alpha = n / b.$$

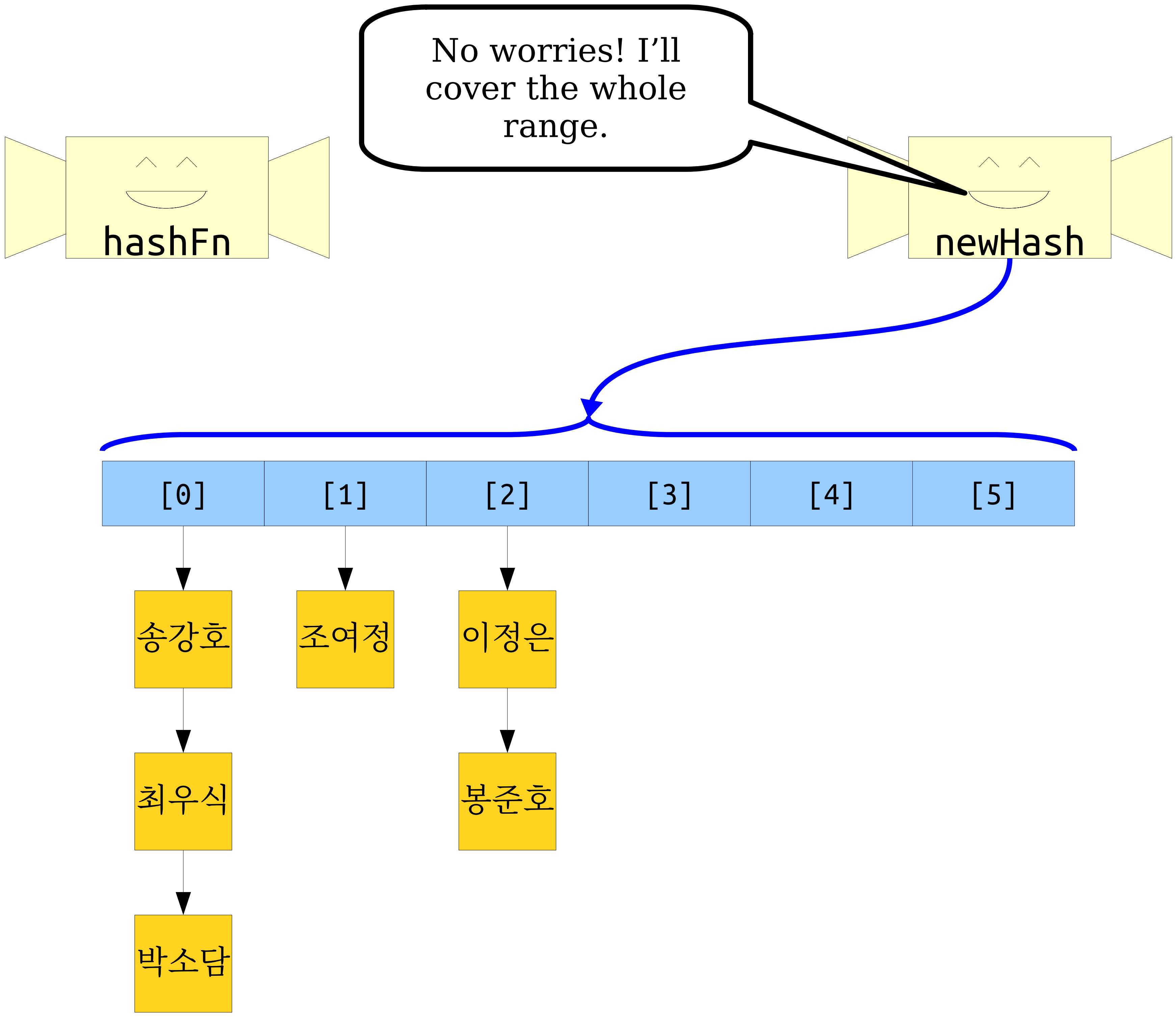
- The expected cost of a lookup in a hash table is $O(1 + n / b) = O(1 + \alpha)$.
  - If α gets too big, the hash table will be too slow.
  - If α gets too low, the hash table will waste too much space.
- How do we balance things?

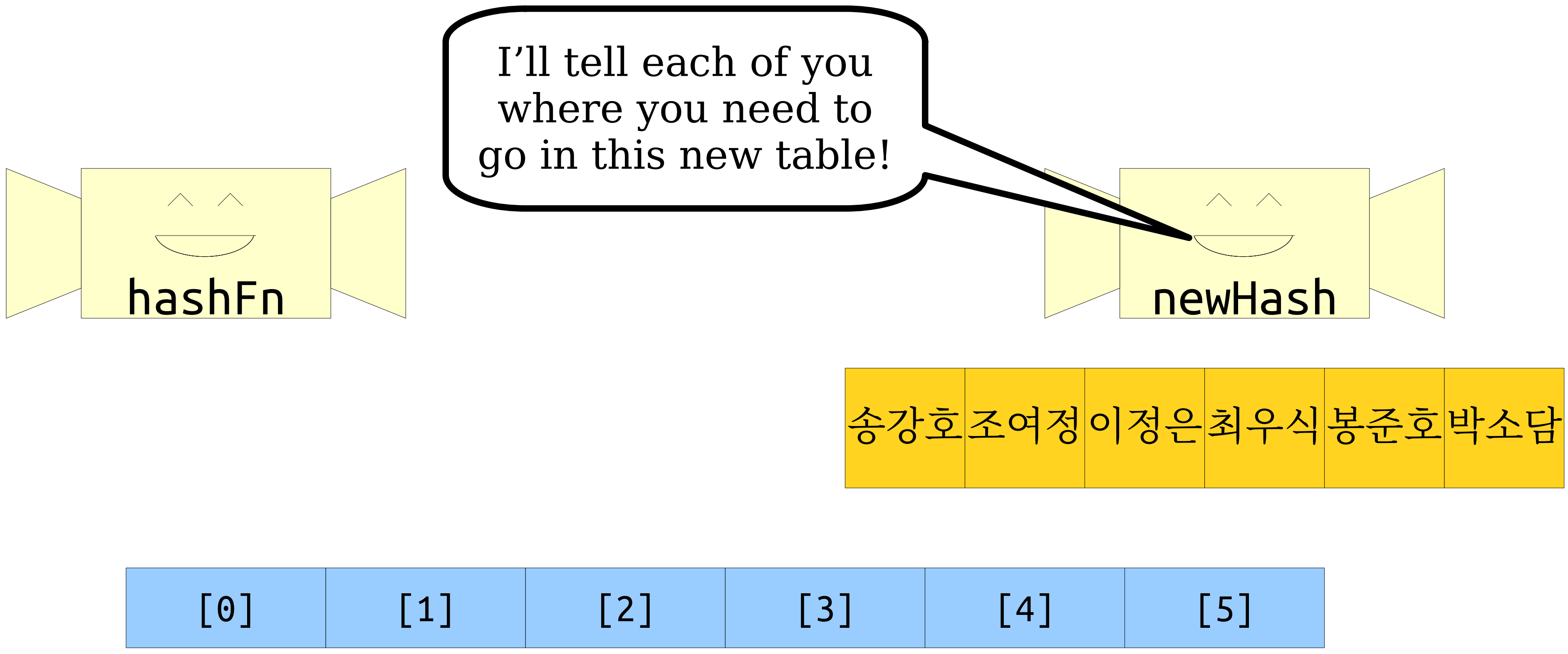
Remember When?

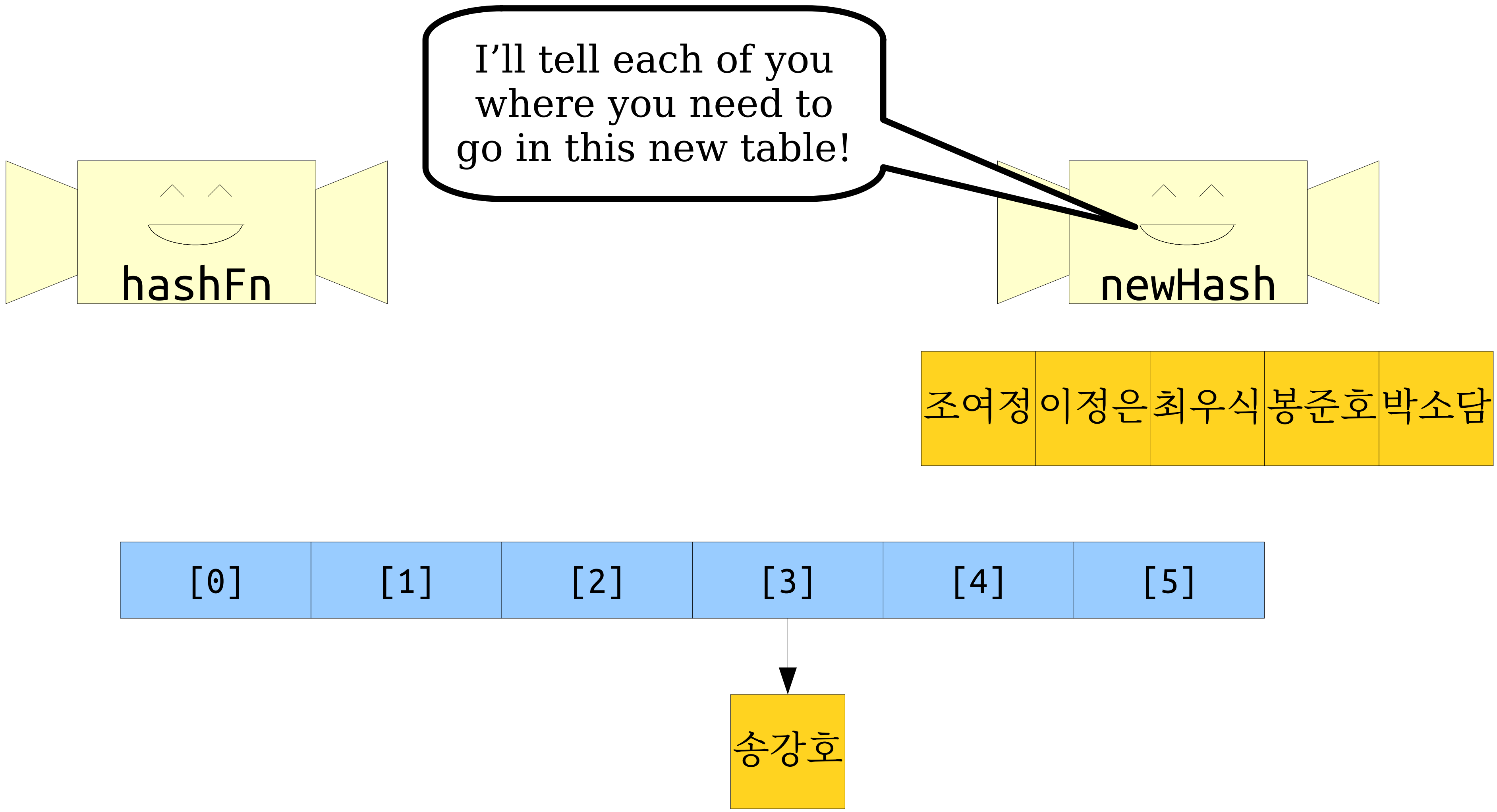
- Think back to how we implemented the `Stack`.
- Initially, we had a fixed number of slots.
- Once we ran out of space, we doubled the number of slots and transferred things over.
- Can we do that here?
- ***Idea:*** Double the table size whenever $n / b \geq 2$.



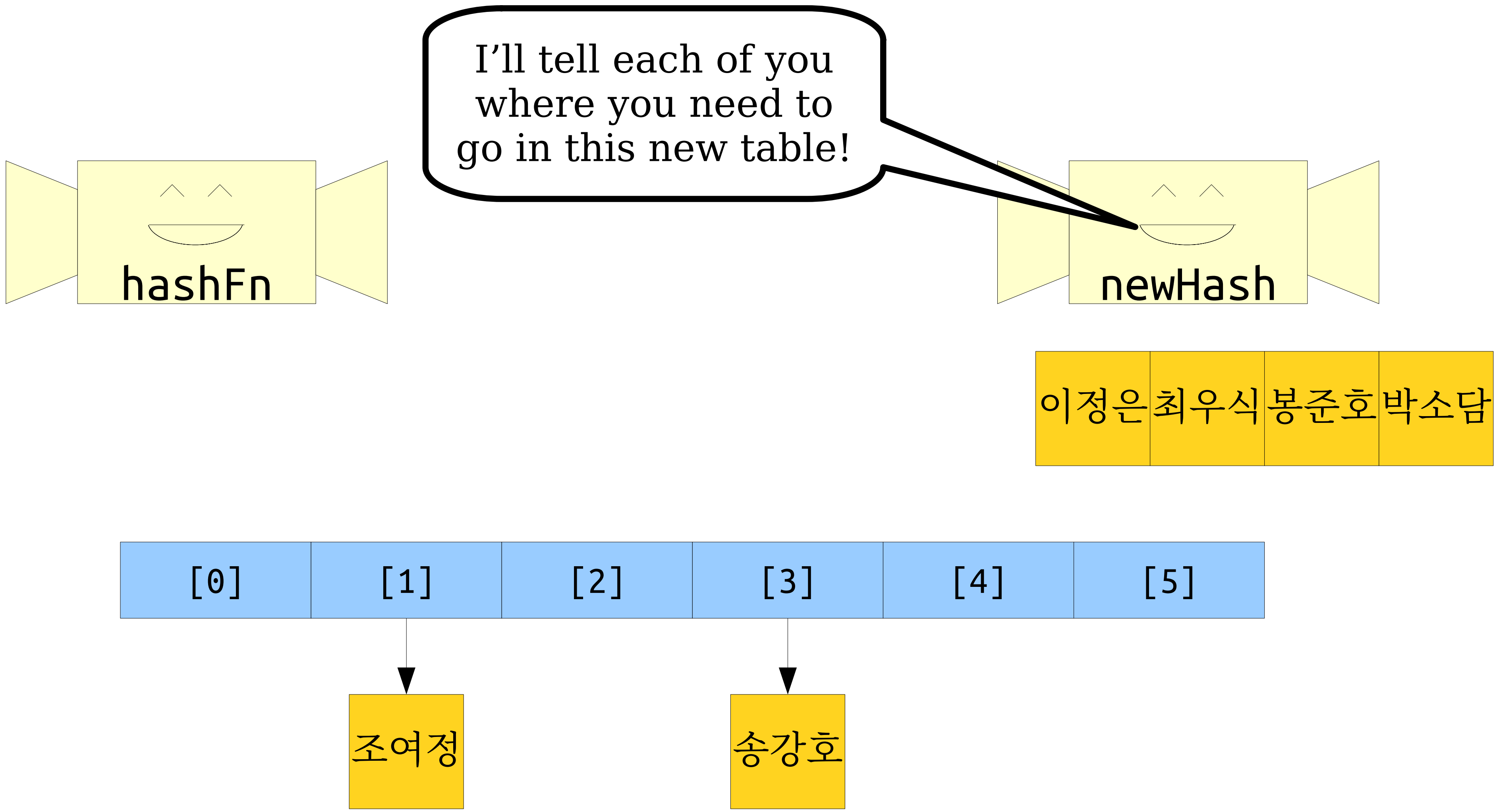

I only produce hash codes in this range!
hashFn
[0]
[1]
[2]
[3]
[4]
[5]
송강호
조여정
이정은
최우식
봉준호
박소담


No worries! I'll cover the whole range.
hashFn
newHash
[0]
[1]
[2]
[3]
[4]
[5]
송강호
조여정
이정은
최우식
봉준호
박소담

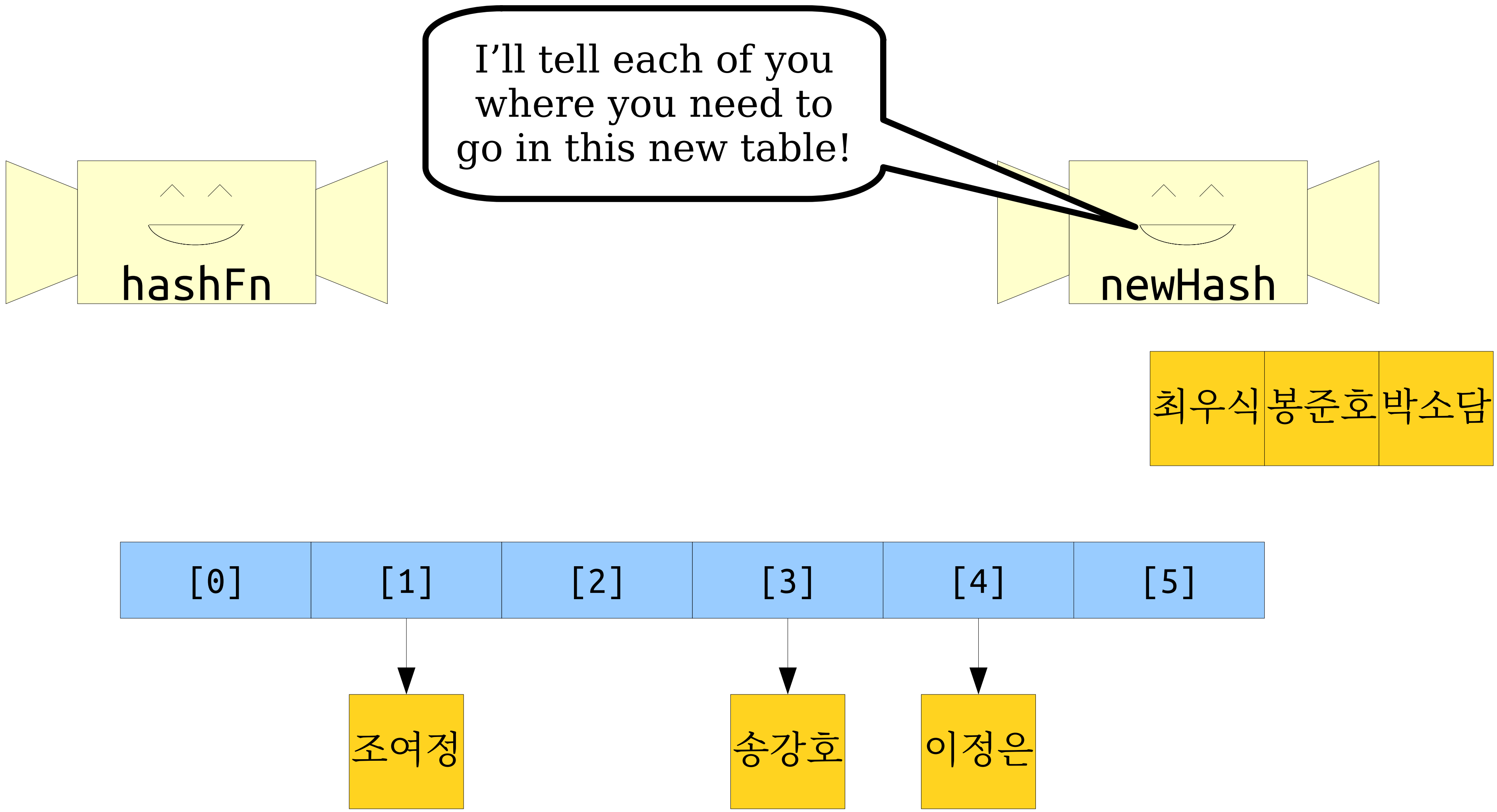

I'll tell each of you where you need to go in this new table!
hashFn
newHash
송강호
조여정
이정은
최우식
봉준호
박소담
[0]
[1]
[2]
[3]
[4]
[5]



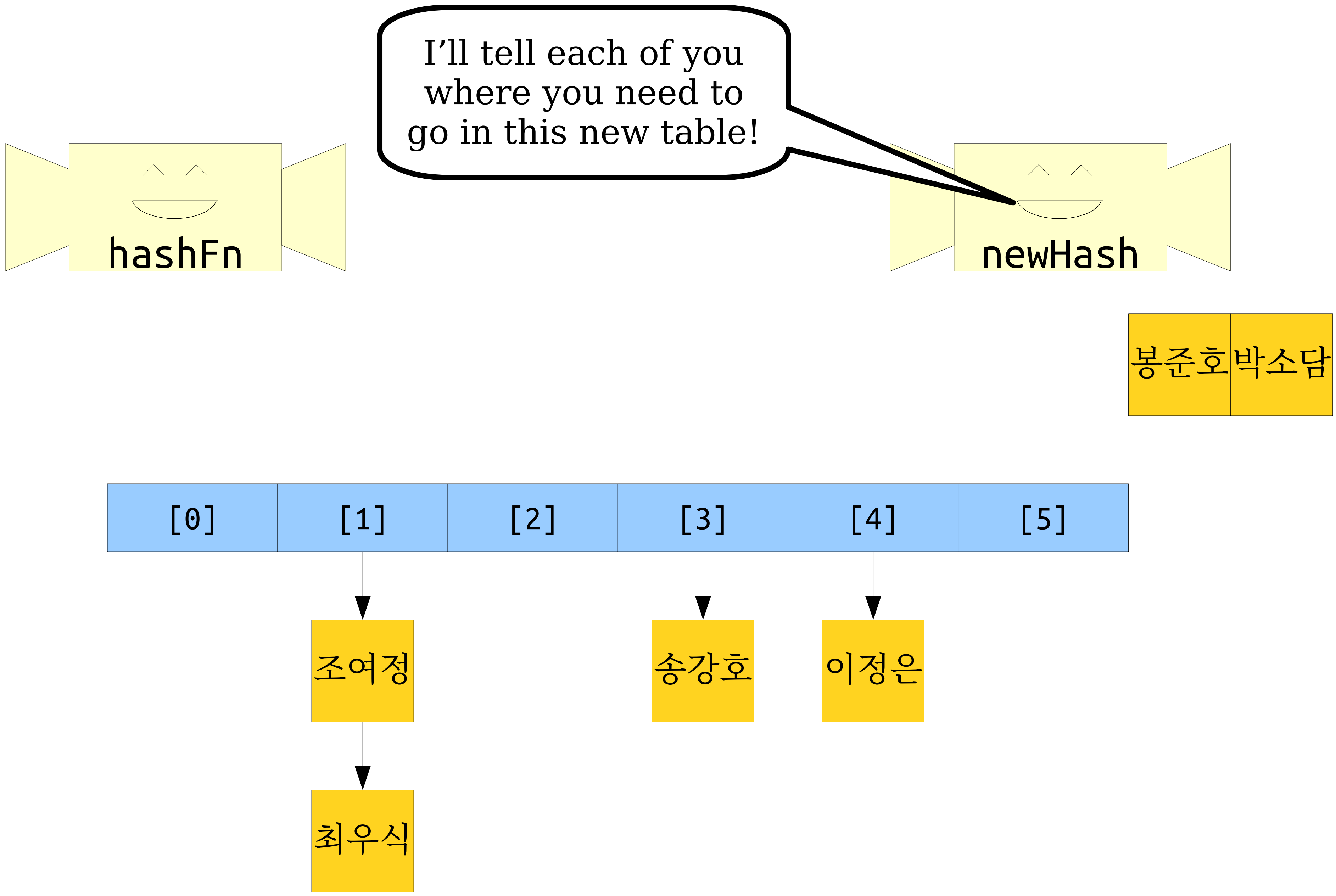
I'll tell each of you where you need to go in this new table!
hashFn
newHash
조여정
이정은
최우식
봉준호
박소담
[0]
[1]
[2]
[3]
[4]
[5]
송강호

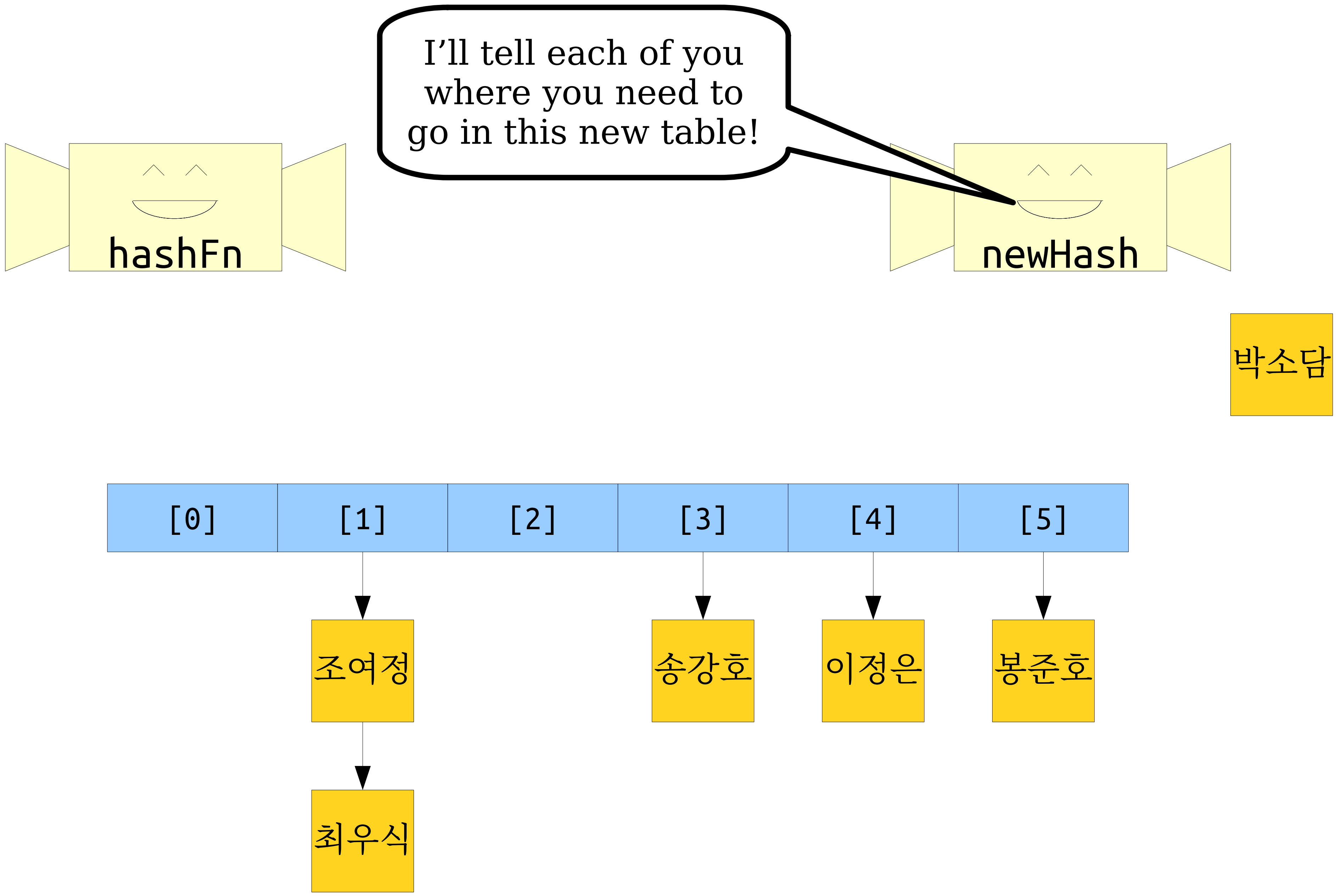


I'll tell each of you where you need to go in this new table!
hashFn
newHash
이정은
최우식
봉준호
박소담
[0]
[1]
[2]
[3]
[4]
[5]
조여정
송강호

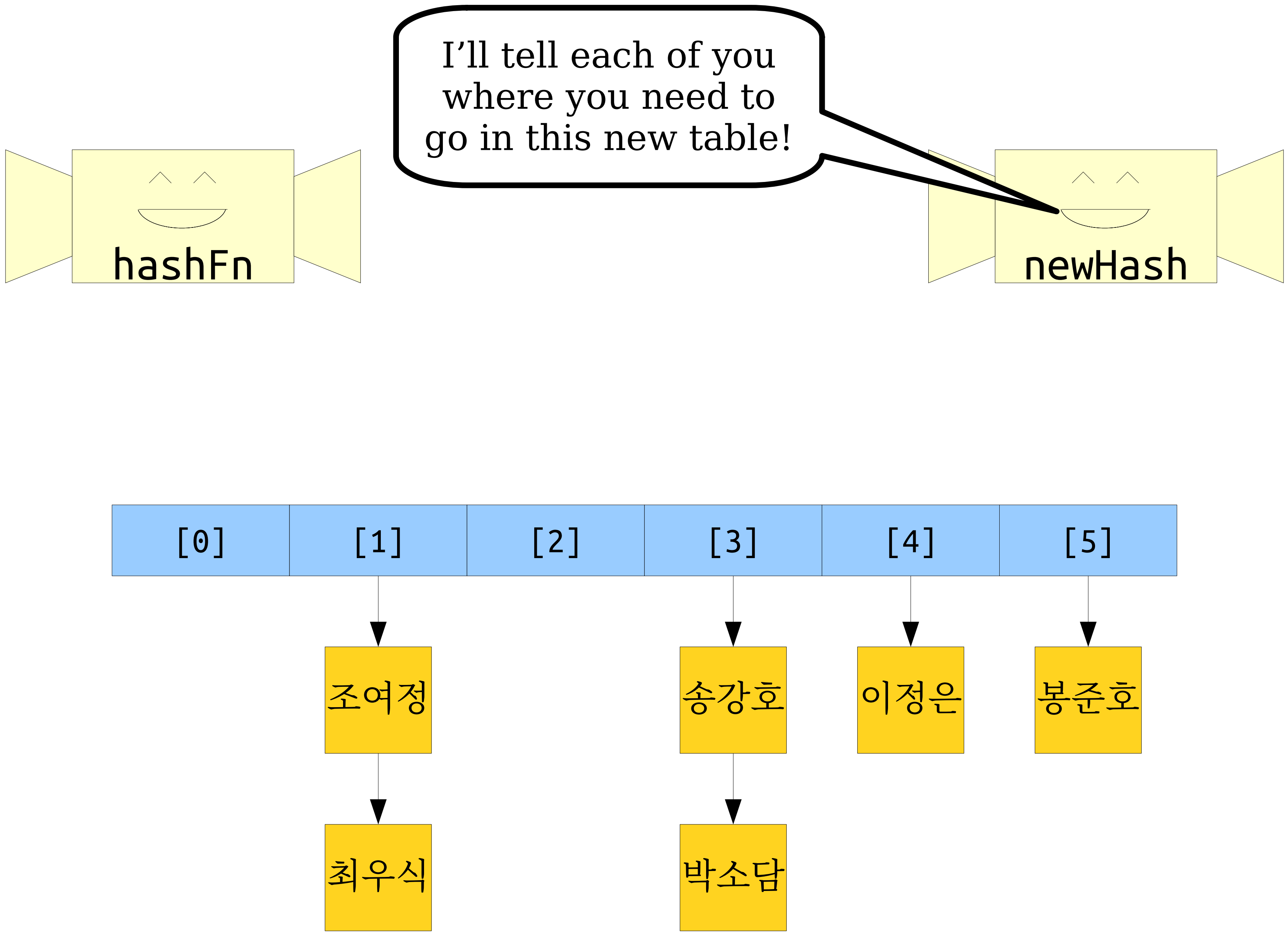


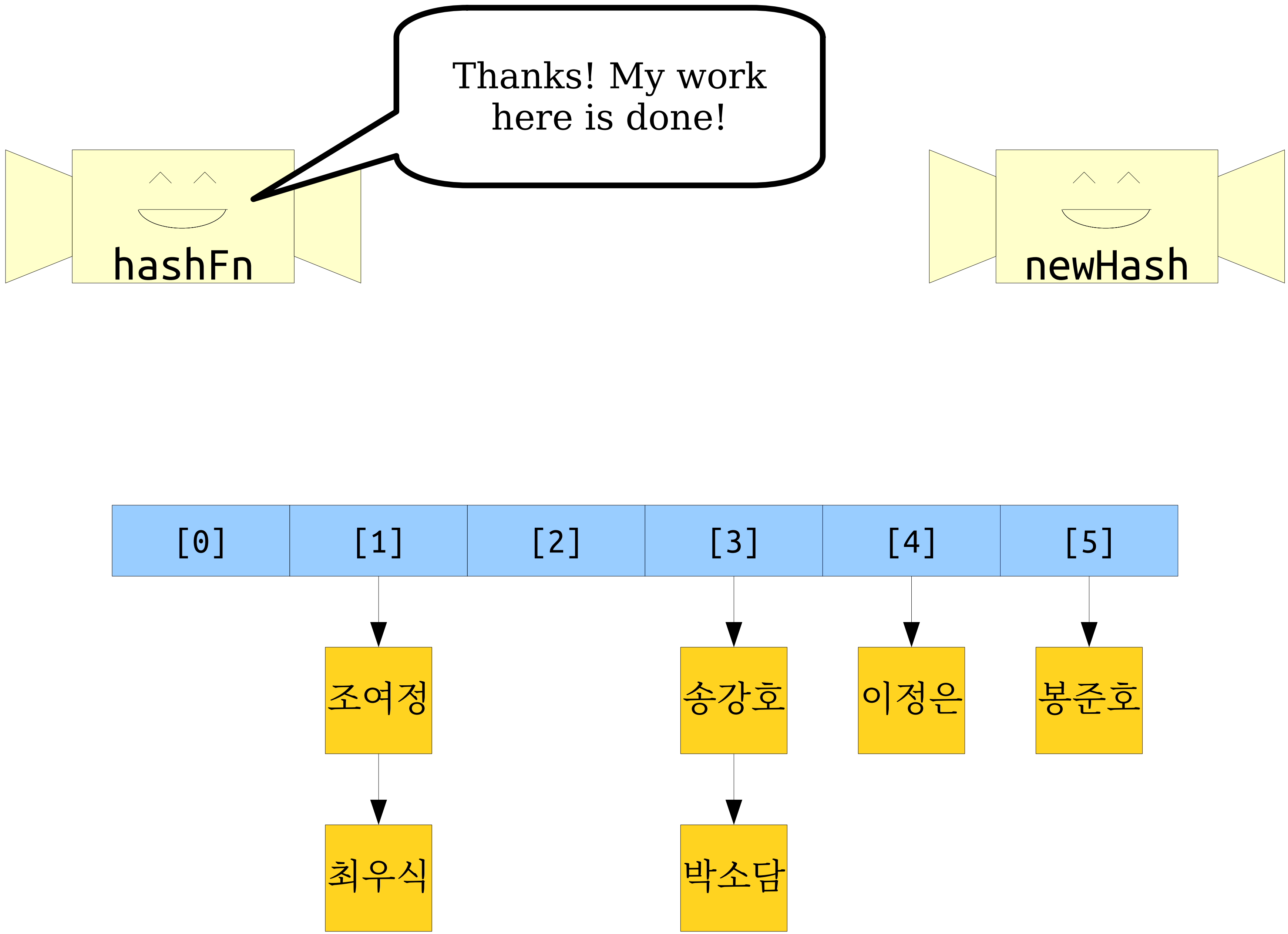
I’ll tell each of you where you need to go in this new table!
hashFn
newHash
최우식
봉준호
박소담
[0]
[1]
[2]
[3]
[4]
[5]
조여정
송강호
이정은

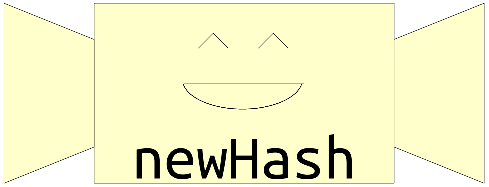

I'll tell each of you where you need to go in this new table!
hashFn
newHash
봉준호
박소담
[0]
[1]
[2]
[3]
[4]
[5]
조여정
최우식
송강호
이정은

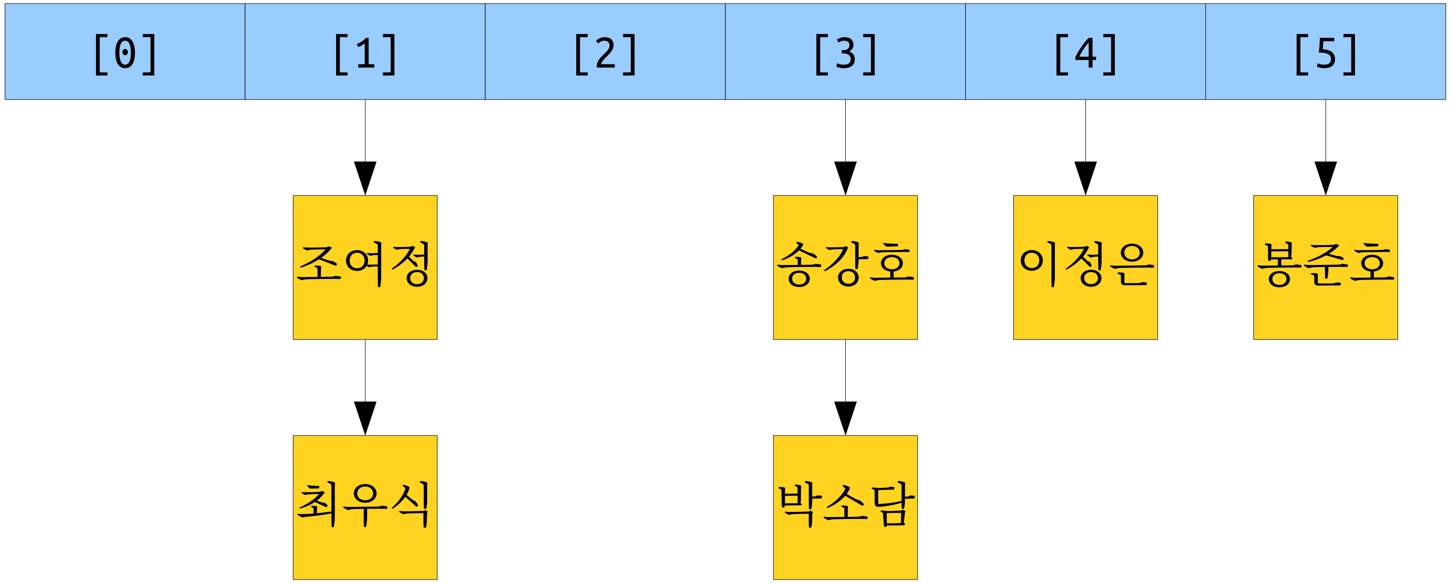


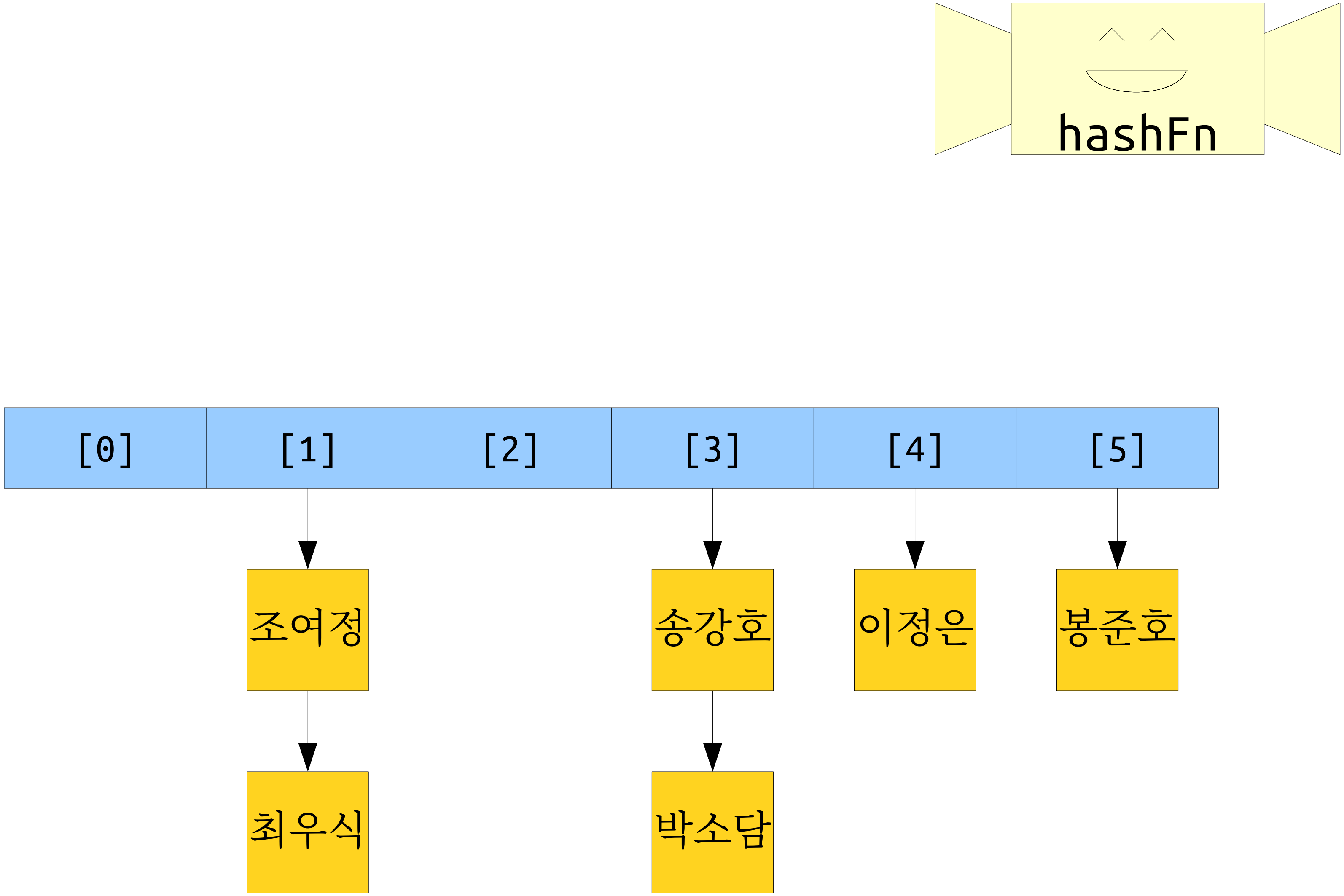
I'll tell each of you where you need to go in this new table!
hashFn
newHash
박소담
[0]
[1]
[2]
[3]
[4]
[5]
조여정
최우식
송강호
이정은
봉준호


I'll tell each of you where you need to go in this new table!
hashFn
newHash
[0]
[1]
[2]
[3]
[4]
[5]
조여정
최우식
송강호
박소담
이정은
봉준호


Thanks! My work here is done!
hashFn
newHash
[0]
[1]
[2]
[3]
[4]
[5]
조여정
최우식
송강호
박소담
이정은
봉준호


newHash


[0]
[1]
[2]
[3]
[4]
[5]
조여정
최우식
송강호
박소담
이정은
봉준호


hashFn
[0]
[1]
[2]
[3]
[4]
[5]
조여정
최우식
송강호
박소담
이정은
봉준호

Rehashing

- To perform a ***rehash***, do the following:
  - Get a new list of buckets, twice as big as before.
  - Get a new hash function that distributes elements across the wider range.
  - Redistribute the elements from the old buckets into the new ones, using the new hash function.
  - Use the new buckets and hash functions going forward.
- Time required is O(n). However, this happens so rarely that the extra work averages out to O(1) per insert.

The Final Scorecard

- Assuming we cap the load factor α at some constant (say, 2), then 1 + α = O(1).
  - That is, 1 + α doesn't grow as a function of *n*, the number of elements in the hash table.
- The expected cost of a lookup is **O(1)**.
- The expected cost of an insertion is **O(1)**.
  - (It's actually *expected amortized* O(1), since we do some work to copy things over, but only very infrequently.)
- This is about as good as it gets!

Your Action Items

- ***Work on Assignment 6***
  - If you're following our proposed timetable, you'll be wrapping up your `HeapPQueue` implementation by Wednesday.
  - Need help or support? Come talk to us at LaIR, in office hours, or over EdStem!

Next Time

- ***Guest Lecture by Katie Creel***
  - Our resident ethicist!
- ***Ethics of Ranking***
  - What happens if you reduce someone to a single number?
- ***Ethics of Priority Queues***
  - What happens when you rank people from highest to lowest priority?