

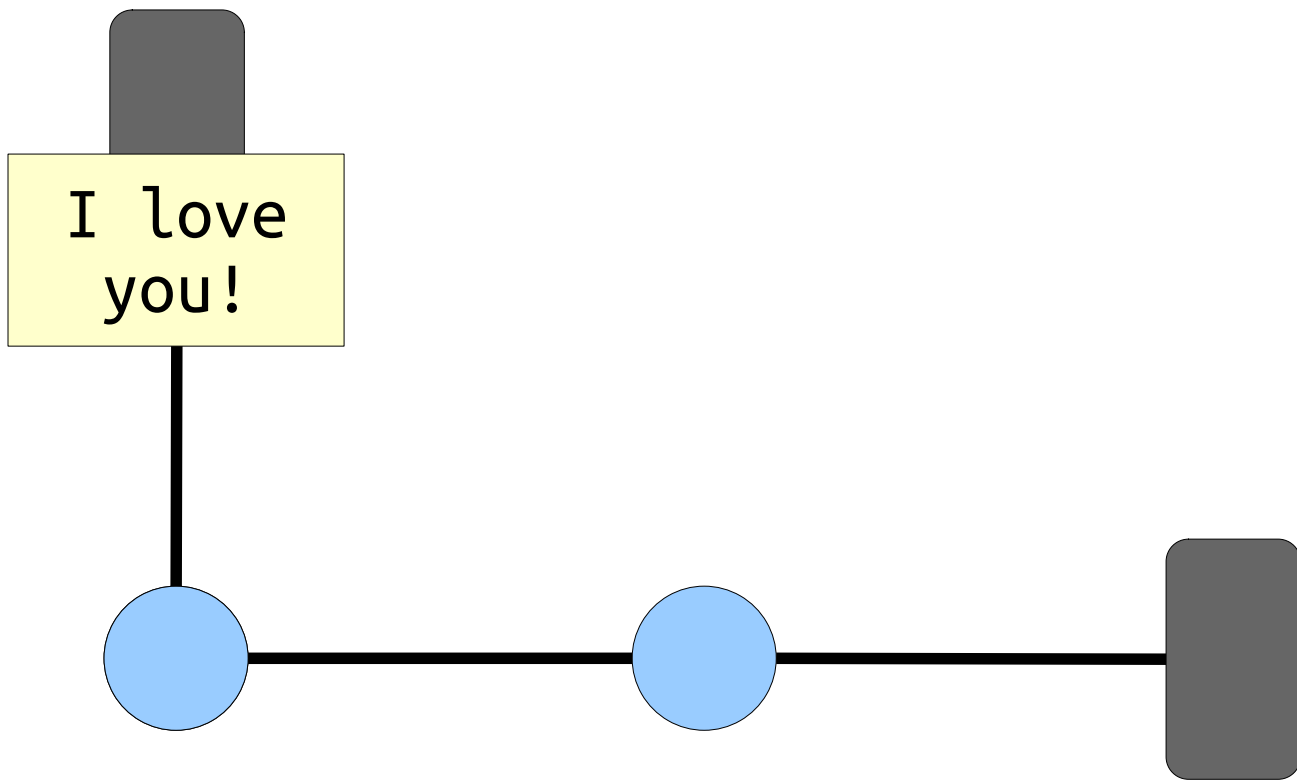
Hashing

Part One

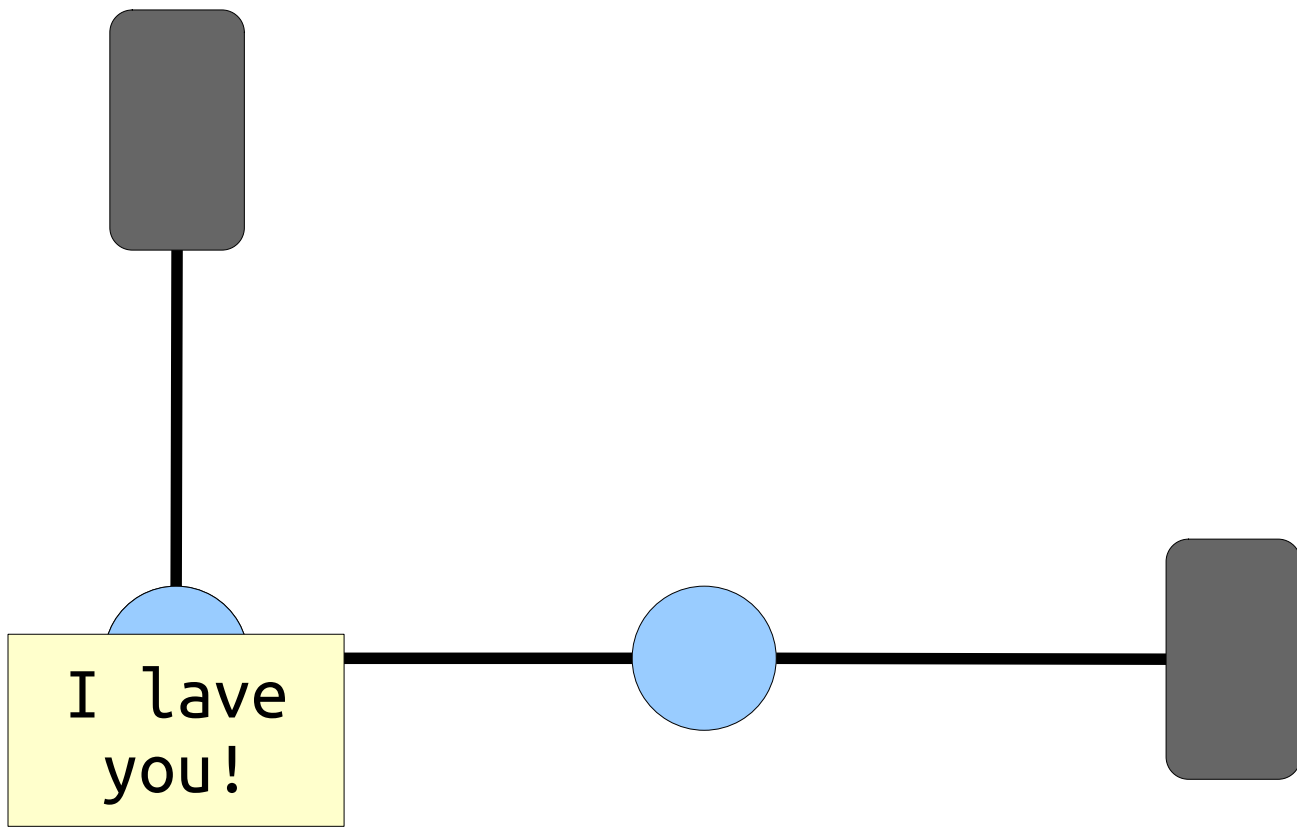
Outline for Today

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

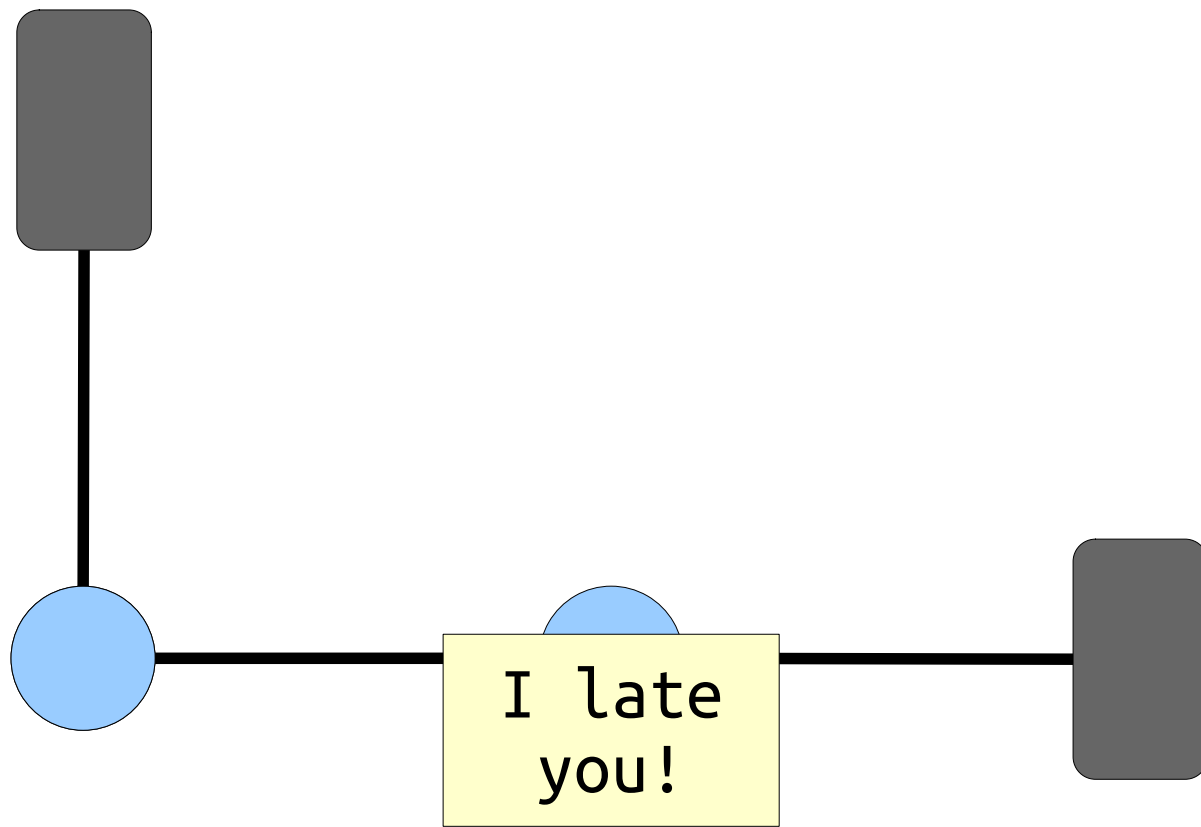
Two Motivating Problems



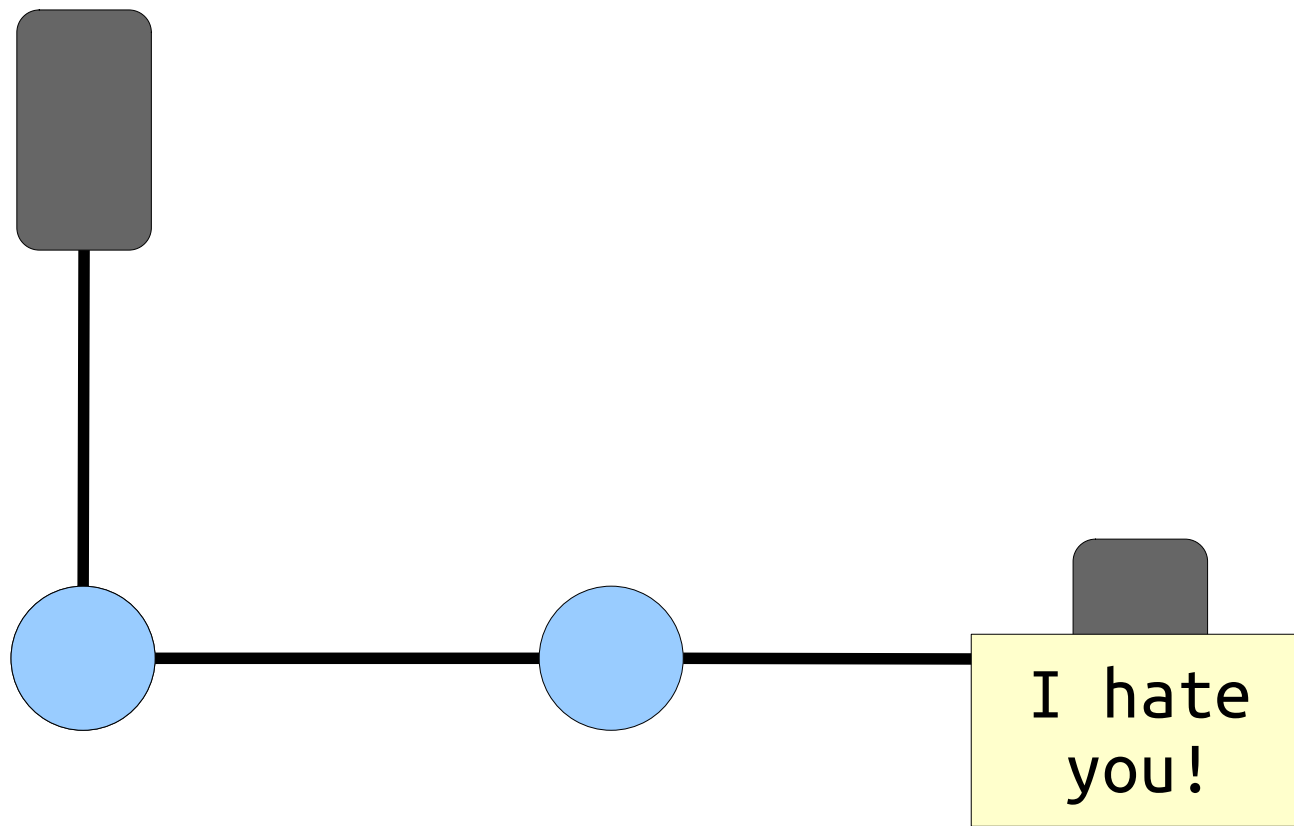
Did my data make it through the network?



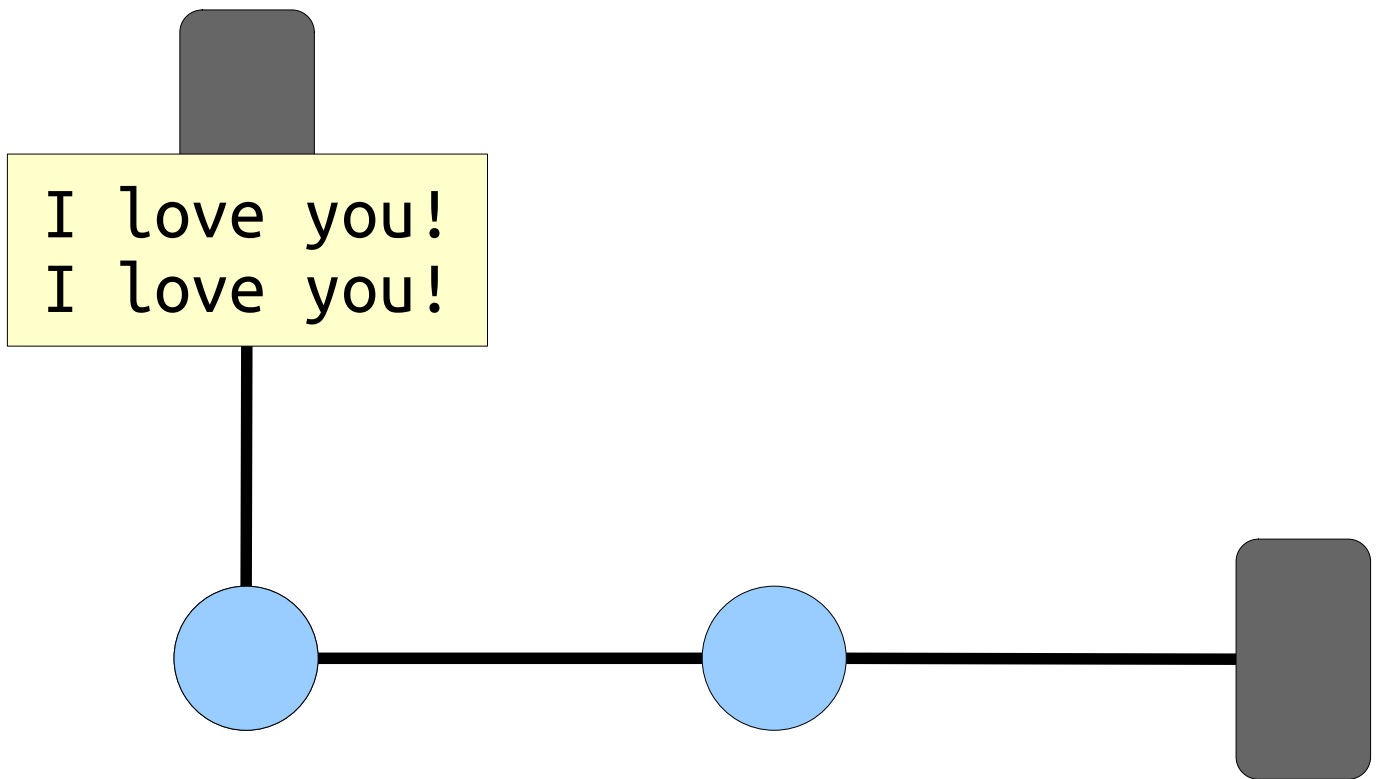
Did my data make it through the network?



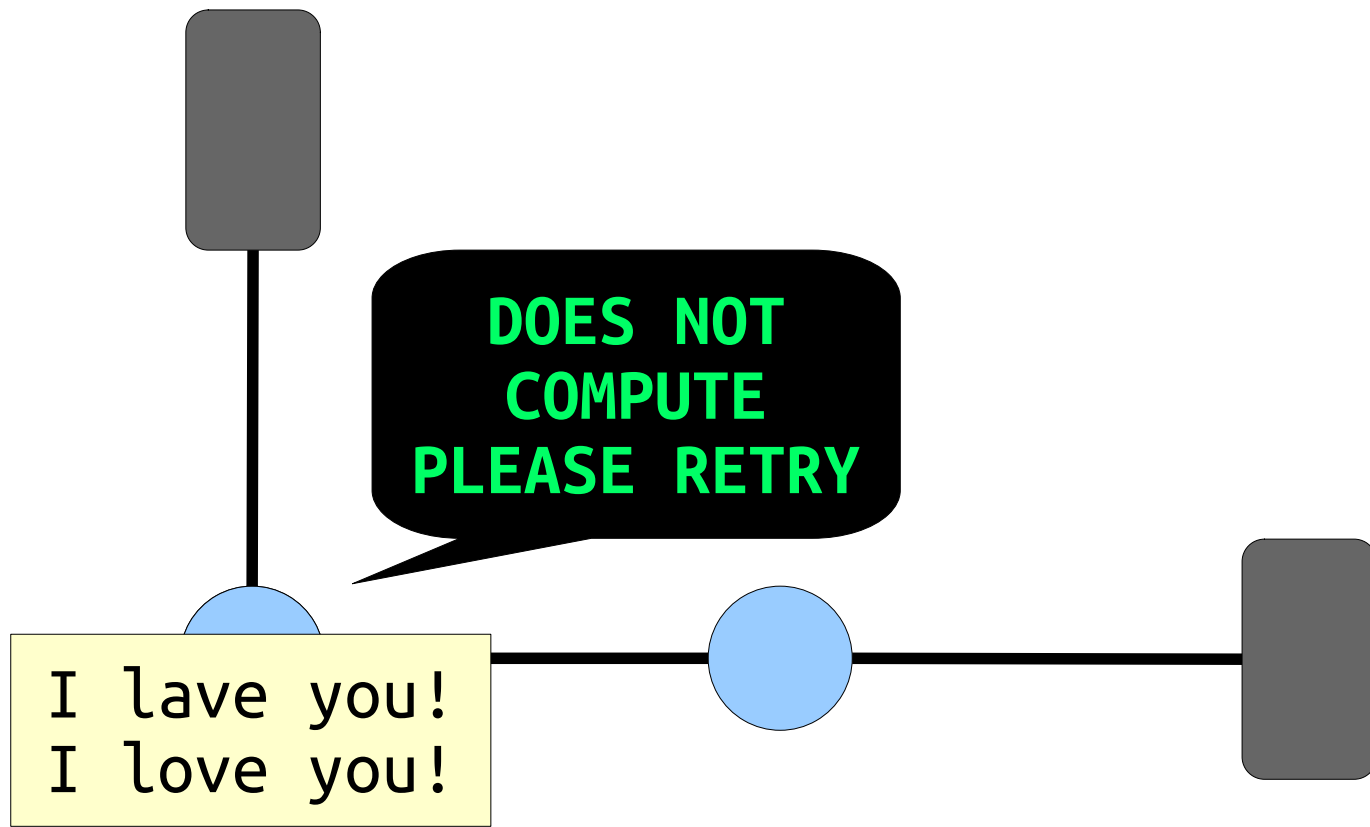
Did my data make it through the network?



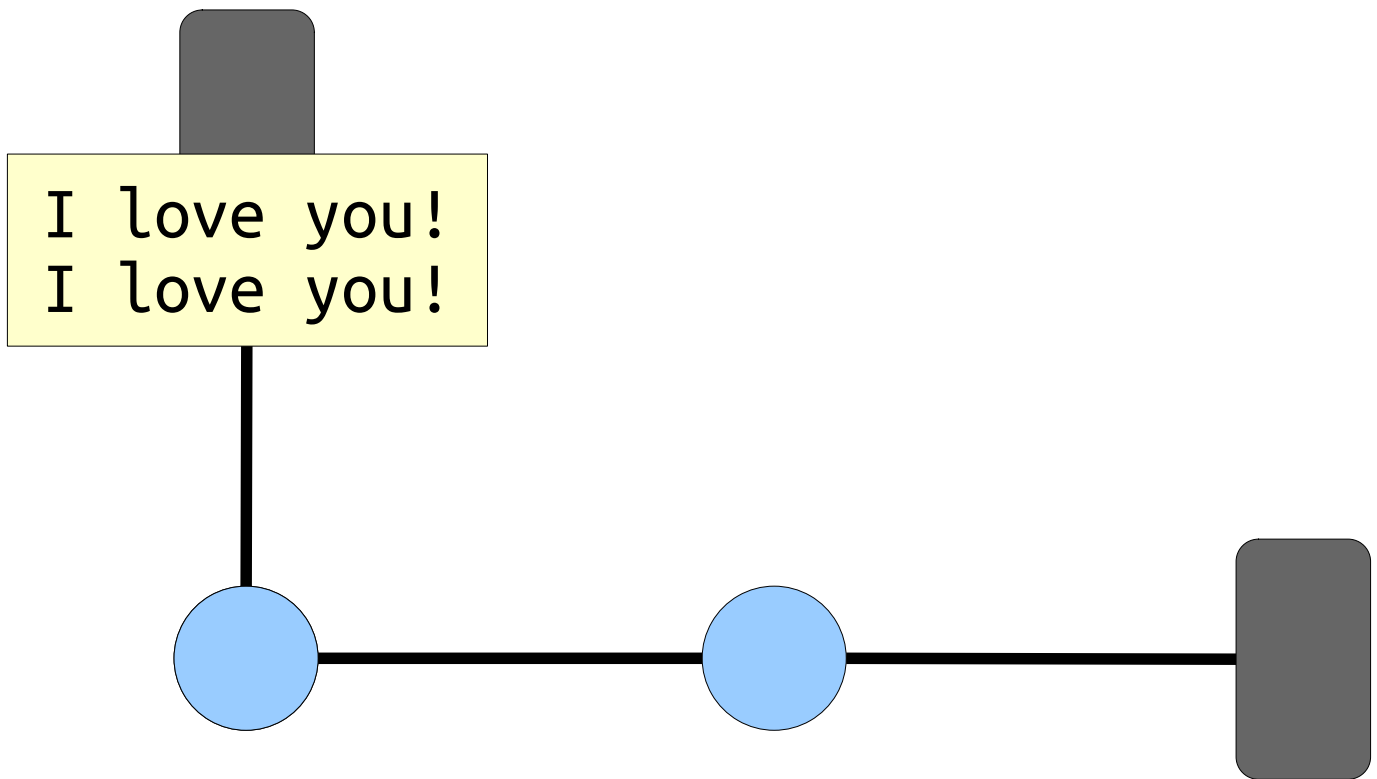
Did my data make it through the network?



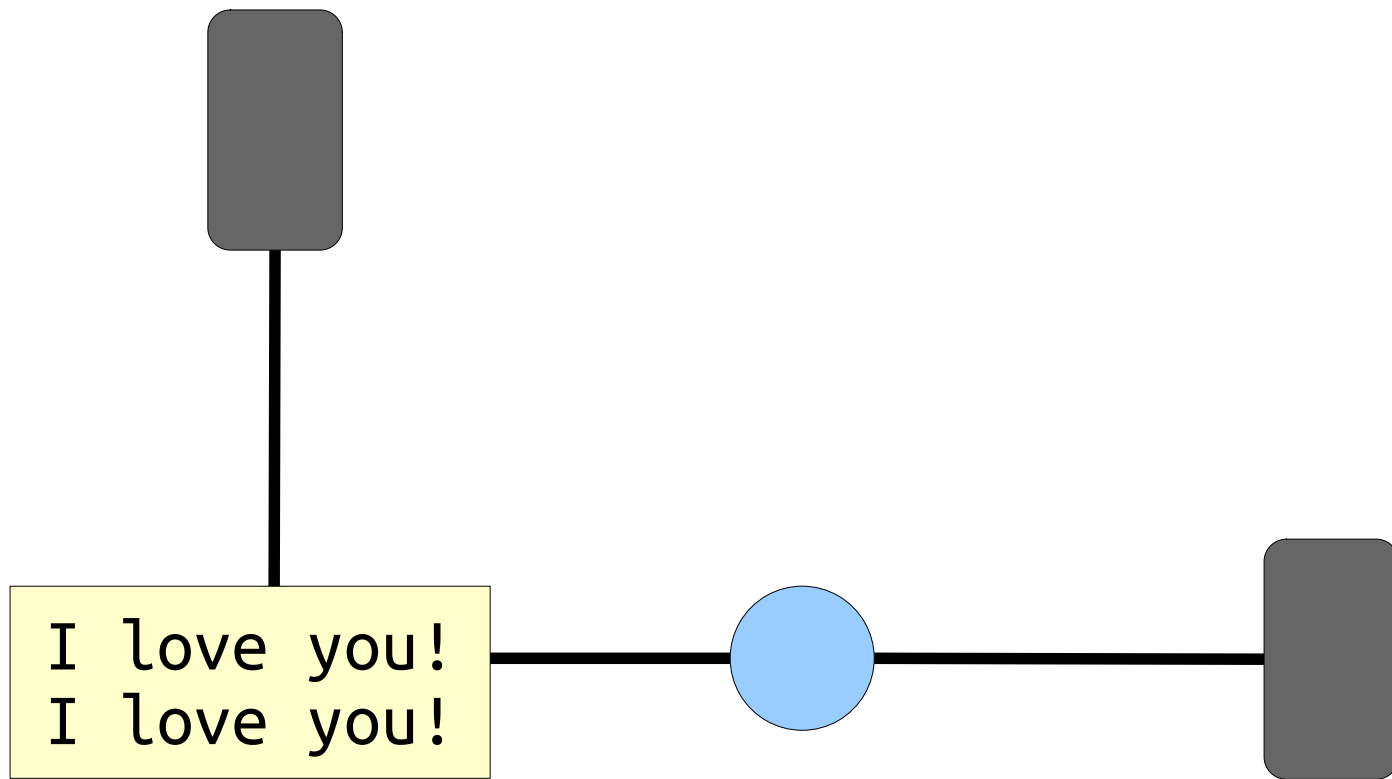
Did my data make it through the network?



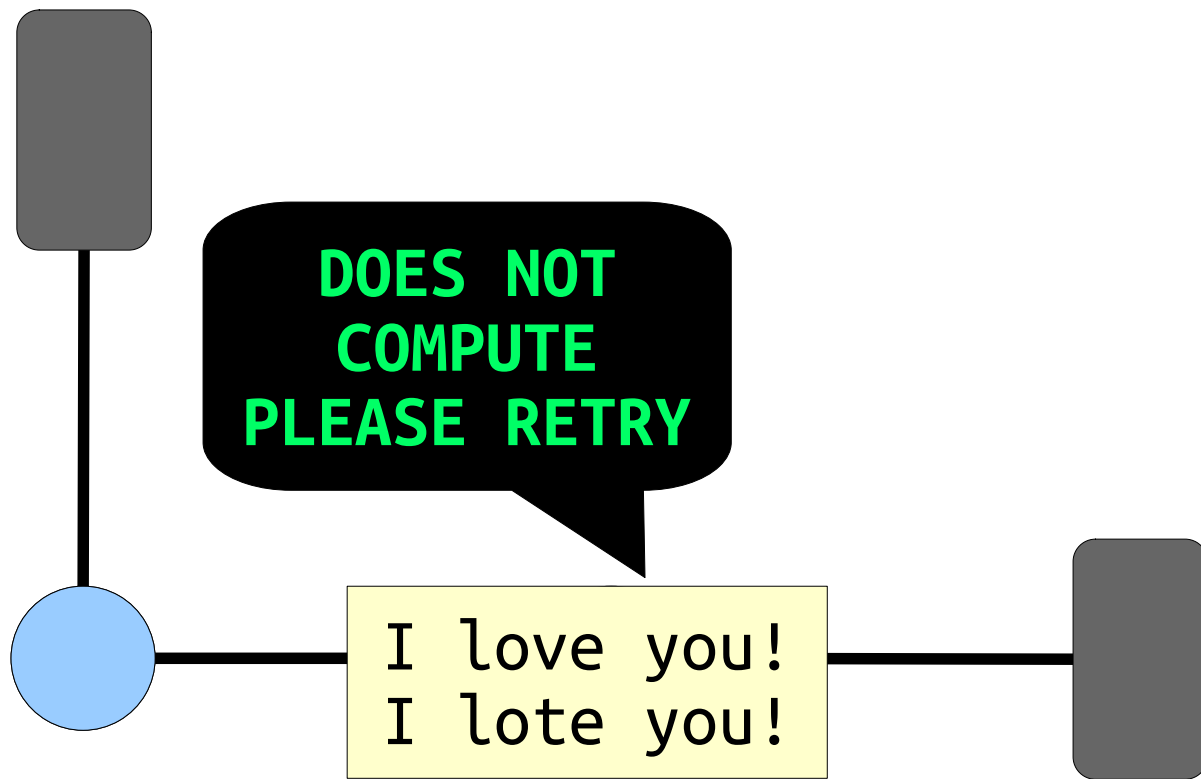
Did my data make it through the network?



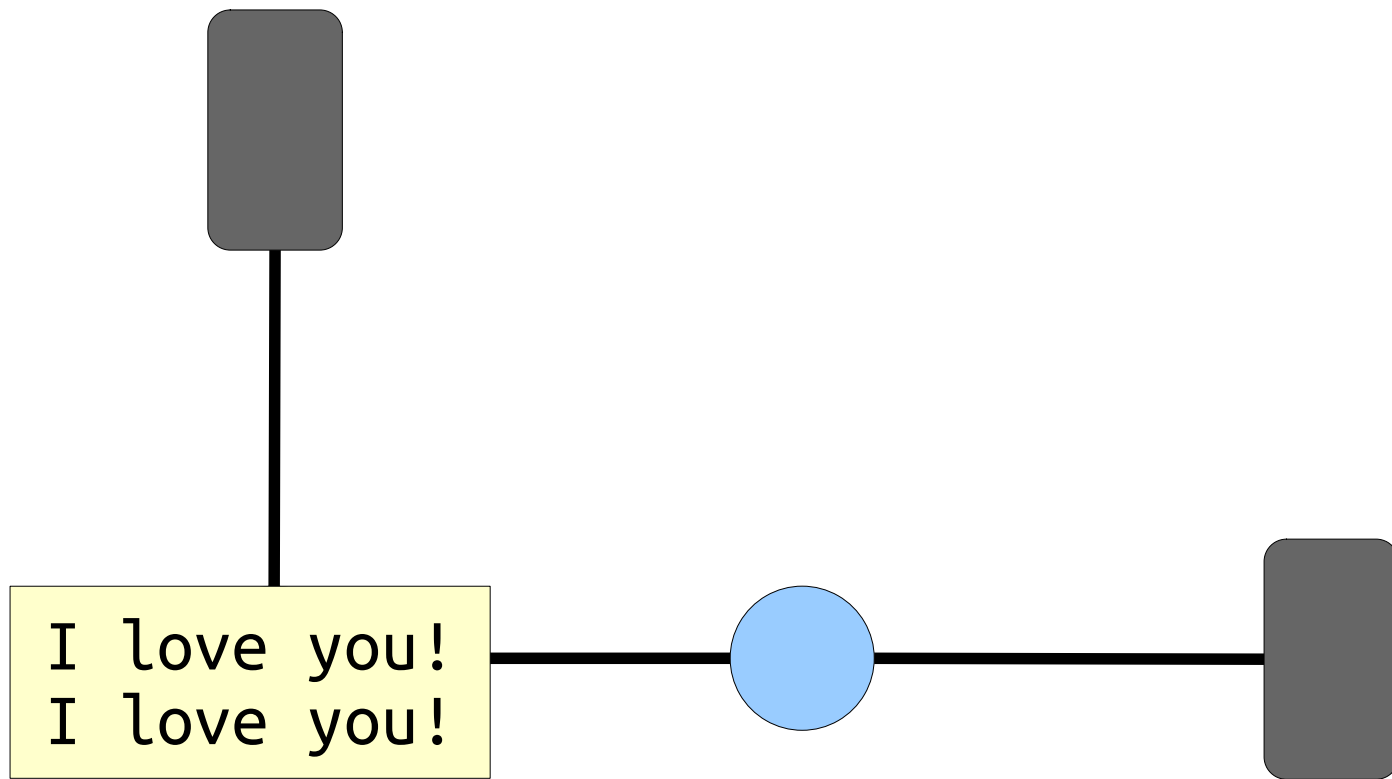
Did my data make it through the network?



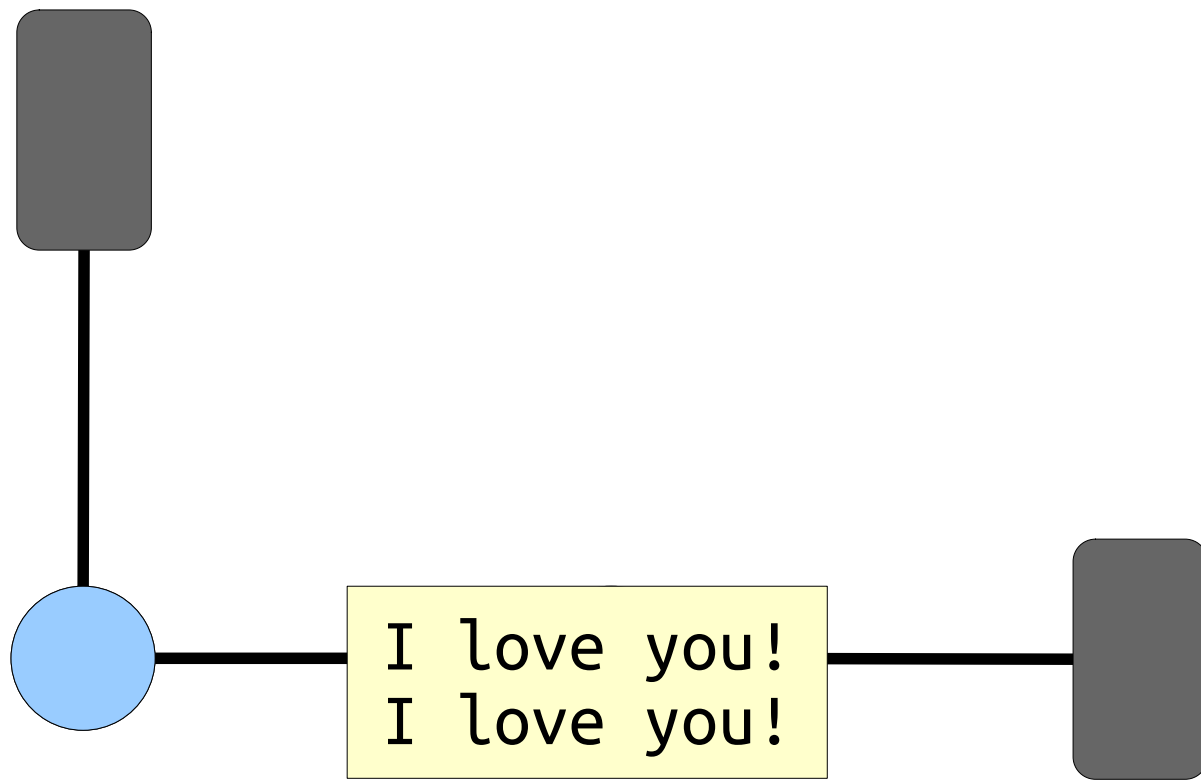
Did my data make it through the network?



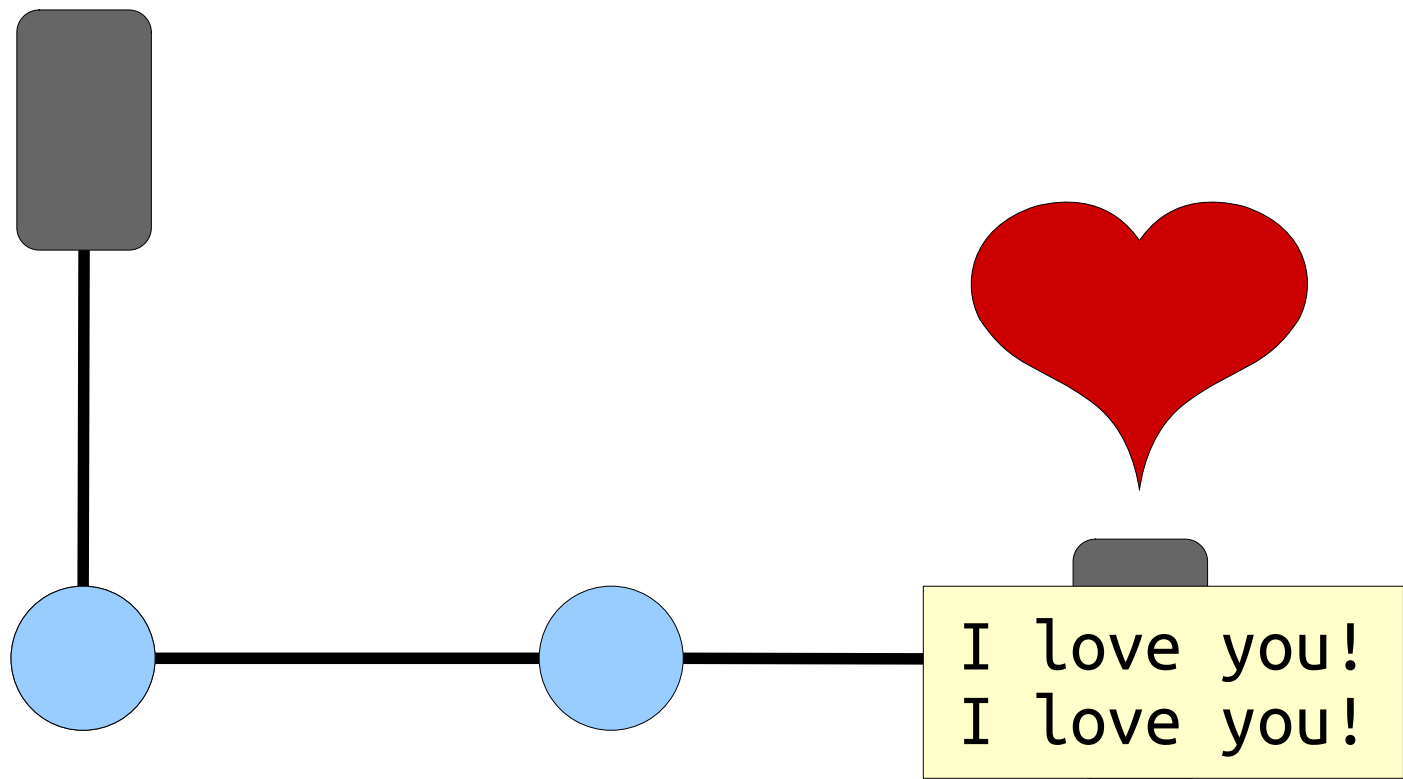
Did my data make it through the network?



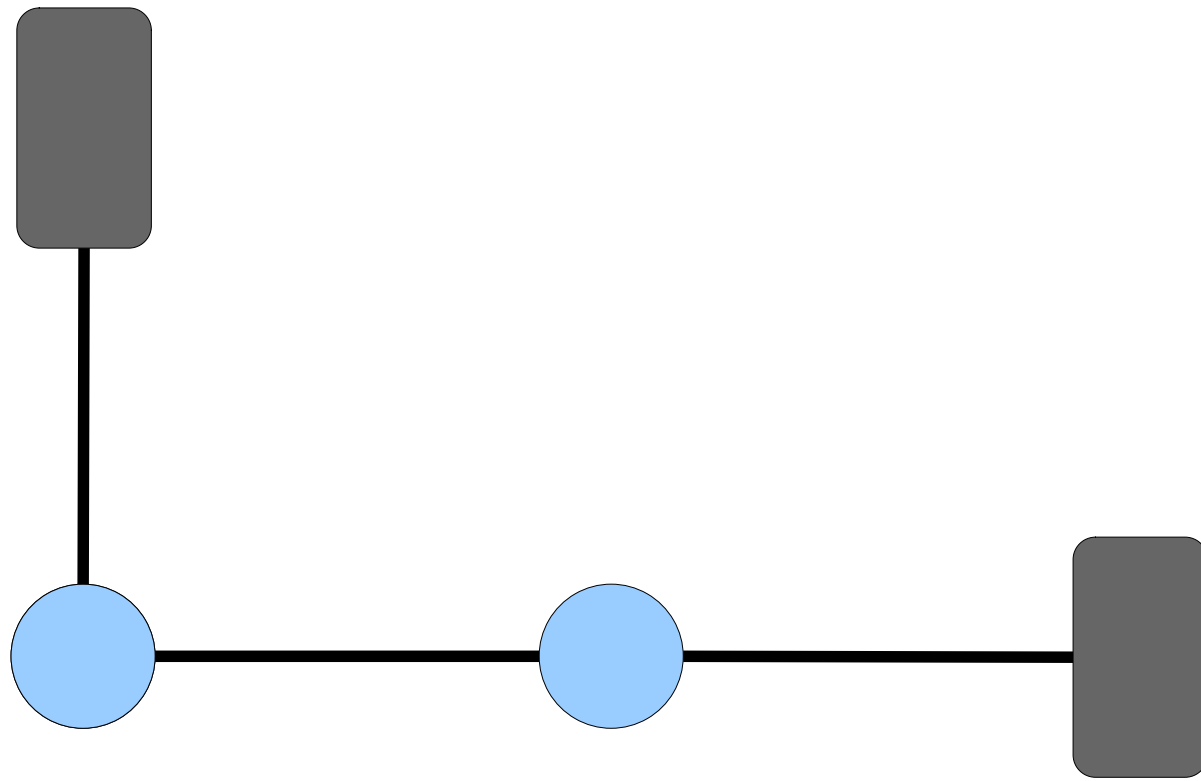
Did my data make it through the network?



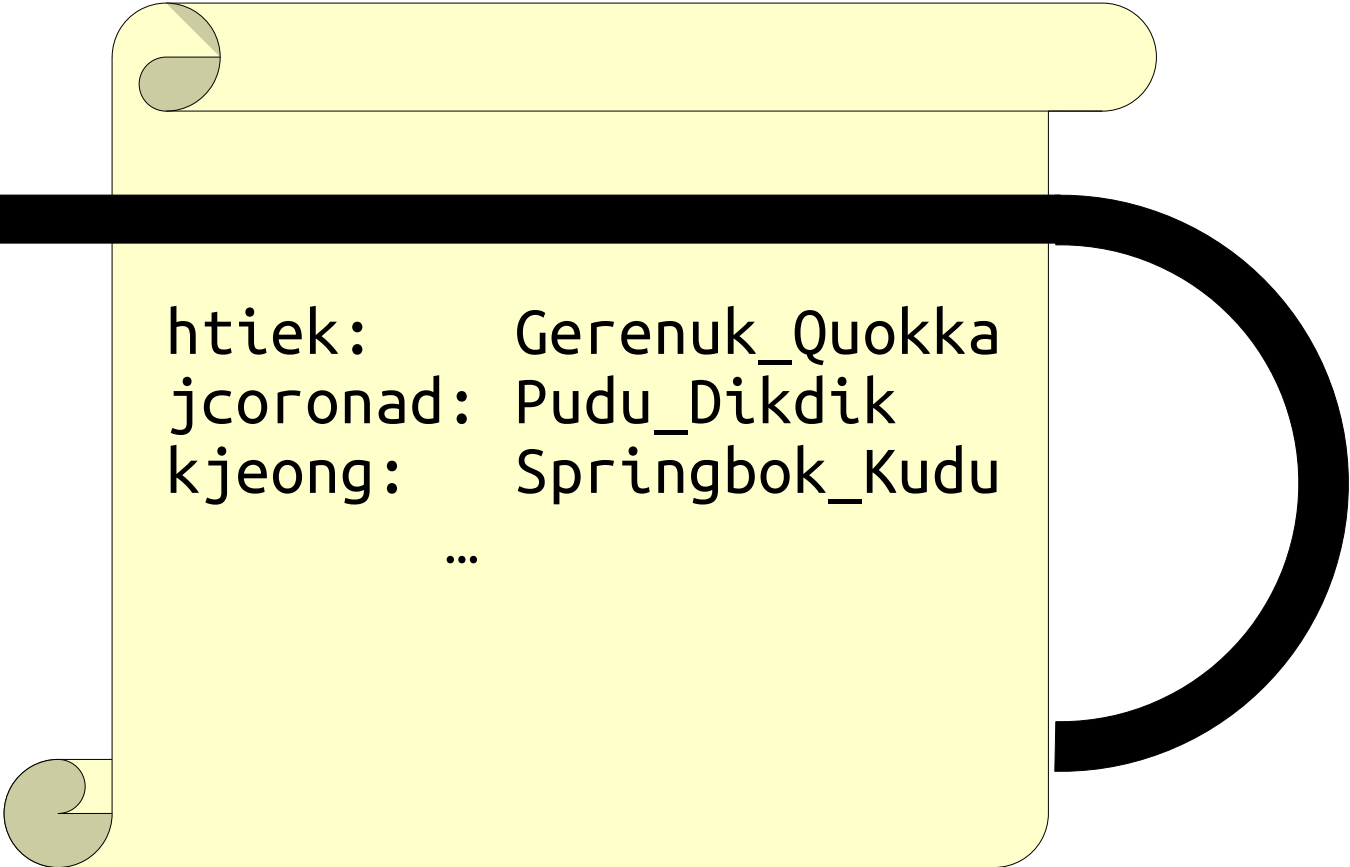
Did my data make it through the network?



Did my data make it through the network?

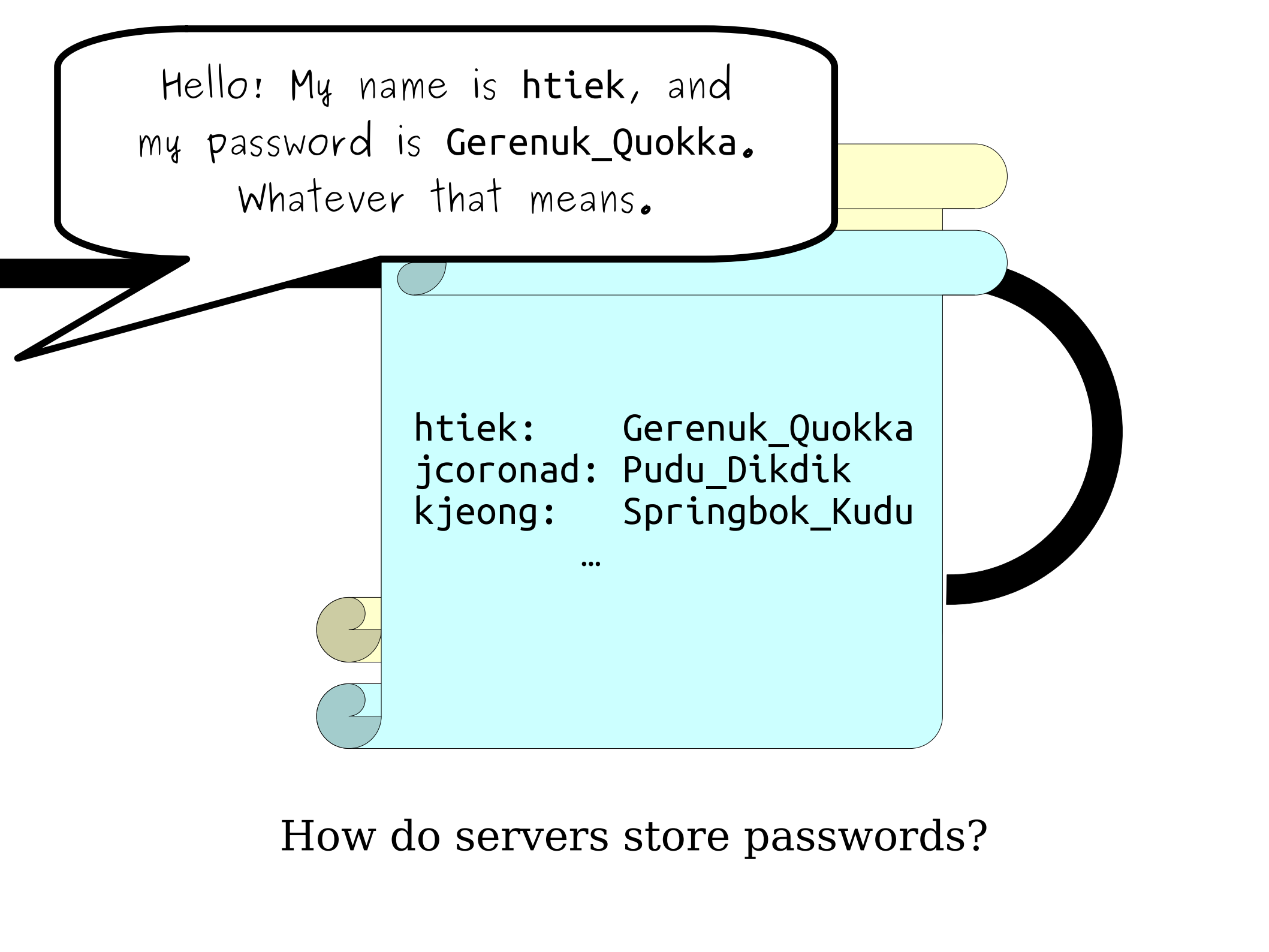


Can we do this without doubling the amount of data transmitted over the network?



```
htiek:      Gerenuk_Quokka  
jcoronad:  Pudu_Dikdik  
kjeong:    Springbok_Kudu  
          ...
```

How do servers store passwords?



Hello! My name is htiiek, and
my password is Gerenuk_Quokka.
Whatever that means.

```
htiek:      Gerenuk_Quokka  
jcoronad:  Pudu_Dikdik  
kjeong:    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  
jcoronad:  Pudu_Dikdik  
kjeong:    Springbok_Kudu  
...
```

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

Way Back When...

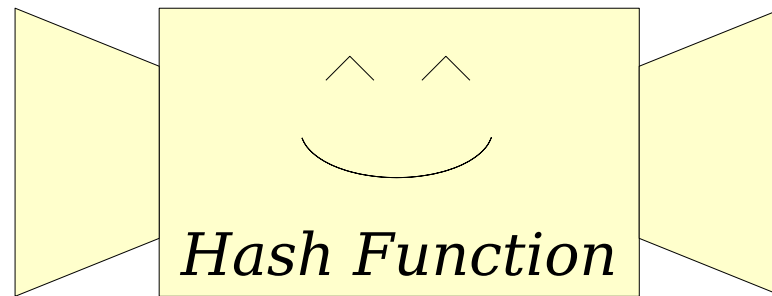
```

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;

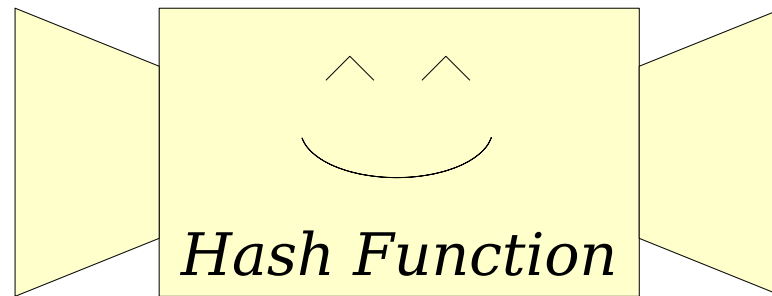
    int hashVal = 0;

    /* 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;
}

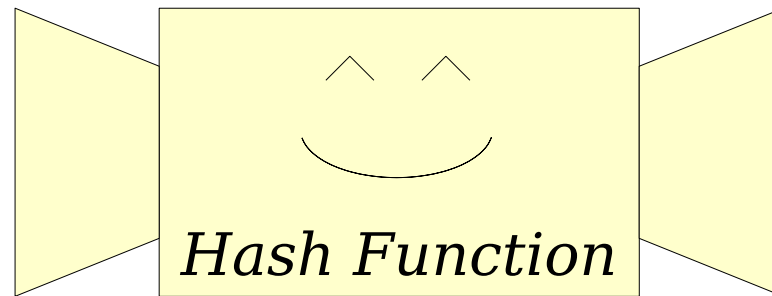
```



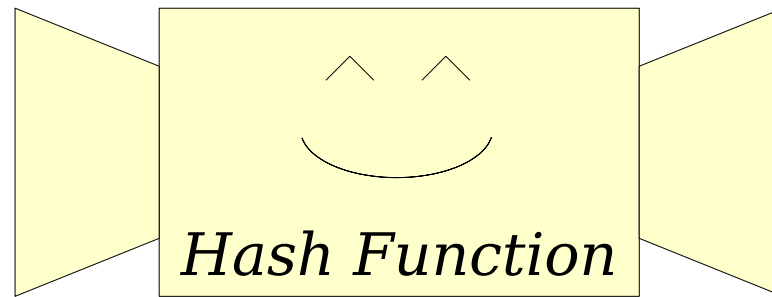
This is a ***hash function***. It's a type of function some smart math and CS people came up with.



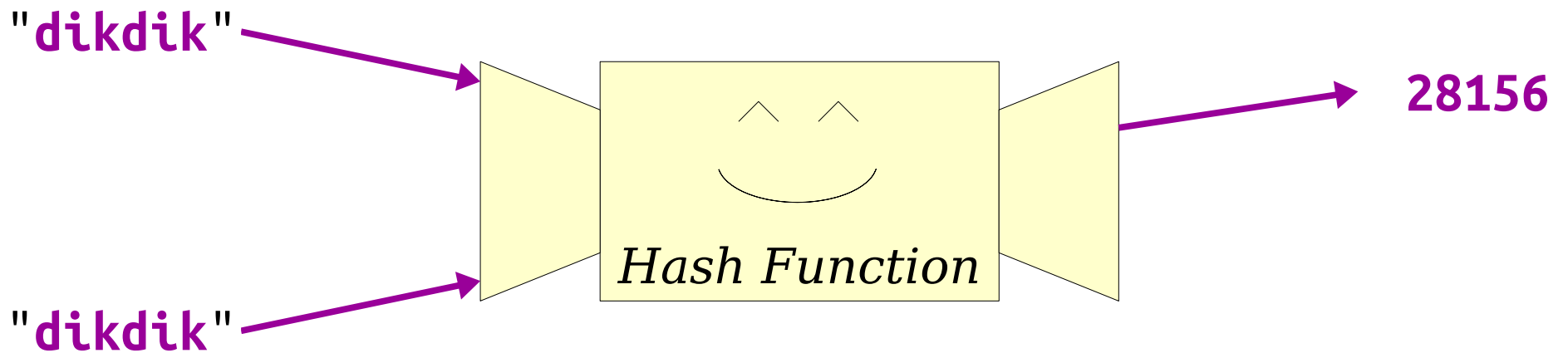
Most hash functions return a number.
In CS106B, we'll use the `int` type.



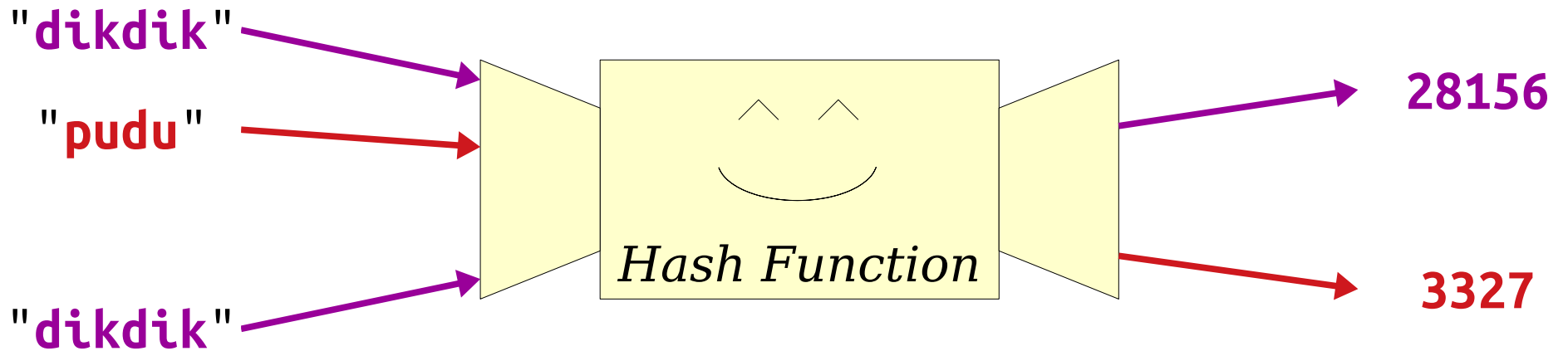
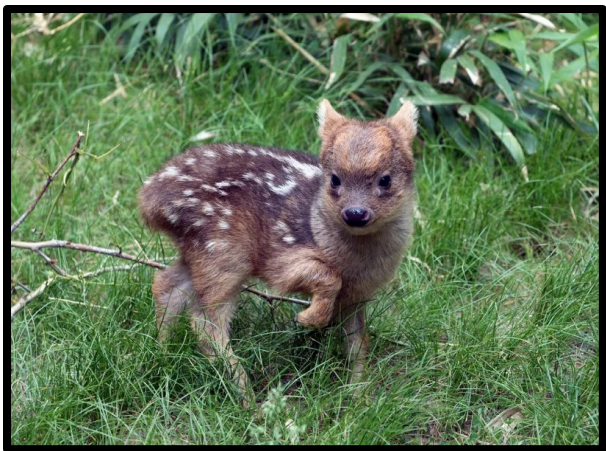
Different hash functions take inputs of different types.
In this example, we'll assume it takes string inputs.



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.



"dikdik"



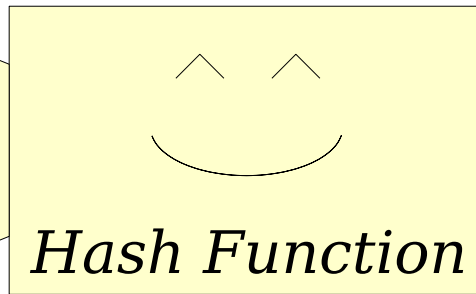
"pudu"



"kudu"



"dikdik"



28156



13985



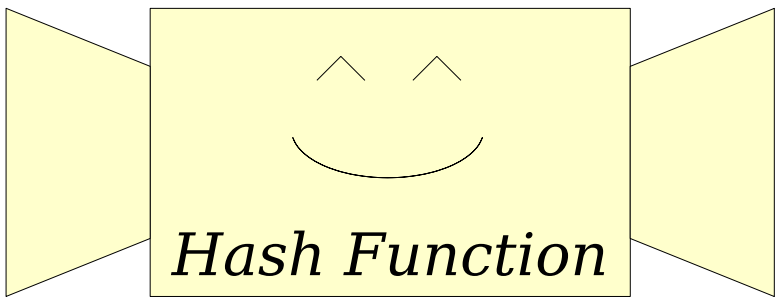
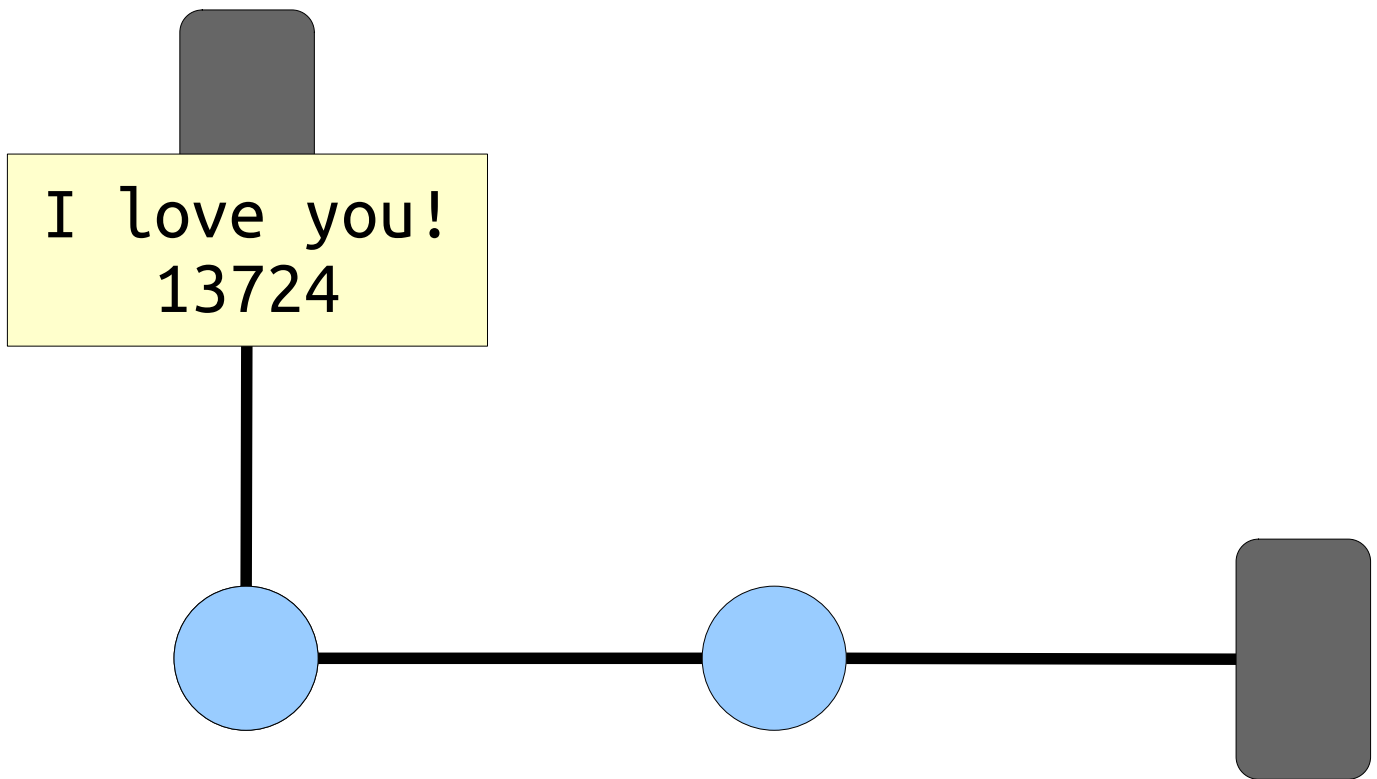
3327

Even very similar inputs give
very different outputs!

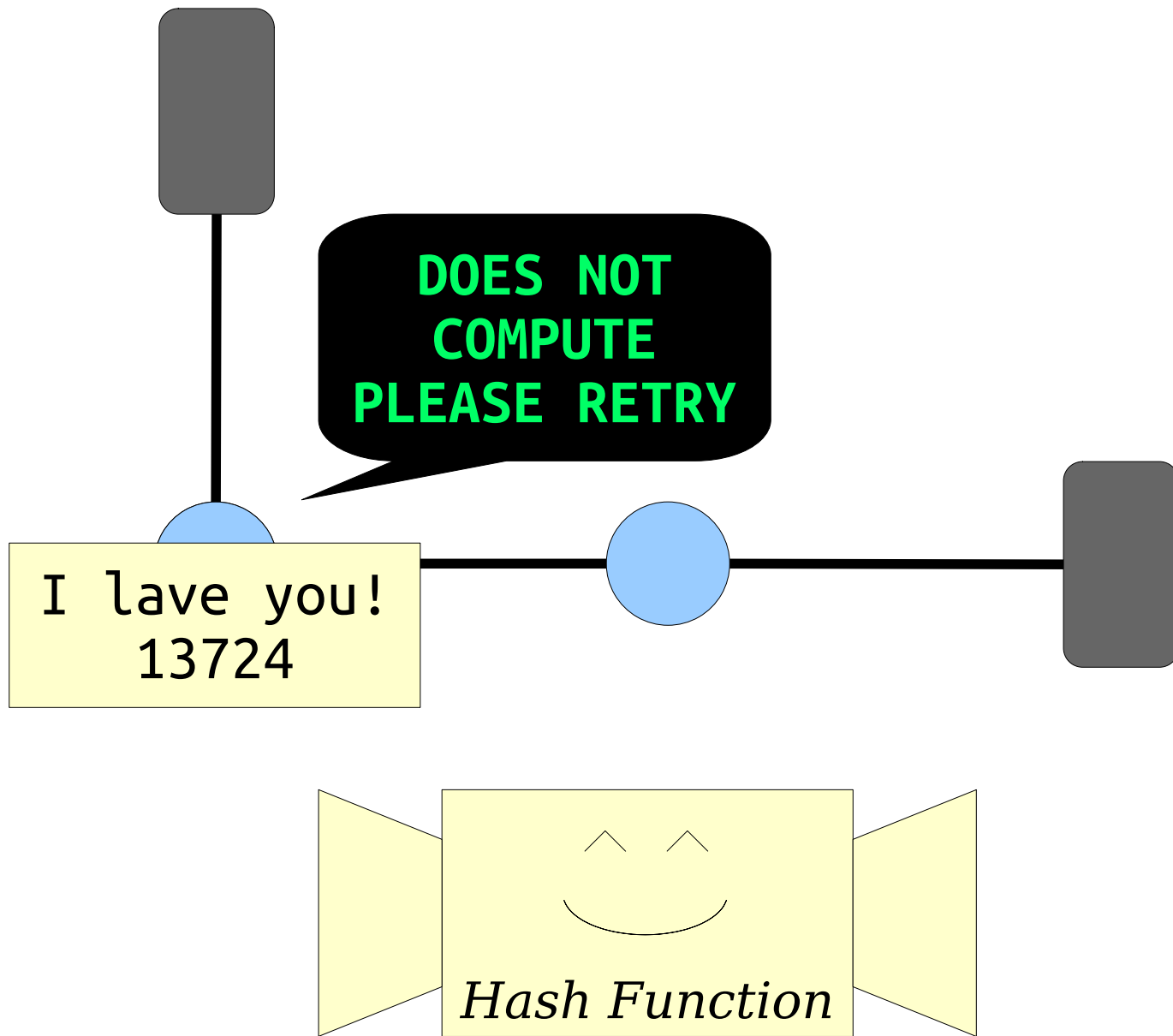
To Recap:

Equal inputs give equal outputs.

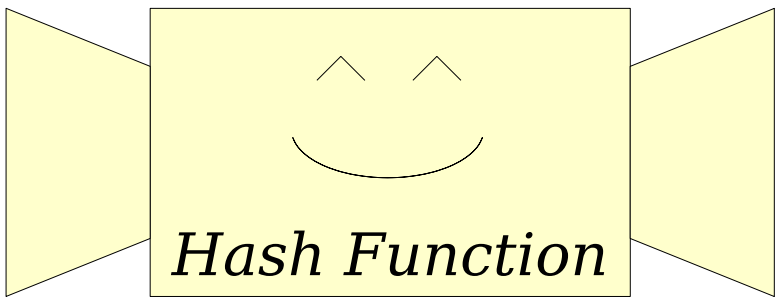
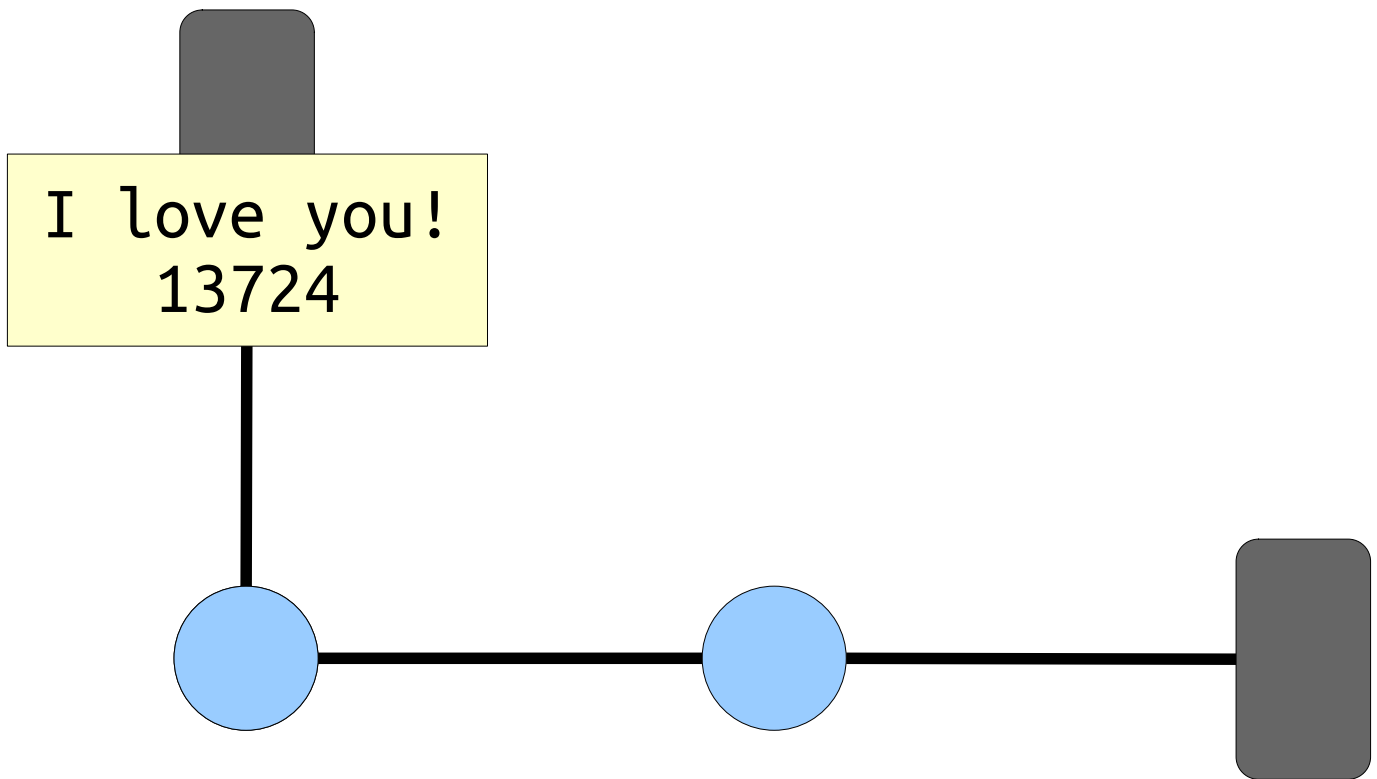
Unequal inputs (usually) give
very different outputs.



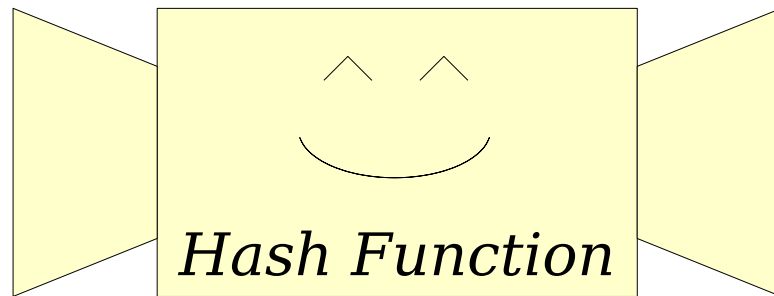
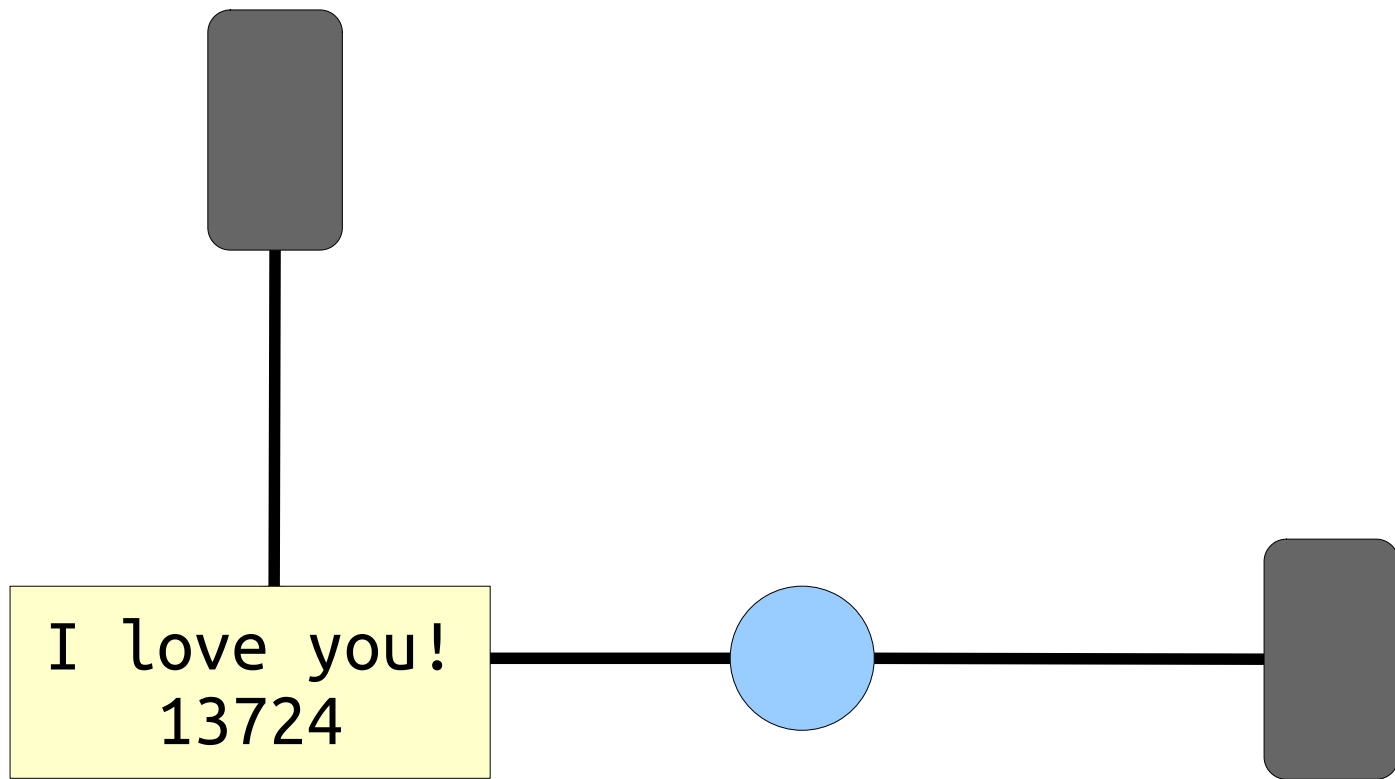
Did my data make it through the network?



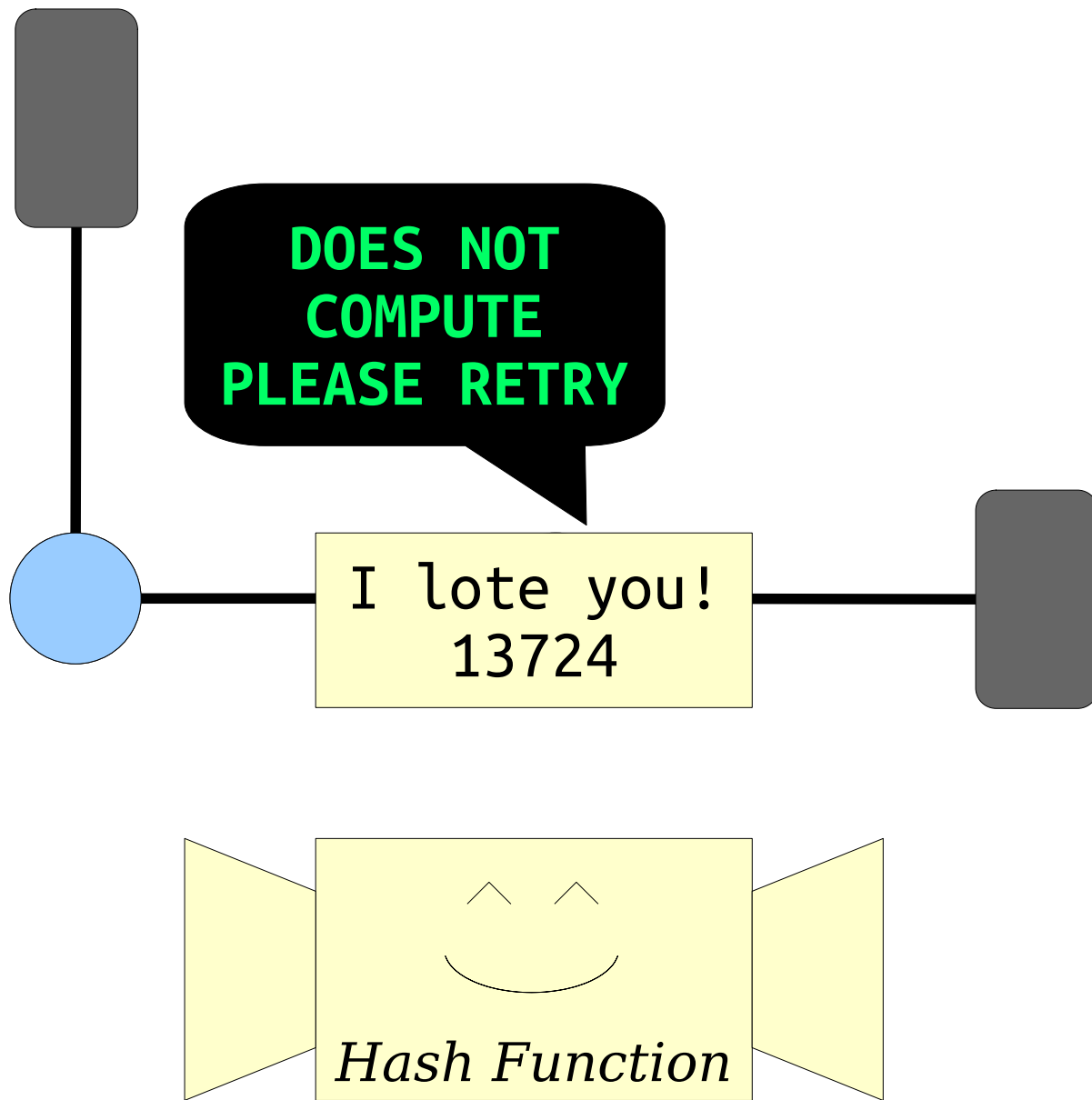
Did my data make it through the network?



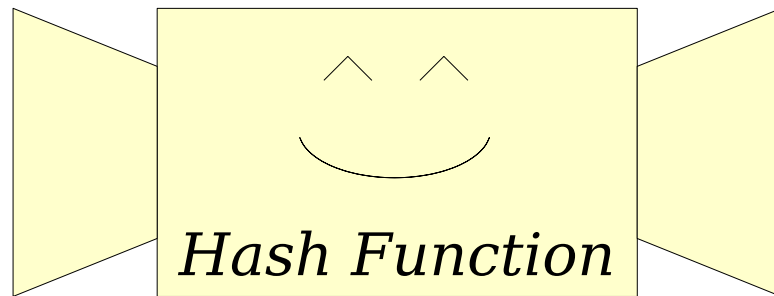
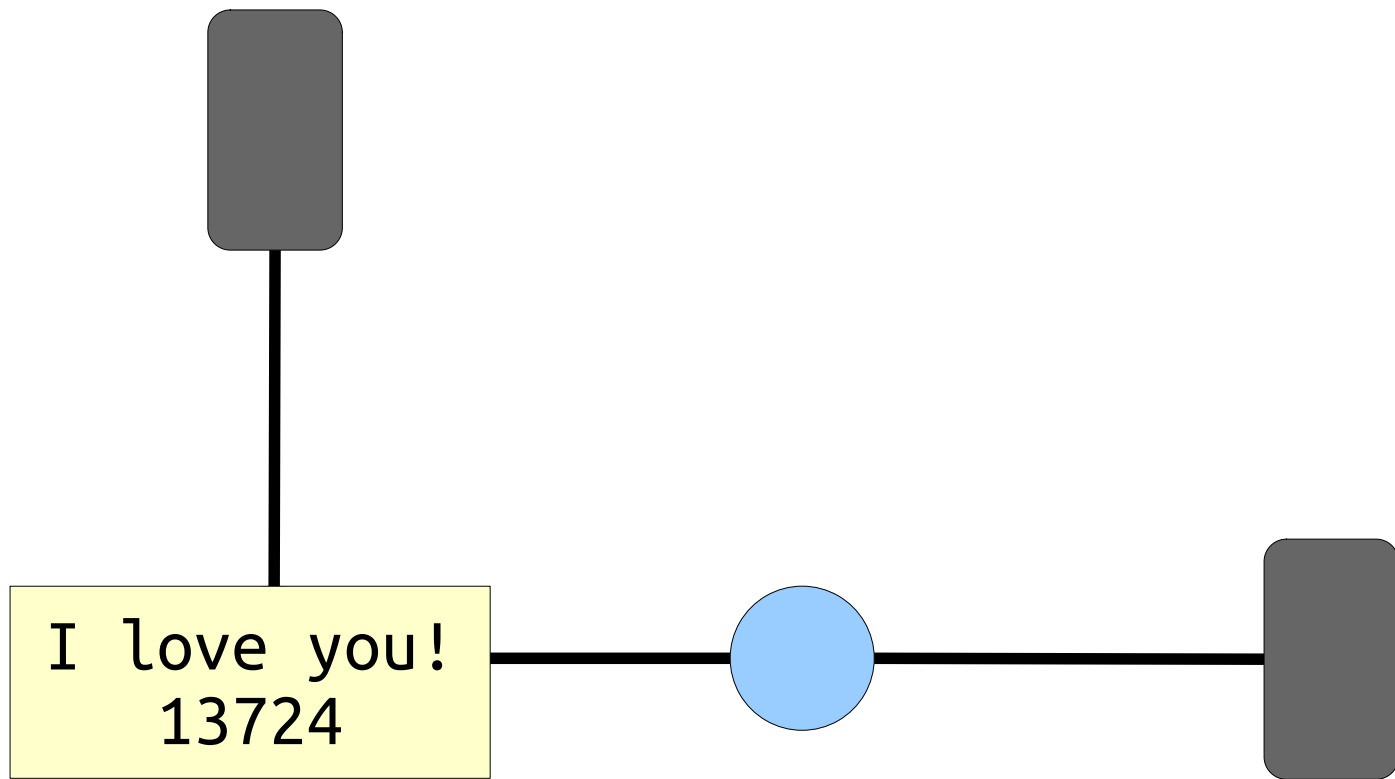
Did my data make it through the network?



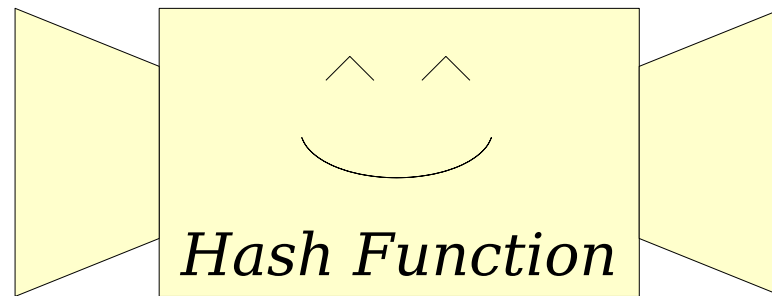
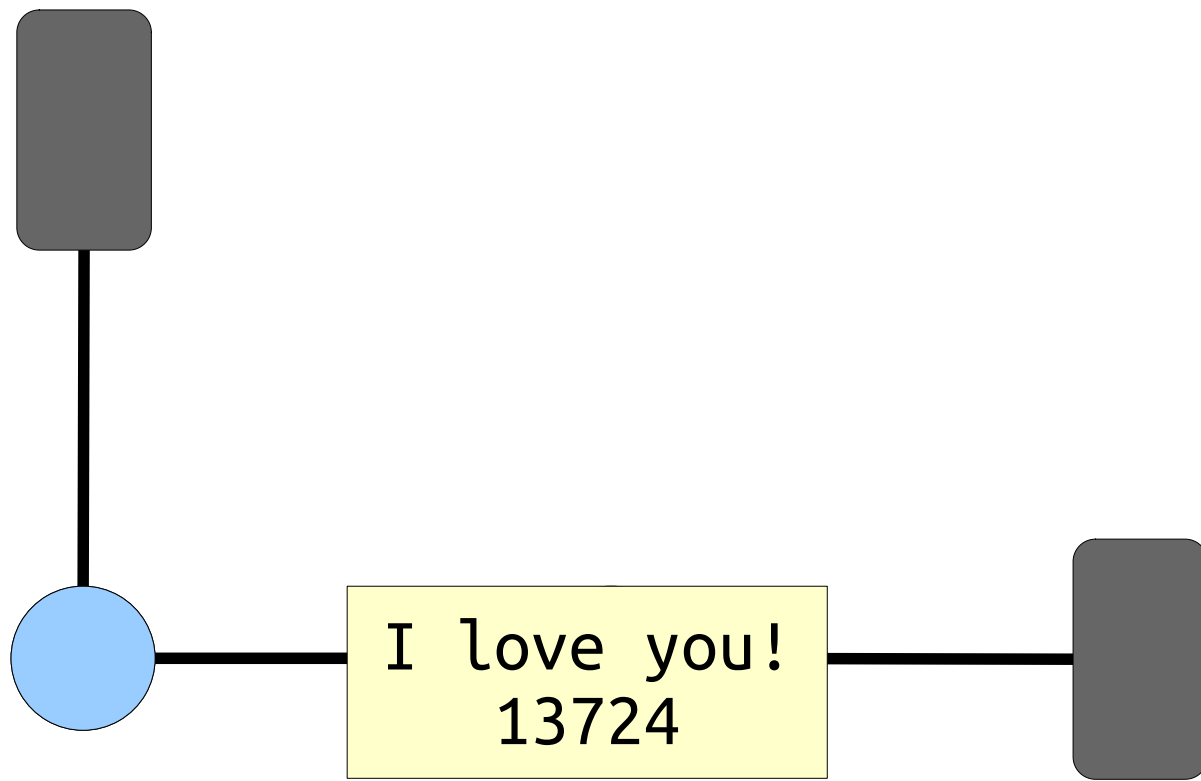
Did my data make it through the network?



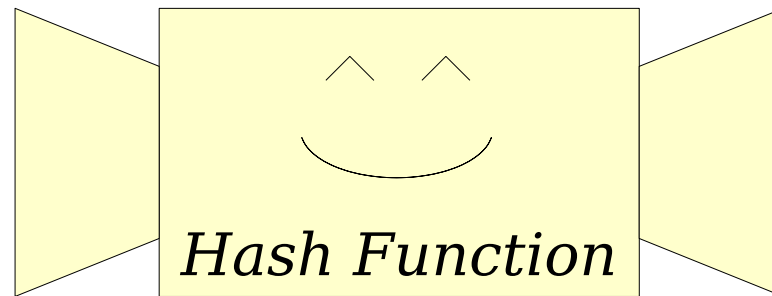
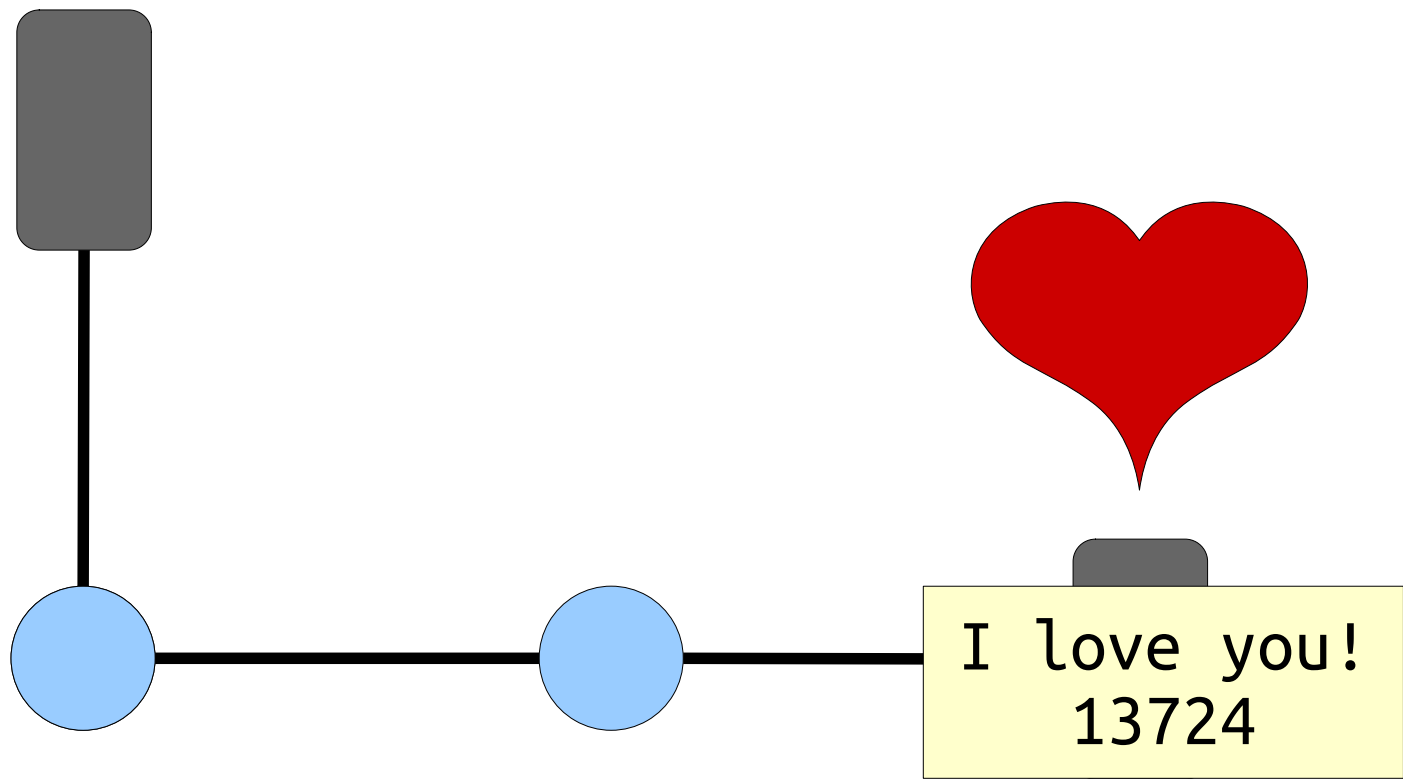
Did my data make it through the network?



Did my data make it through the network?



Did my data make it through the network?



Did my data make it through the network?

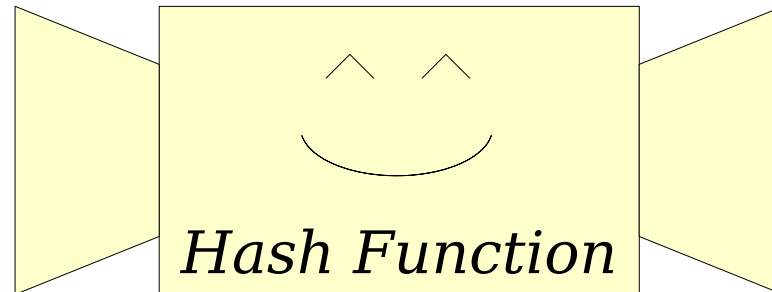
This is done in practice!

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

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

```
htiek:      2915738932396303  
jcoronad:  5416204120152480  
kjeong:    3096517106352733
```

...



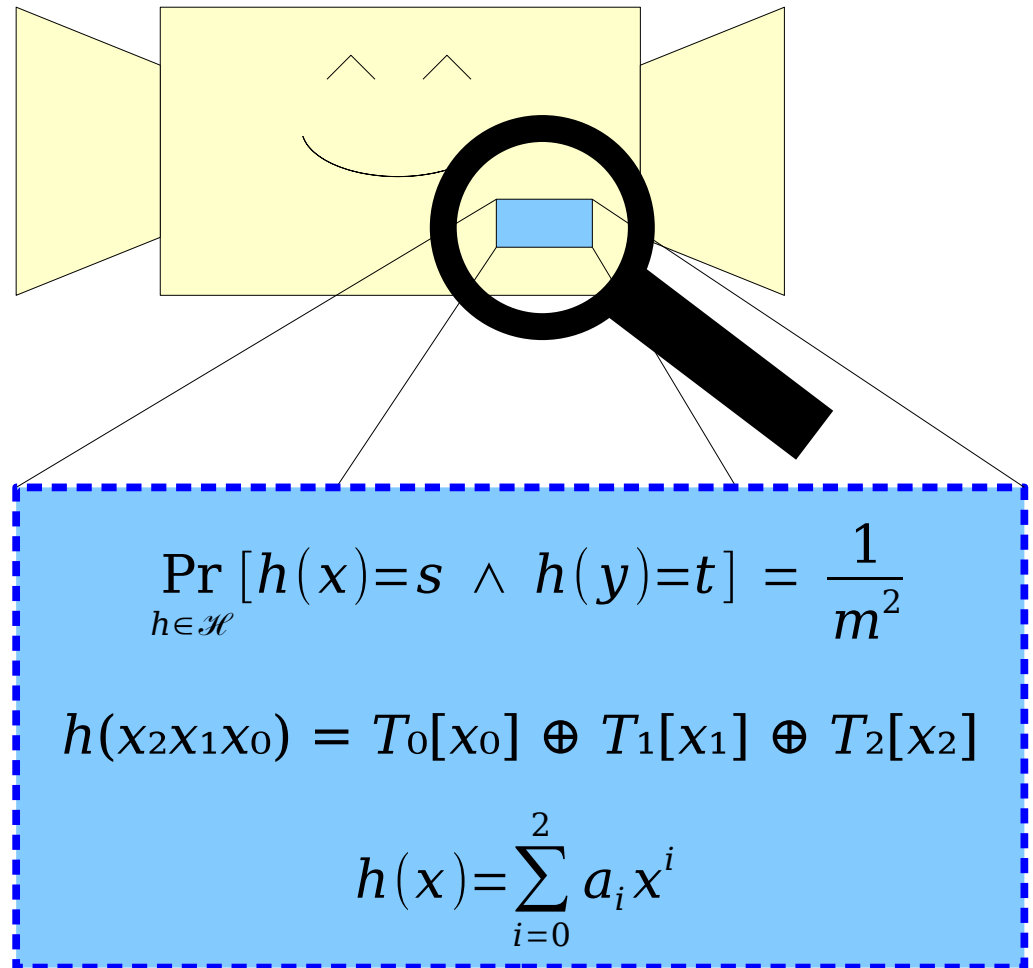
How do servers store passwords?

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

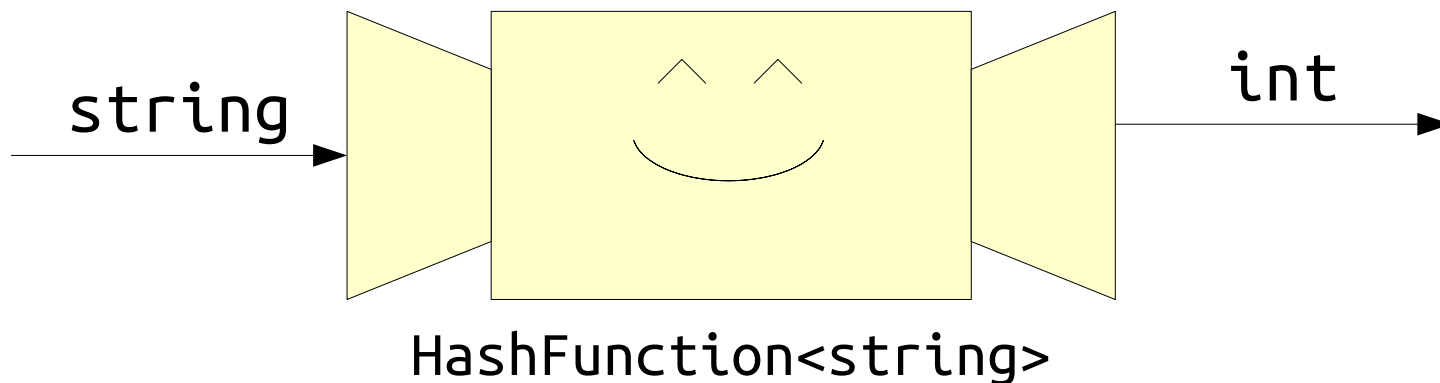
- Designing good hash functions is challenging, and it's beyond the scope of what we'll explore in CS106B.
- We will assume that some Smart, Attractive, Witty person has created the hash functions we'll use this quarter and won't look into how they work.
- Like finite fields and abstract algebra? Stick around after class and I can share more of the technical details.



Working with Hash Functions

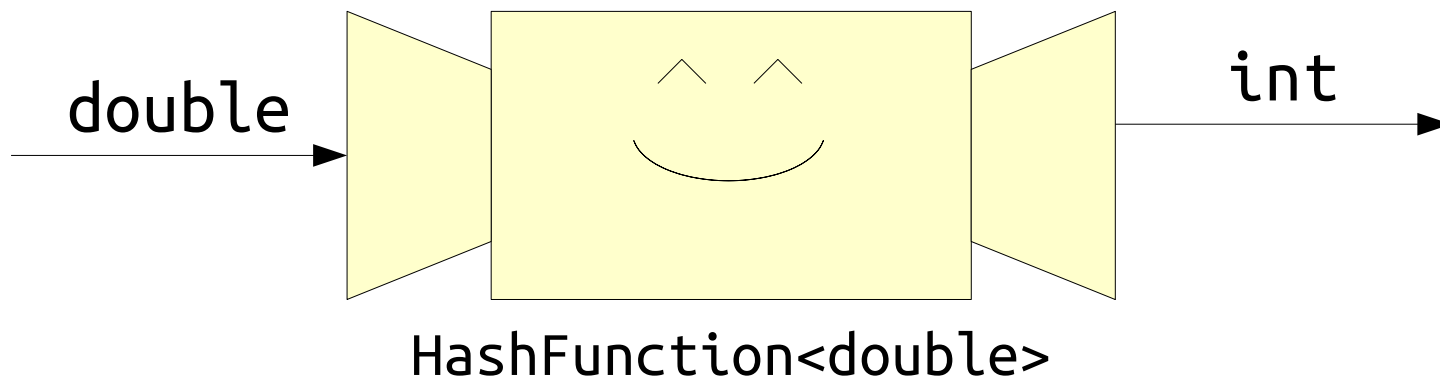
Working with Hash Functions

- 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>`.



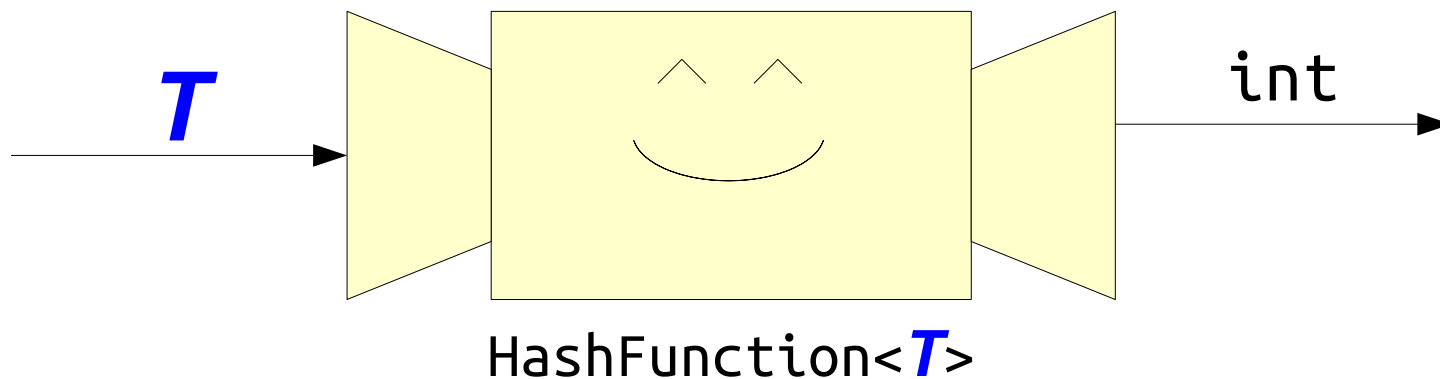
Working with Hash Functions

- 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

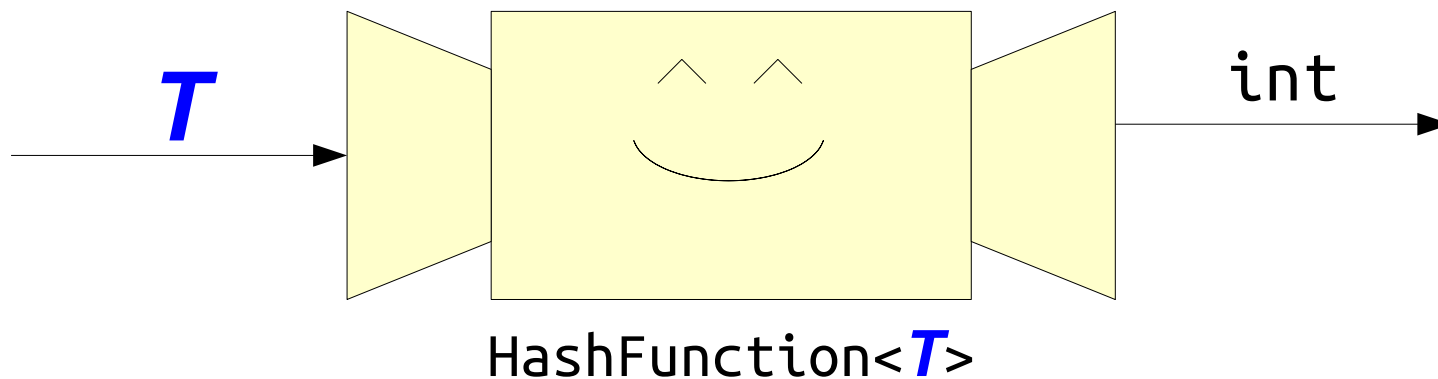
- Every programming language has a different way for programmers to work with hash functions.
- In CS106B, we'll represent hash functions using the type `HashFunction<T>`.



Working with Hash Functions

- In many applications, we need a hash function that outputs values in a small range.
- To create a hash function that outputs values between 0 and $n - 1$, inclusive, use this syntax:

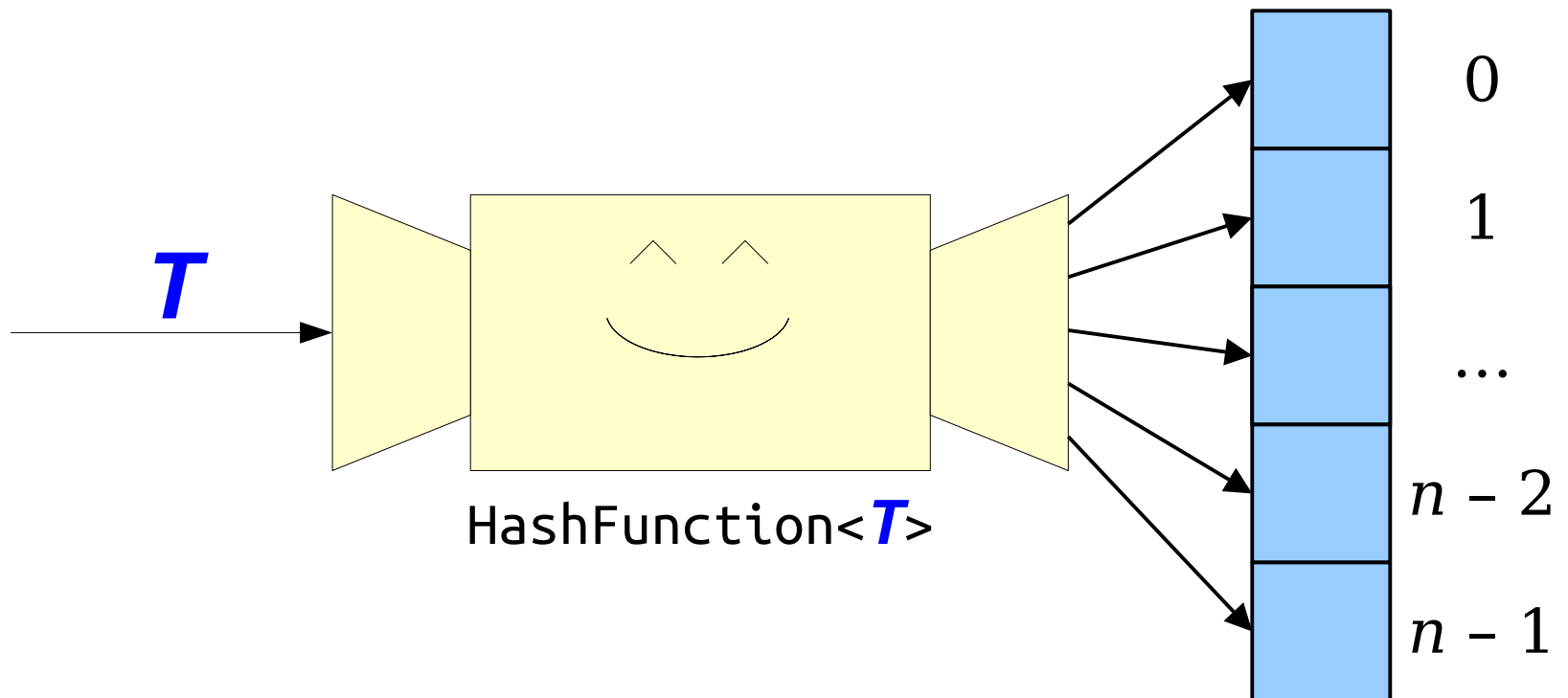
```
HashFunction<T> hashFn = forSize(n);
```



Working with Hash Functions

- In many applications, we need a hash function that outputs values in a small range.
- To create a hash function that outputs values between 0 and $n - 1$, inclusive, use this syntax:

```
HashFunction<T> hashFn = forSize(n);
```



Hash Collisions

- A ***hash collision*** is a pair of inputs to a hash function that produce the same outputs.
- When working with hash functions over a constrained range, hash collisions are unavoidable.
- This isn't the fault of the hash function. If you only have n possible outputs and drop in $n+1$ inputs, you're guaranteed to have a collision.

Hash Collisions

- Think back to the two examples we saw earlier (sending data and storing passwords).
- What bad things might happen in those examples if there are hash collisions?

Answer at

<https://cs106b.stanford.edu/pollev>

Time-Out for Announcements!

Midterm Debrief

- We graded the midterm exam over the weekend and scores are now available on Gradescope. Solutions and statistics are up on the course website.
- Please reach out to Jonathan, to your section leader, or to me if you want to set up a time to chat about your exam.
- Regrade requests are open. Check EdStem for information about how to submit a request.

Midquarter Check-In

- This part of the quarter can be a stressful time.
- We are all part of a broader campus community and we all need to look out for each other.
- If you're feeling stressed or overwhelmed, please feel free to reach out to me. I'm happy to help however I can.
- If you know someone in your dorm who's having a rough time, check in with them and make sure they're doing okay.
- You are so much more than your academics and your well-being takes precedence over your coursework. If you're feeling like those are in tension, please reach out to me.

A Note on the Honor Code

Back to CS106B!

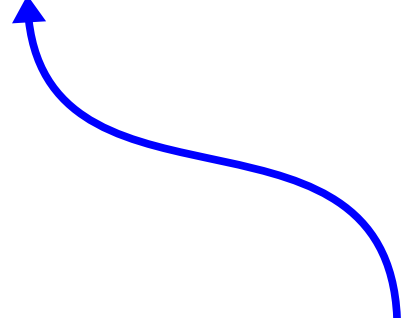
An Application:
Map and Set

```
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.


```
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? */

};
```

```
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).

Use a hash function!

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

buckets

[0]

[1]

[2]

[3]

[4]

[5]



buckets

[0]

[1]

[2]

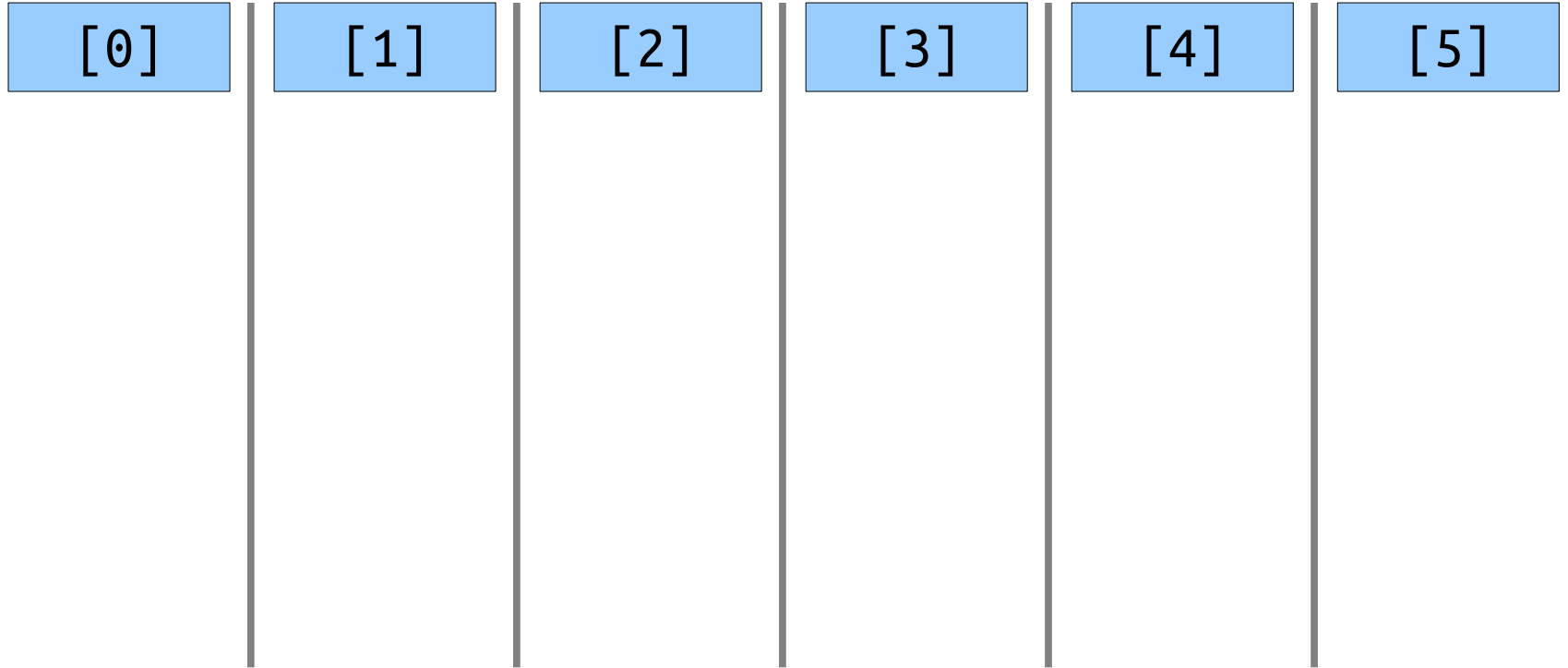
[3]

[4]

[5]

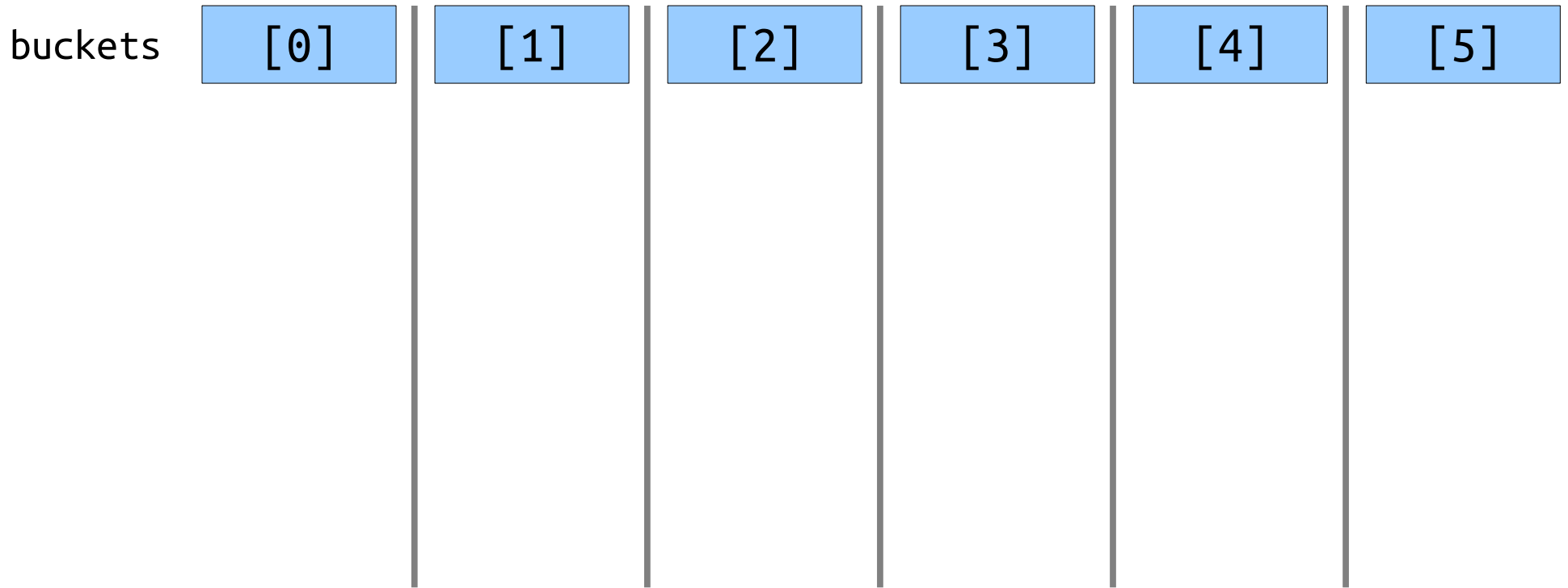
urania

buckets



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

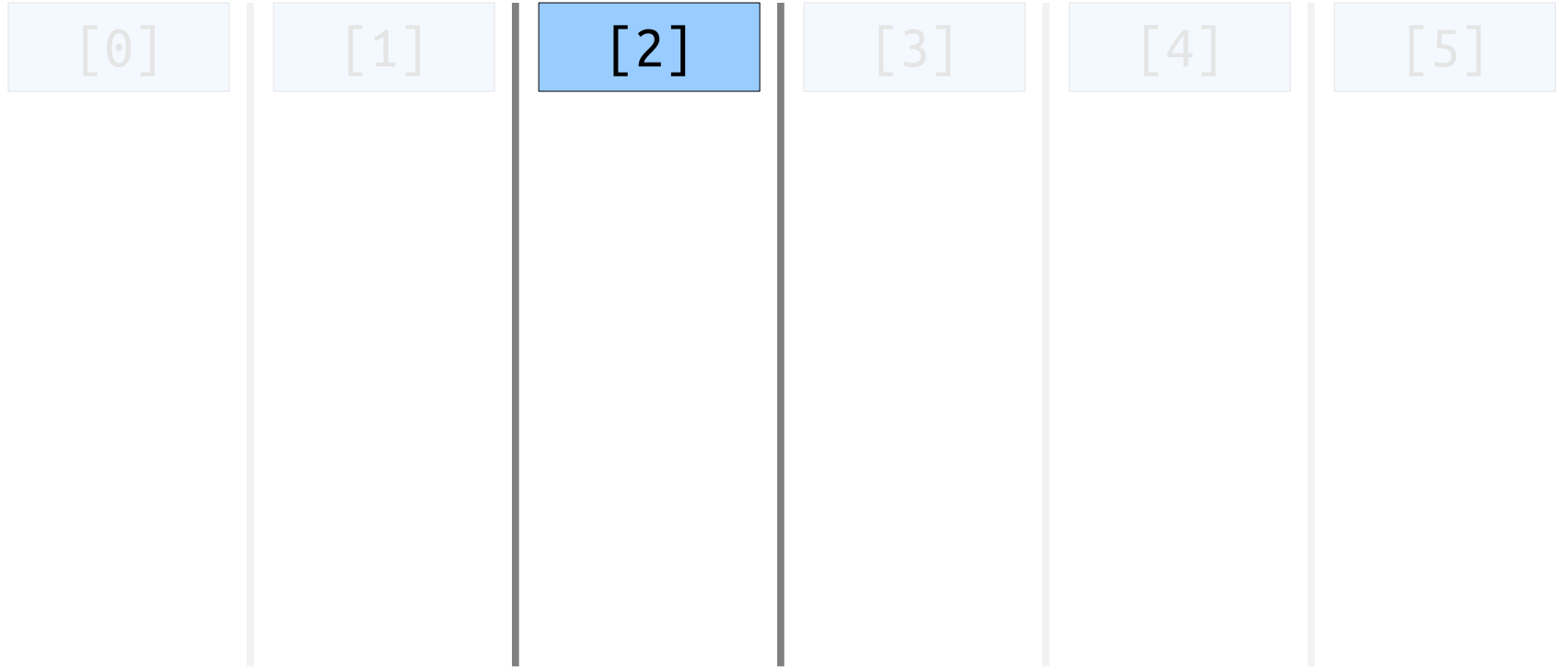
urania



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

urania

buckets

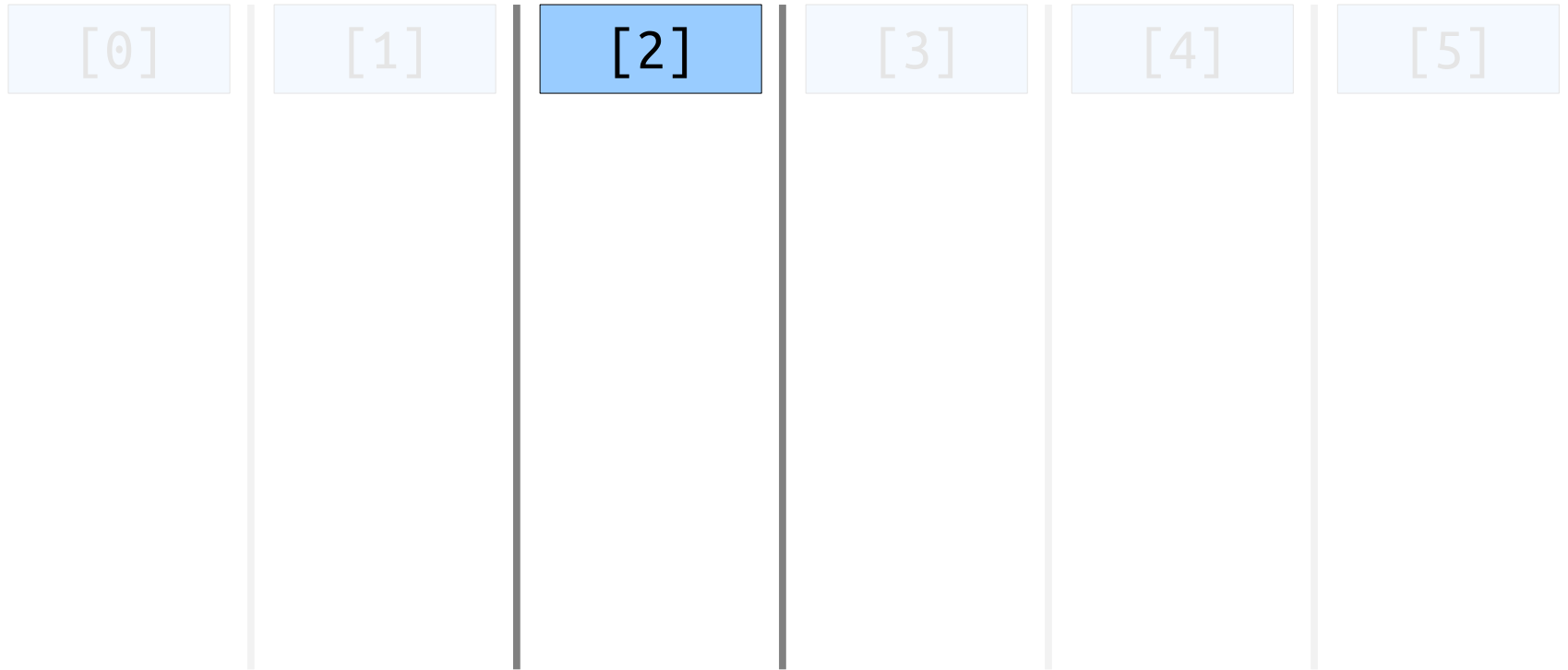


```
void OurSet::add(const string& value) {  
    int bucket = hashFn(value);  
  
}
```

urania

(bucket 2)

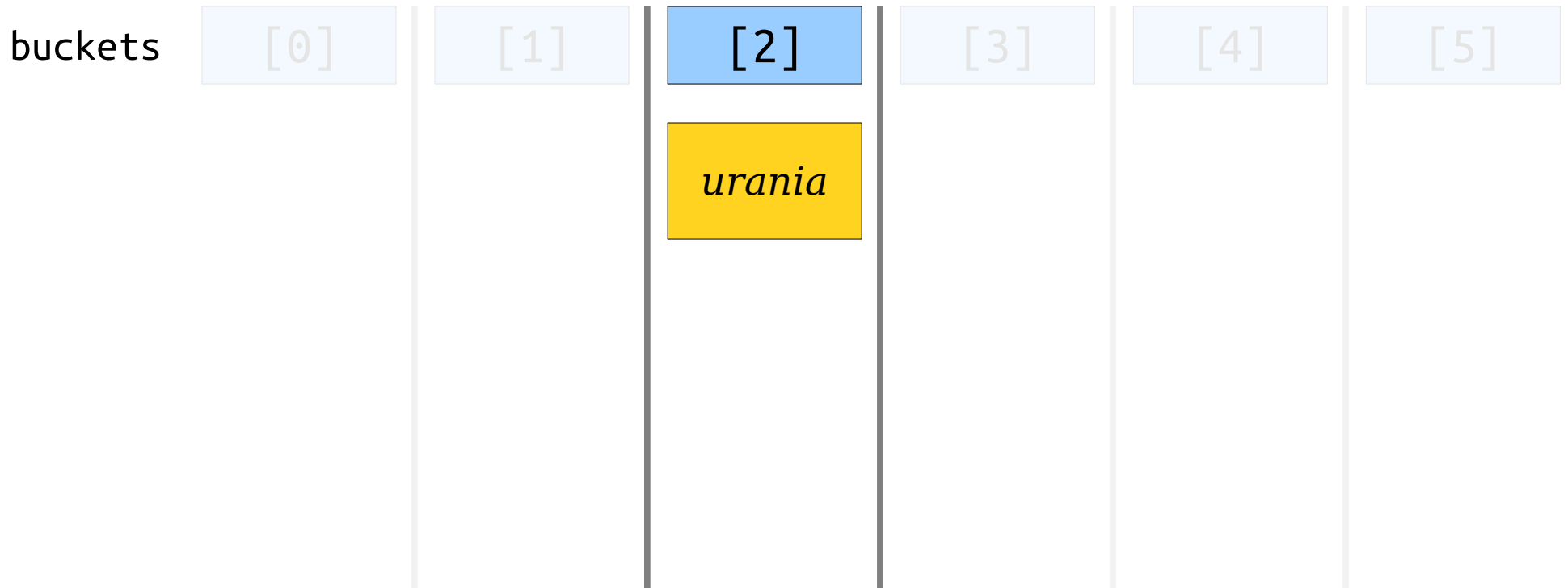
buckets



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

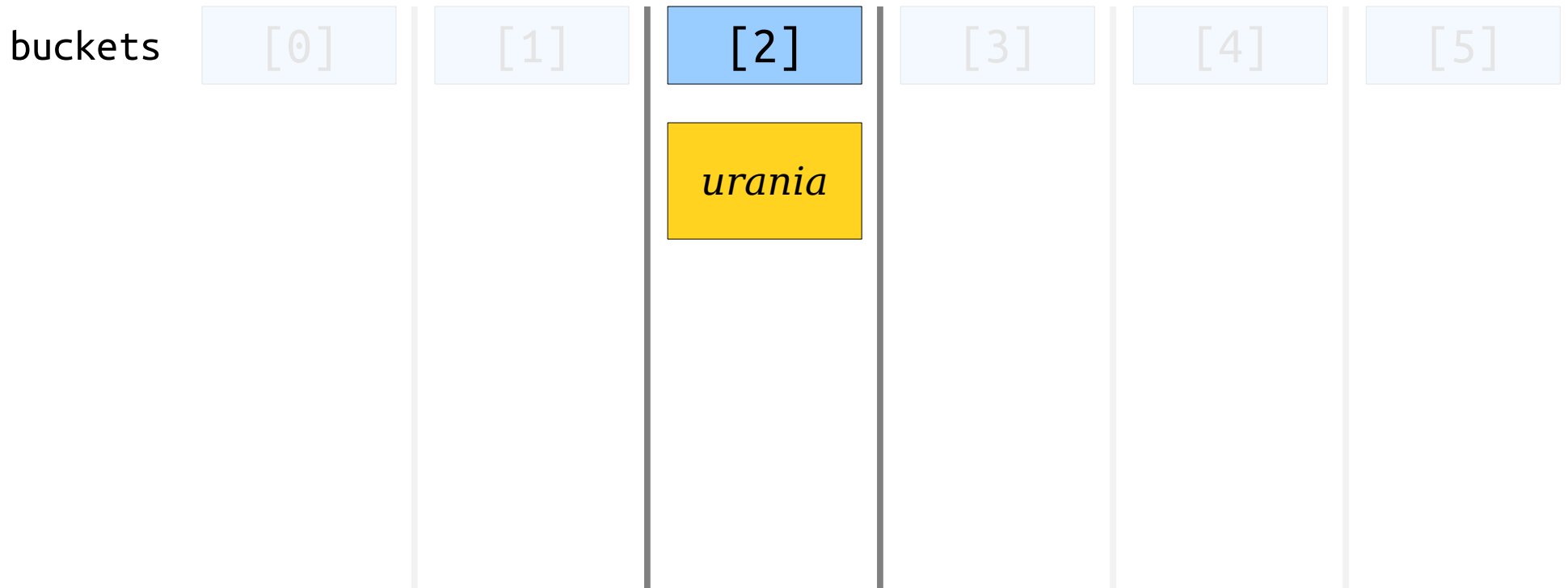
urania

(bucket 2)



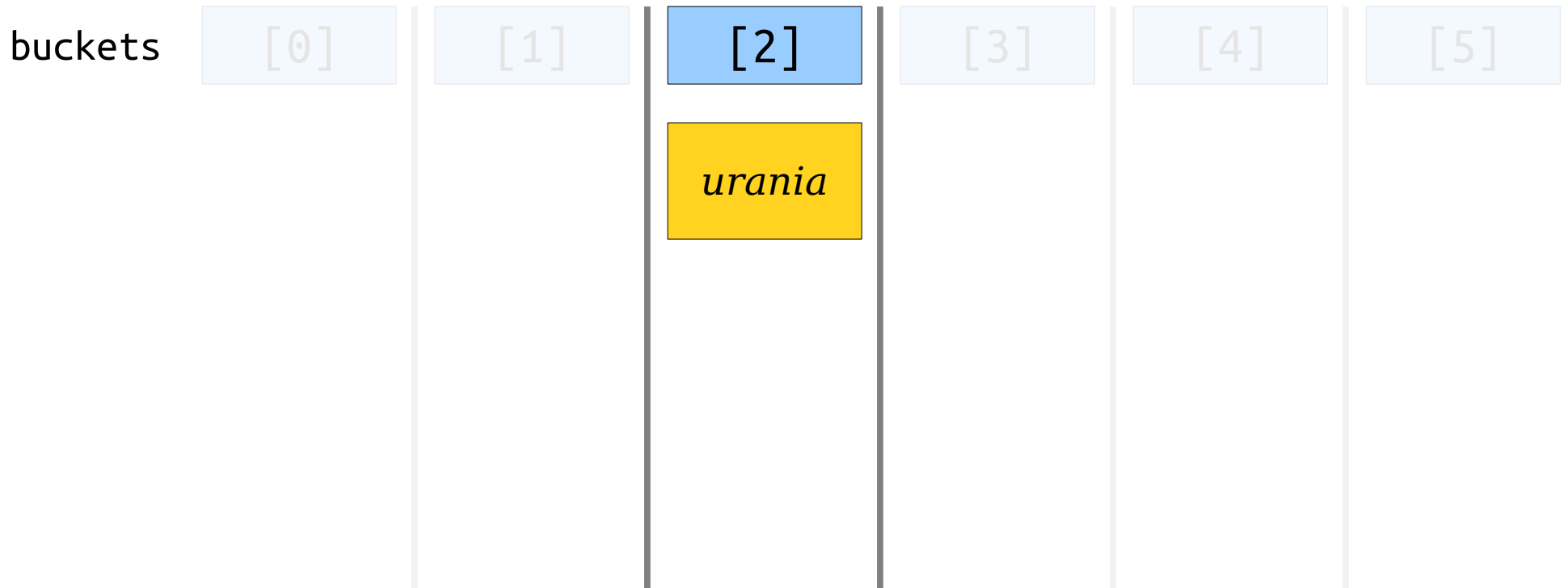
```
void OurSet::add(const string& value) {  
    int bucket = hashFn(value);  
    buckets[bucket] += value;  
}
```

urania
(bucket 2)



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

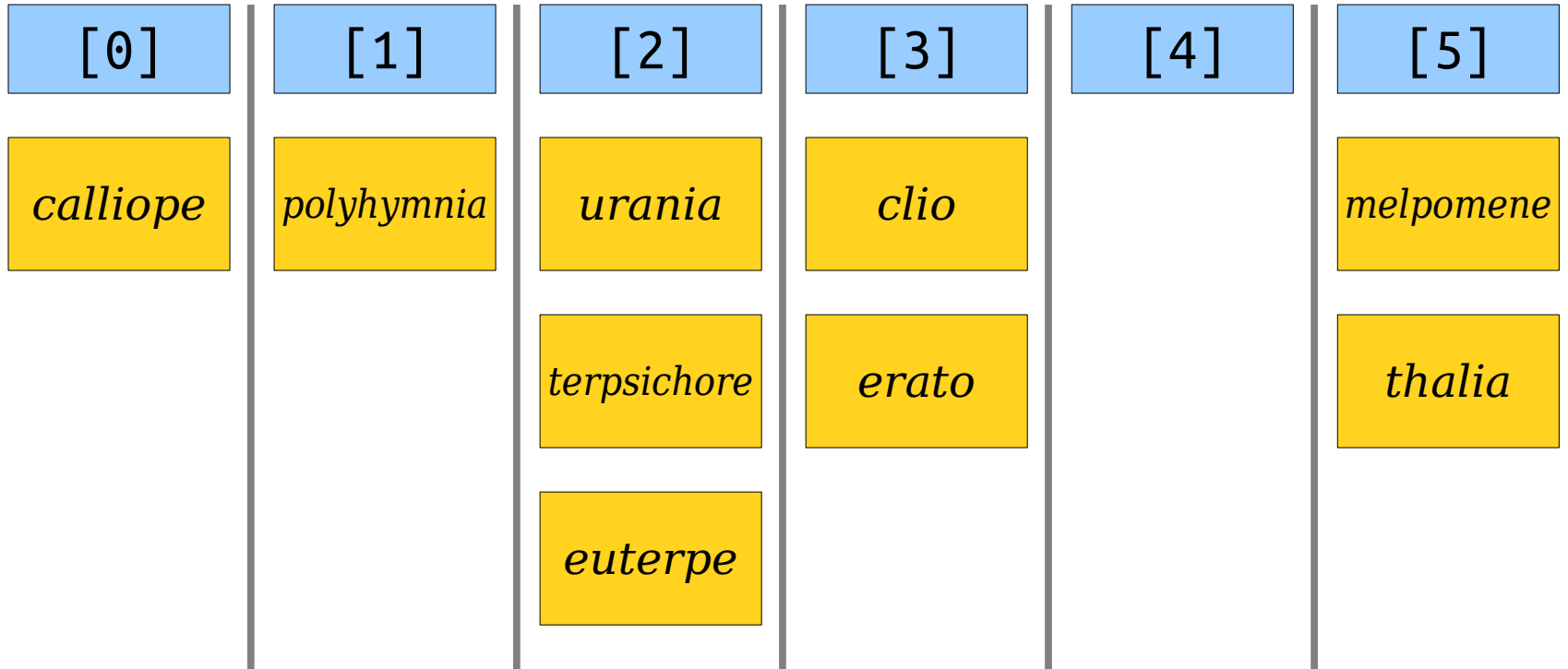
urania
(bucket 2)



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

urania
(bucket 2)

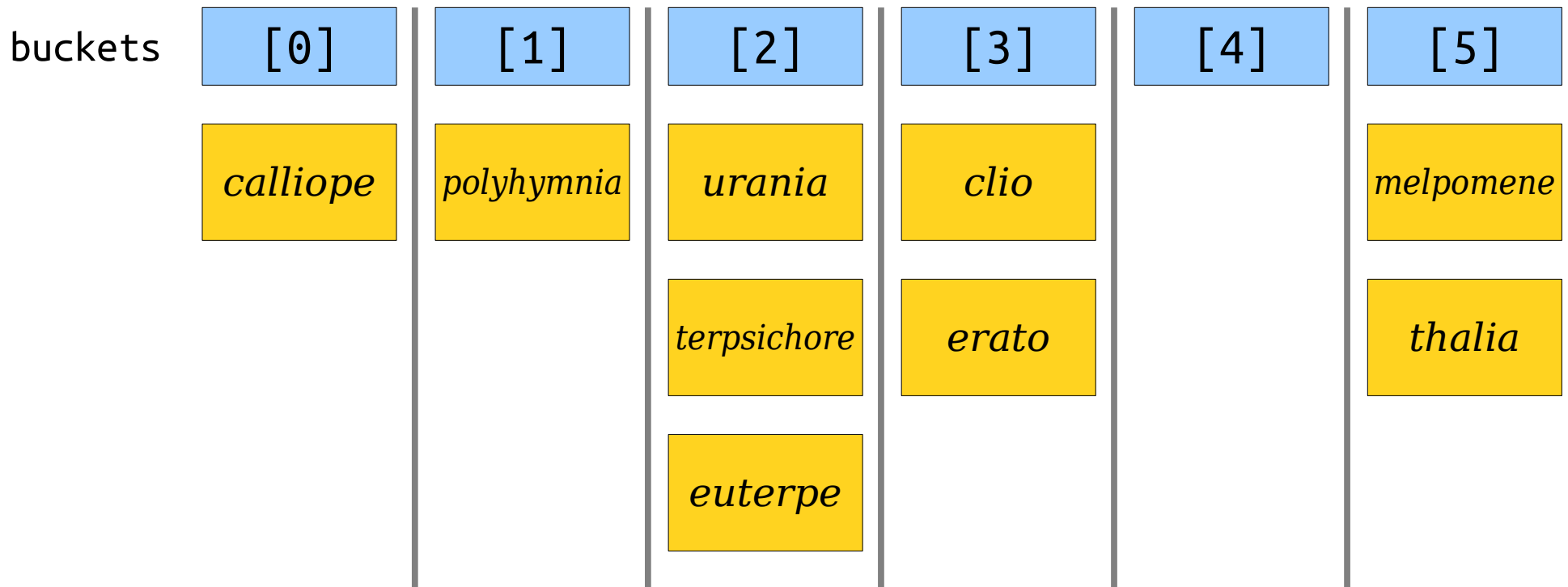
buckets



How efficient is this?

Analyzing our Efficiency

- Each hash table operation
 - chooses a bucket and jumps there, then
 - potentially scans everything in the bucket.
- Choosing the bucket only requires us to hash the input, which is decently quick. So what's needed for that next step?



Analyzing our Efficiency

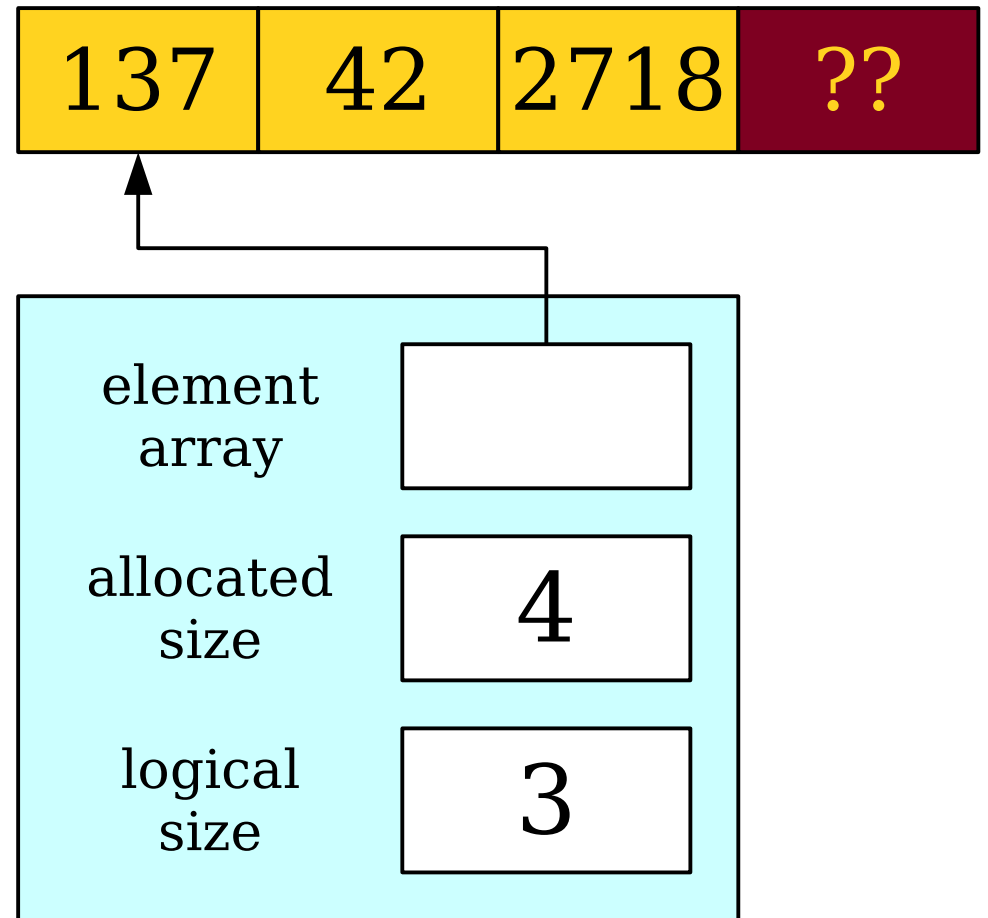
- Imagine we have b buckets and n elements in our table.
- On average, we'll have n / b items in each bucket.
- The average cost of an insertion, deletion, or lookup is therefore

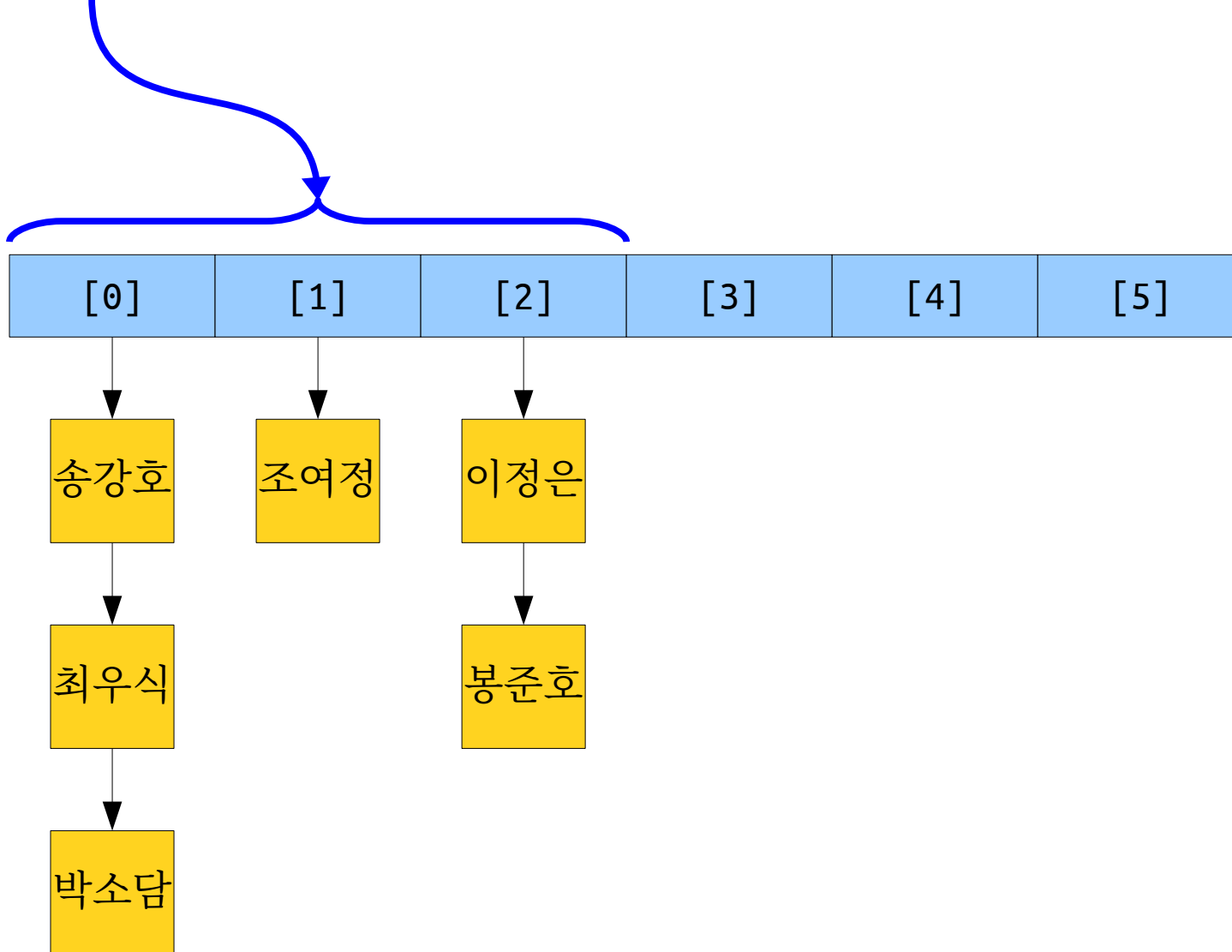
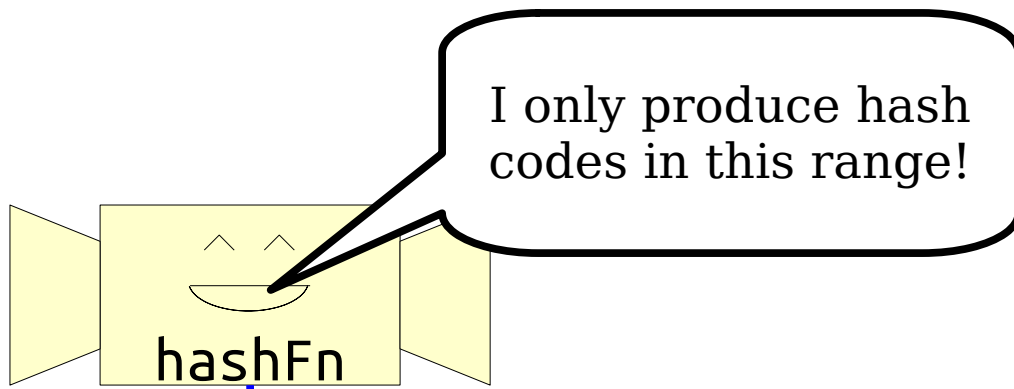
$$O(1 + n / b).$$

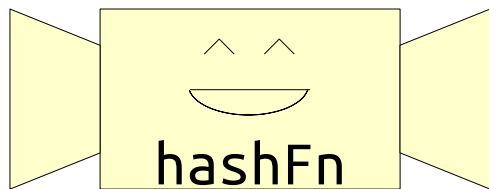
- The *load factor* of a hash table, denoted α , is the ratio of items to buckets ($\alpha = n / b$).
- If we keep α small (say, $\alpha \leq 2$), then the average operation cost is $O(1)$. That's as good as it can possibly get! How do we do this?

Remember When?

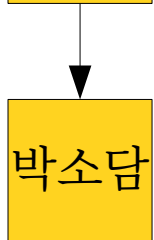
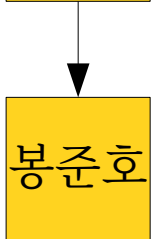
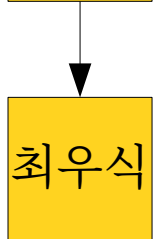
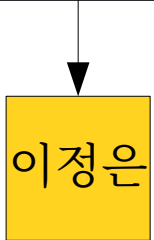
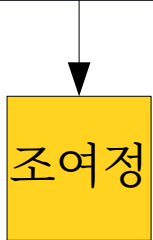
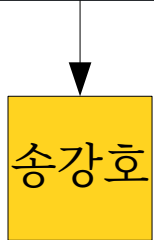
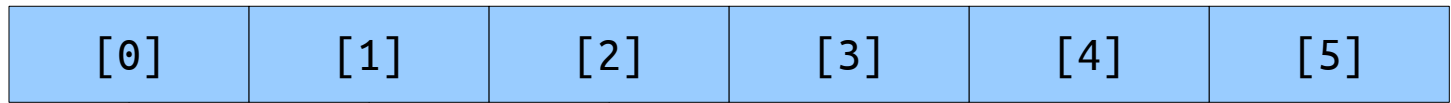
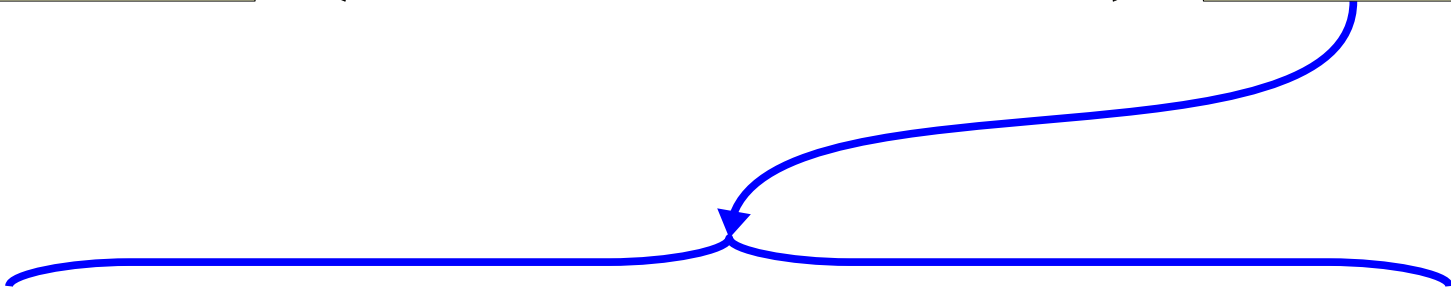
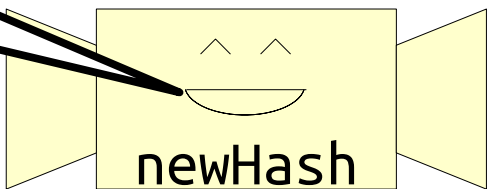
- Think back to how we implemented the Stack.
- Initially, we had a fixed number of slots.
- Once we ran out of space, we doubled the number of slots and transferred things over.
- **Idea:** Whenever $\alpha > 2$, double the number of buckets. This keeps α below two and makes all operations take average time $O(1)$.

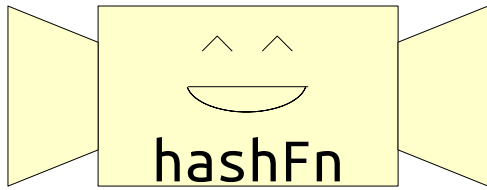




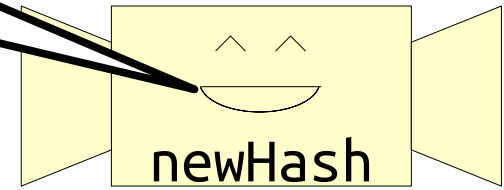


No worries! I'll cover the whole range.



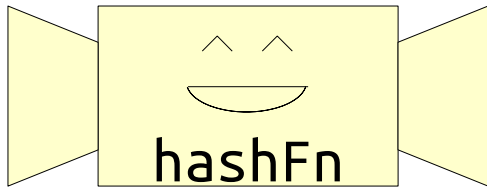


I'll tell each of you
where you need to
go in this new table!

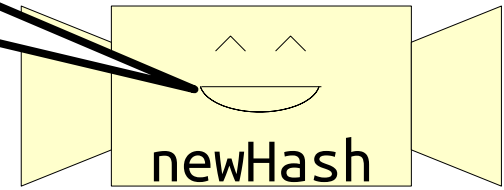


송강호	조여정	이정은	최우식	봉준호	박소담
-----	-----	-----	-----	-----	-----

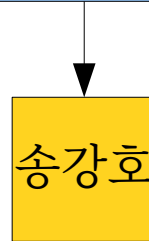
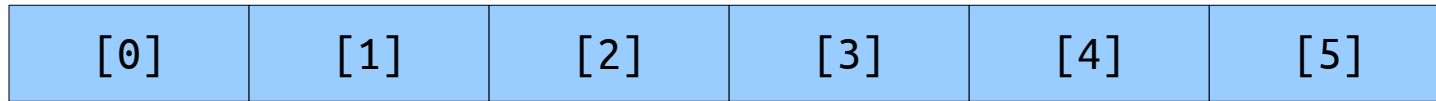
[0]	[1]	[2]	[3]	[4]	[5]
-----	-----	-----	-----	-----	-----

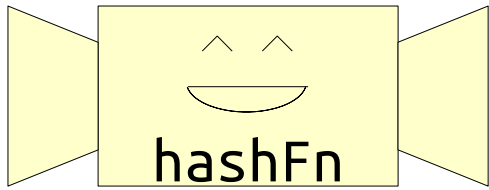


I'll tell each of you
where you need to
go in this new table!

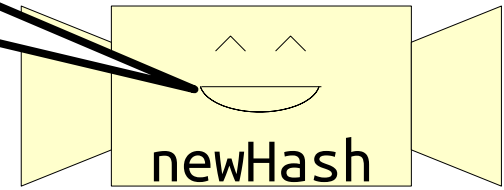


조여정 이정은 최우식 봉준호 박소담

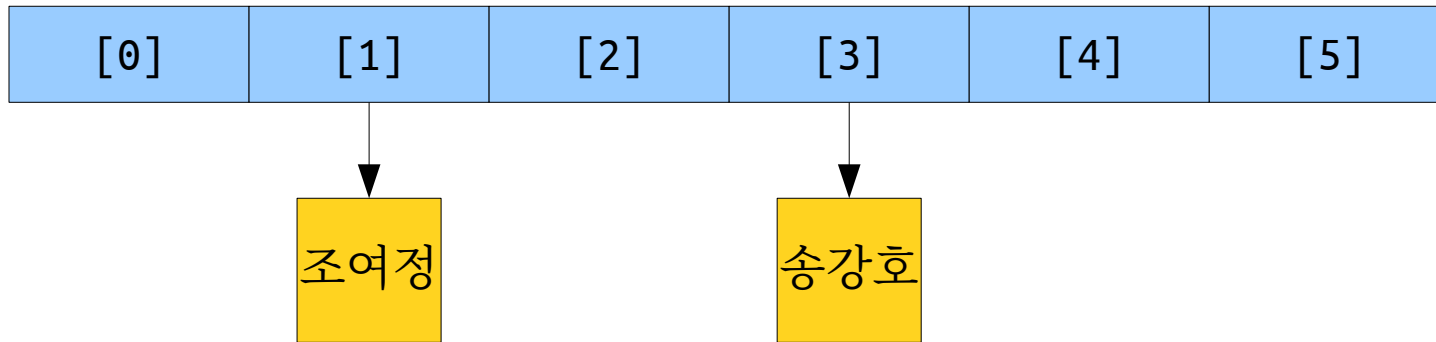


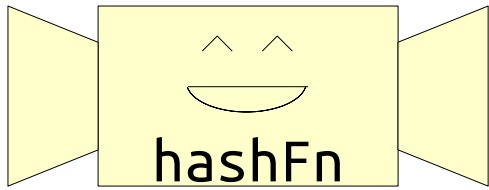


I'll tell each of you
where you need to
go in this new table!

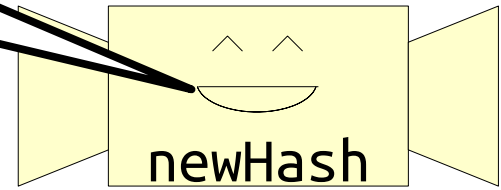


이정은 최우식 봉준호 박소담

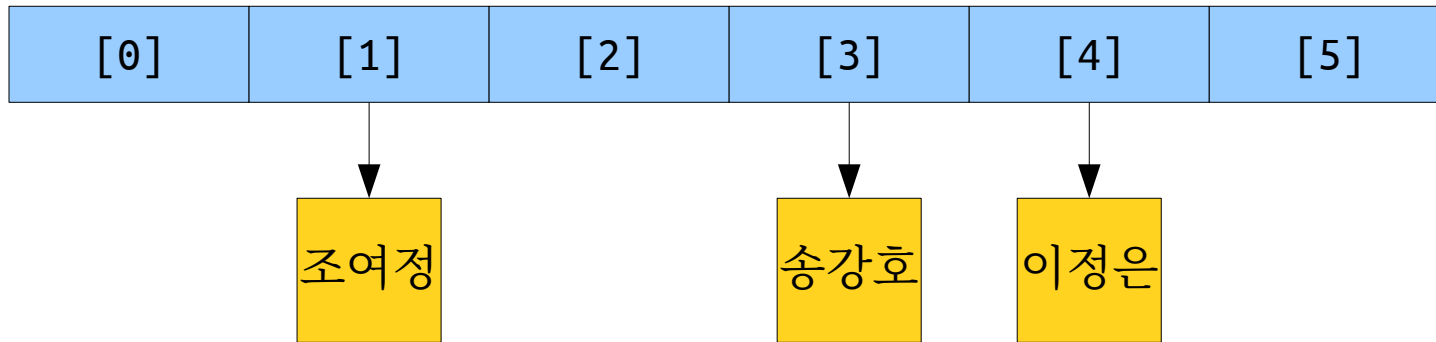


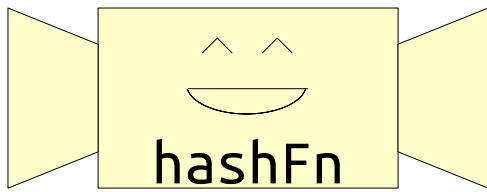


I'll tell each of you
where you need to
go in this new table!

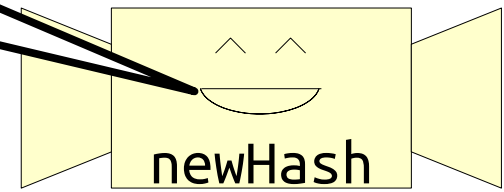


최우식	봉준호	박소담
-----	-----	-----

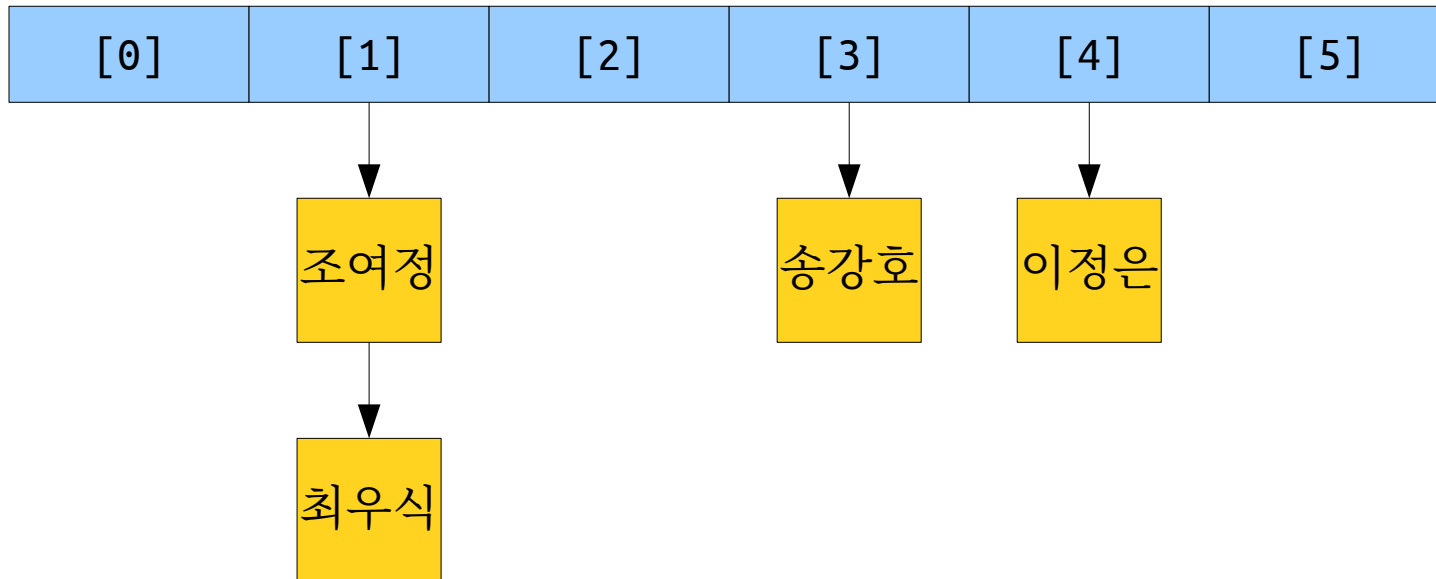


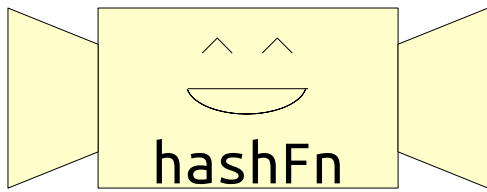


I'll tell each of you
where you need to
go in this new table!

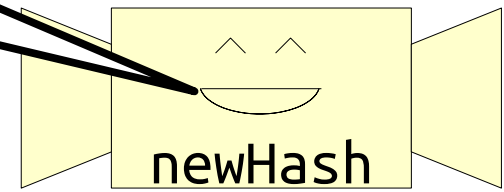


봉준호 박소담

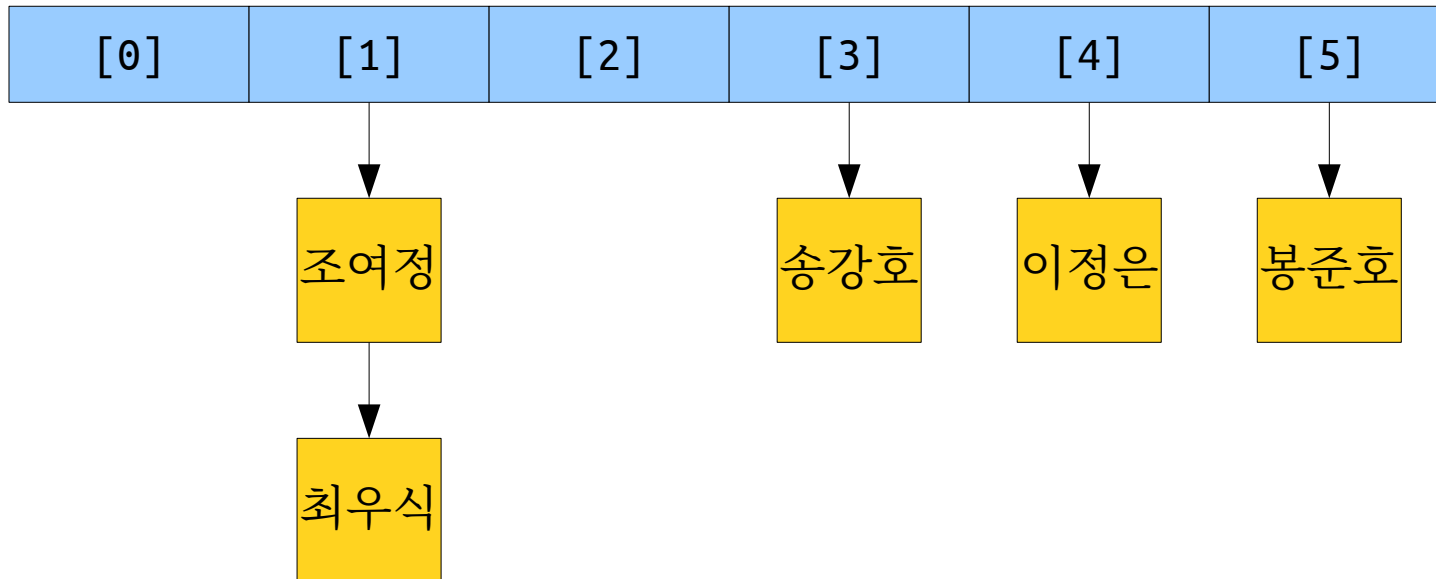


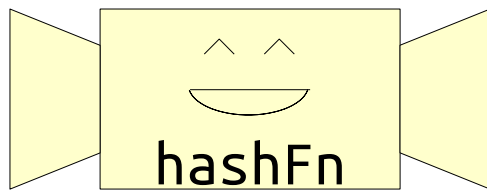


I'll tell each of you
where you need to
go in this new table!

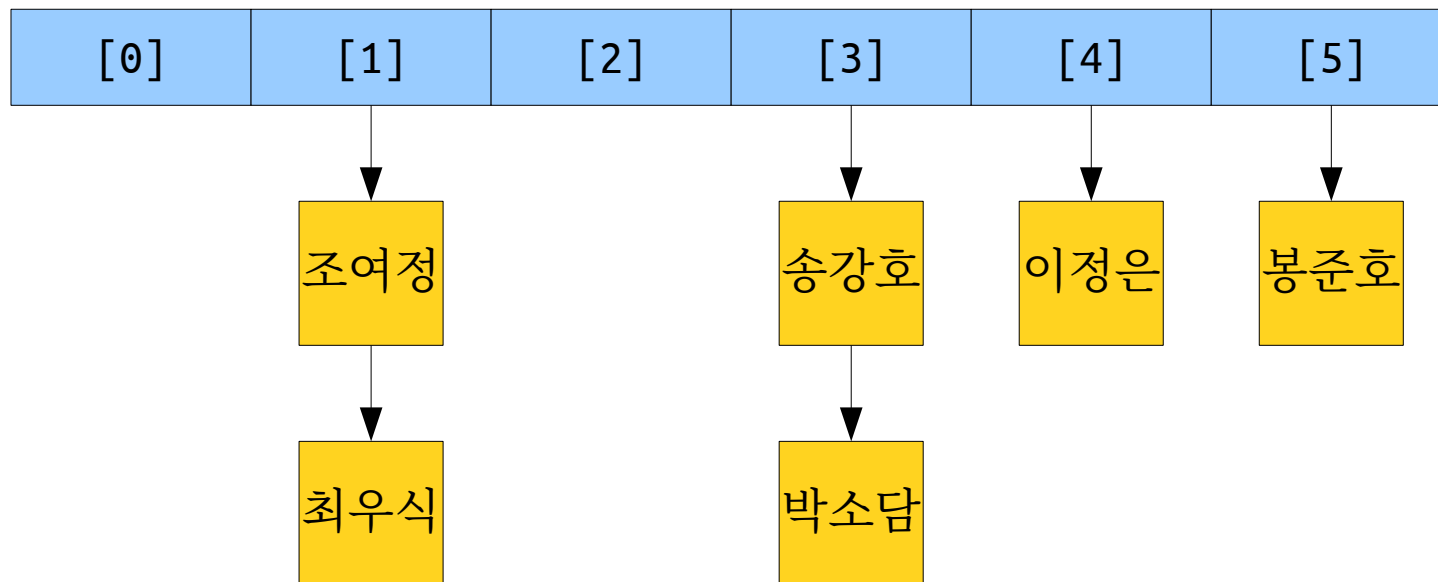
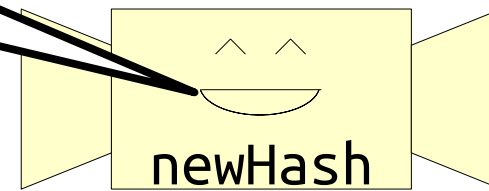


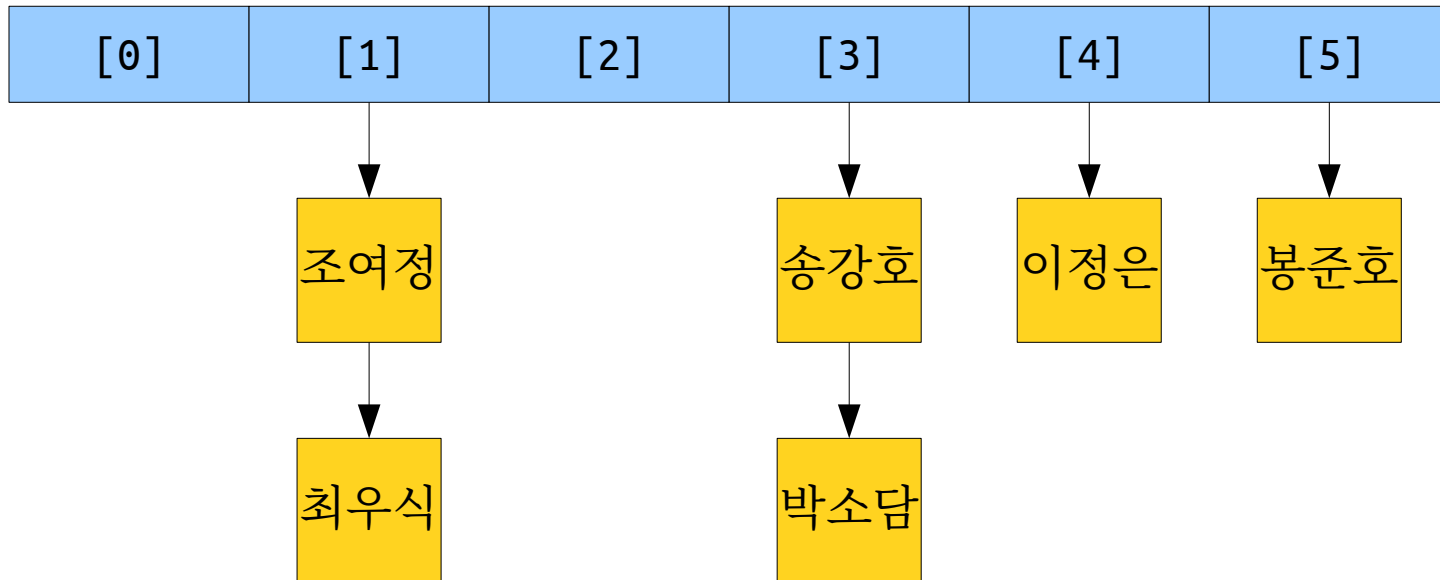
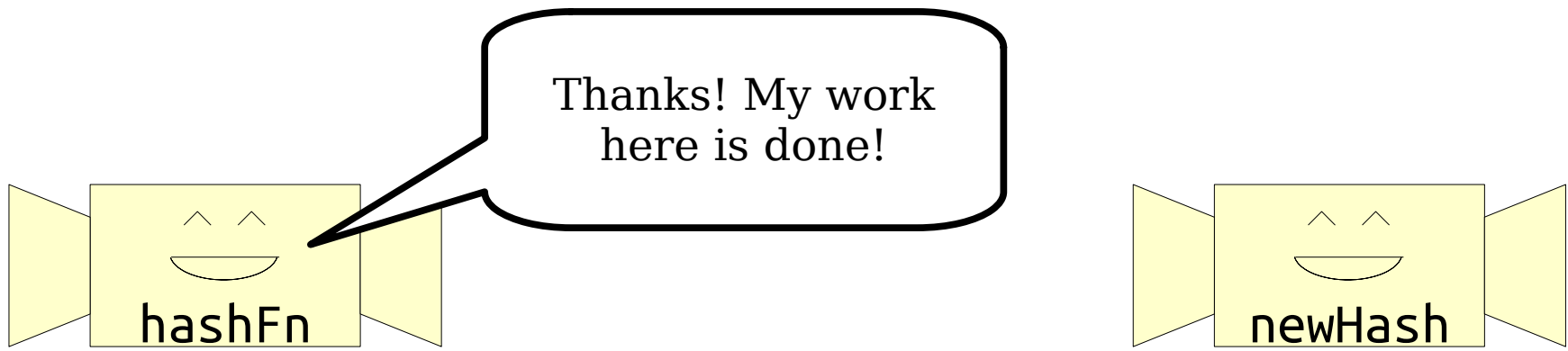
박소담

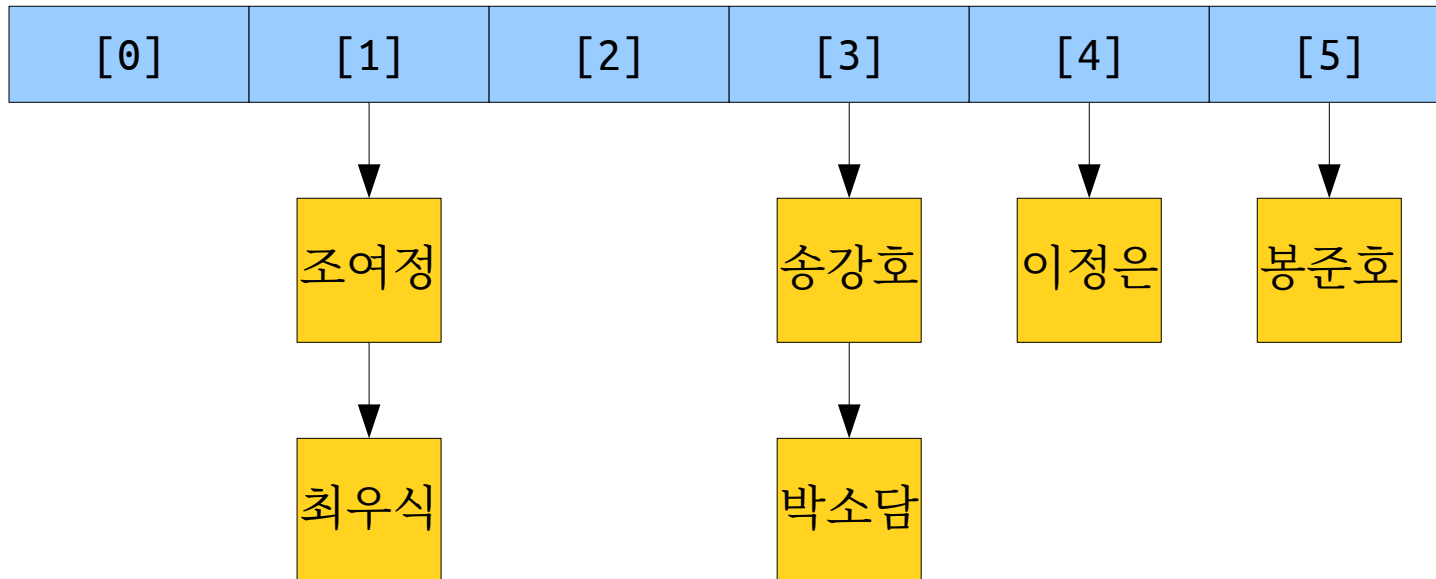
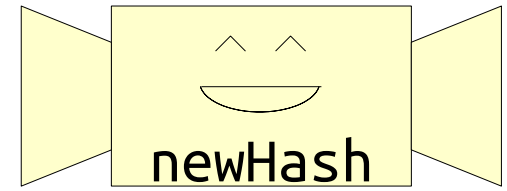


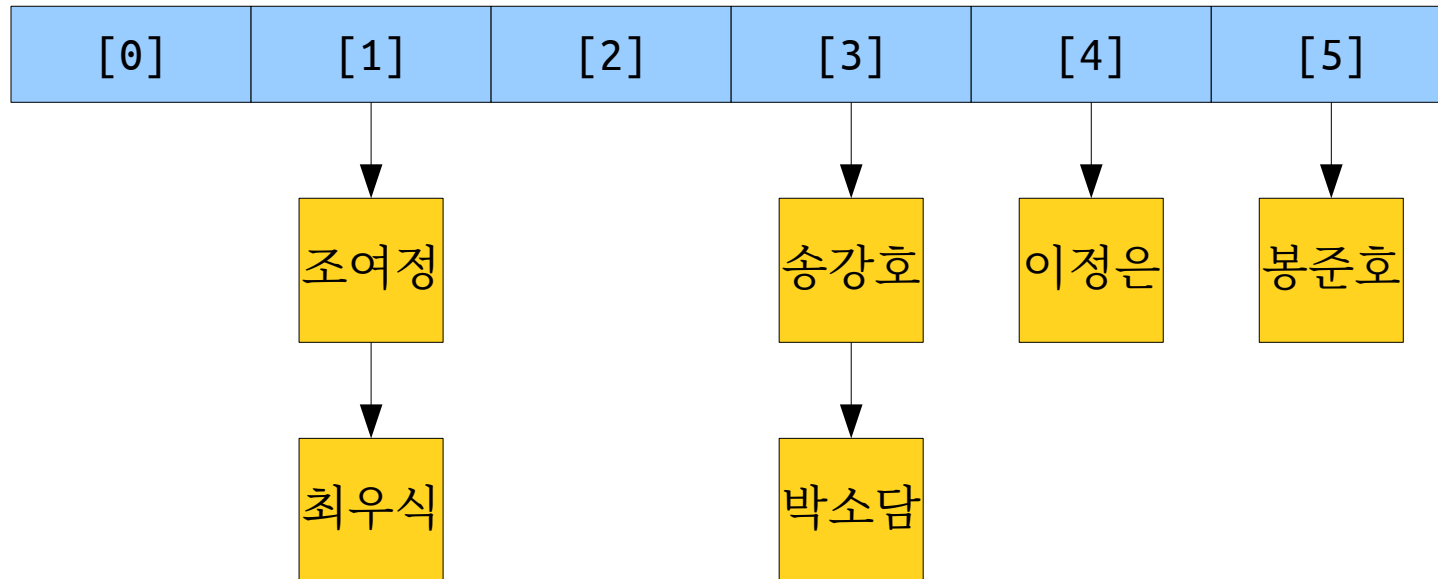
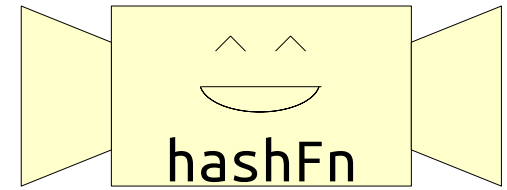


I'll tell each of you where you need to go in this new table!









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.

Your Action Items

- ***Work on Assignment 5***
 - If you're following our timetable, by this point you should be done with the Debugger Warmup and String Simulation and making progress through Tone Matrix.
 - Need help or support? Come talk to us at LaIR, in office hours, or over EdStem!

Next Time

- ***Open Addressing***
 - A different conception of hash tables.
- ***Linear Probing***
 - A fast, flexible hash table.
- ***Robin Hood Hashing***
 - A fairer way to hash items.