CodingLab
Powered by RBK

Junior Phase

Hash tables

An introduction to hash tables

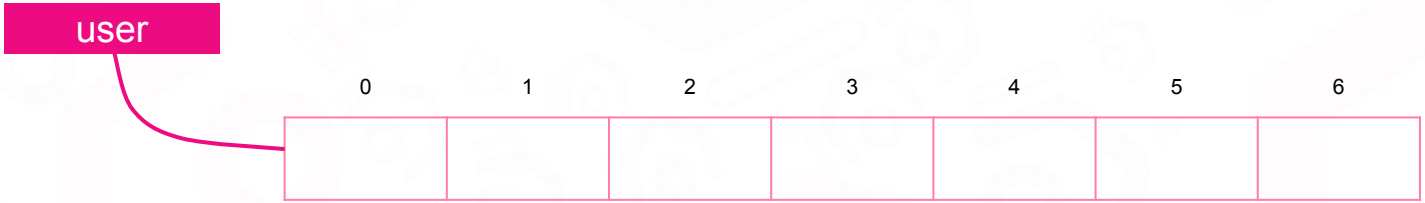
Duration: 30 minutes :
Q&A: 5 minutes by the end of the lecture

Definition

A Hash Table is a **data structure** that maintains **associations** between **two data values**. The data values being associated are commonly referred to as the **key** and **value**. A single instance of a Hash Table may store many associations.

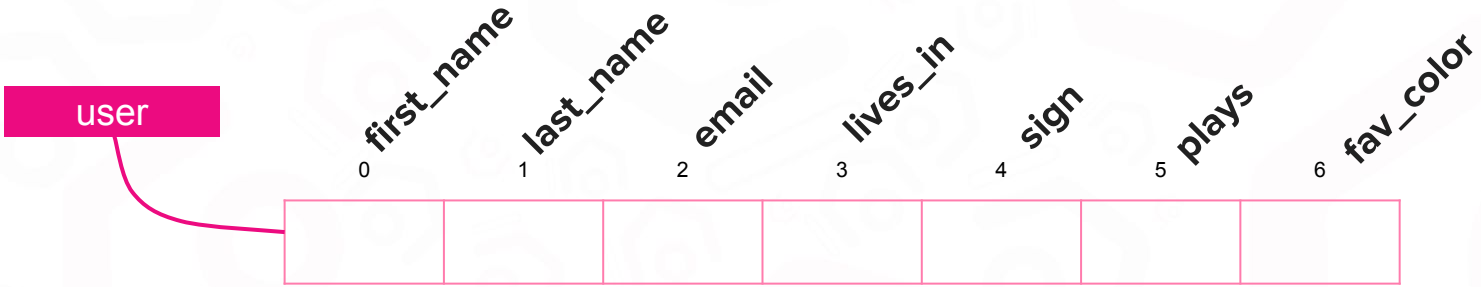
A Hash Table has other names too, such as an Associative Array, Dictionary, Map, or Hash.

Data Association - The Problem



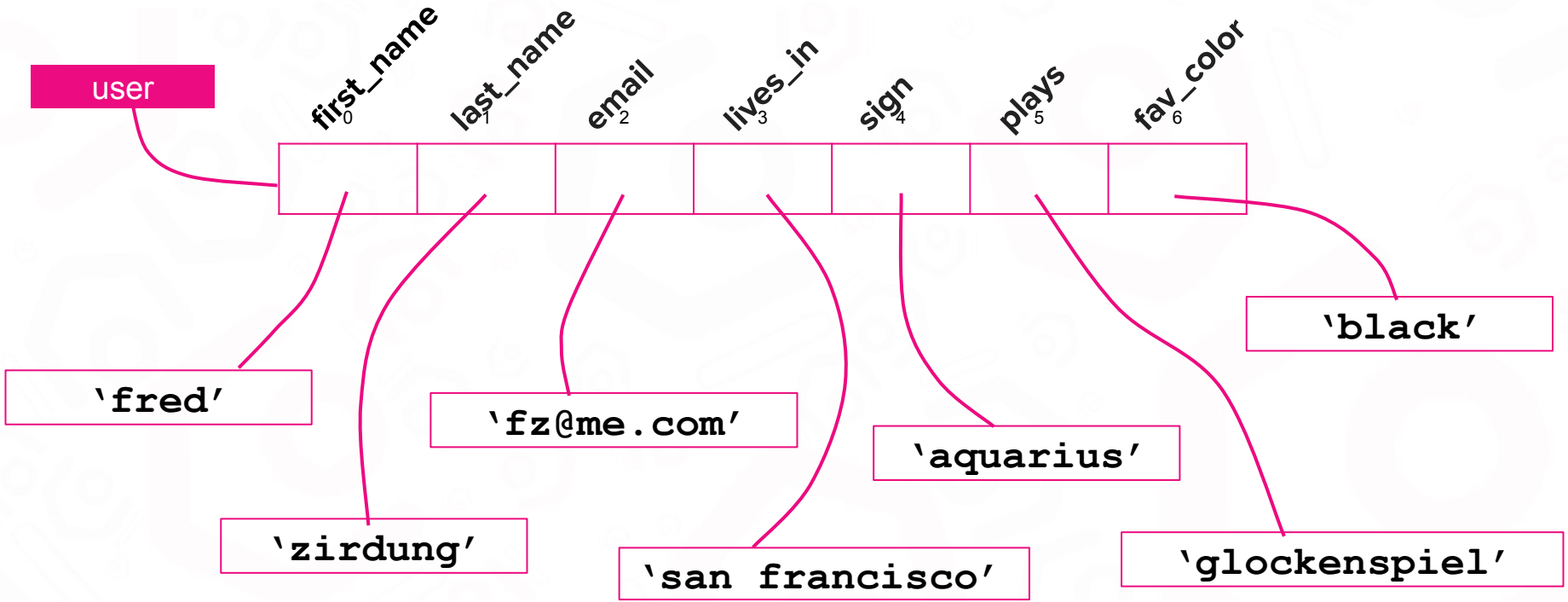
In order to understand the importance of associative data, let's start by creating a data structure to store user information. For this task, we'll use the most basic of data structures - an array.

Data Association - The Problem



A typical usage of arrays is in the form of a **tuple**. A tuple is a data structure where the ordinal position has a specific meaning and each instance follows the same pattern. In this example first_name is always at index 0.

Data Association - The Problem



Next, we'll populate our data structure with some data about a user. Each index in the array will point to a predetermined aspect of the user data about the user (first name in 0, last name in 1, then email in 2, and so on).

Data Association - The Problem

To understand how to make use of this data structure, let's write some code that will check if this user lives in **san francisco** and likes the color **black**.

Data Association - The Problem

Q: What index values should be used to lookup the corresponding values in our user array?

Data Association - The Problem

user

```
if (user[3] === 'san francisco' &&
   user[6] === `black') {

 //...

}
```

'fred'

'fz@me.com'

'zirdung'

'san francisco'

'aquarius'

'black'

'glockenspiel'

A: The indexes for lives_in and fav_color are 3 and 6. The code here is straightforward, but the data layout is unintuitive for humans. It requires you to remember which index is associated with each kind of data.

Data Association - The Problem


```
if (user[3] === 'san francisco' &&
    user[6] === `black') {

  //...

}
```

user

'fred'

'fz@me.com'

'zirdung'

'san francisco'

'aquarius'

'black'

'glockenspiel'

For small tuples, this scheme works reasonably well. When a tuple contains many properties, it becomes tedious and error prone to remember and/or look up mappings between field meaning and index value.

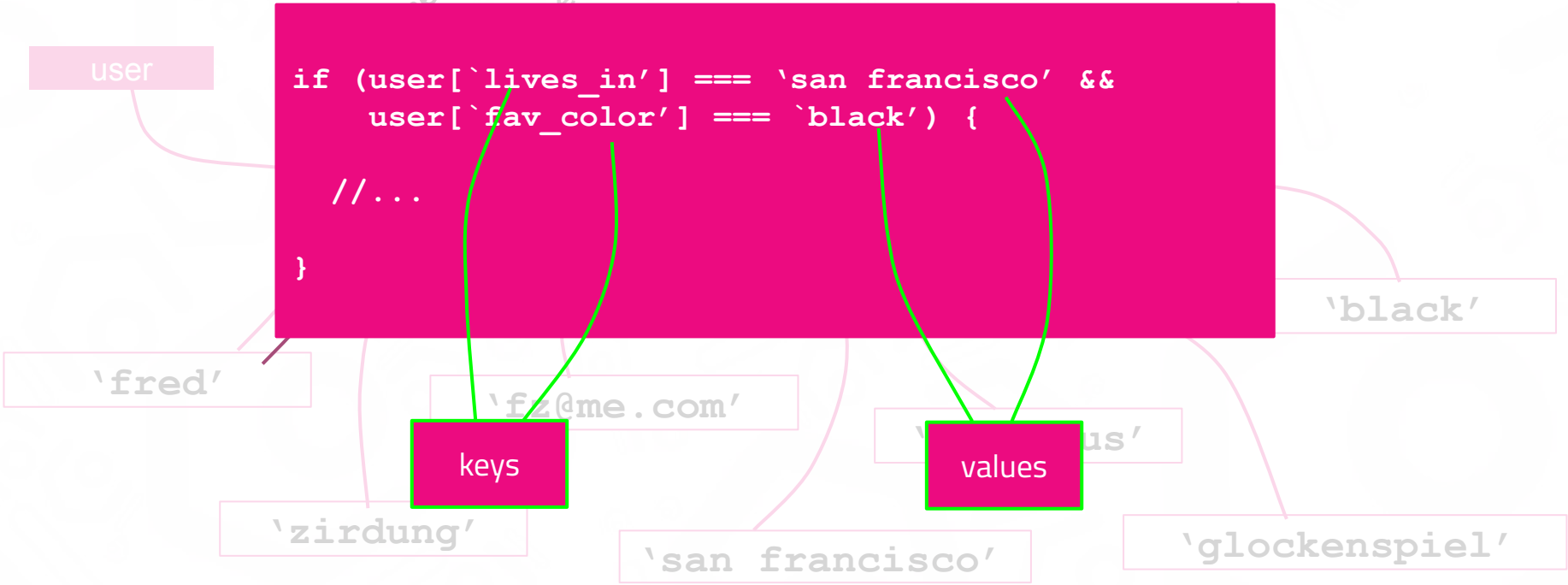
Data Association - The Problem

As humans, we can relate information to string-based labels more easily than numerical labels. **Q:** Can you think of a way to change this code to make it easier to maintain and understand?

Data Association - The Problem

A: Use string-based labels instead of numerical index values. In particular, we'll want to use the field name as the identifier for any given property.

Data Association - The Problem



This code should look very familiar -- it is JavaScript object syntax, using **keys** and **values**. Under the hood, JavaScript objects are implemented using Hash Tables.

Data Association - The Problem


```
if (user[`lives_in’] === ‘san francisco’ &&
    user[`fav_color’] === `black’) {

  //...

}
```

What about time complexity? Let’s compare this to the original array-based version...

Data Association - The Problem

user

```
if (user[`lives_in'] === 'san francisco' &&
    user[`fav_color'] === `black') {

  //...

}
```

'fred'

constant time lookup

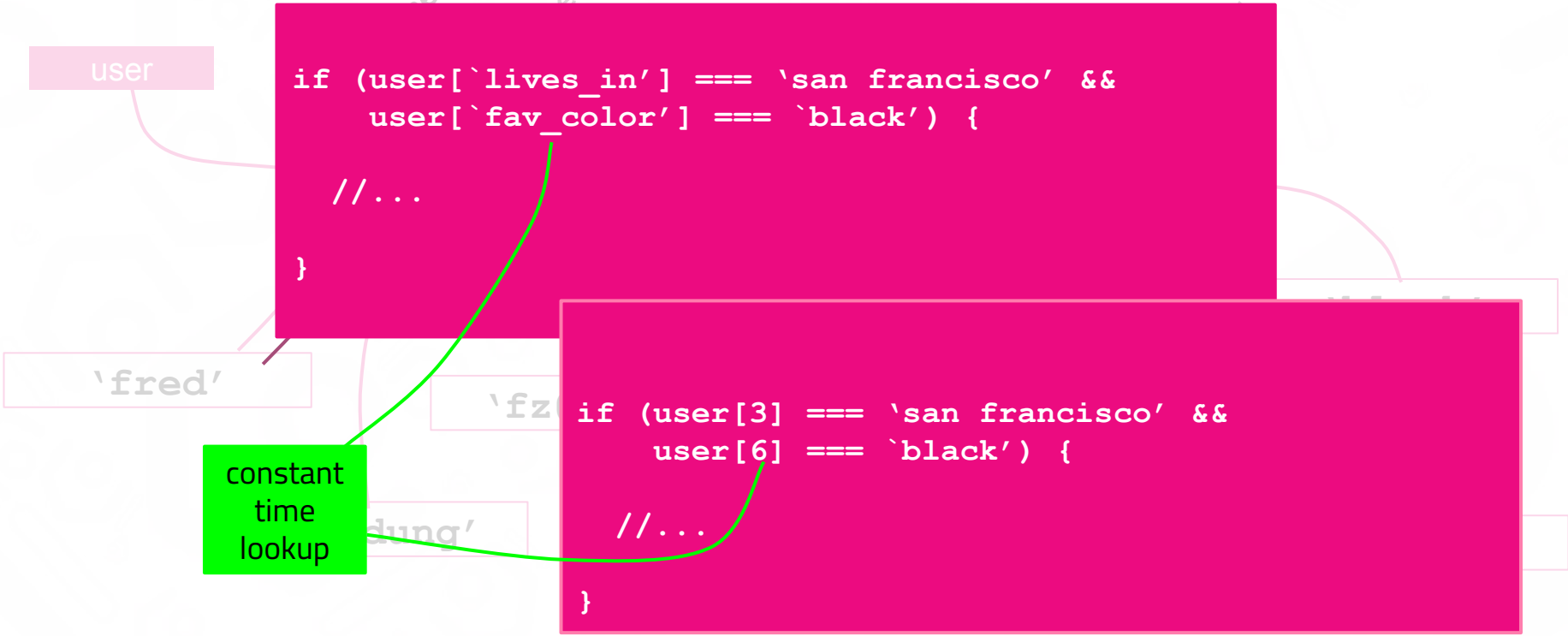
```
if (user[3] === 'san francisco' &&
    user[6] === `black') {

  //...

}
```

We know that array lookup for any ordinal position is a constant time operation. What about the objects?

Data Association - The Problem



Key lookups on an object are constant time operations too! These two characteristics, string-based keys and constant time lookup, are what make Hash Tables so powerful.

Data Association - The Problem

```
if (user[`lives_in'] === 'san francisco' &&
    user[`fav_color'] === `black') {

  //...

}
```

Now that we understand the motivation for Hash Tables, let's dive into the details of how they work, and how to achieve constant time access.

Objects vs Hash Tables

The interface of an object as compared to a hash table.

	JavaScript Object	Hash Table
Insert		
Retrieve		
Remove		

Let's compare the standard operations done on objects and hash tables. Besides syntax, we're going to discover very little difference in how these operations behave for objects and hash tables.

Objects vs Hash Tables

The interface of an object as compared to a hash table.

	JavaScript Object	Hash Table
Insert	obj[‘key’] = ‘value’	ht.insert(‘key’, ‘value’)
Retrieve		
Remove		

Objects vs Hash Tables

The interface of an object as compared to a hash table.

	JavaScript Object	Hash Table
Insert	obj['key'] = 'value'	ht.insert('key', 'value')
Retrieve	obj['key']	ht.retrieve('key')
Remove		

Objects vs Hash Tables

The interface of an object as compared to a hash table.

	JavaScript Object	Hash Table
Insert	obj['key'] = 'value'	ht.insert('key', 'value')
Retrieve	obj['key']	ht.retrieve('key')
Remove	delete obj['key']	ht.delete('key')

Objects vs Hash Tables

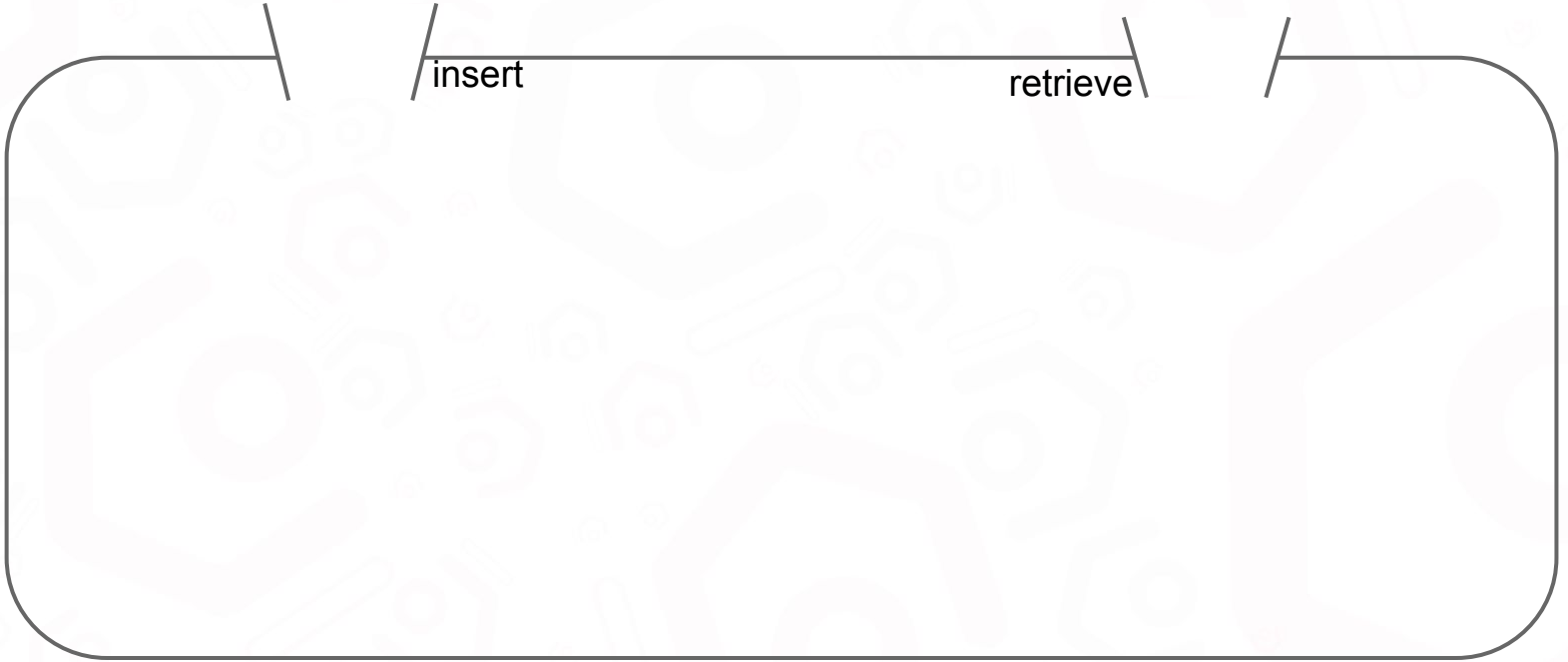
The interface of an object as compared to a hash table.

	JavaScript Object	Hash Table
Insert	obj['key'] = 'value'	ht.insert('key', 'value')
Retrieve	obj['key']	ht.retrieve('key')
Remove	delete obj['key']	ht.delete('key')

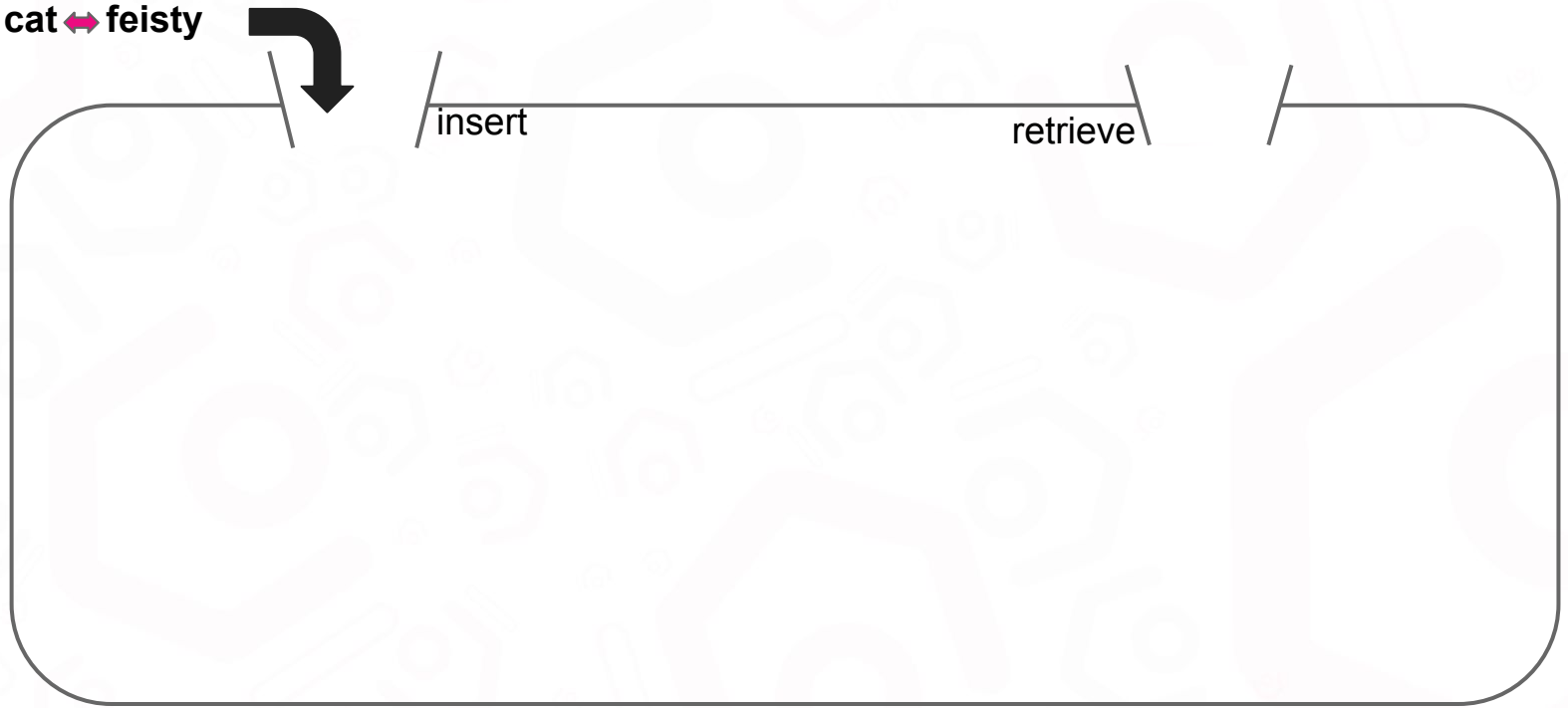
Remember, all three functions described here can run in **constant time**. This is the most valuable aspect of a hash table, so focus on how to achieve that as we explore the implementation for a hash table.

Let’s design a Hash Table

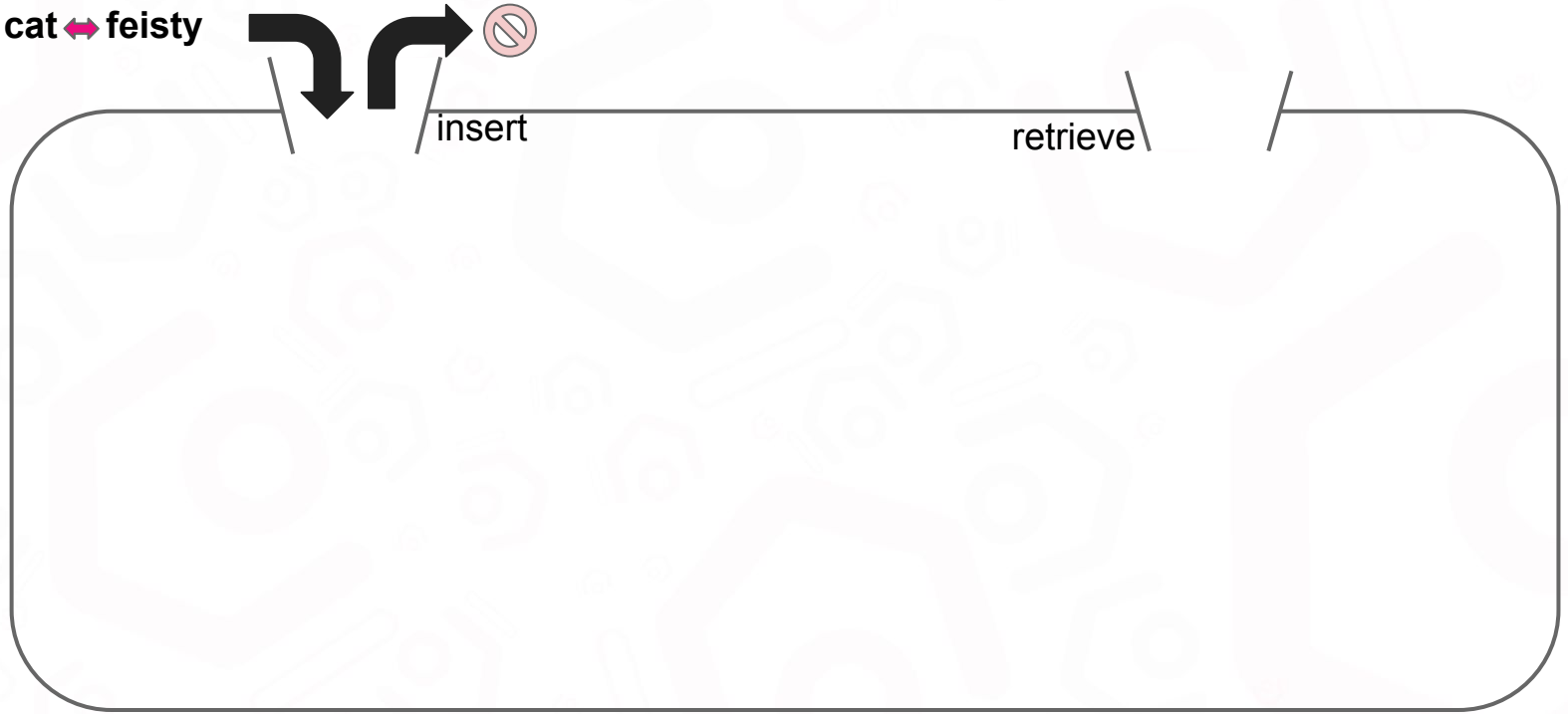
We'll represent the hash table data structure as a box...



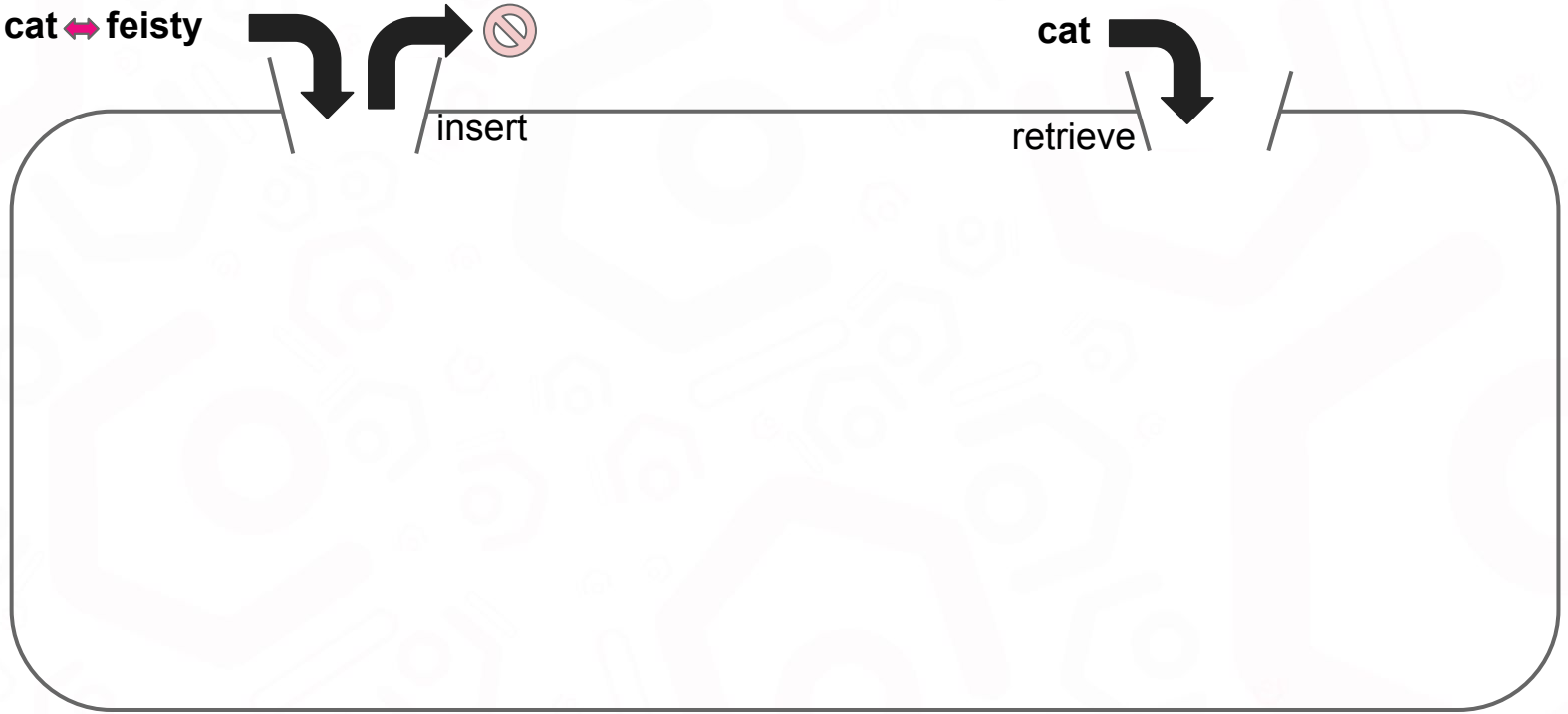
...and add two portals that represent method interfaces available on every hash table. (We won't cover the remove method here, since it's implementation does not follow any different rules.)



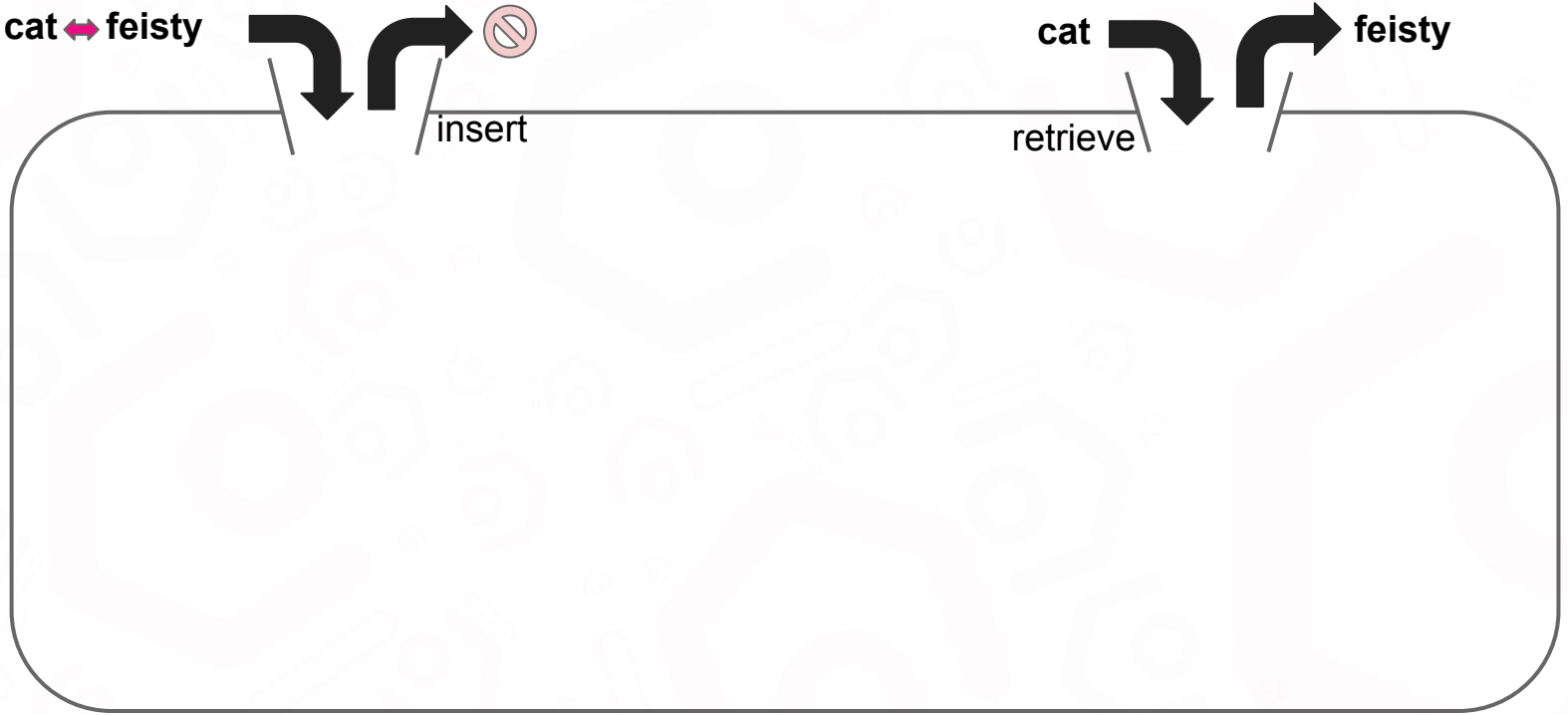
An insert operation takes a key and a value. In this example, the key is 'cat' and the value is 'feisty'.



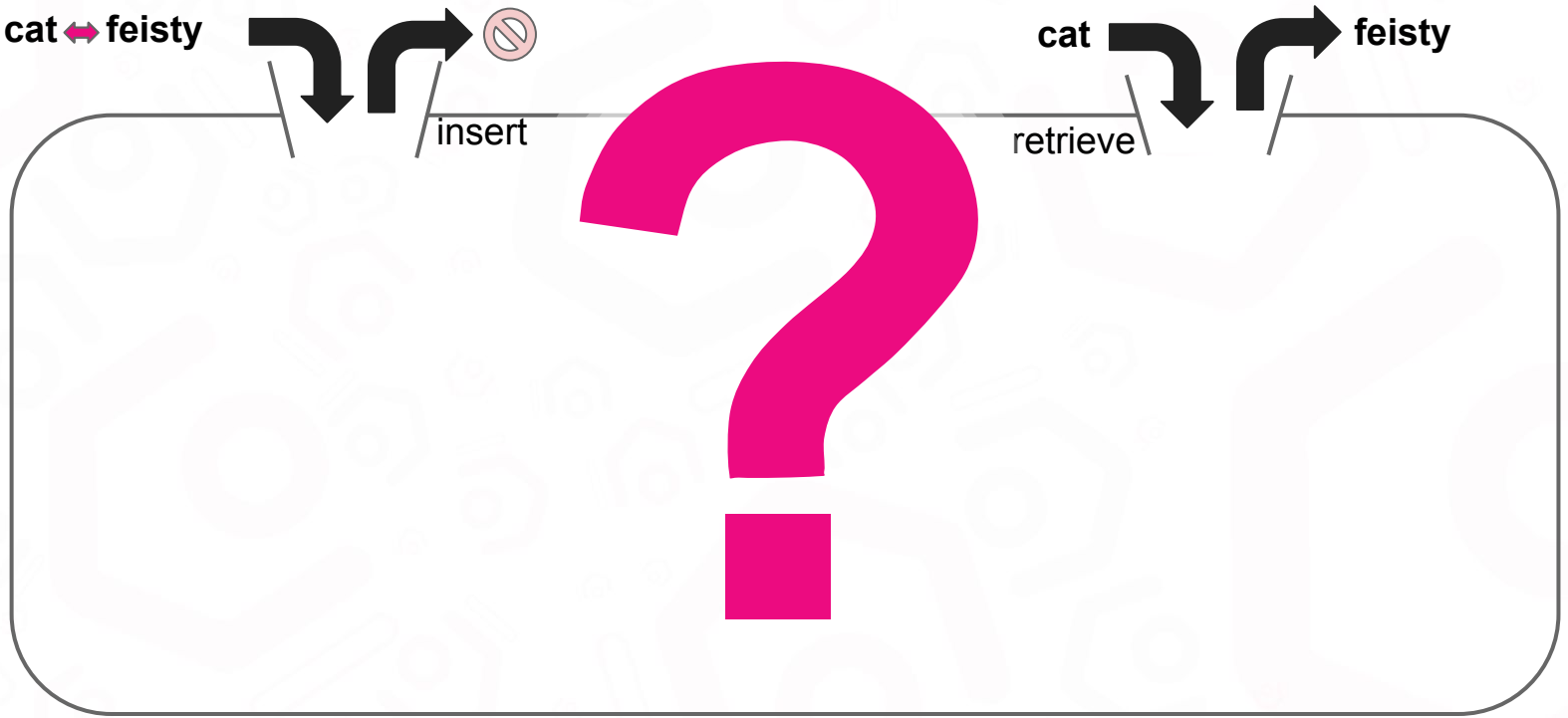
Since there's little new information generated by an assignment operation, it's hard to think of any useful return values. So, assignment won't need to return any value (and in JavaScript, would return undefined).



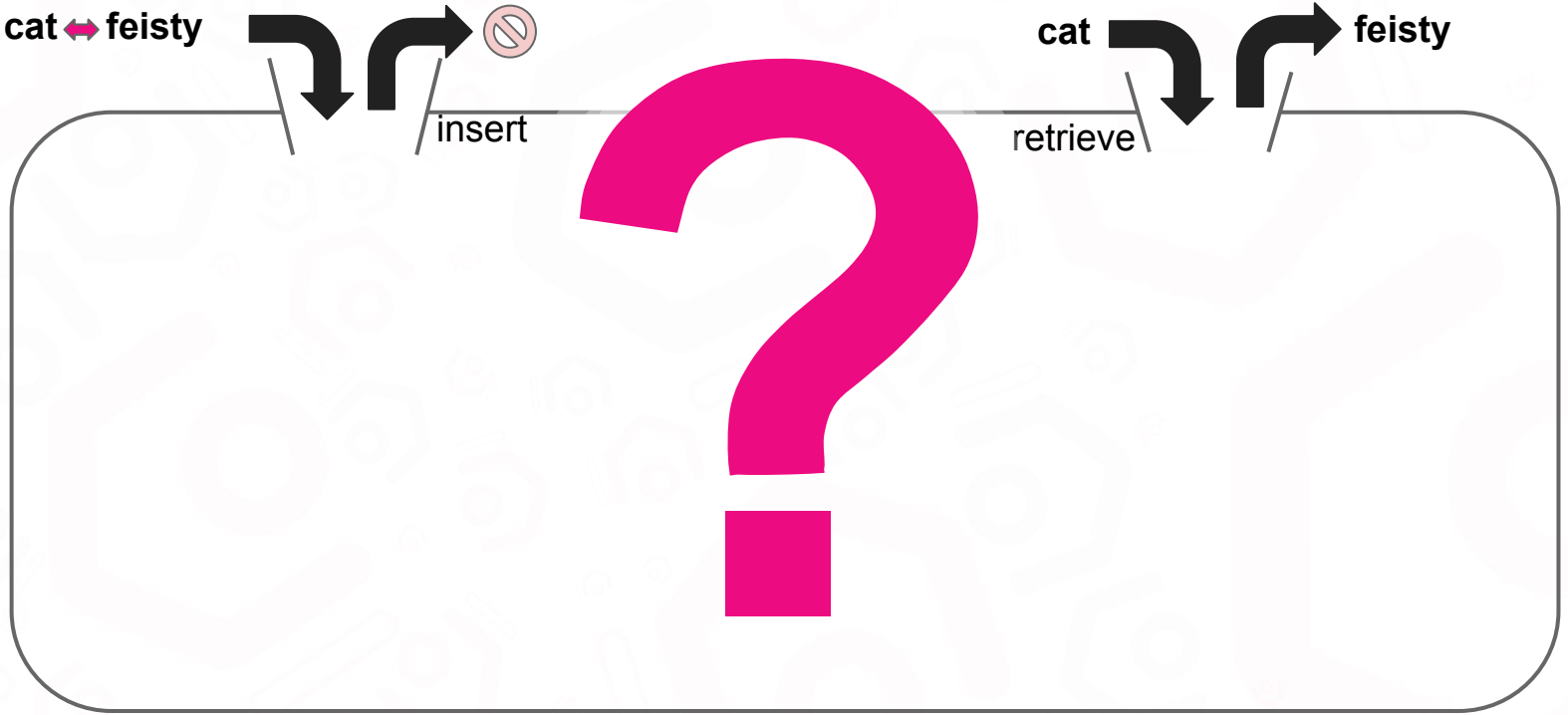
At some point, we might ask the hash table for the value associated with ‘cat’.



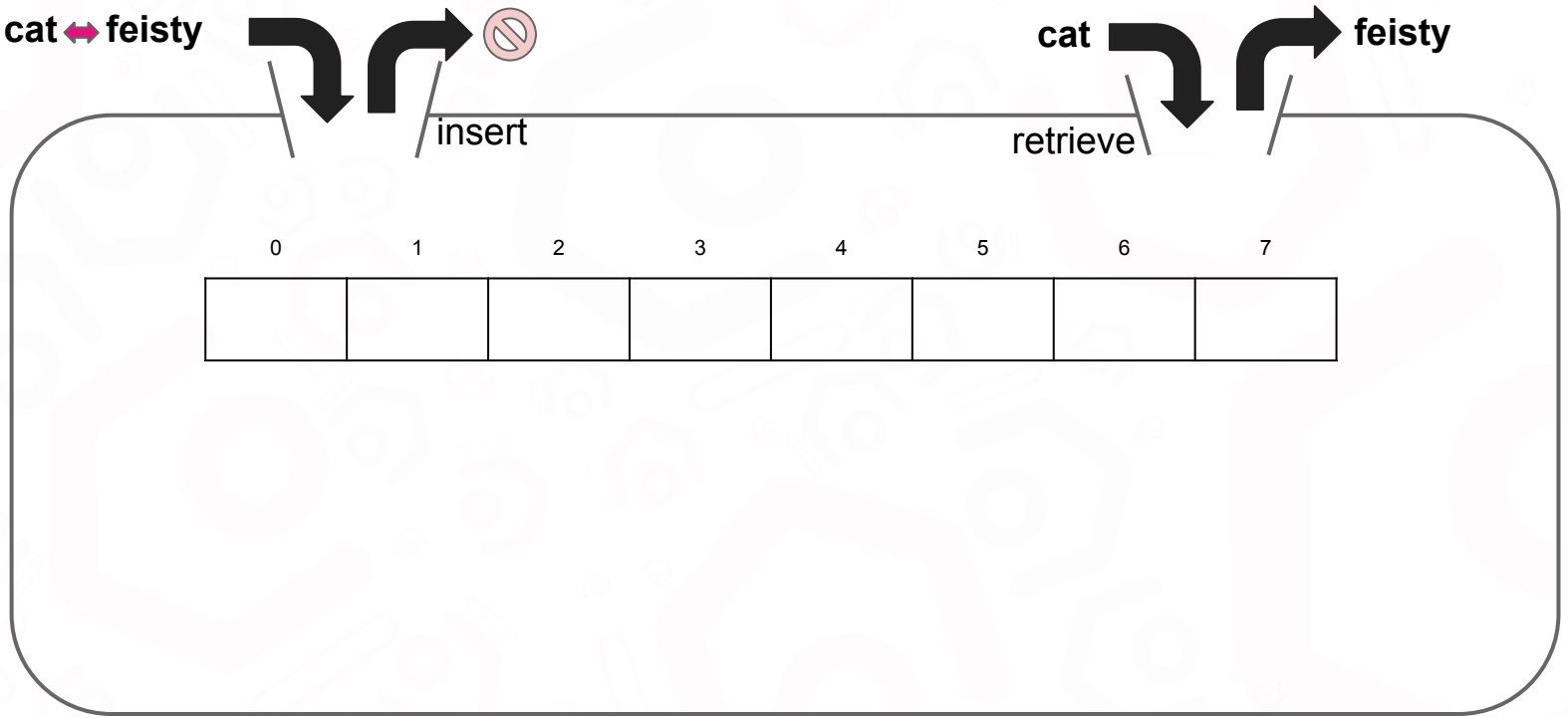
And of course, the return value of that call should be 'feisty'.



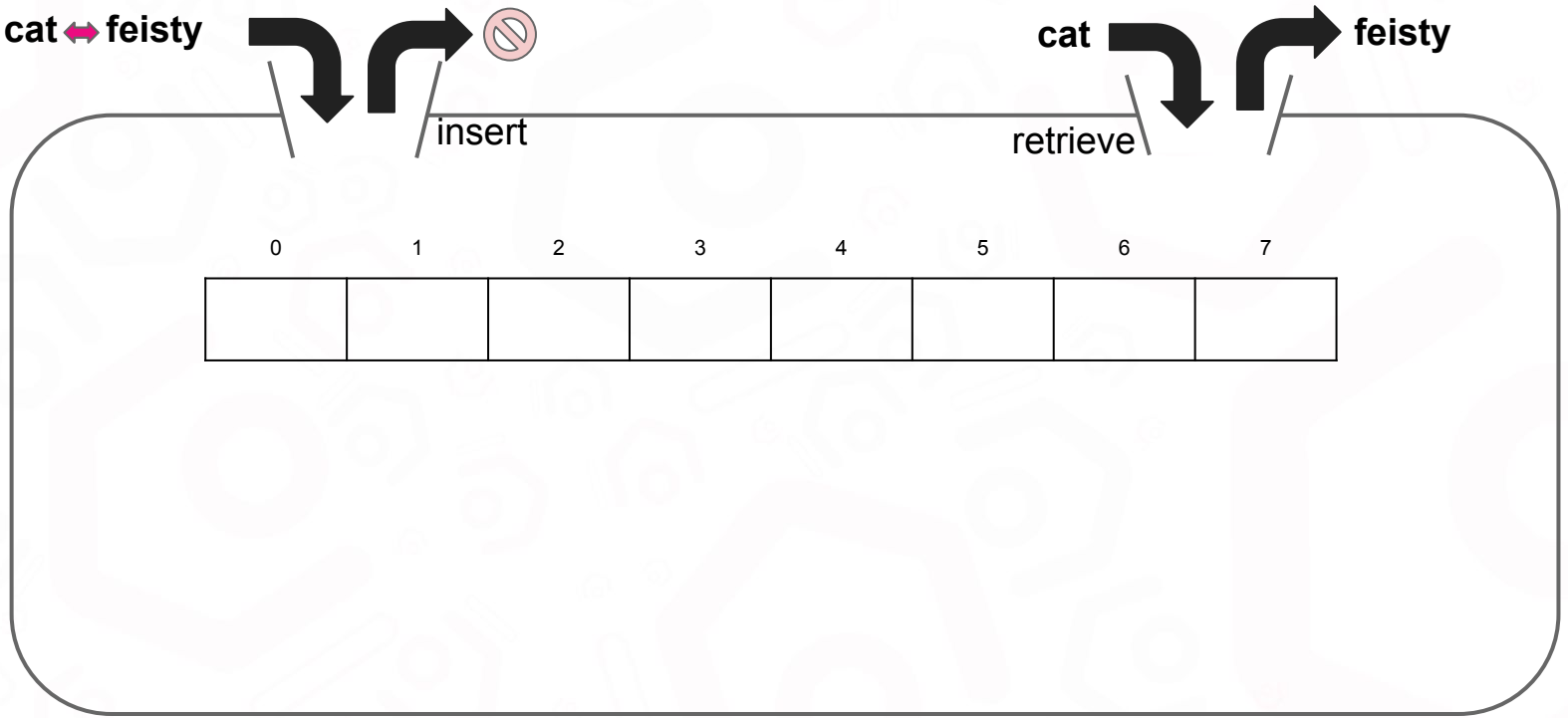
Based on our understanding of the hash table's interface, we know we must store some information. **Q:** What data structure could be used internally, to store and retrieve the data presented on the interface?



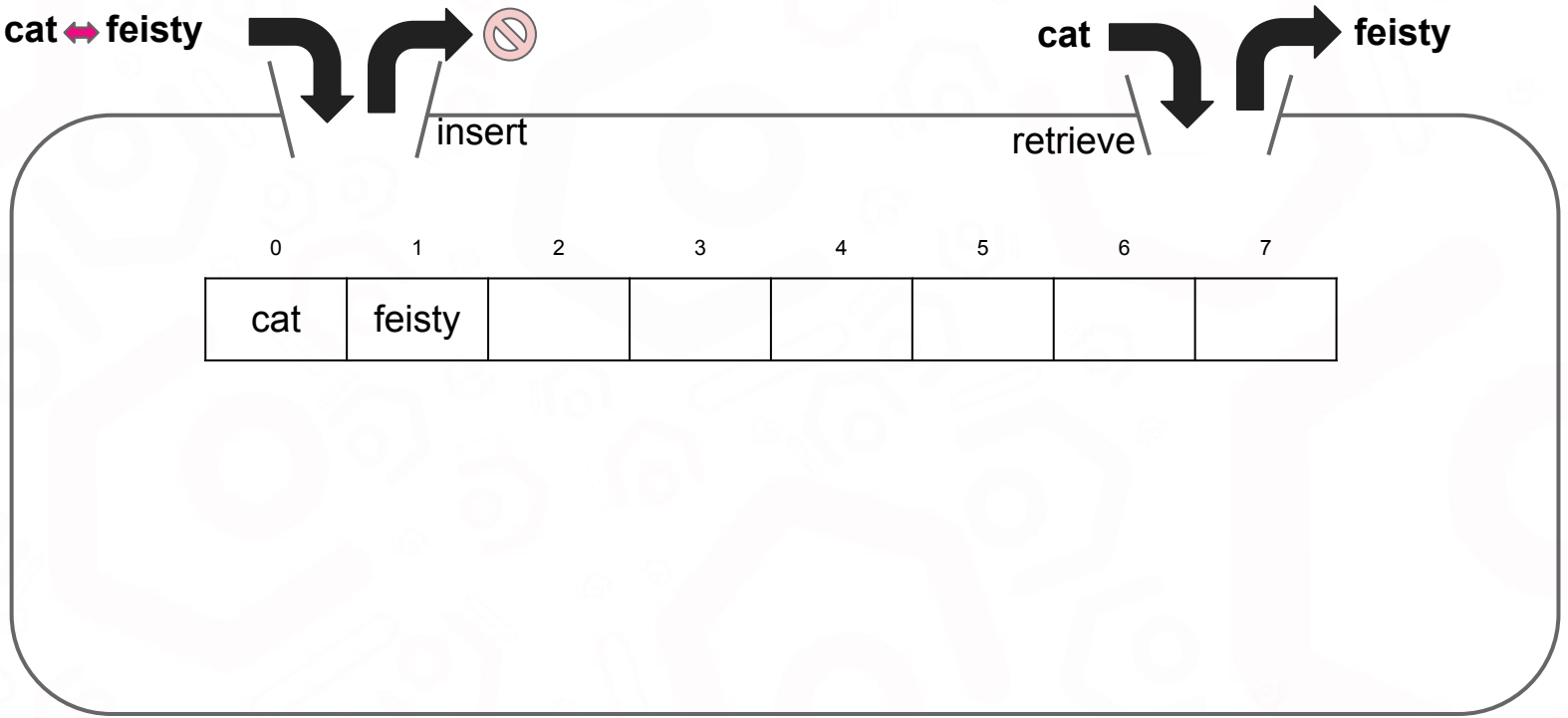
Recall that the insert and retrieve operations are constant time operations. This requirement will constrain our choice of candidate data structures. Additionally, recall that a hash table may store many associations.



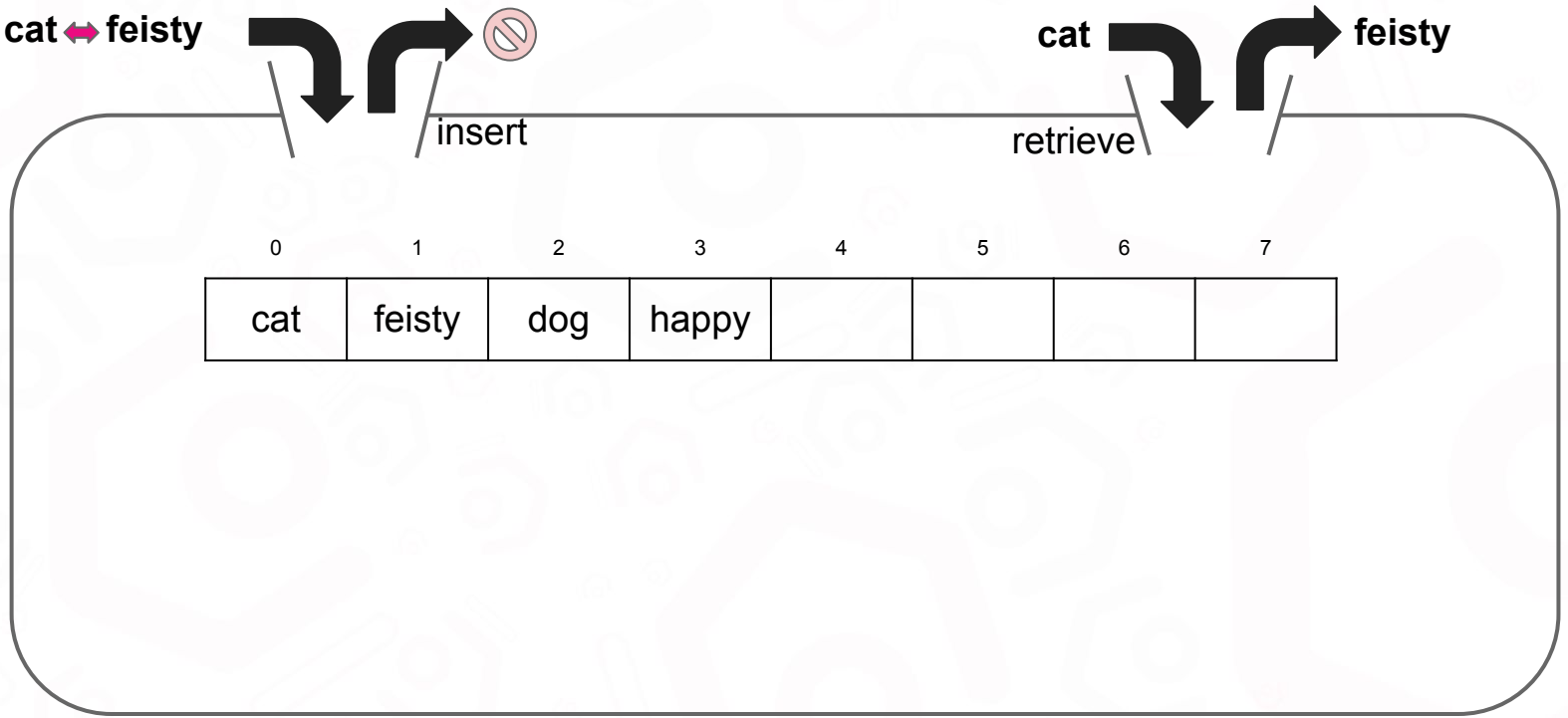
A: If you guessed an array, you are correct. An array has constant time lookup and assignment at any index -- a key characteristic -- since at any moment we may be asked to lookup any previously stored key/value combination.



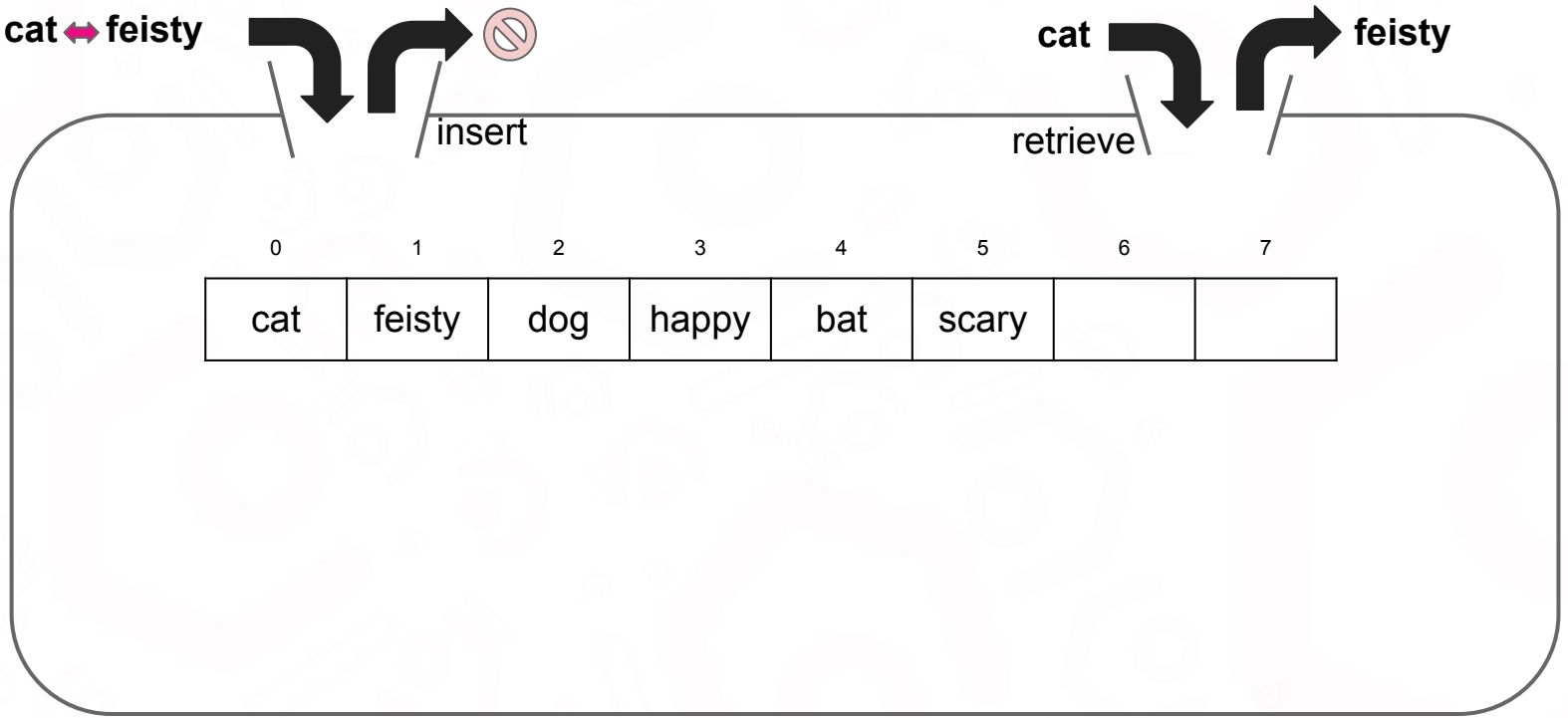
Q: What strategy should we employ to store our key/value pairs in this array?



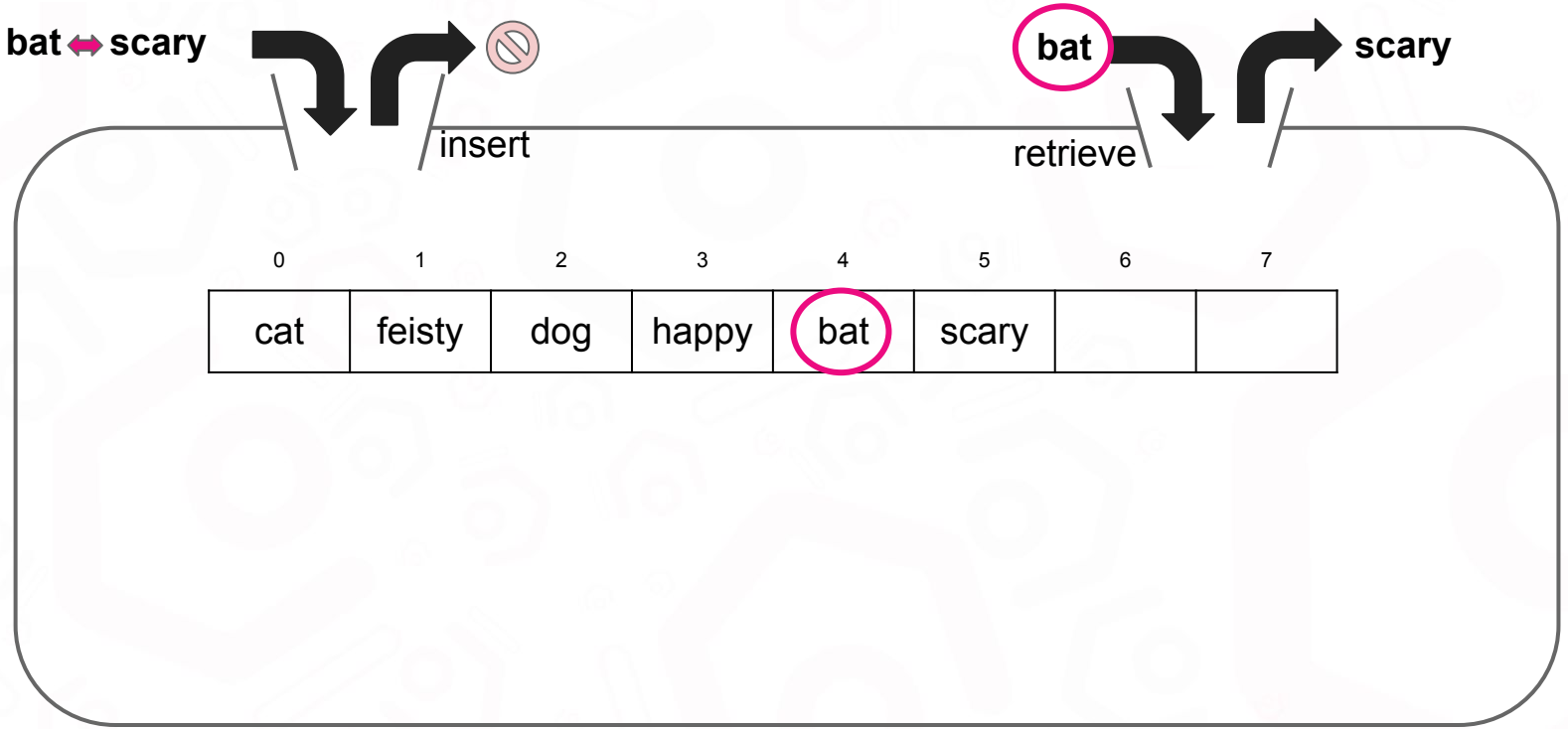
One possible approach is to append each new key/value pair to the array.



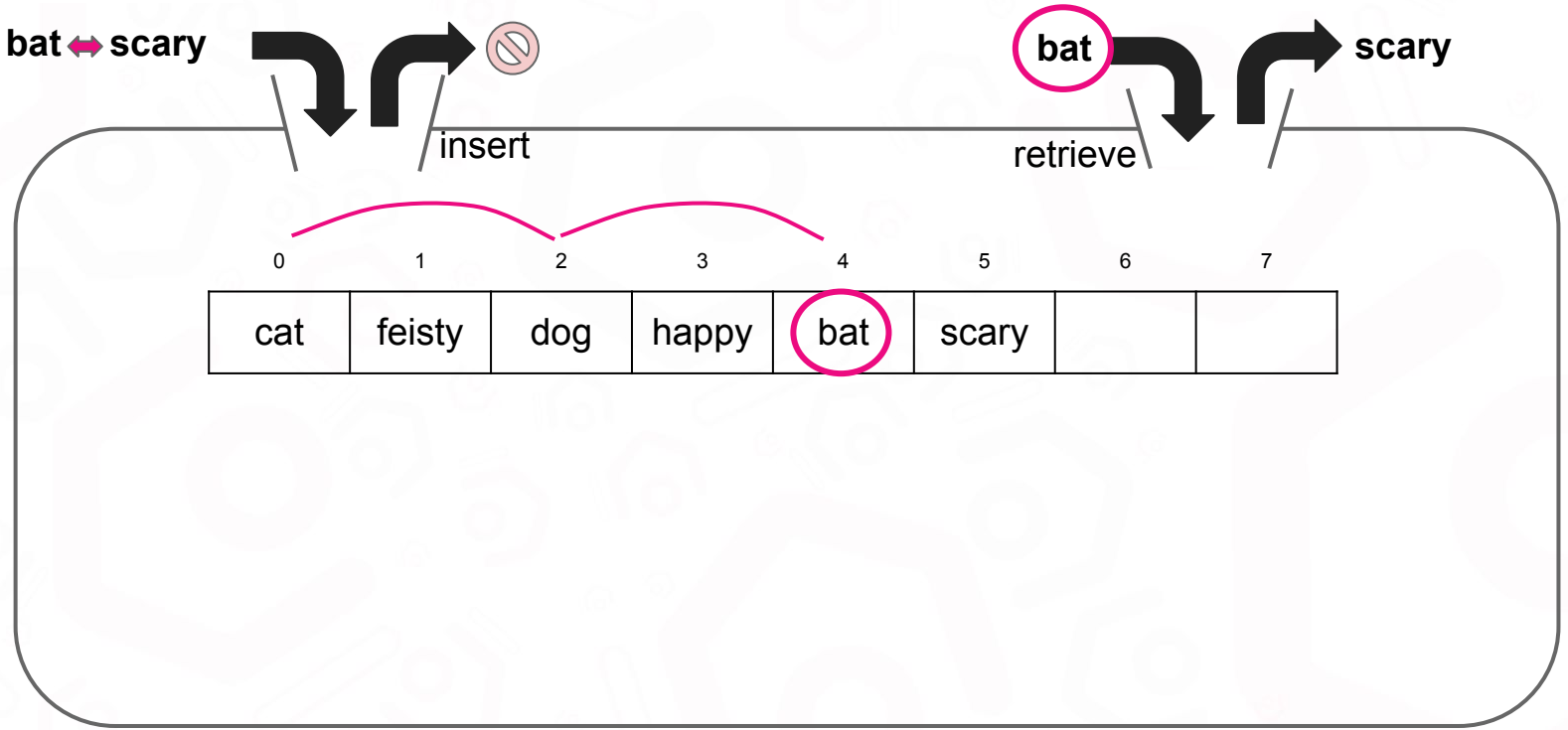
Every time a new key/value pair is offered on the interface, it is added to the end of the array.


cat ⇔ feisty
insert
cat
feisty
retrieve
0
1
2
3
4
5
6
7
cat
feisty
dog
happy
bat
scary

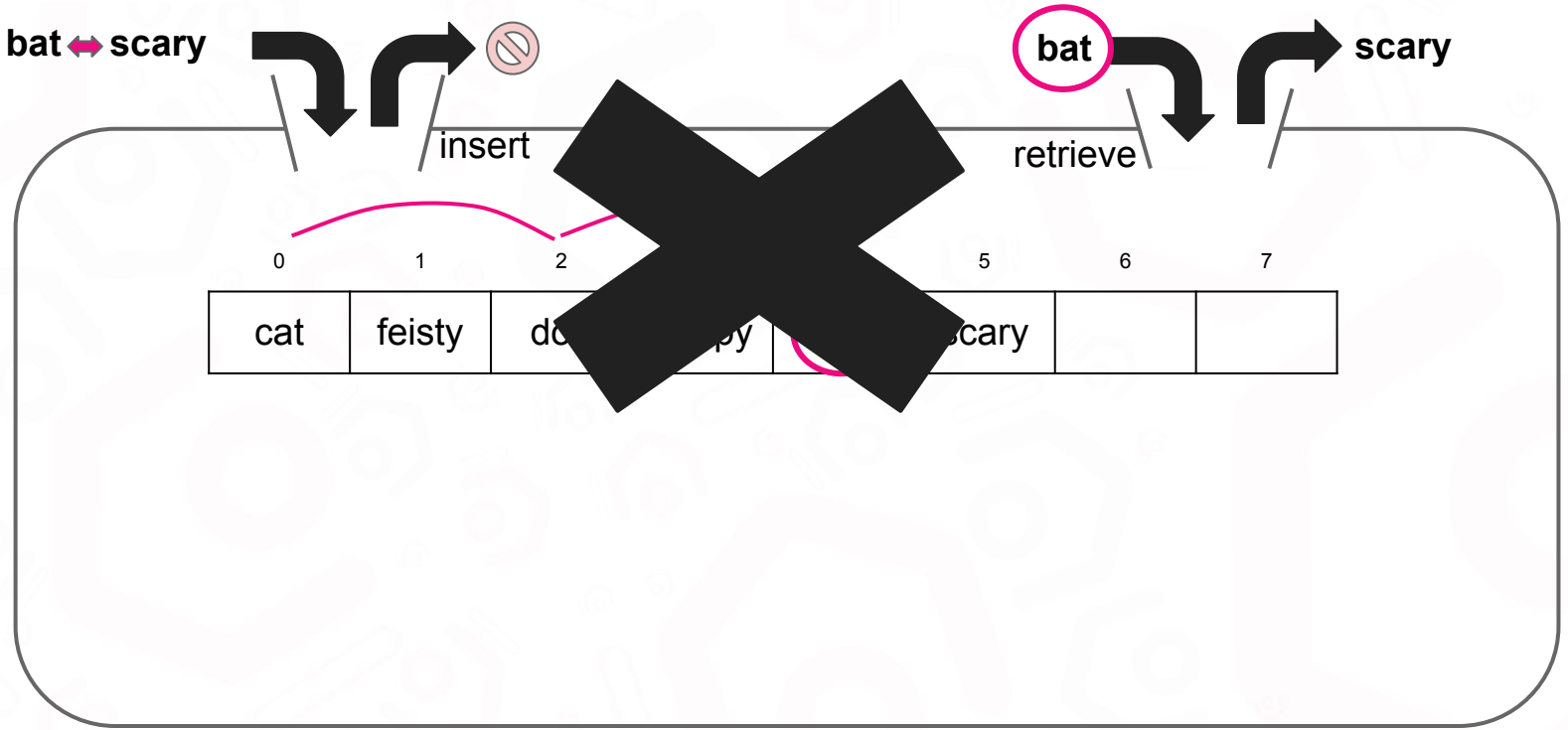
And so forth



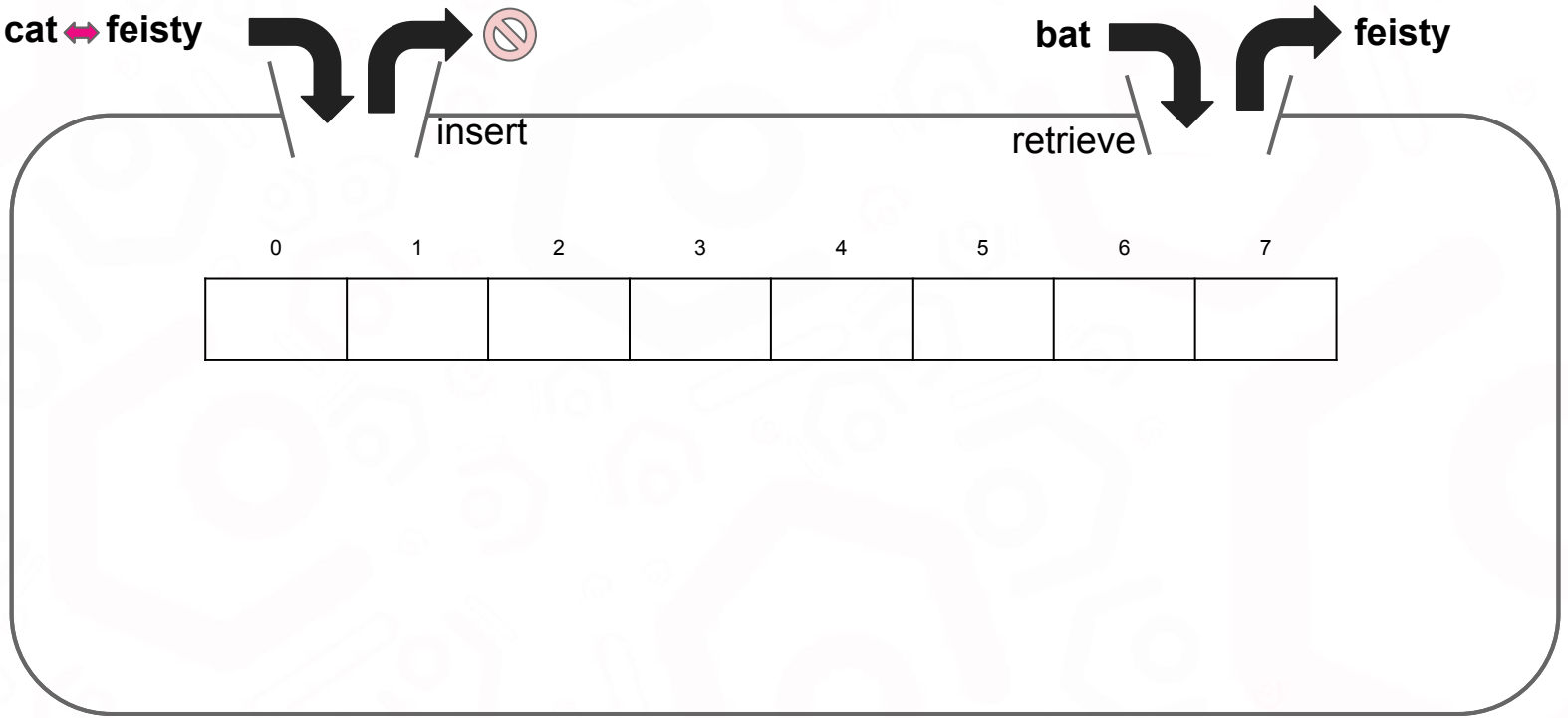
Consider what effect this design choice has on our retrieve operation. **Q:** What is the time complexity of finding the key 'bat' in this array?



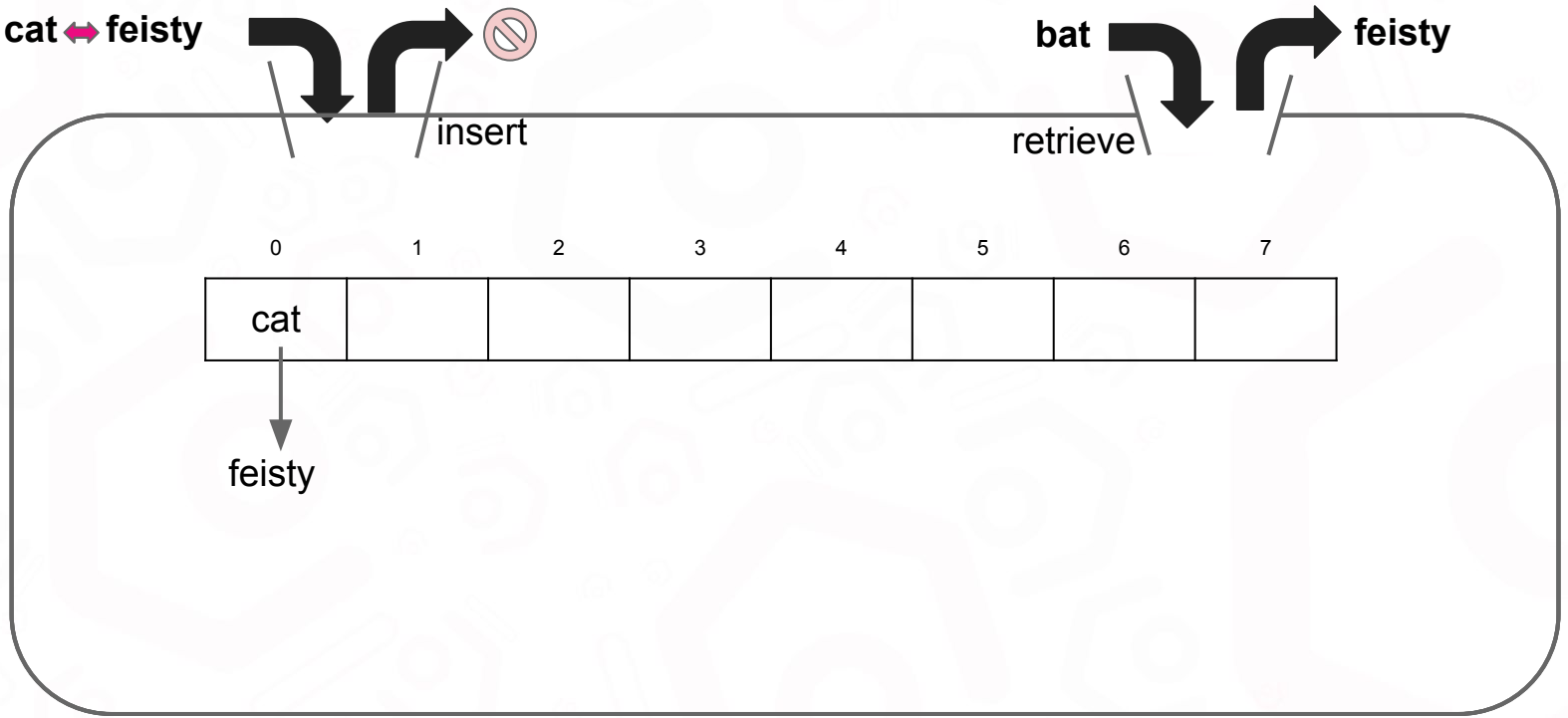
A: This is a linear time operation. We must examine sequential locations in the array to determine if it contains the value 'bat'. Even though we are considering alternating locations in the array, this is still a linear time operation.



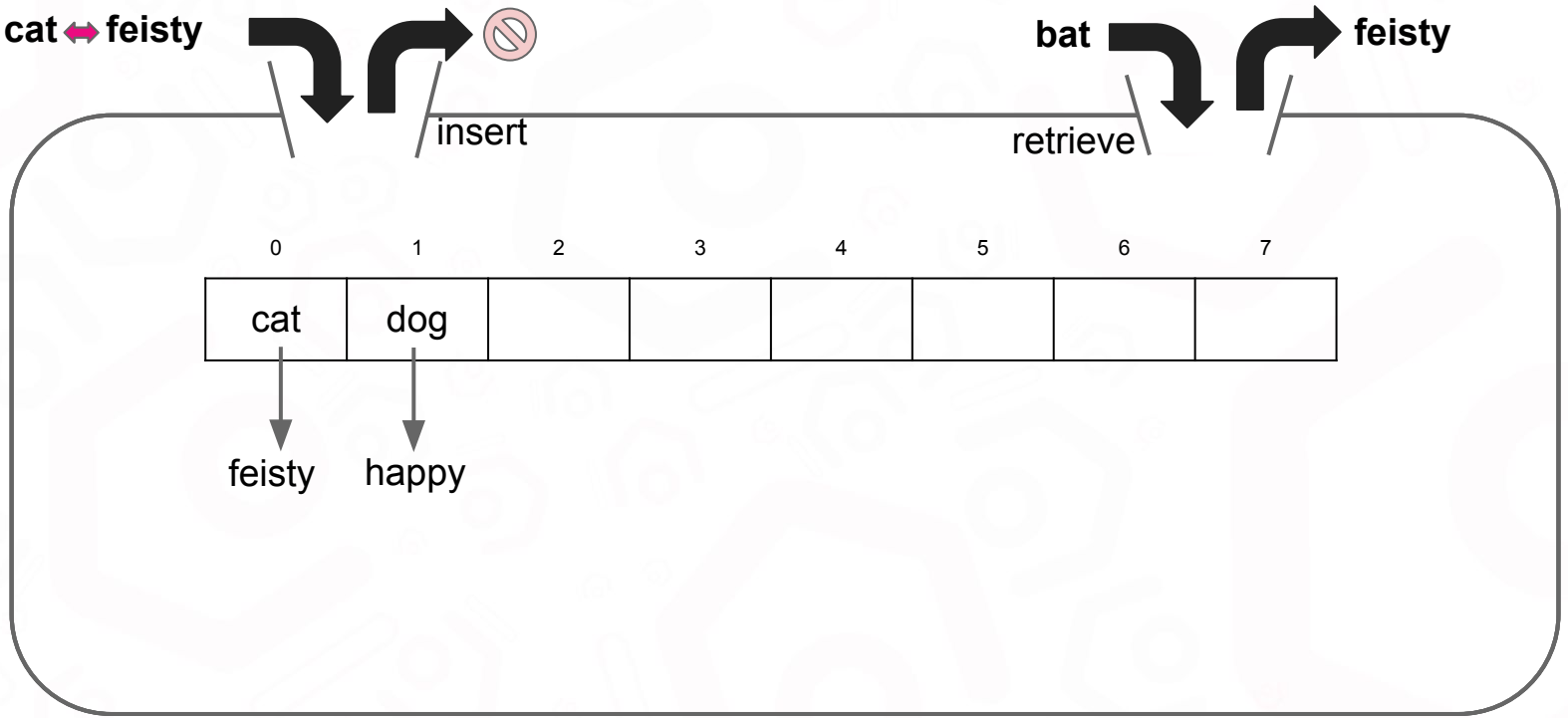
This approach does not conform to the established requirement of retrieve operating with constant time.



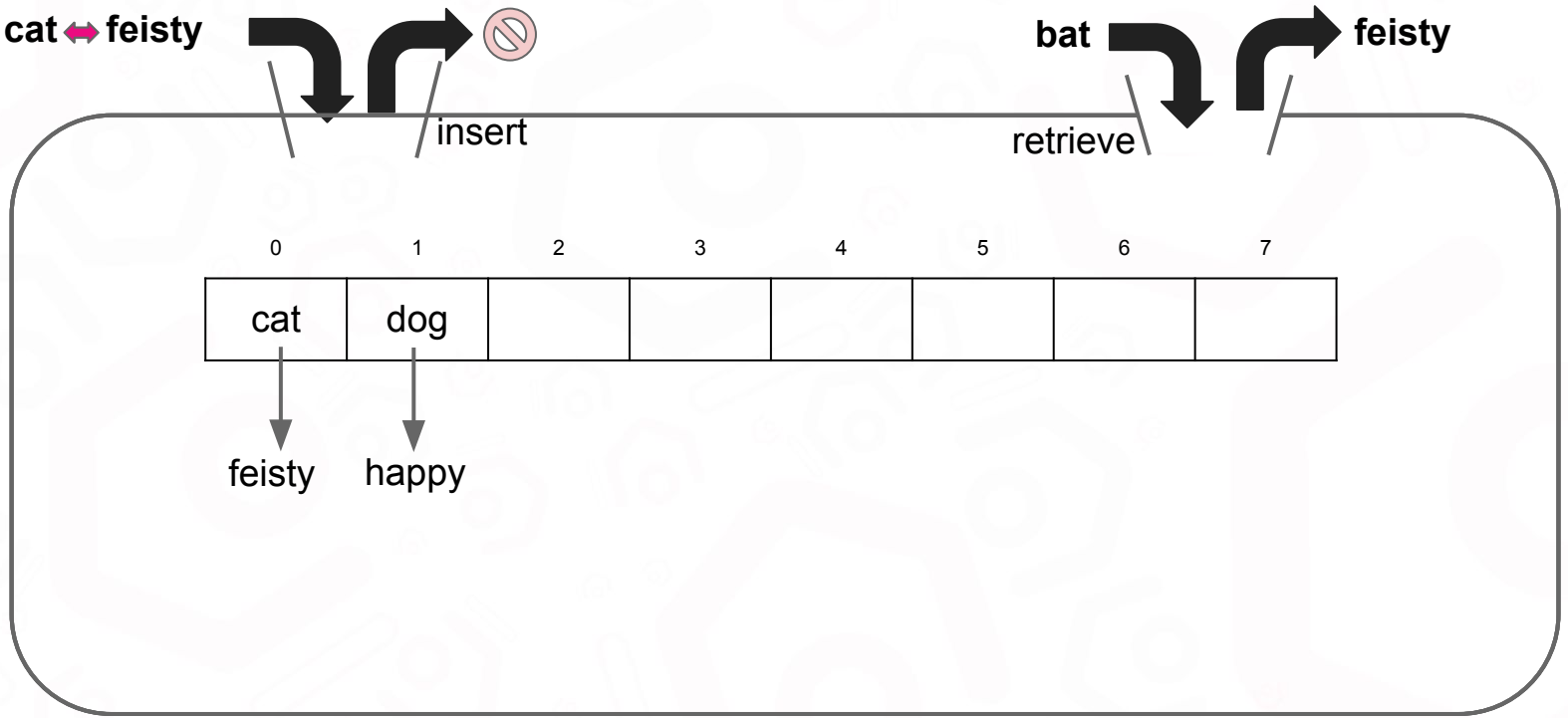
Let's try another approach!



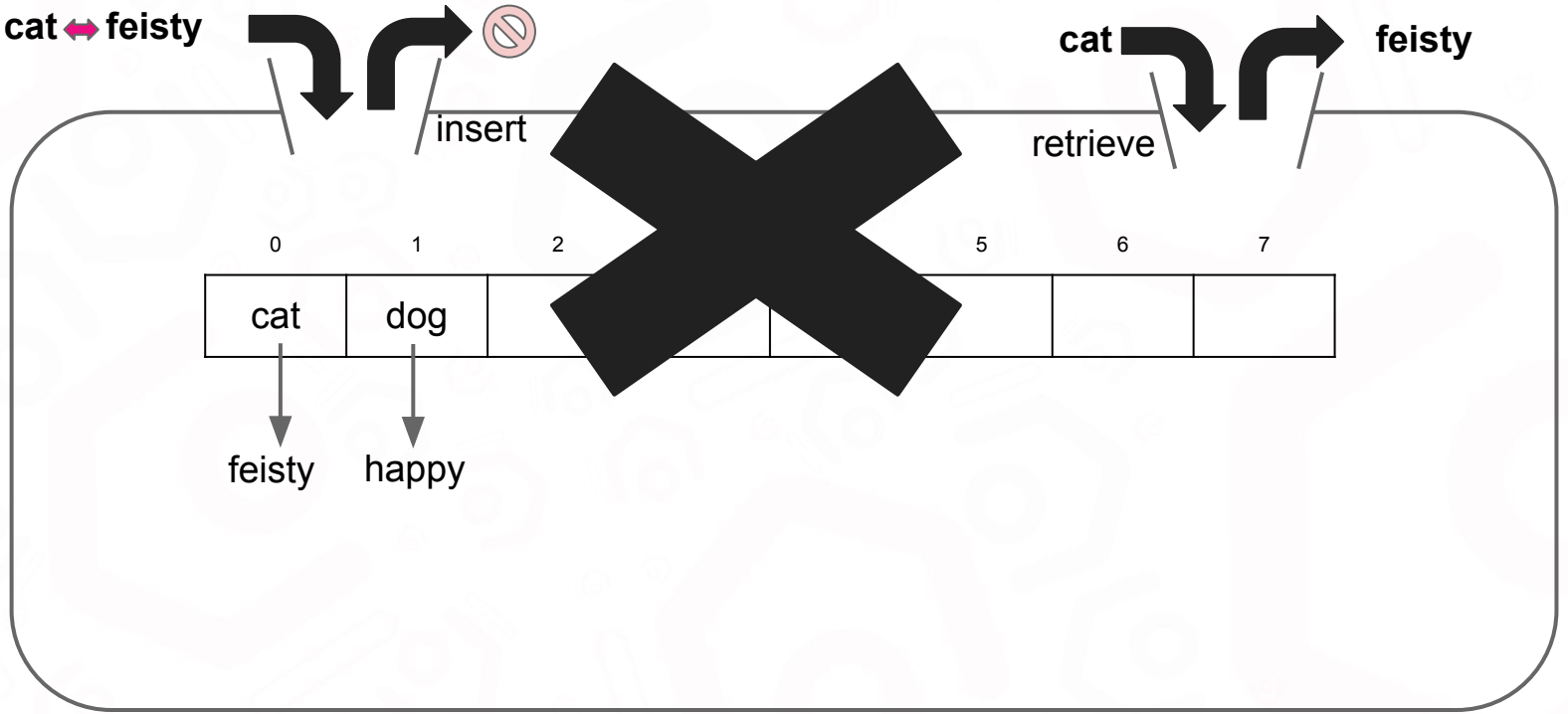
Since we're using JavaScript, it's tempting to want to use an object to store these associations.



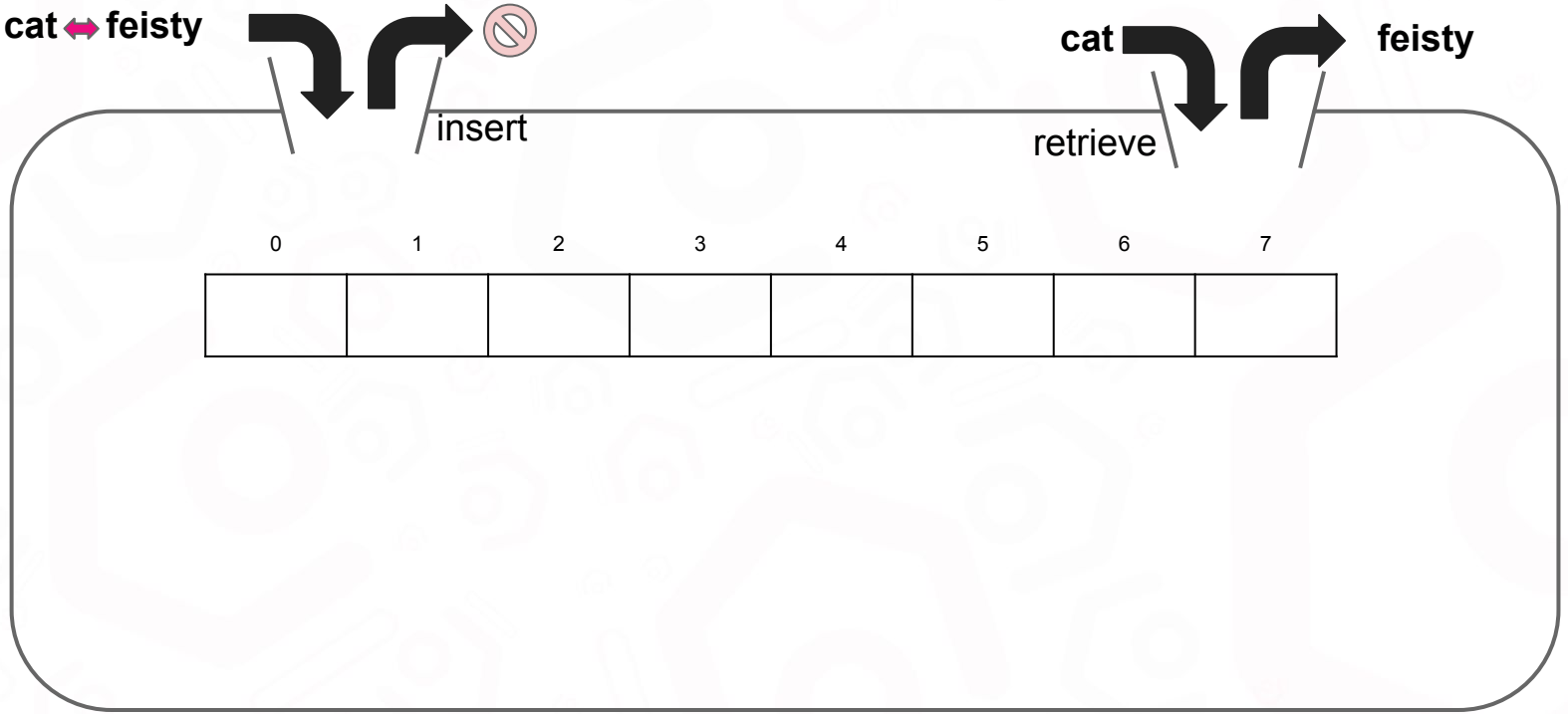
Adding additional key/value pairs as they are presented on the interface.



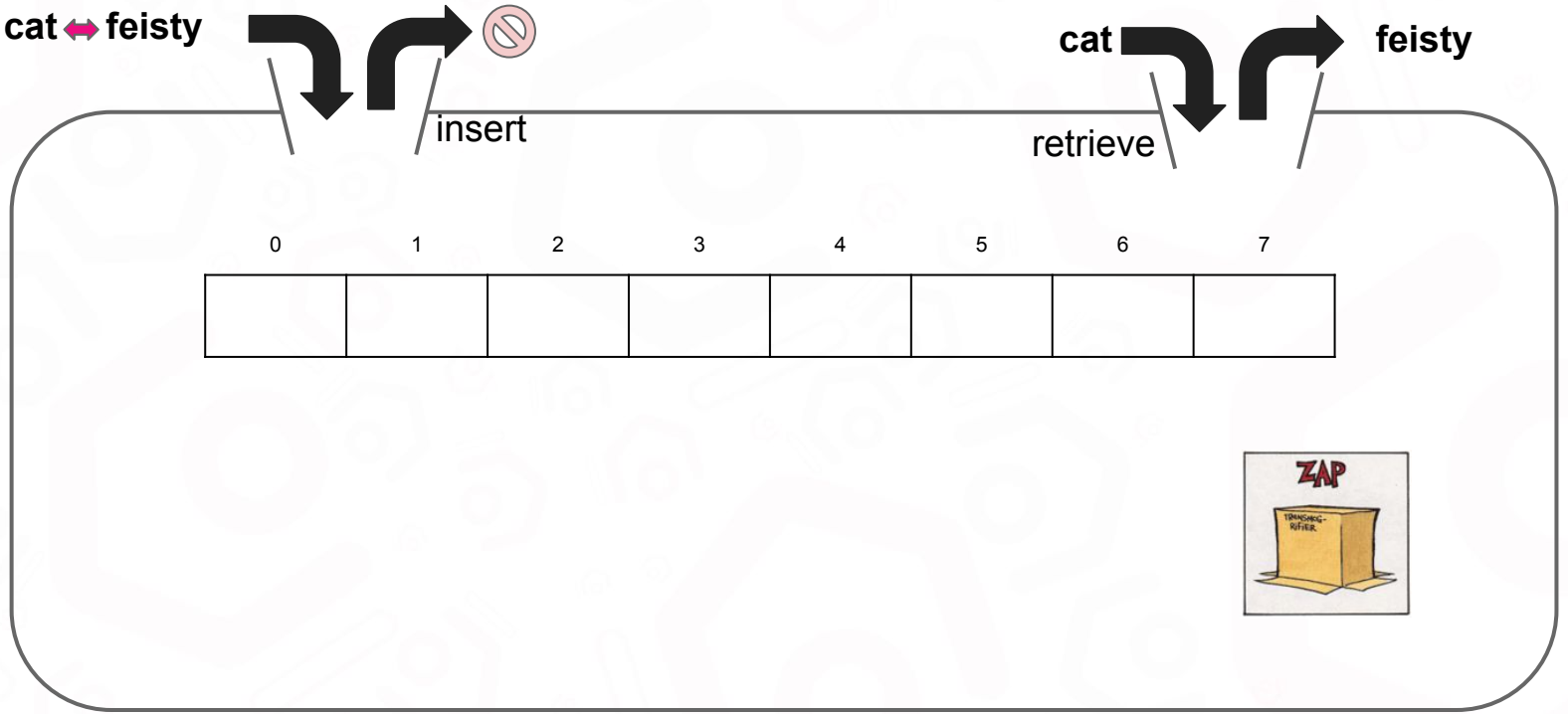
While tempting, this isn't possible. If you recall, JavaScript objects are implemented using hash tables, so we can't use an object to implement a hash table.



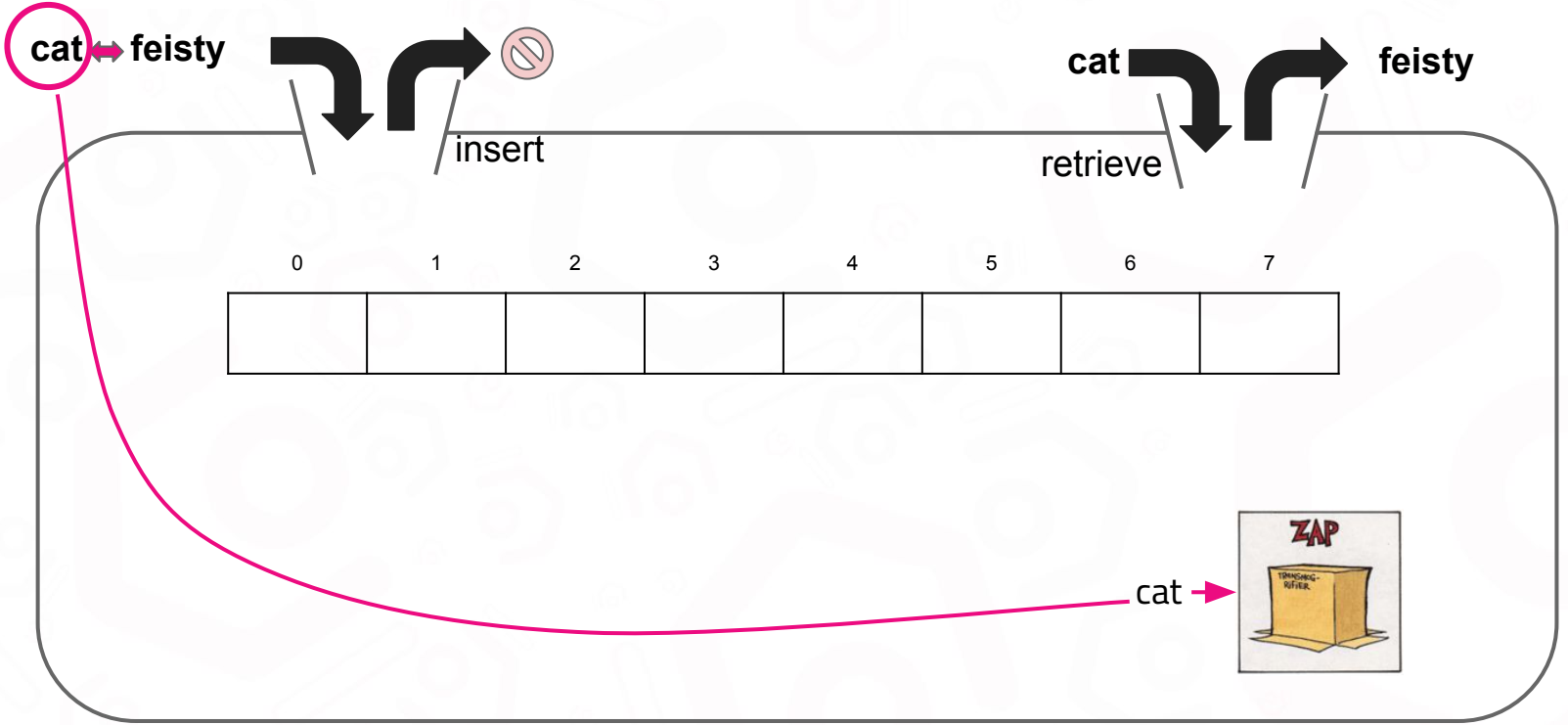
This approach doesn't work either as we find ourselves in a chicken/egg situation.



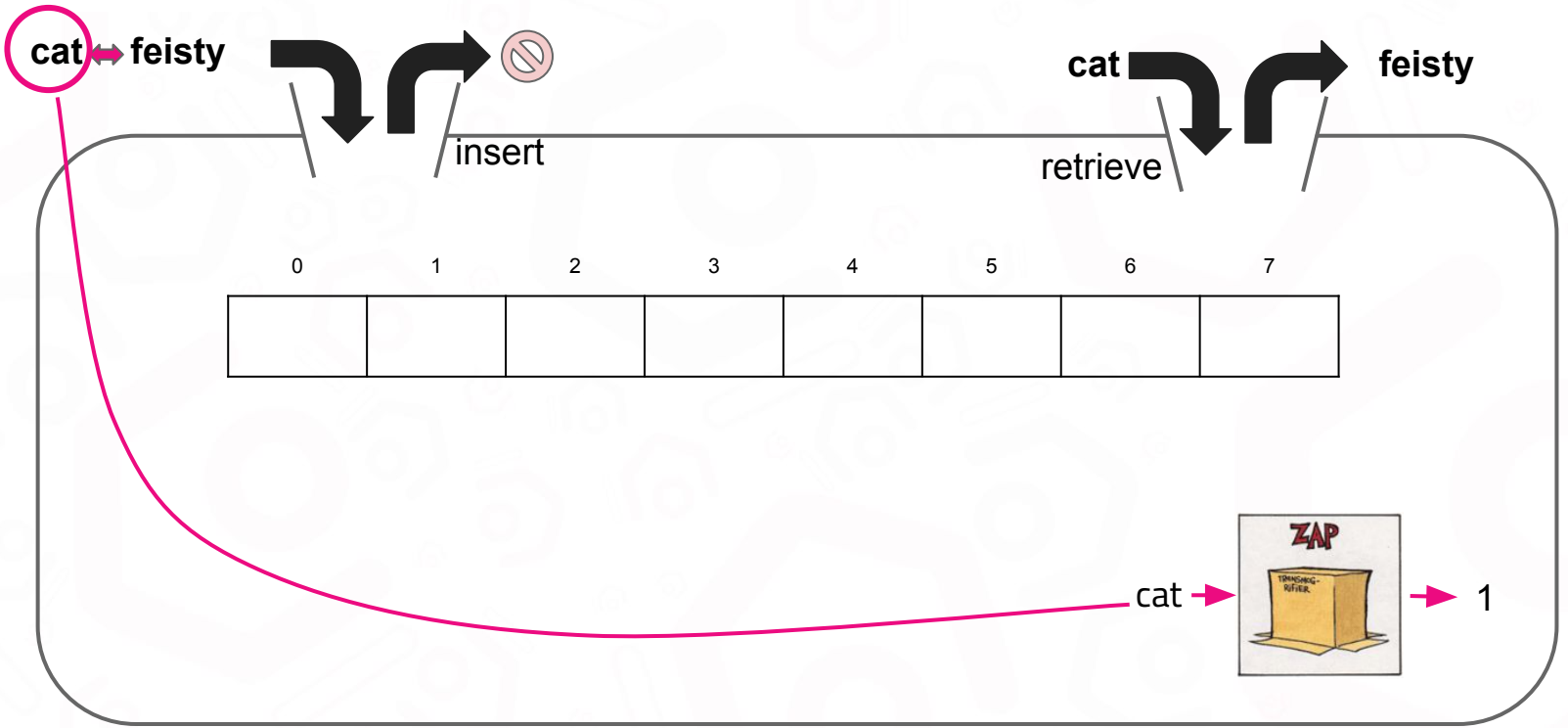
Let's try yet another approach! What if we could magically turn the key to an array index?



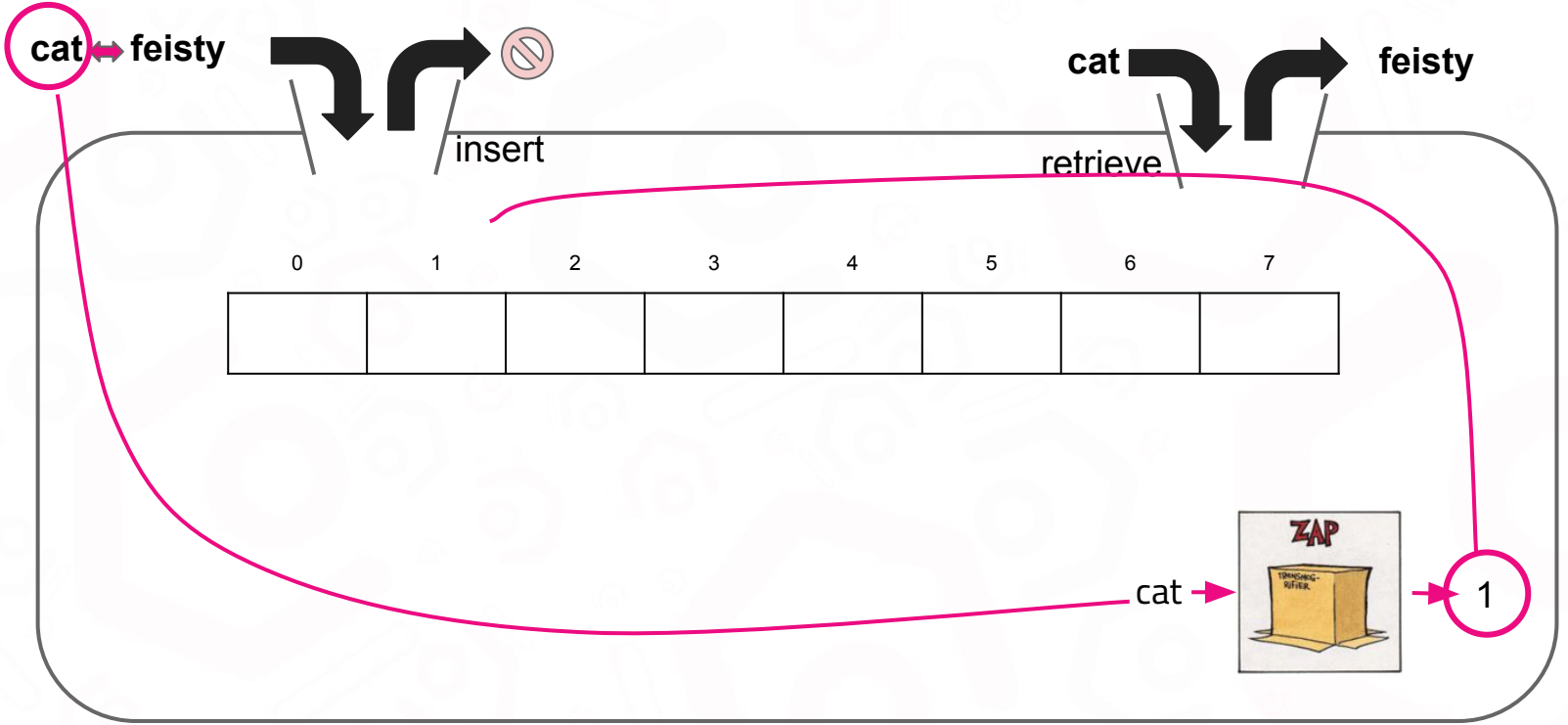
It turns out the we can! We can use a transmogrifier to do this.



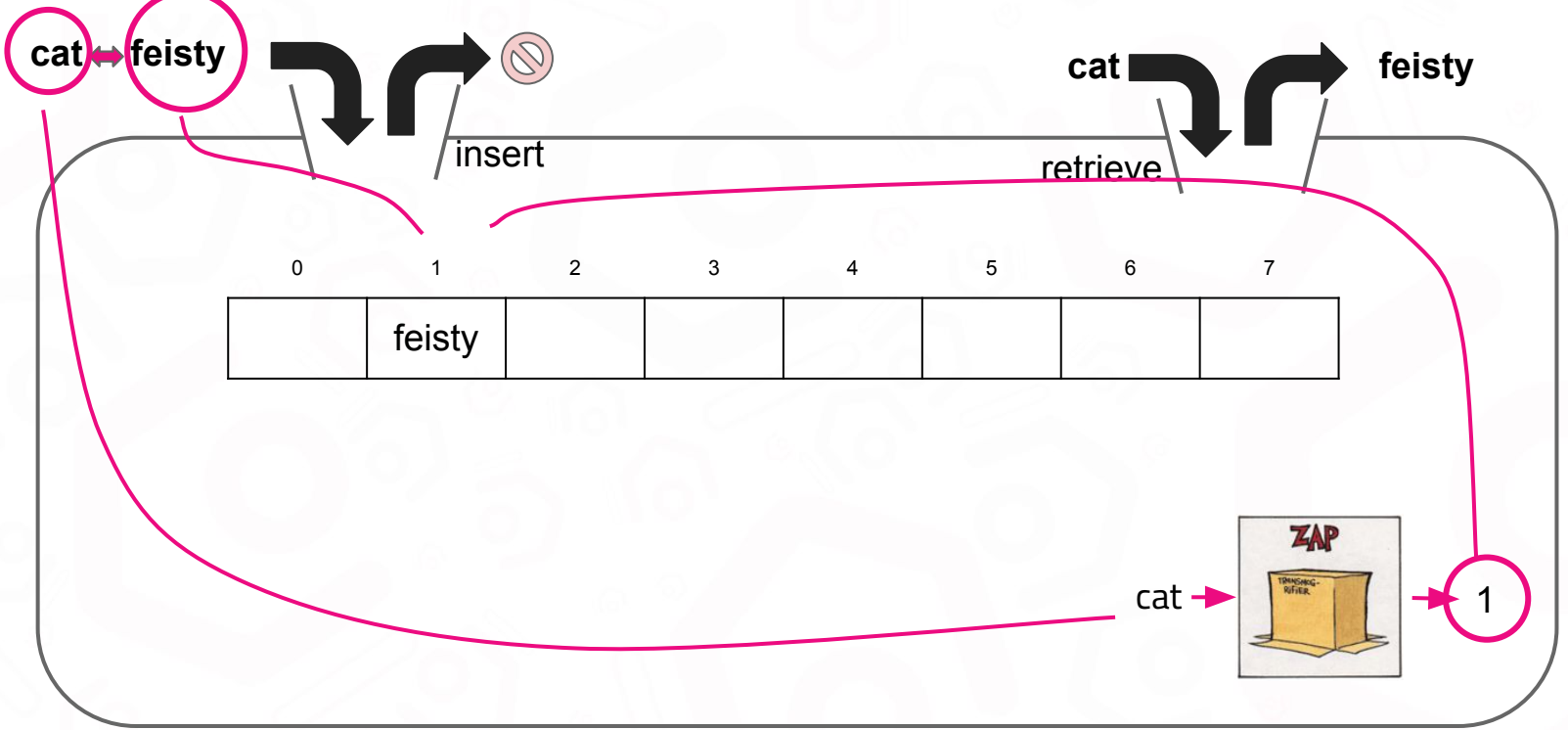
The transmogrifier accepts the key as its input...



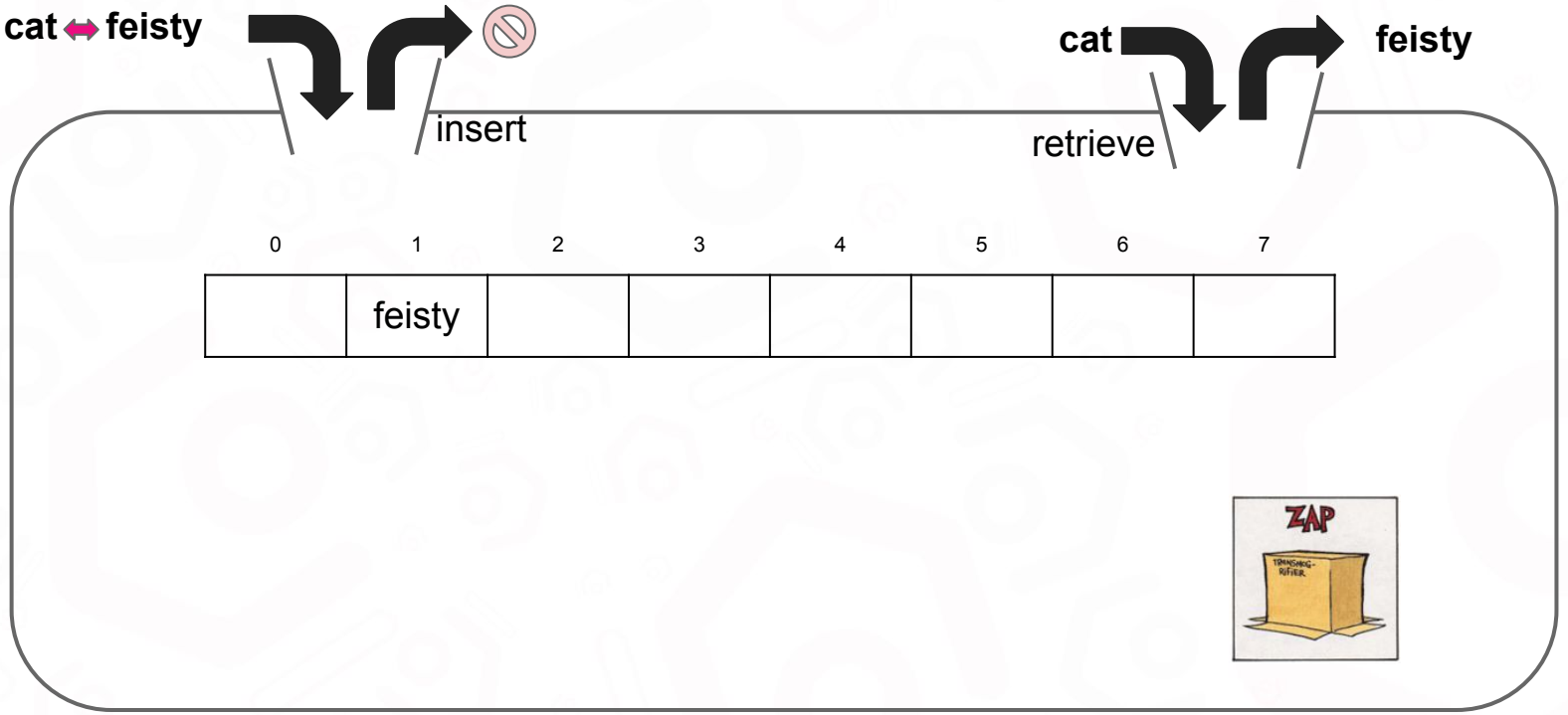
... and turns that key into a number.



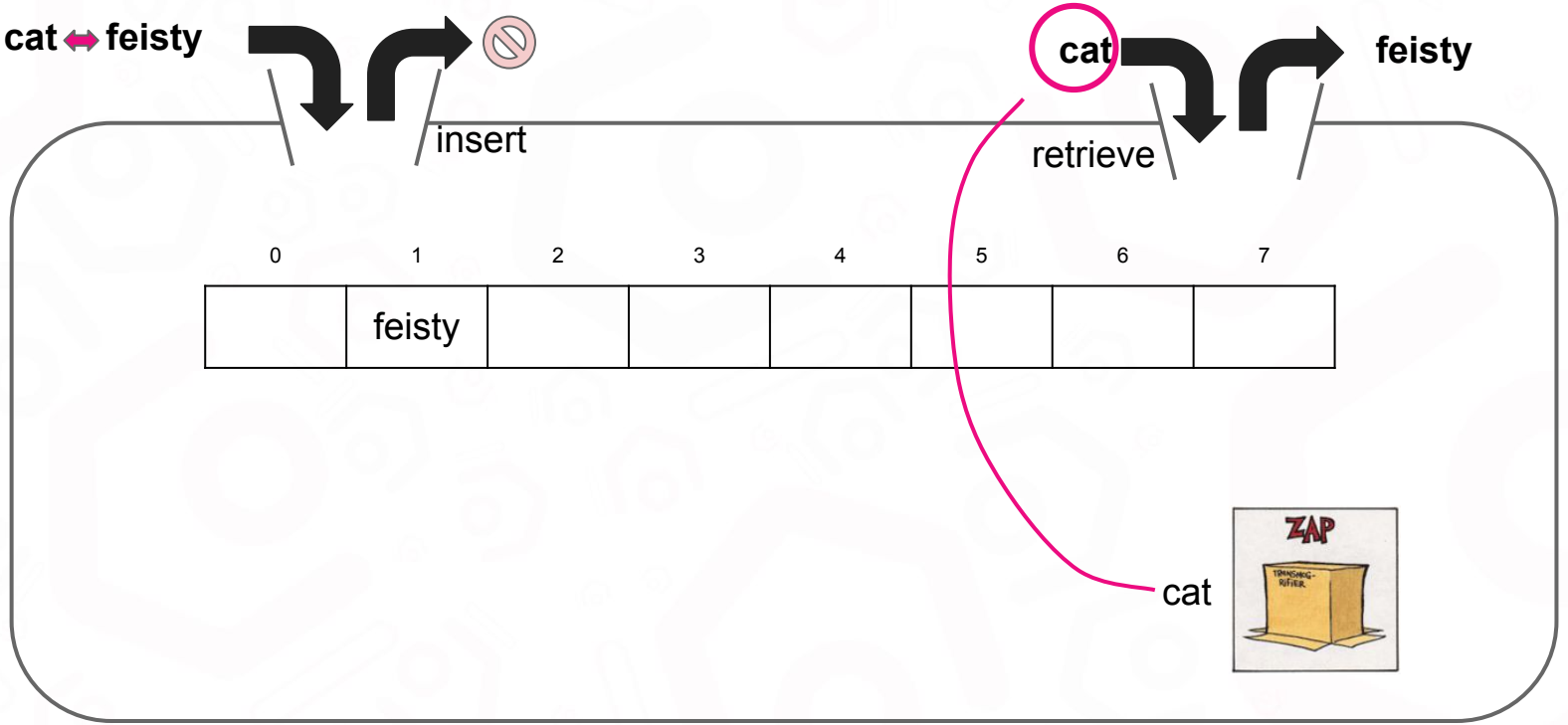
The number is used an index in the array...



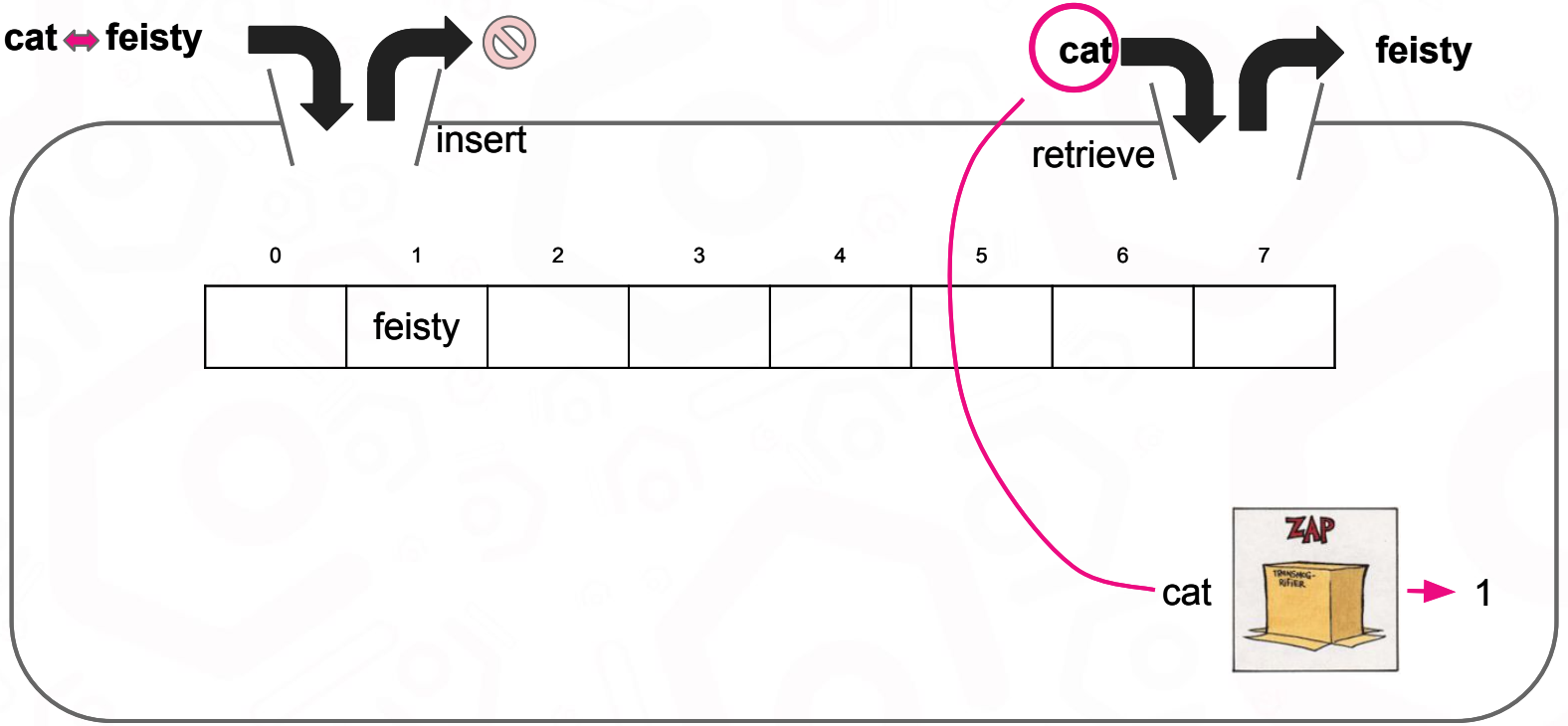
... where we assign the value in constant time.



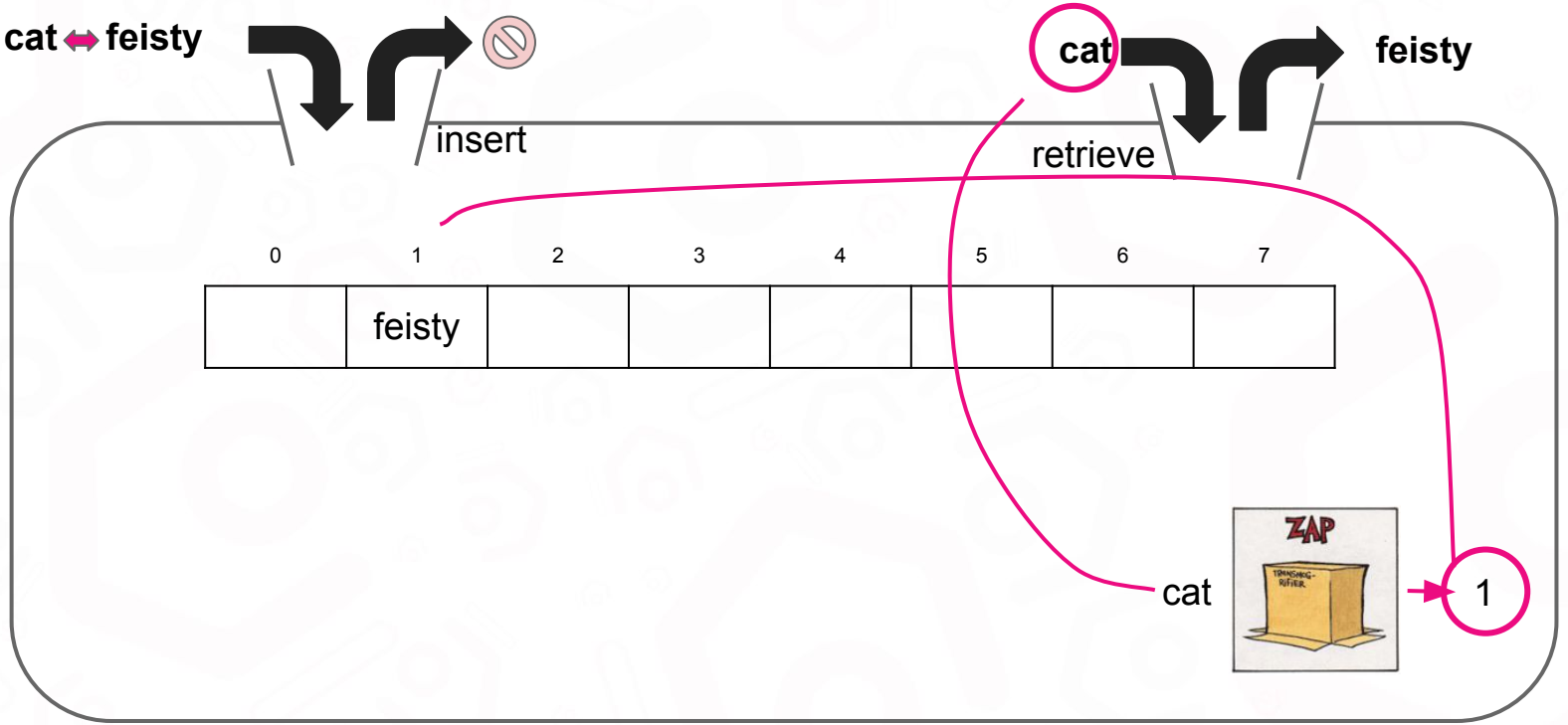
What about retreive?



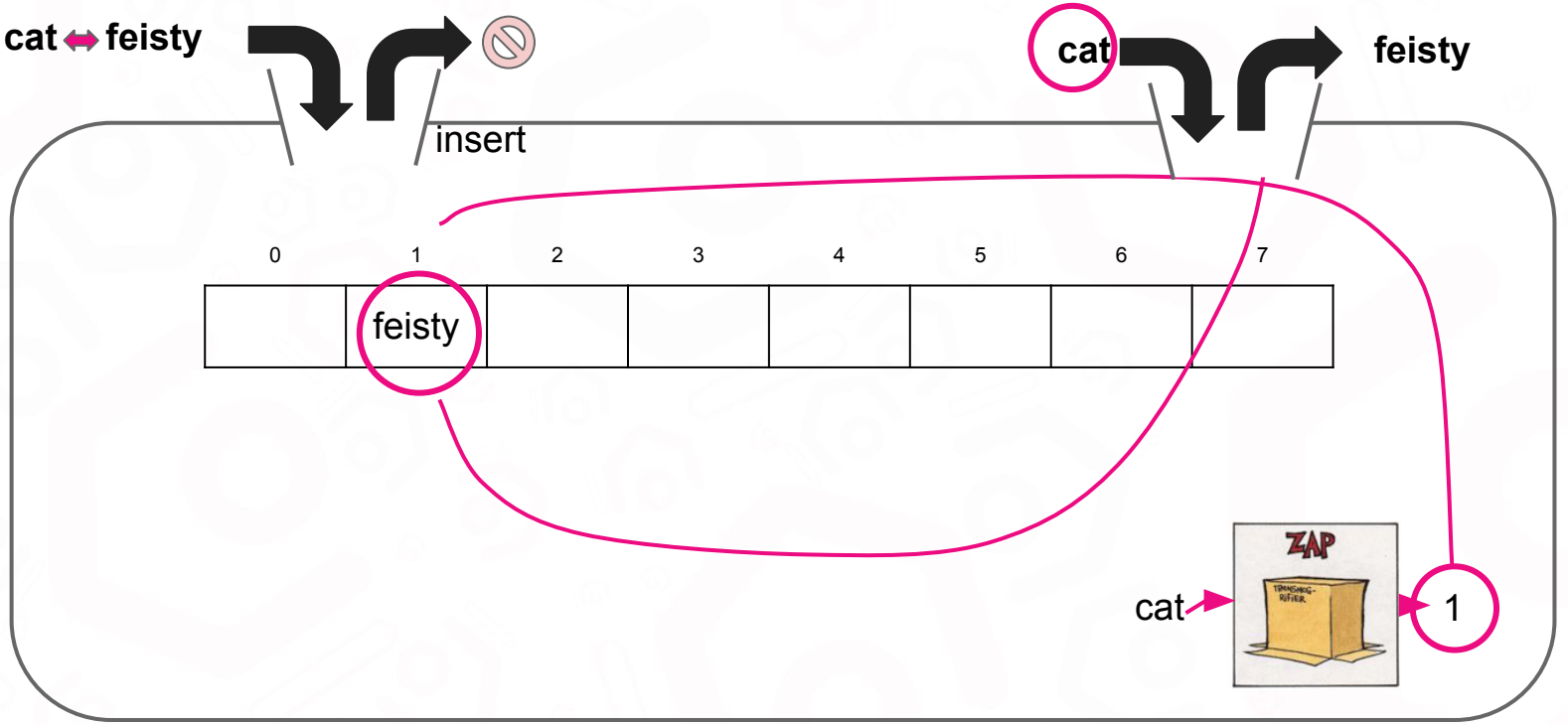
The transmogrifier accepts the key as its input...



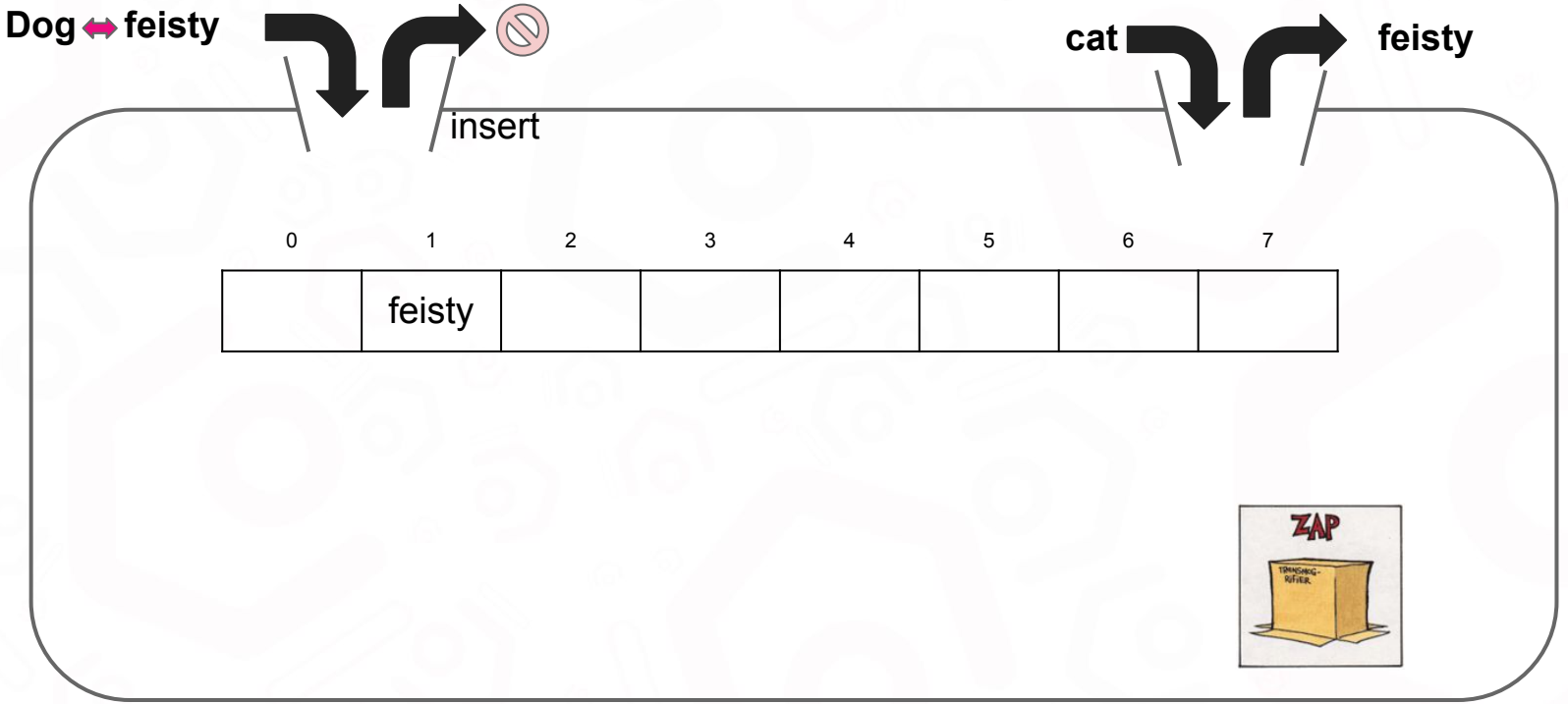
... and turns that key into a number.



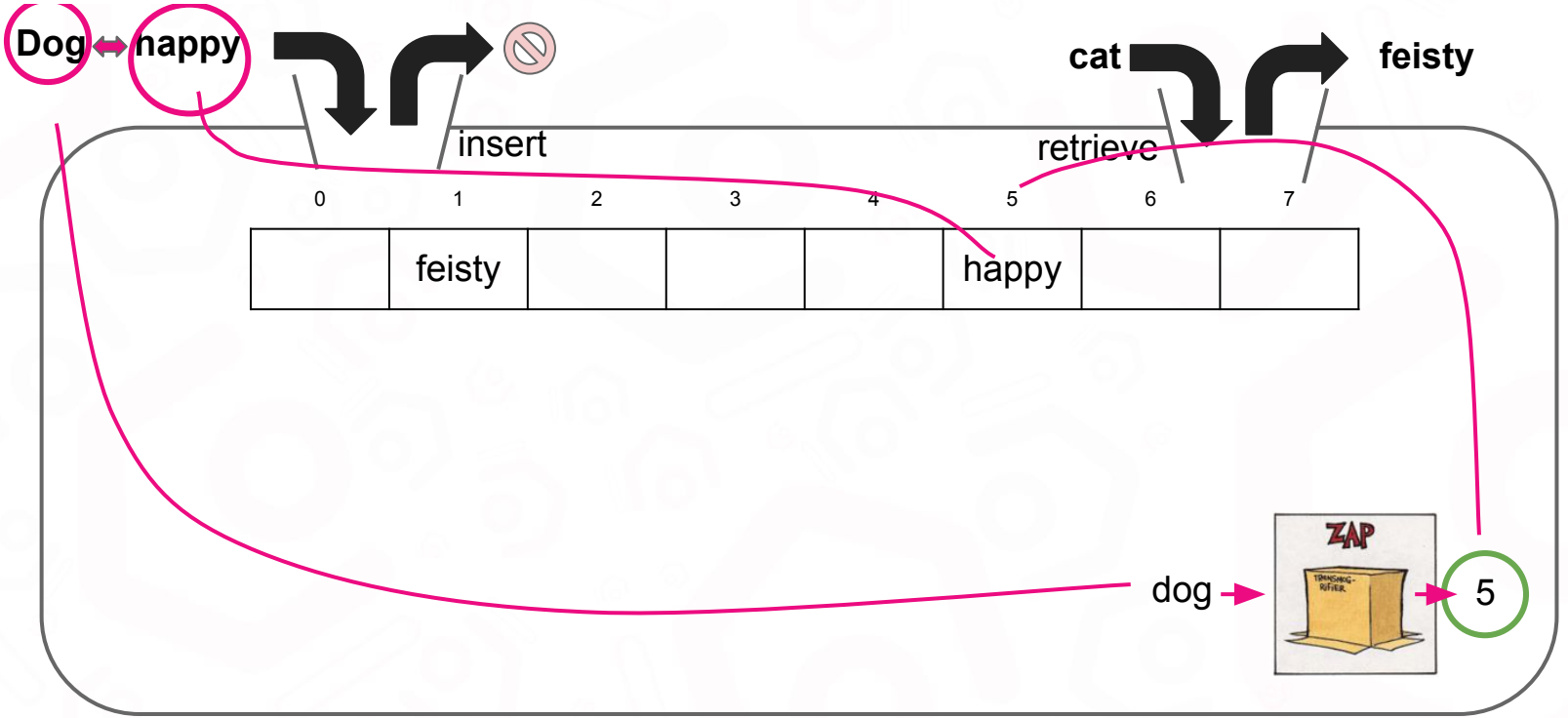
The number is used as an index in the array...



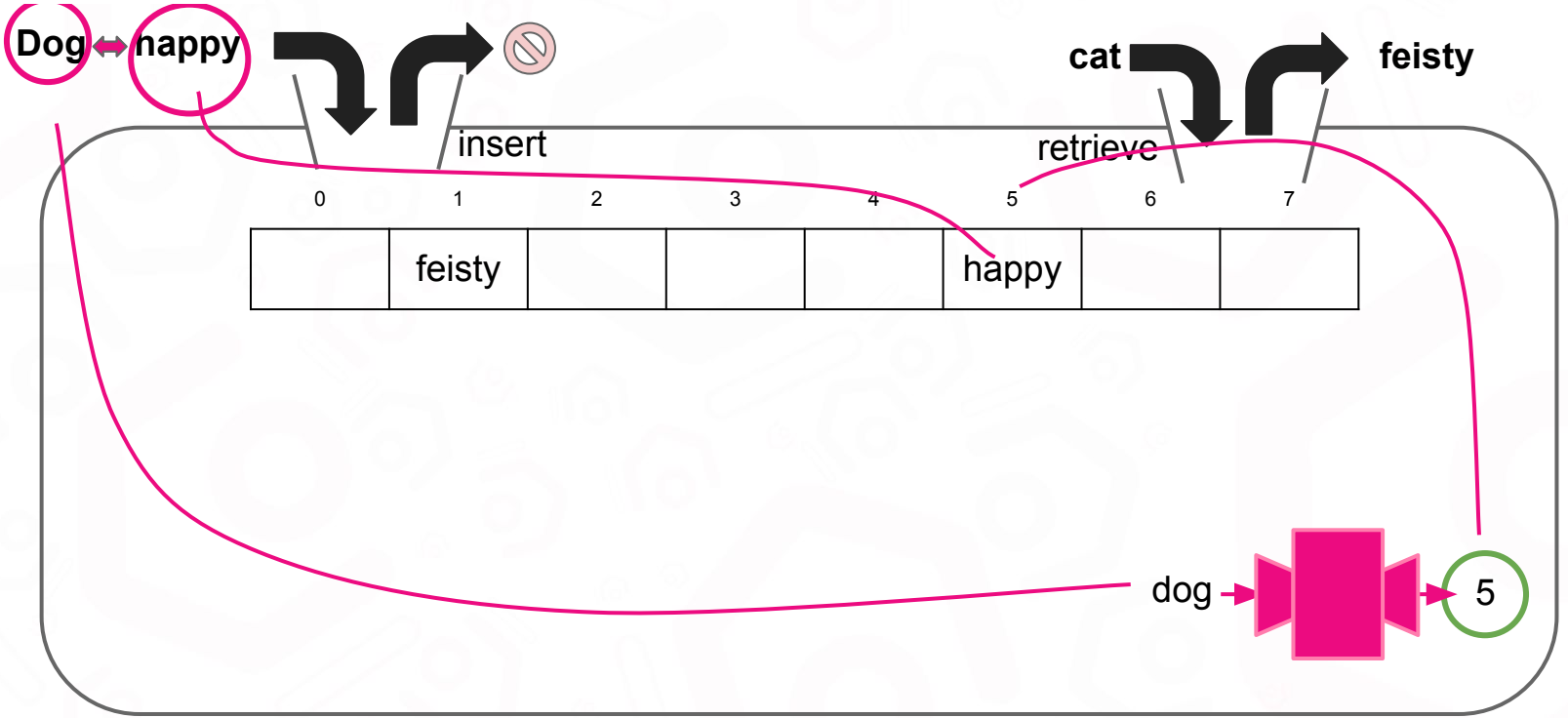
... where we lookup the value in constant time and return it.



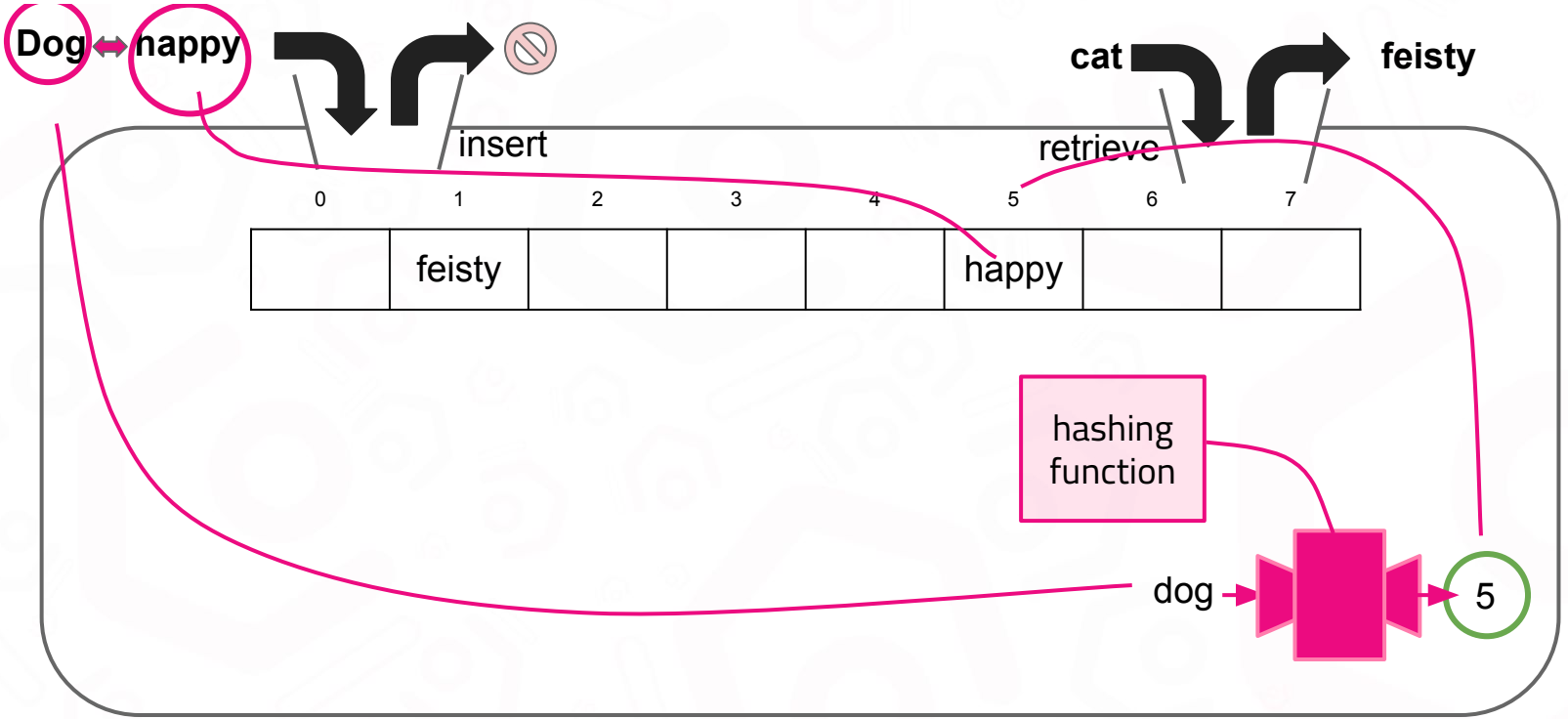
When we are asked to insert another key-value pair, we can repeat this same process.



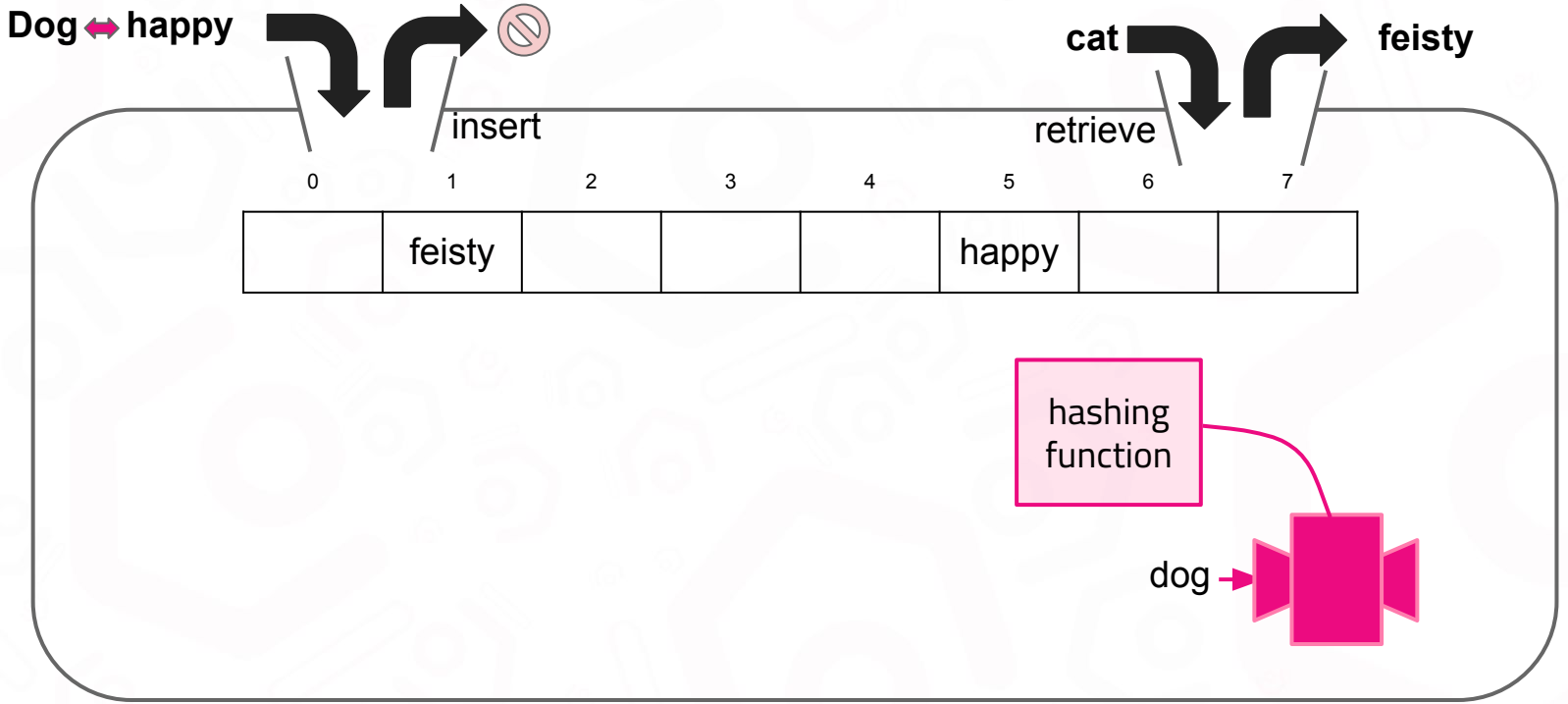
The transmogrifier accepts the key as its input… and turns that key into a number. The number is used an index in the array… where we assign the value in constant time.



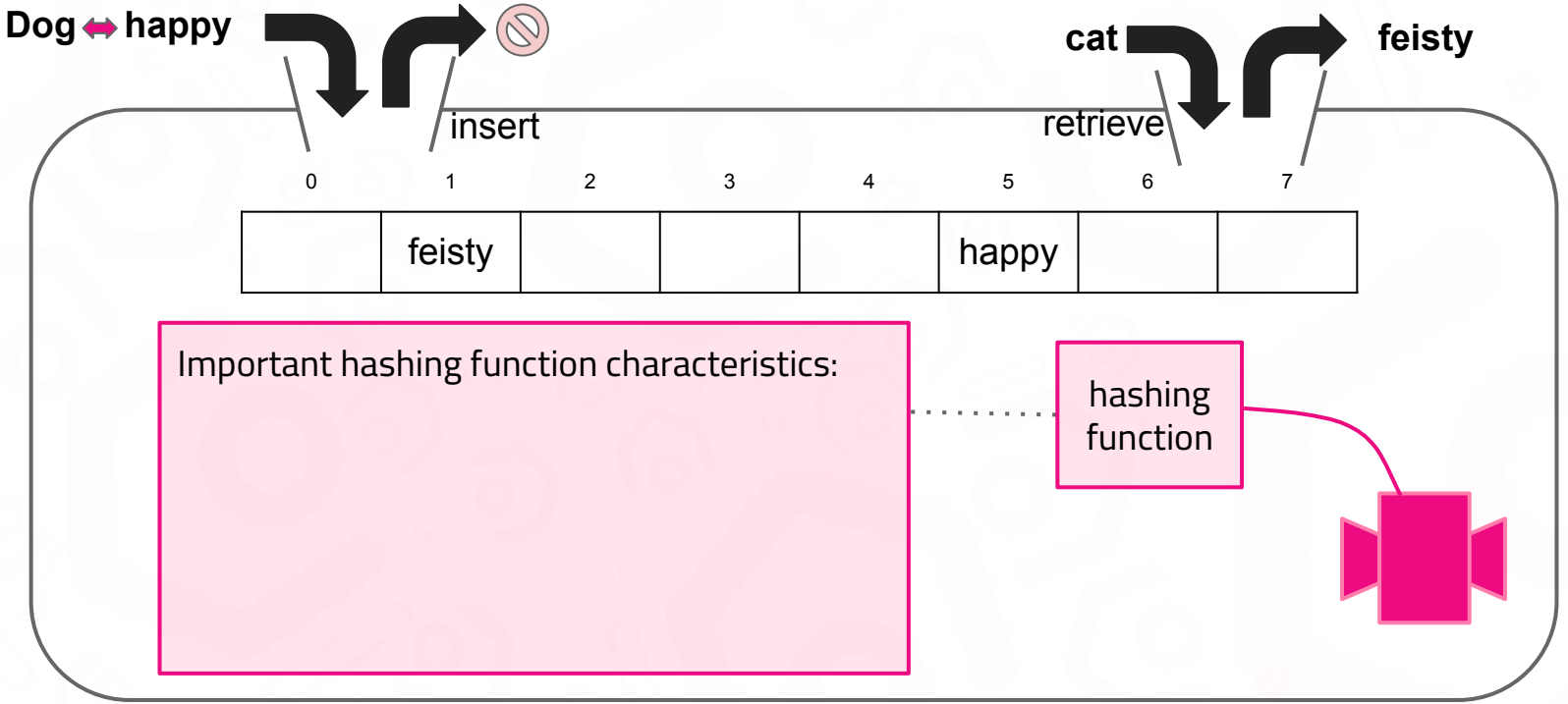
After observing a few of these interactions, you might have noticed the transmogrifier isn't all that magical after all -- it is just a function that performs a transformation. It transforms a string input to a numerical output.



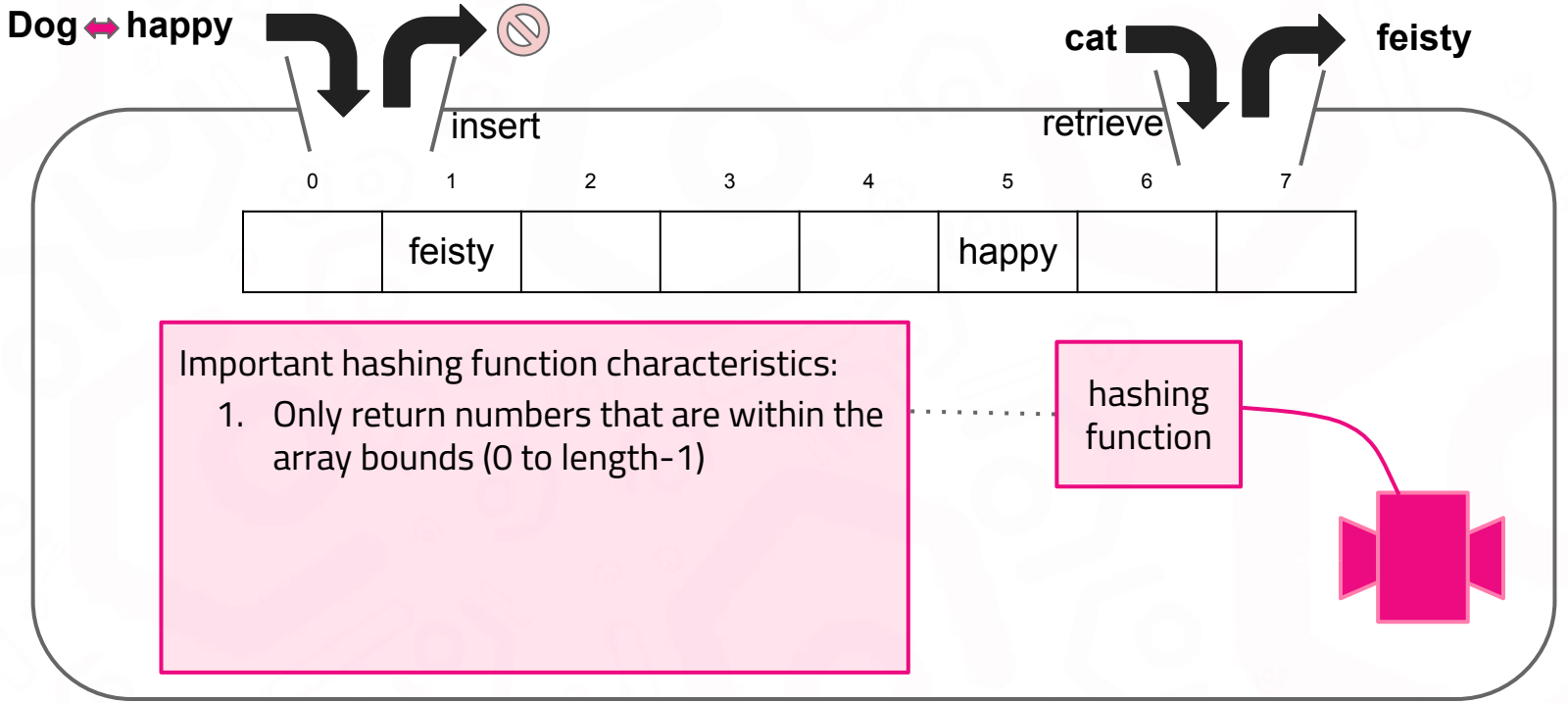
This function is more commonly referred to as a hashing function.



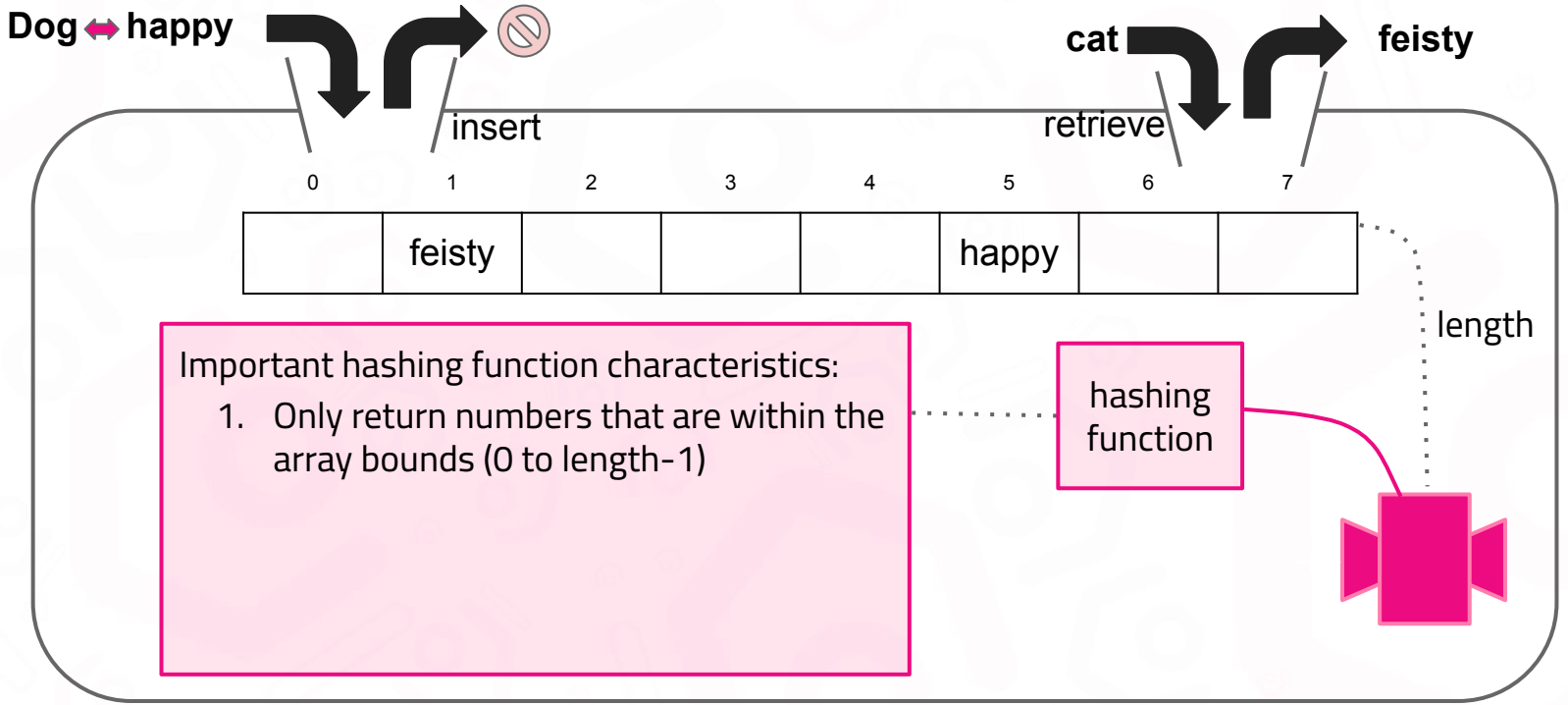
Before diving into the remaining hash table implementation, it's worth investing some time into understanding more details about the hashing function.



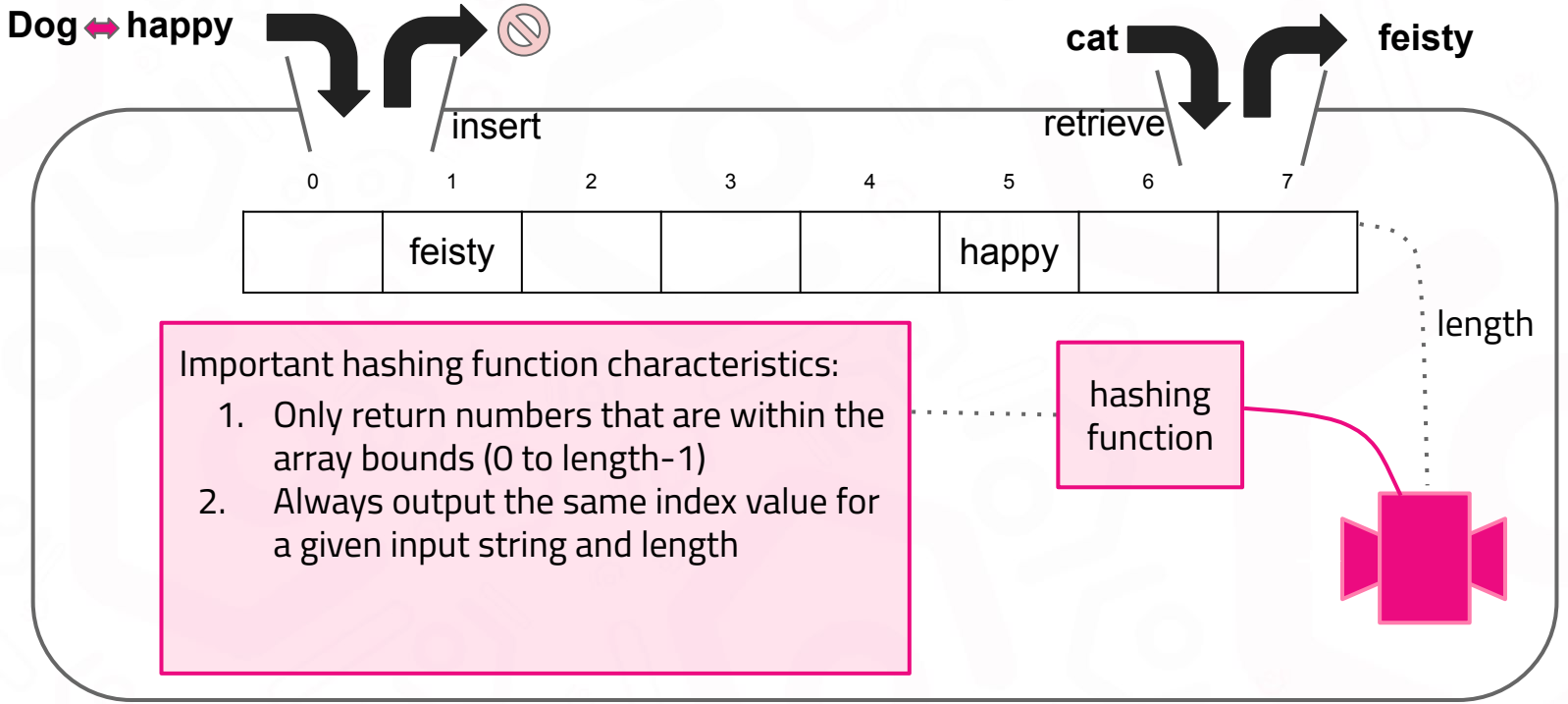
For the hashing function to operate within the constraints of our system, it must exhibit certain characteristics.
Q: Can you think of anything that must be true about the hashing function for our hash table to work correctly?



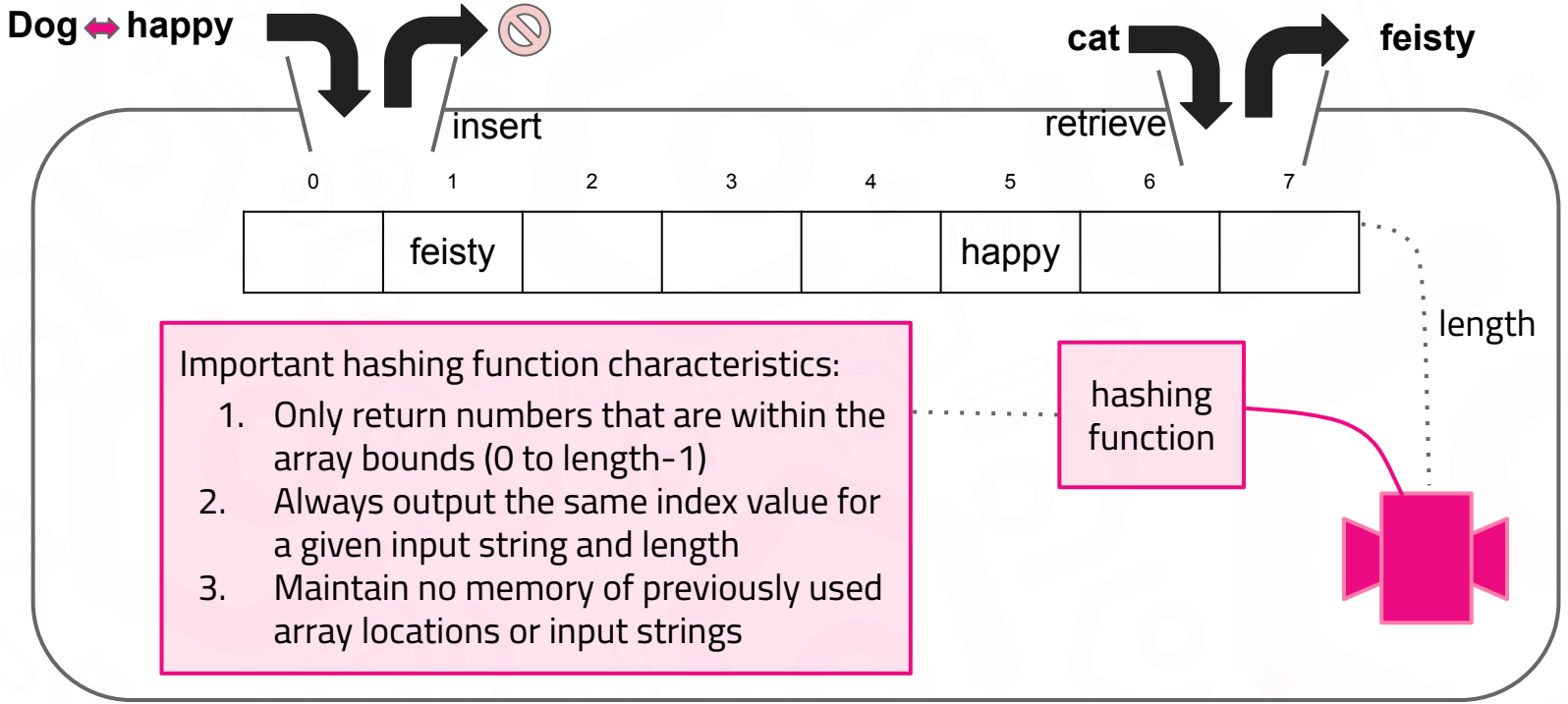
A: It can only return numbers that are within the array boundaries.



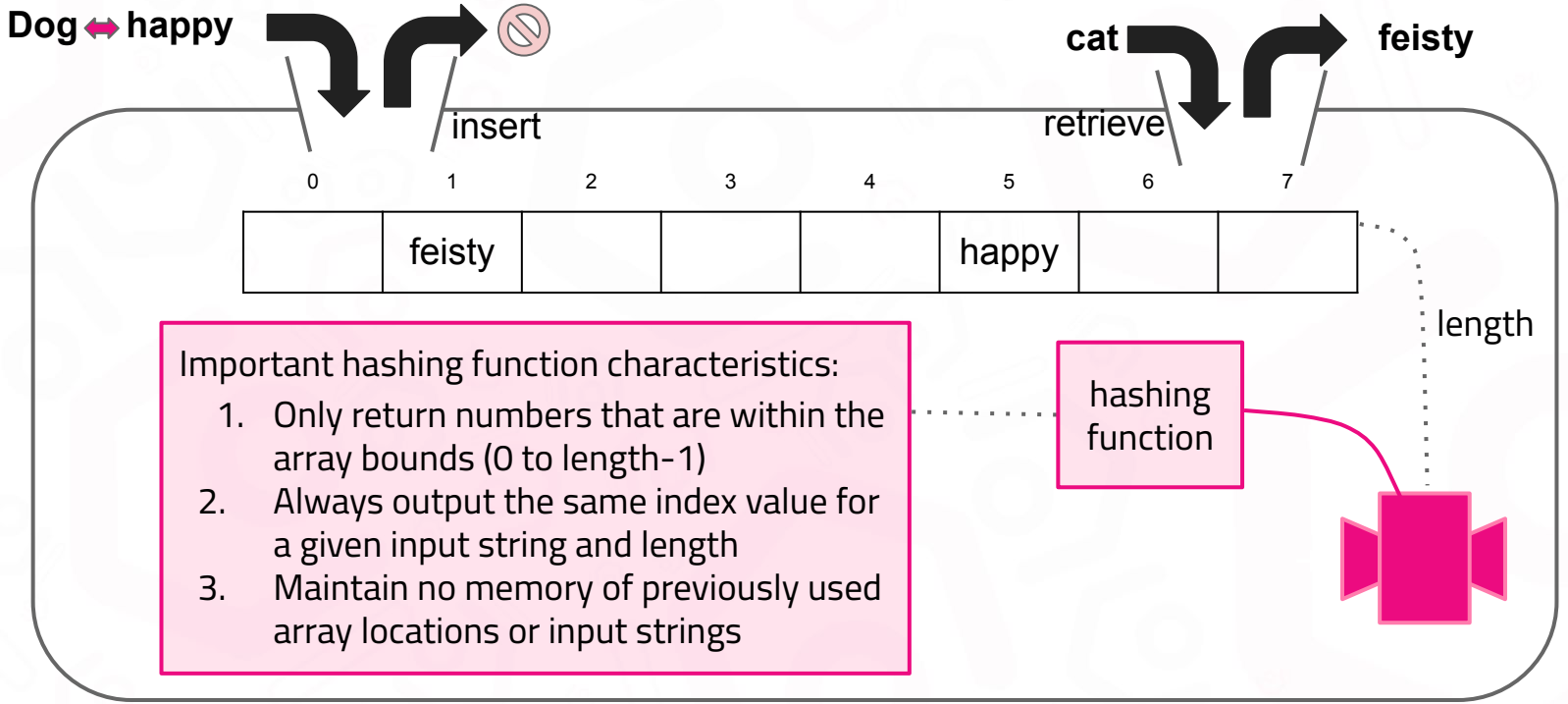
For this to be possible, the hashing function needs to know the length array. In fact, the array's length is one of the inputs to the hashing function.



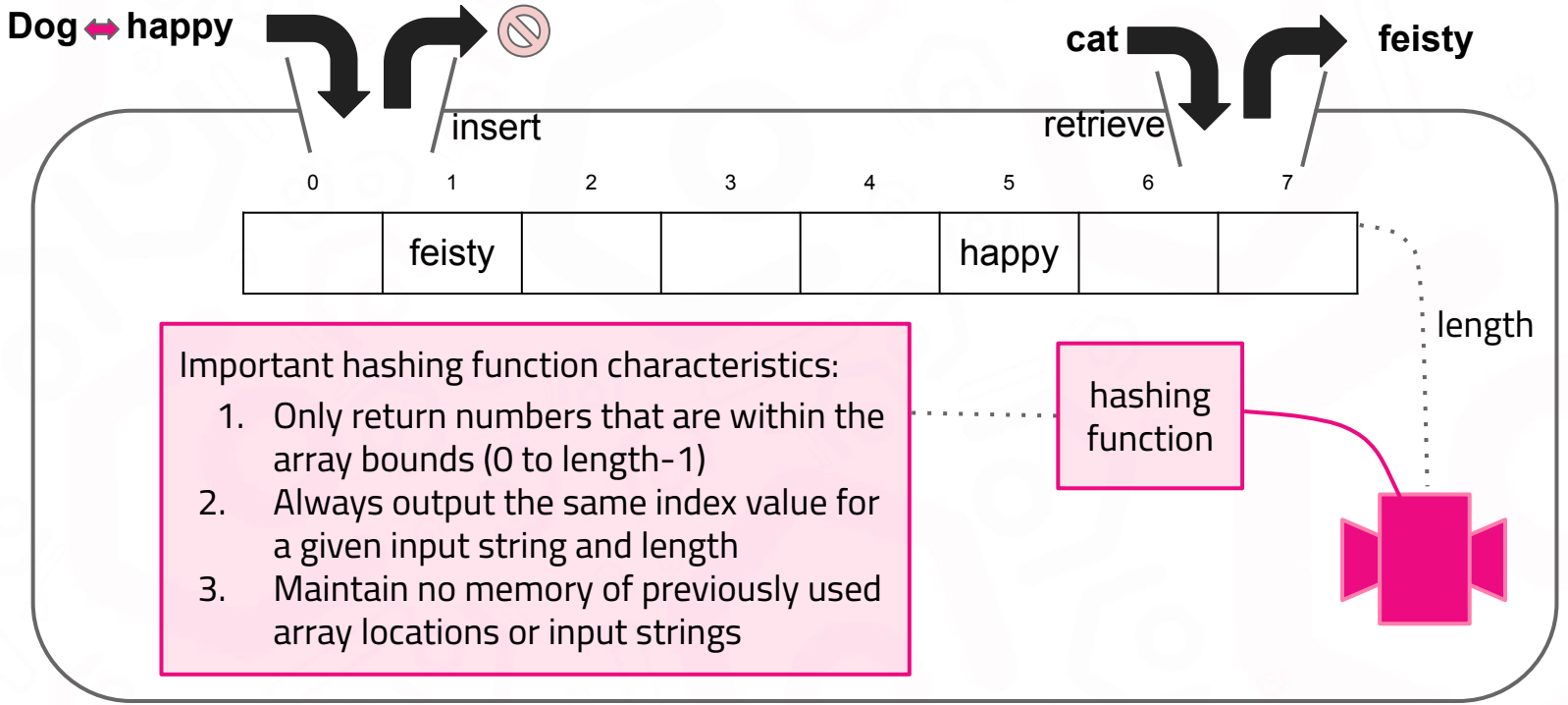
A: The hashing function must be deterministic. More specifically, it must always output the same index value for a given input. For example, for an array of size 8 and a key of 'cat', the output value must always be 1.



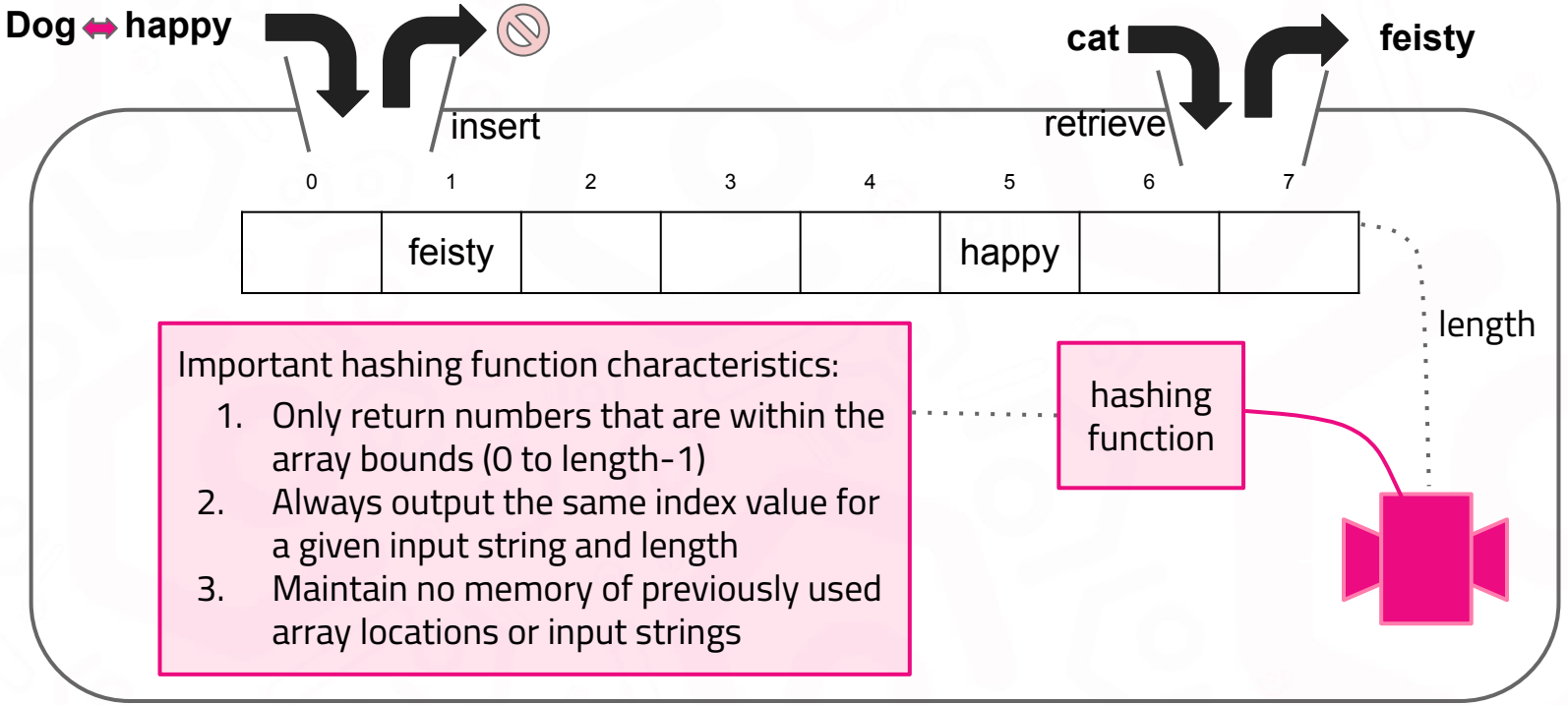
A: The hashing function cannot remember any previously used array locations or input strings. You might find this claim to be confusing and unexpected -- that's ok. Let's expound on those points of confusion.



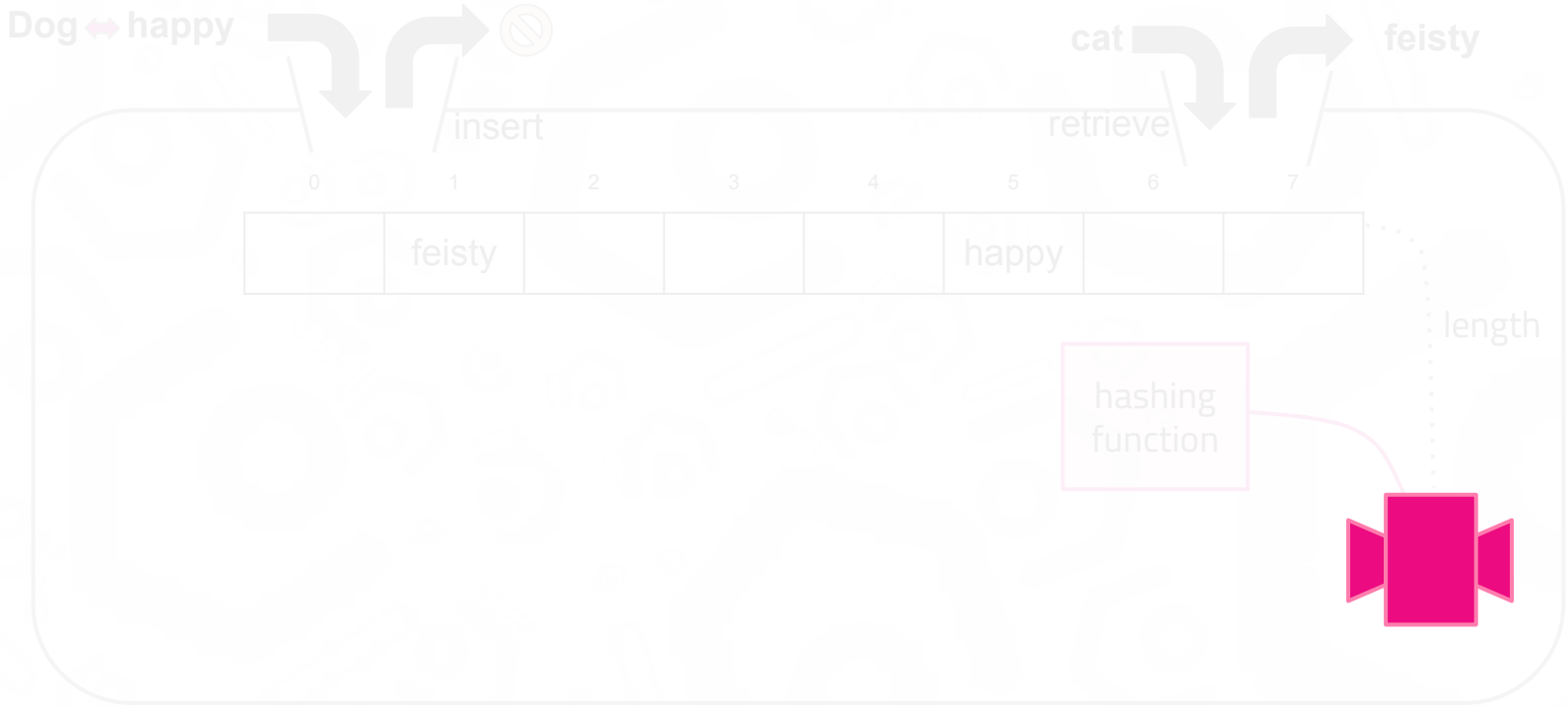
Q: If the hashing function cannot remember the previously used array locations or the supplied input strings, how can it output a unique index -- one that is not in use? Couldn't that cause an existing value to be overwritten?



A: We don't expect the hashing function to always produce unique values. In fact, we aim to design a hash table that expects the hashing function to *sometimes* produce non-unique values. We'll explore how works in just a moment.



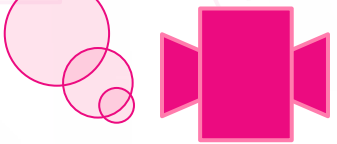
The main motivation item #3 is: **time complexity**. Any time you create a list of things to remember, you must iterate over the list to determine existence -- an O(n) operation. Our requirements are O(1) for insert and retrieve.



A brief detour... how exactly does a hashing function work?

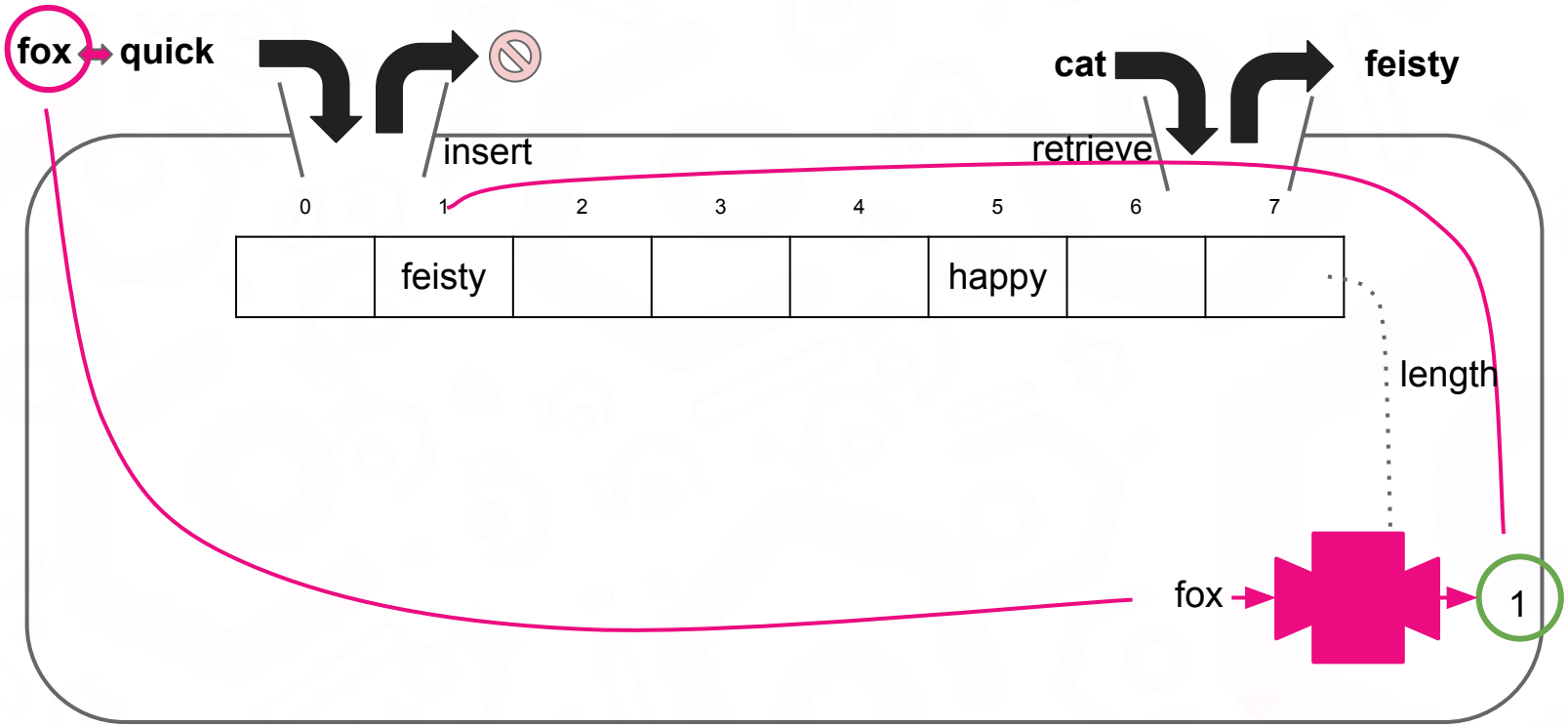
hash.js

```
var hash = function(string, max){
  var result = 0;
  for(var i = 0; i < string.length; i++){
    var char = string.charCodeAt(i);
    result = (result ^ char) + result;
  }
  return result % max;
}
```

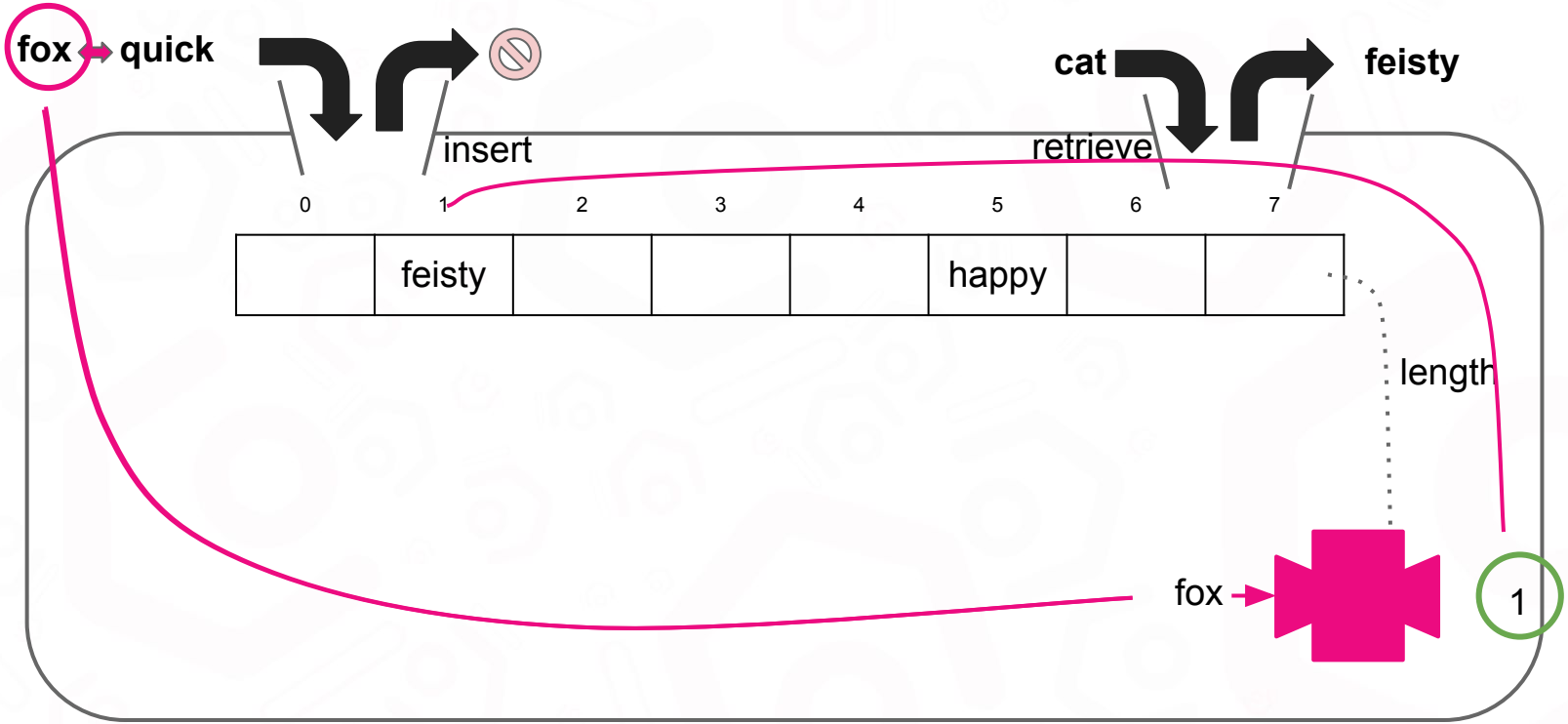


A typical approach is to perform a mathematical calculation on the ordinal value of each character in the string and return a remainder using the array's length (max) as the divisor. Some hashing functions are superior to others.

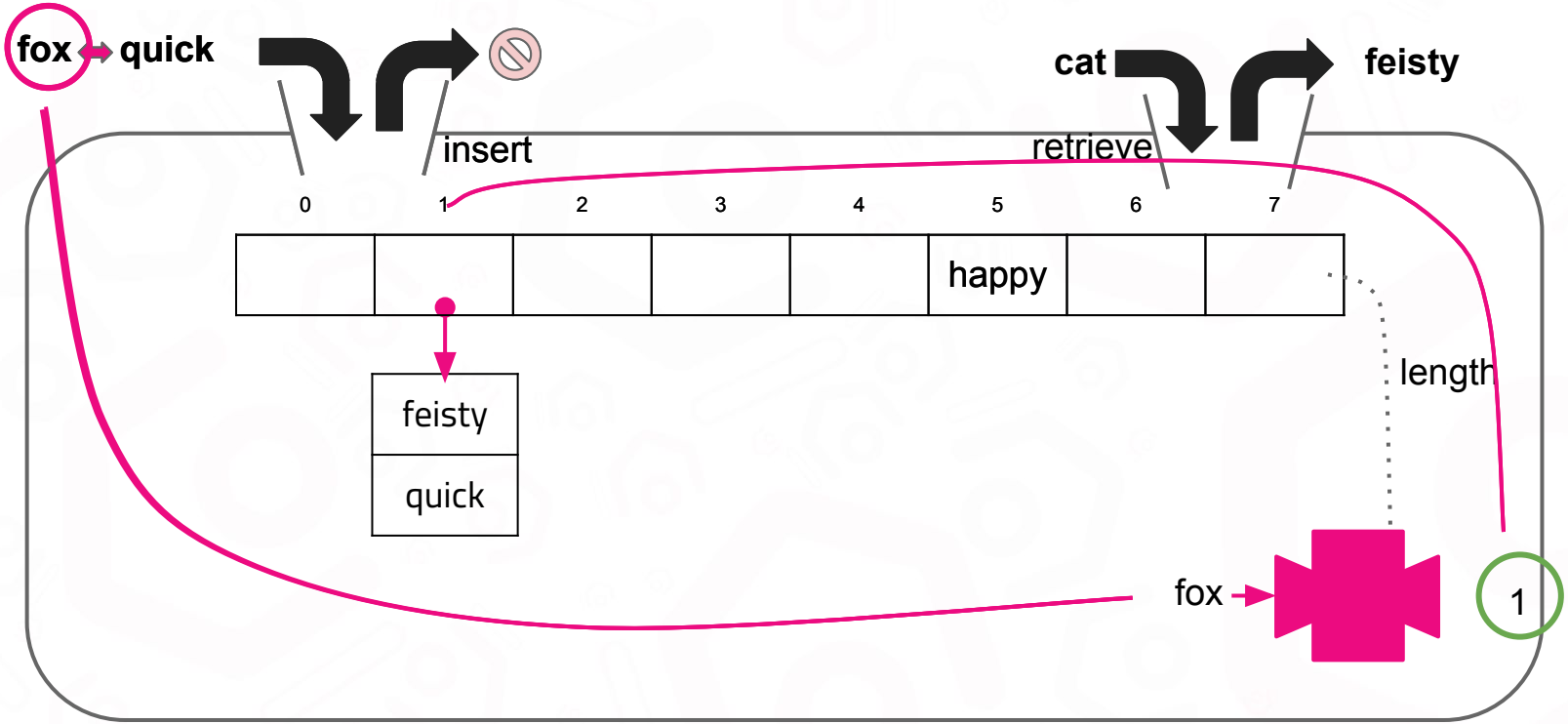
We're ready to explore this idea of hashing functions sometimes returning the same index. When a hashing function returns an index already in use, we call this a **collision**. Let's look at how collisions happen and how to handle them.



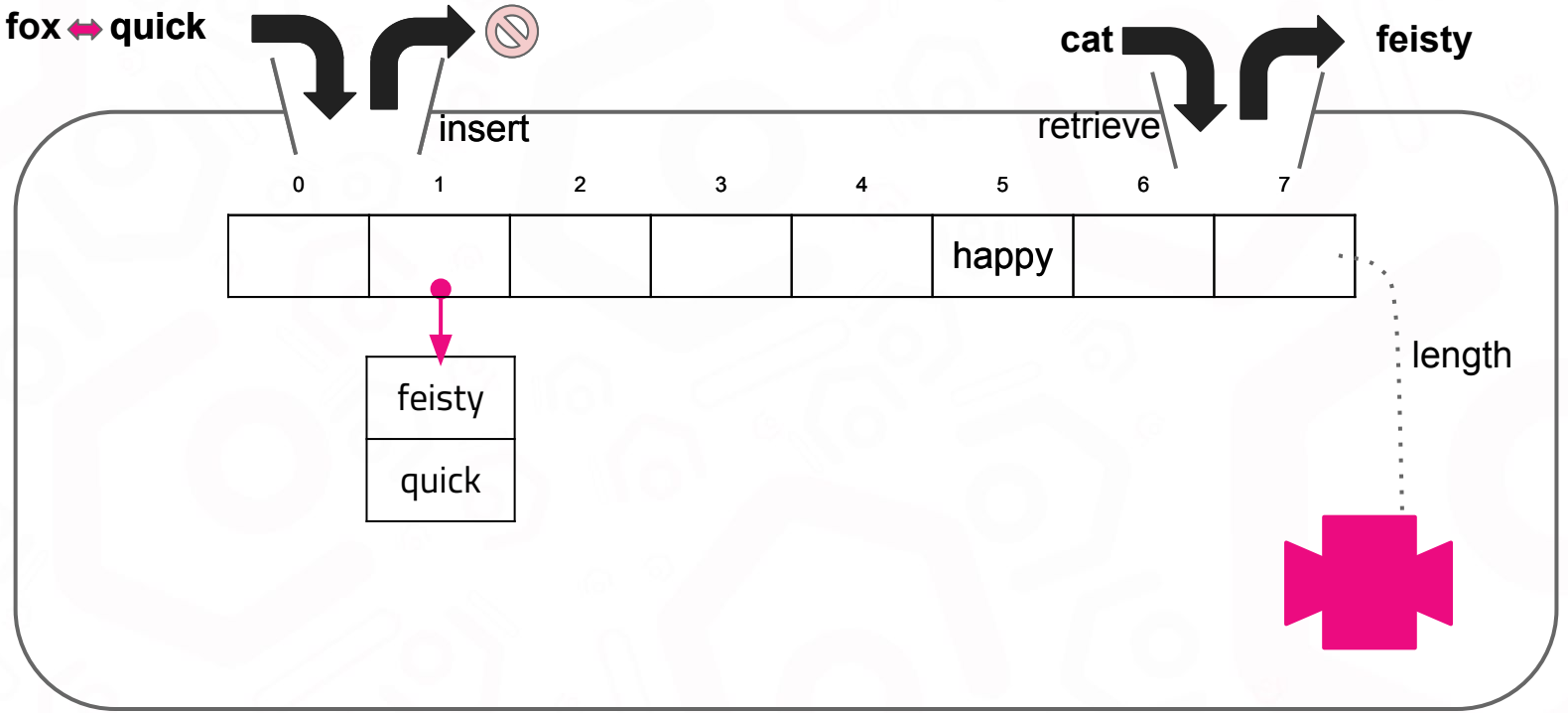
Since the hashing function is based on a mathematical formula, there exists some set of input strings that will always produce the same index value. For example, both ‘cat’ and ‘fox’ might both evaluate to index 1.



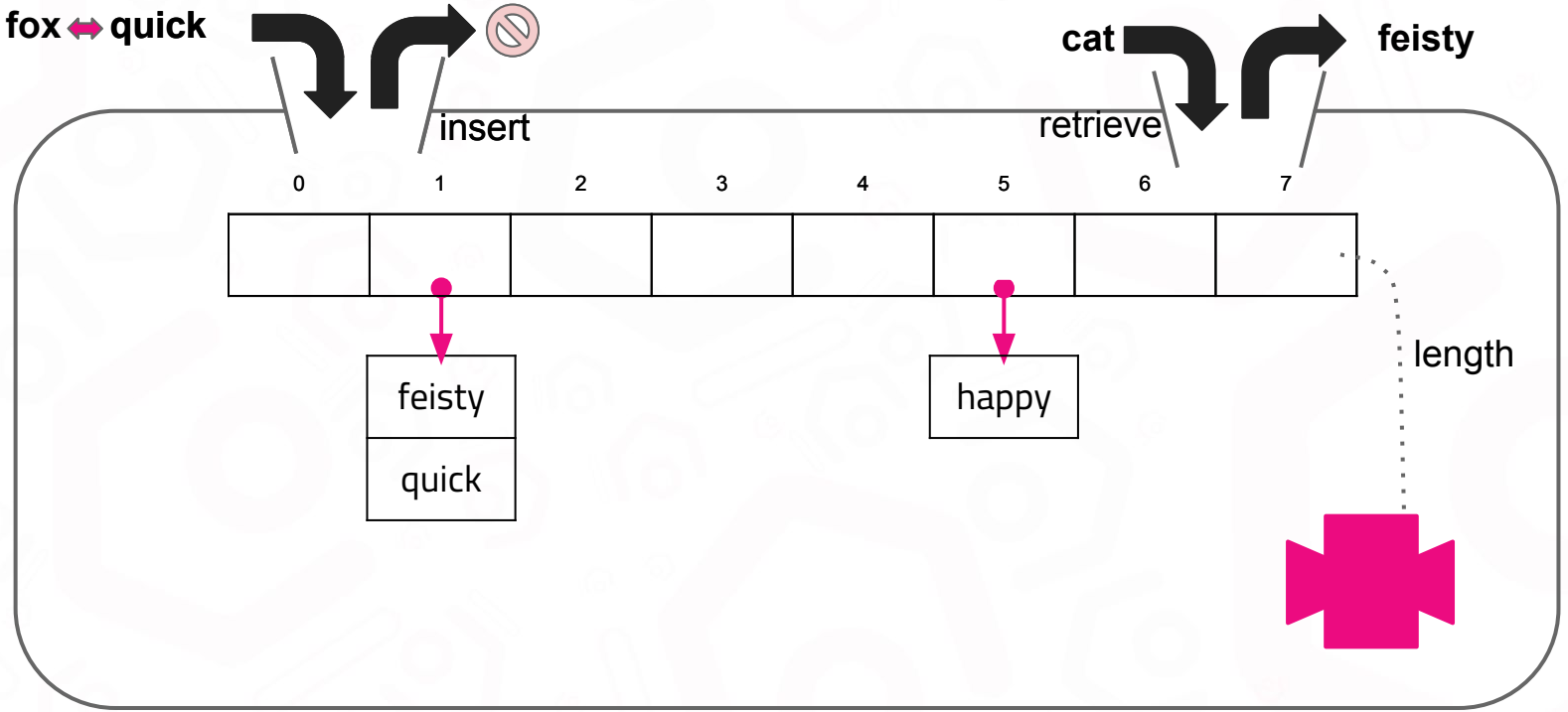
Q: Since index 1 is already occupied by a value, can you think of a way to store a second (and potentially more) values at index 1?



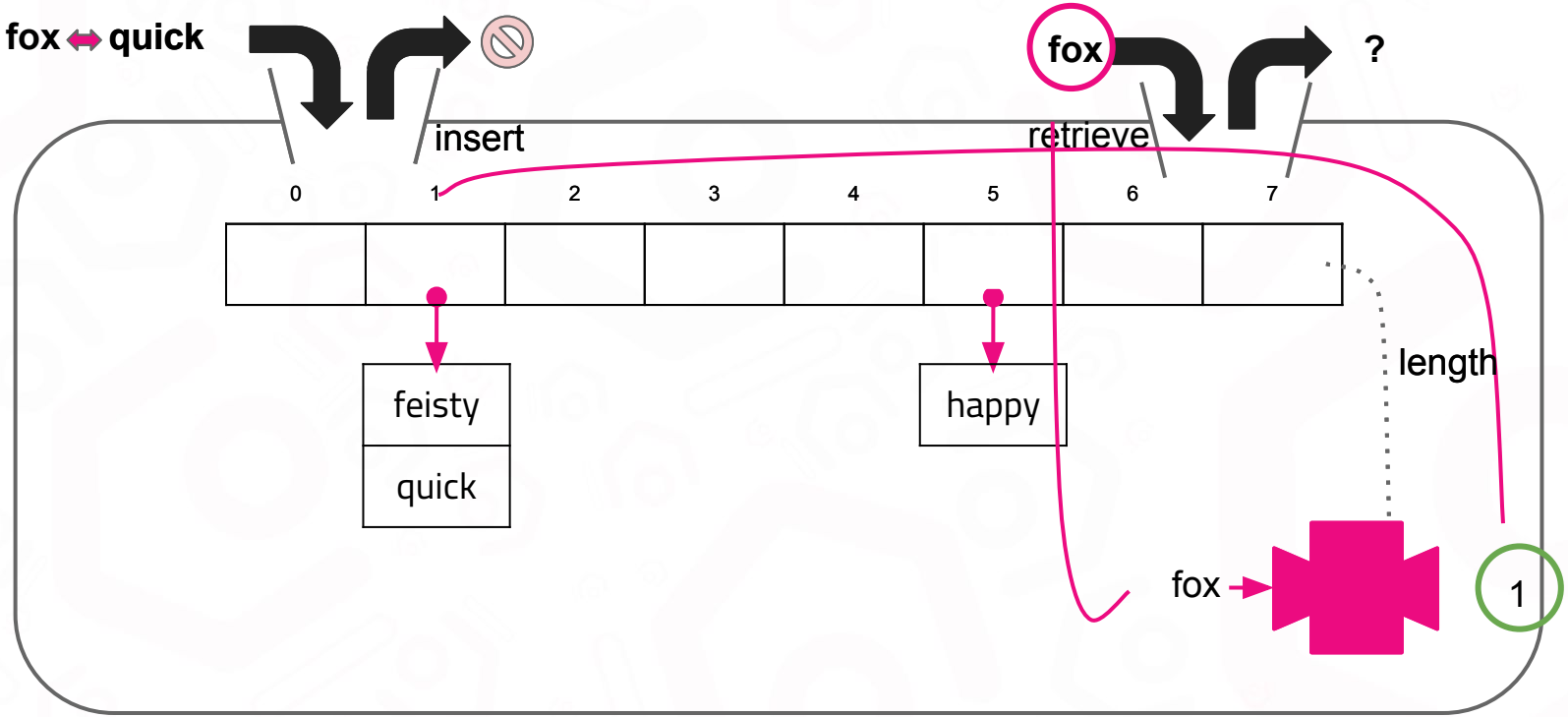
A: Instead of storing a single value at this index, we can store an array of values.



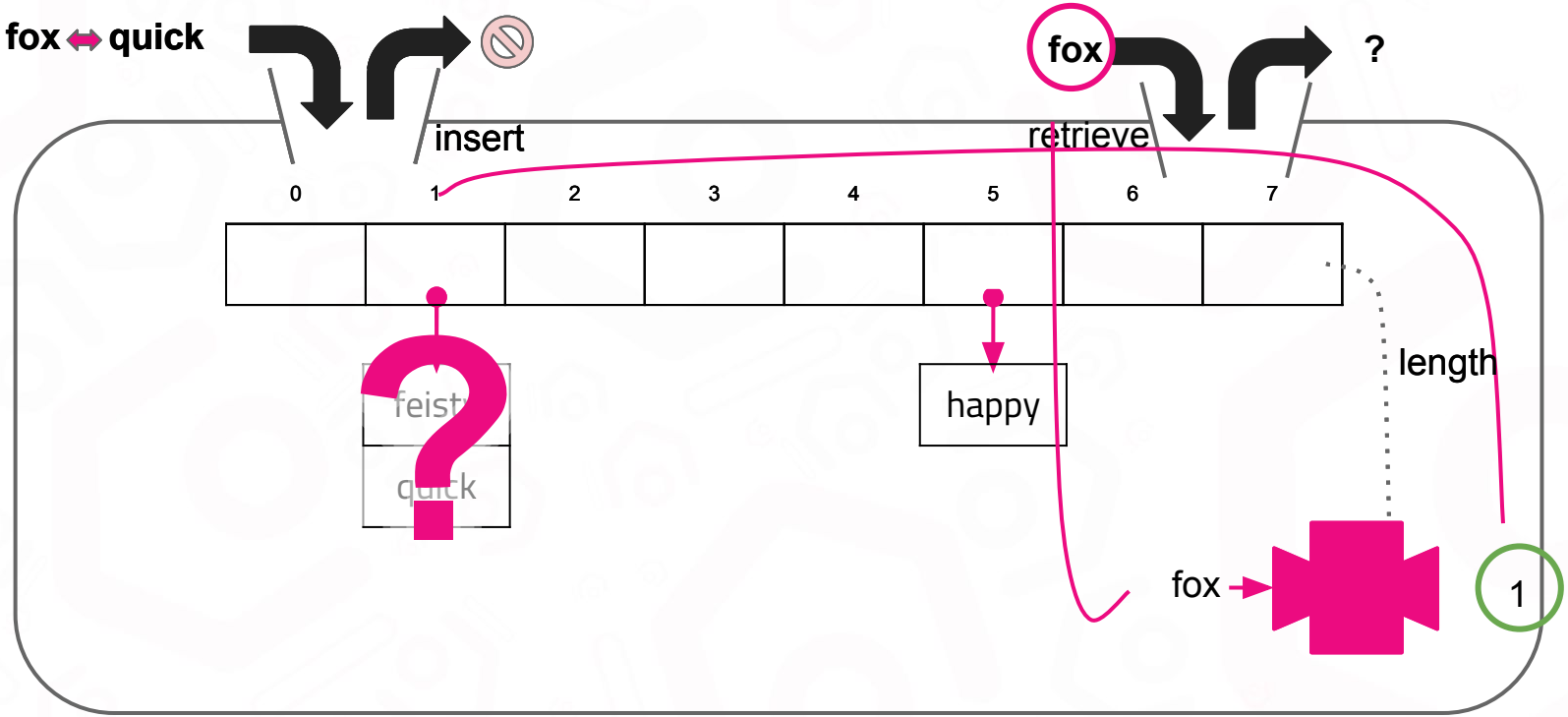
It's very important to maintain our data structure in a consistent way. Even though we have only one value at index 5, we'll ideally use the same strategy at all indexes.



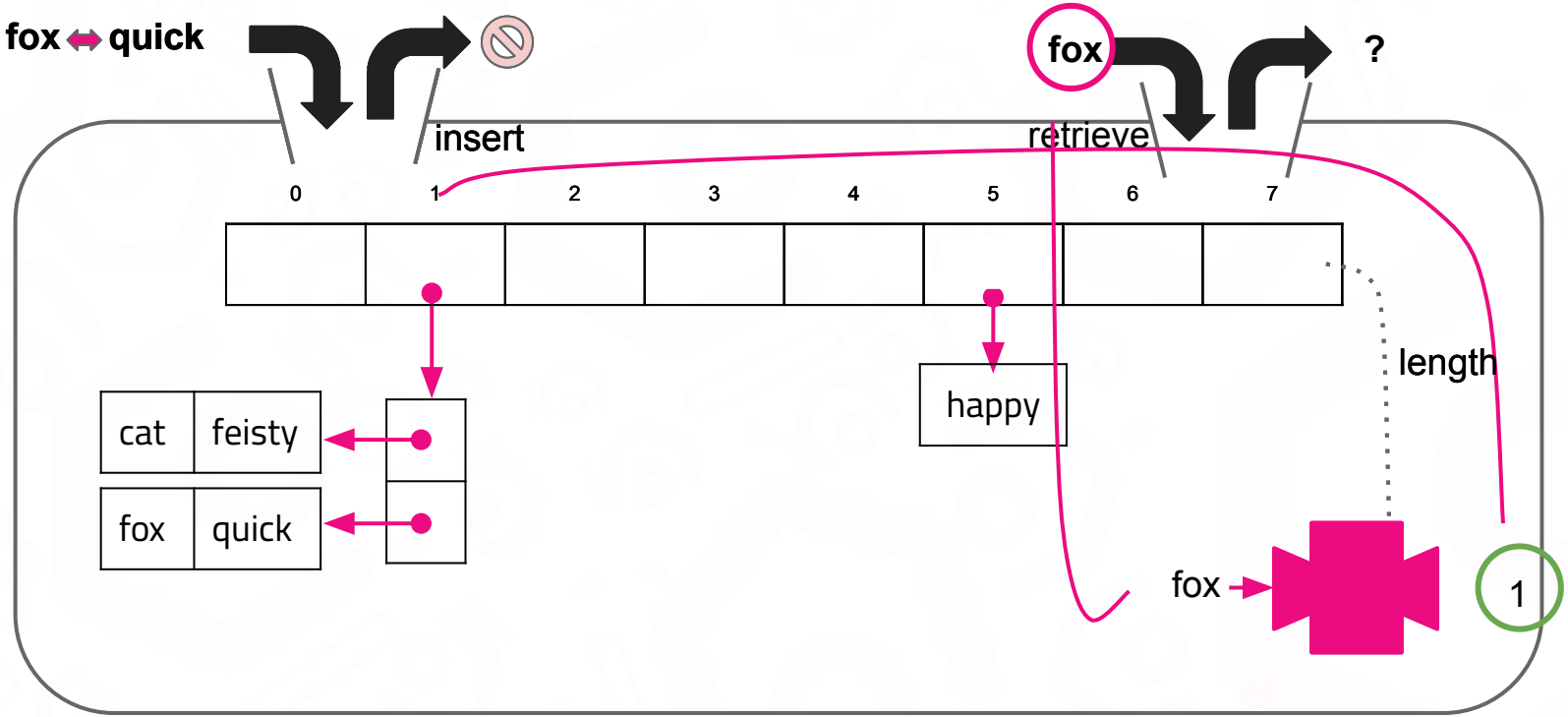
Let's refactor our topology so that all indexes are treated in the same way. Doing so allows our code to operate in the same way at any index, regardless of how many items are stored there. Simpler code is easier to understand.



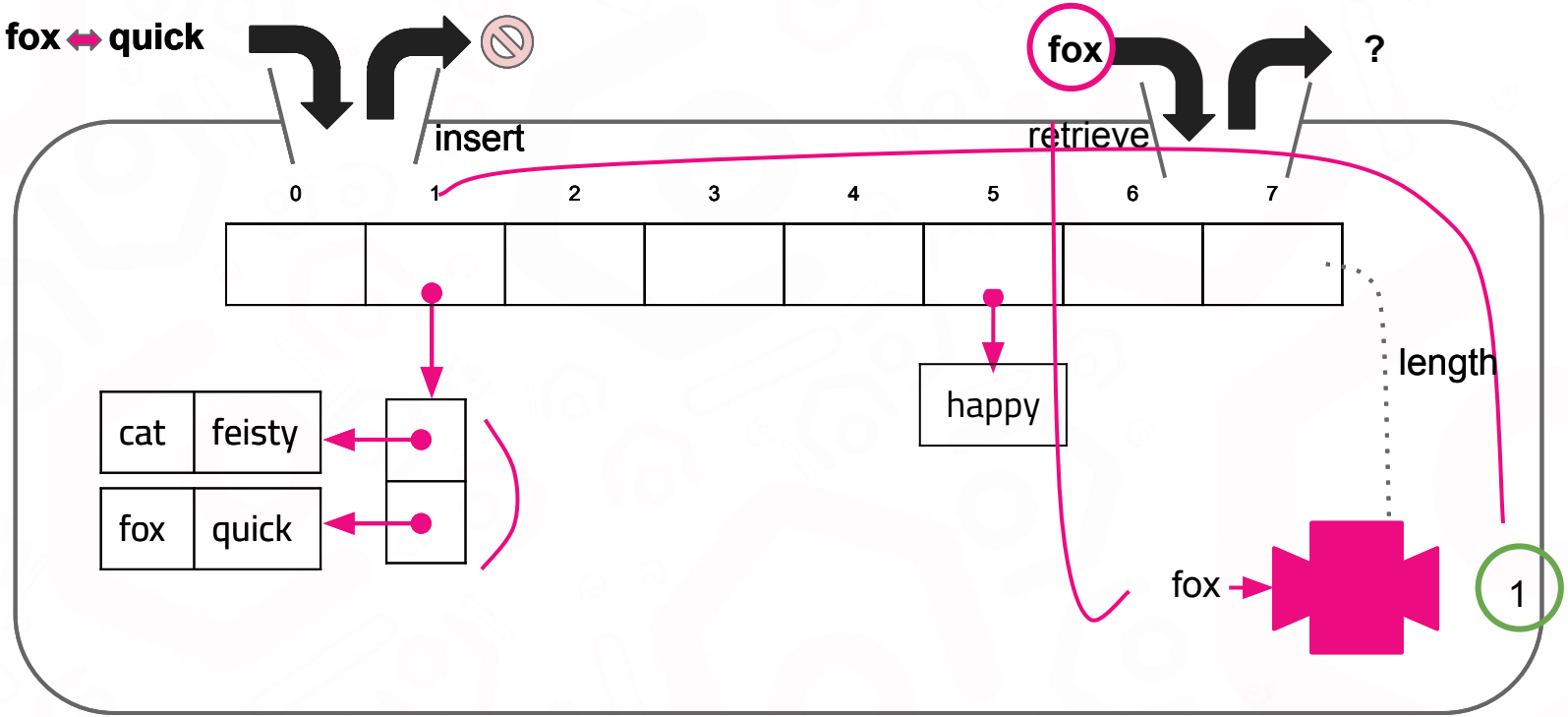
Now let's attempt to retrieve 'fox' from our hash table. Recall that we just inserted the value for 'fox' at index 1.



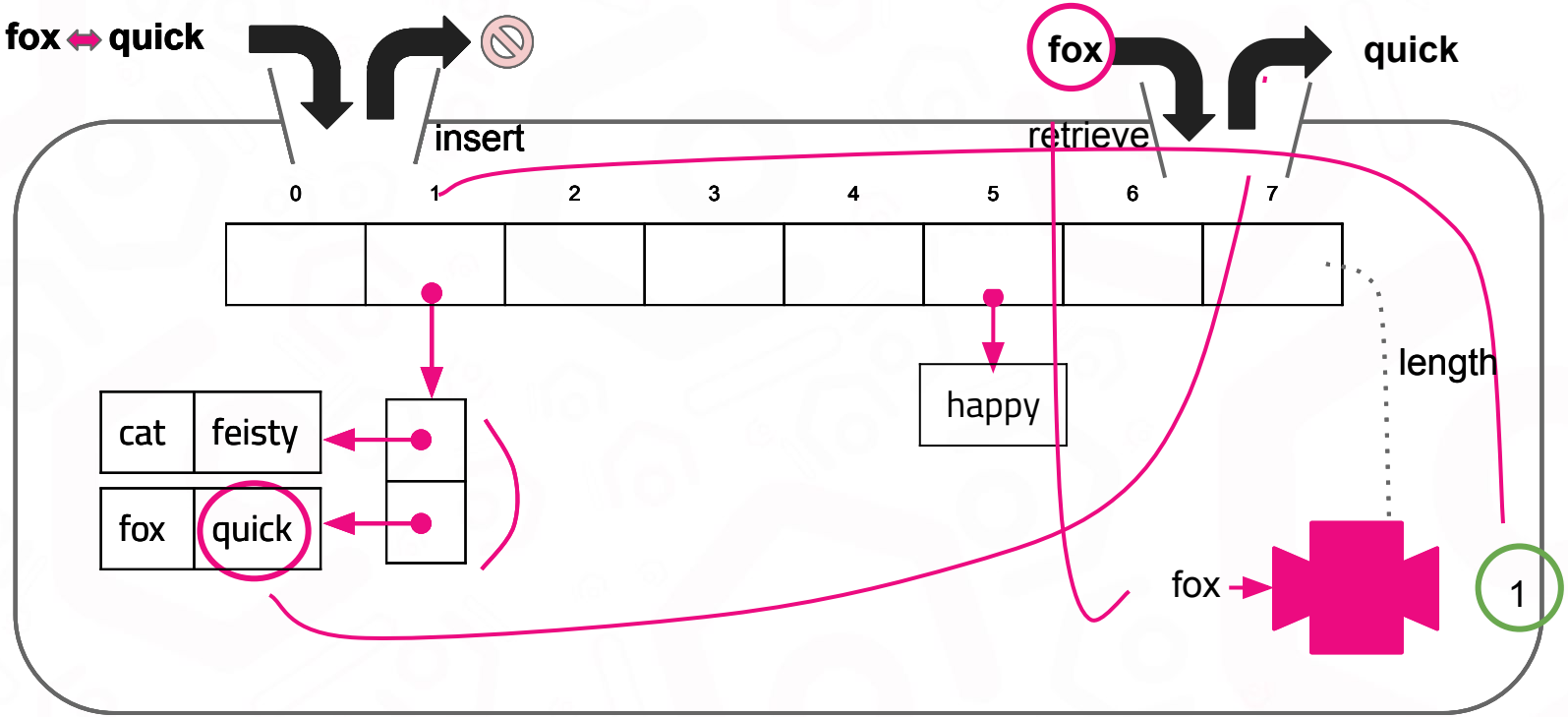
There are two values stored at index 1. Which value should the hash table return? It looks like we have a problem -- we don't know which value was originally associated with 'fox'. **Q:** Can you think of a way to solve this problem?



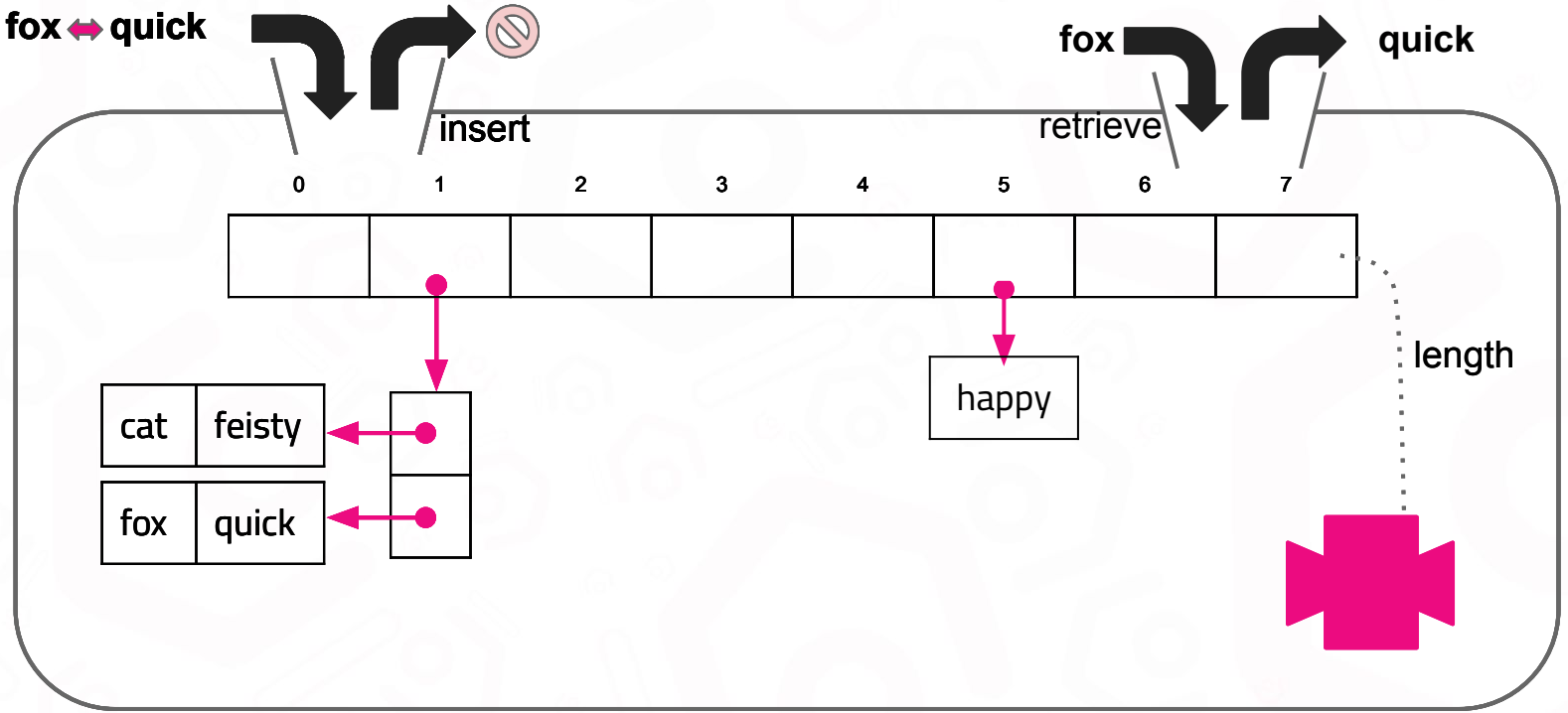
A: Instead of only storing the value, we need to store both the key and value. We do that by introducing another array that stores each key/value pair.



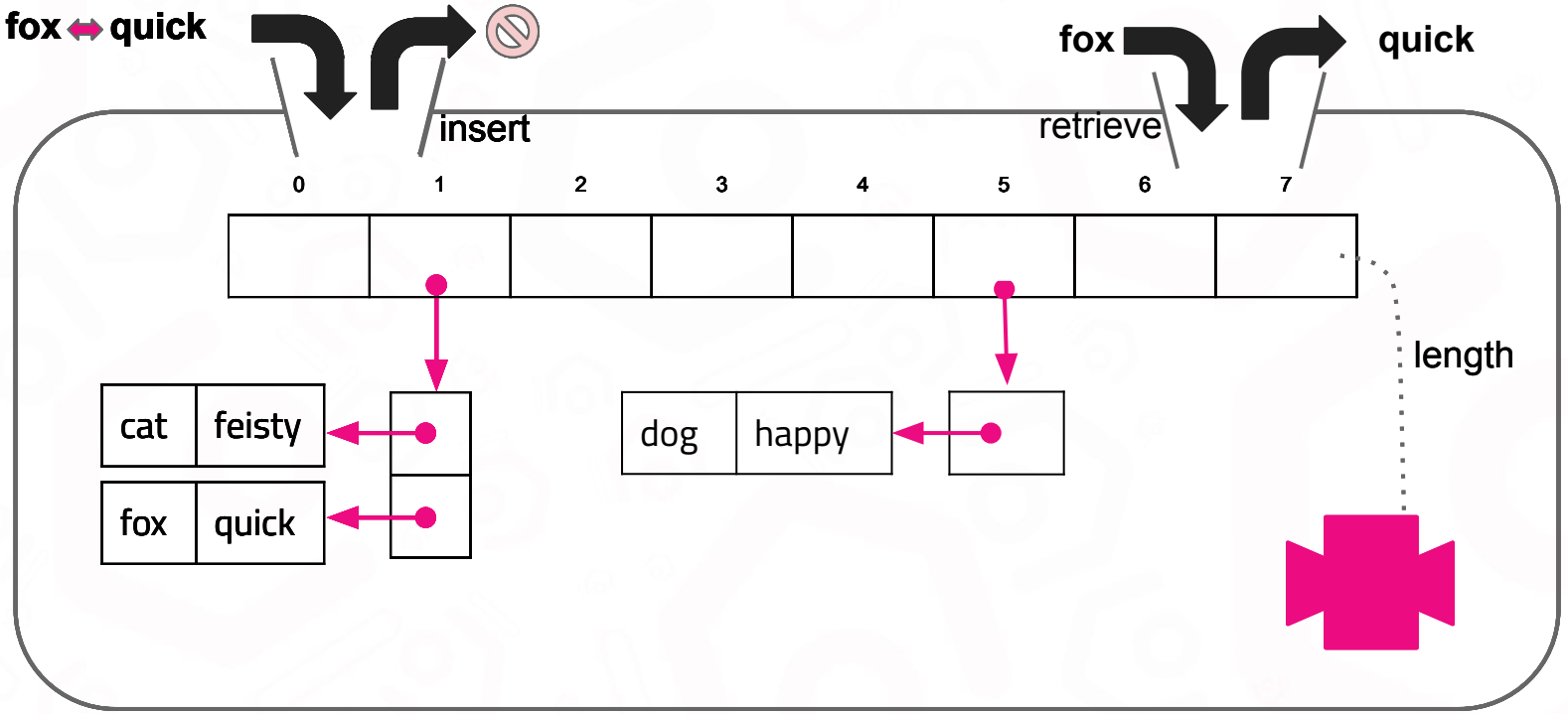
Now we can iterate over the items at index 1, looking for ‘fox’.



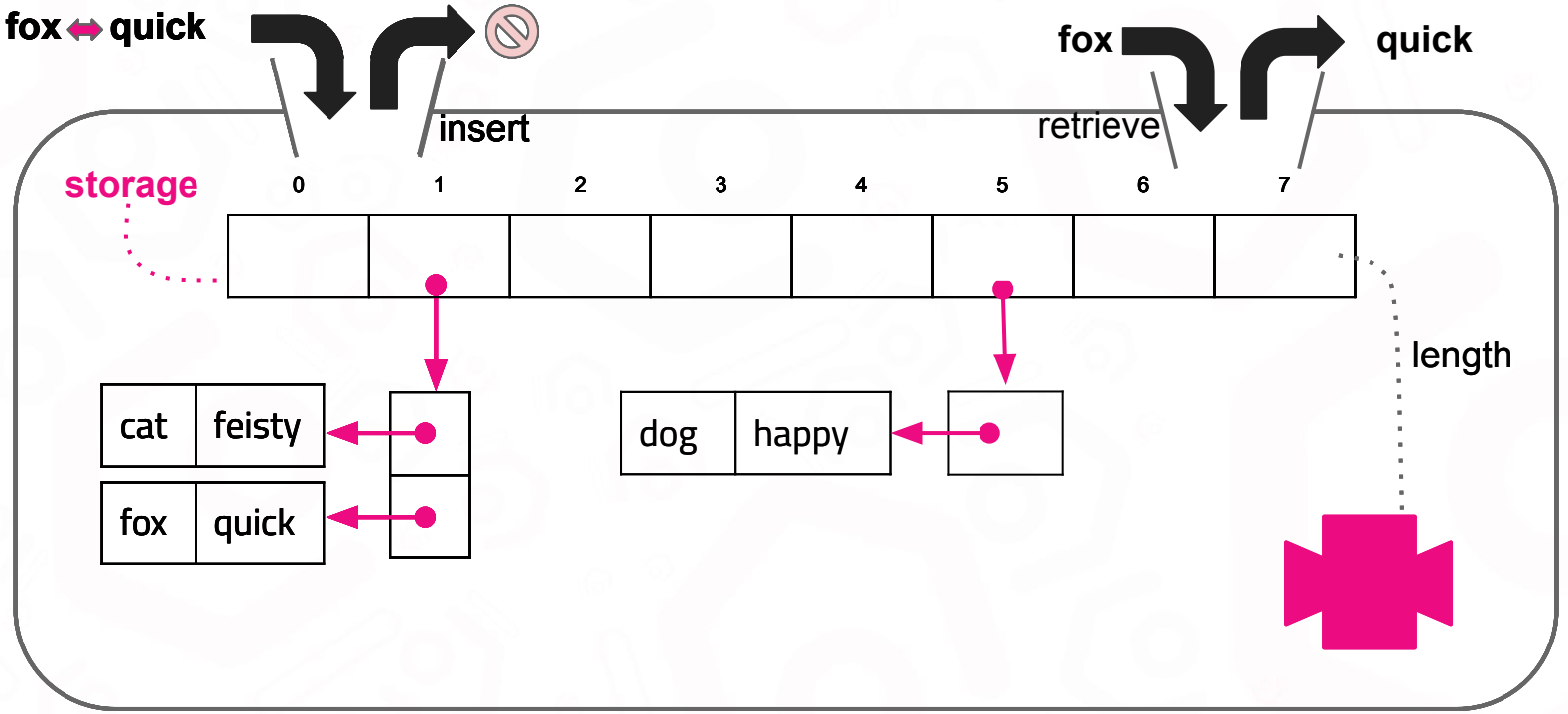
When we find the key we are looking for -- in this case 'fox' -- we can return the correct associated value.



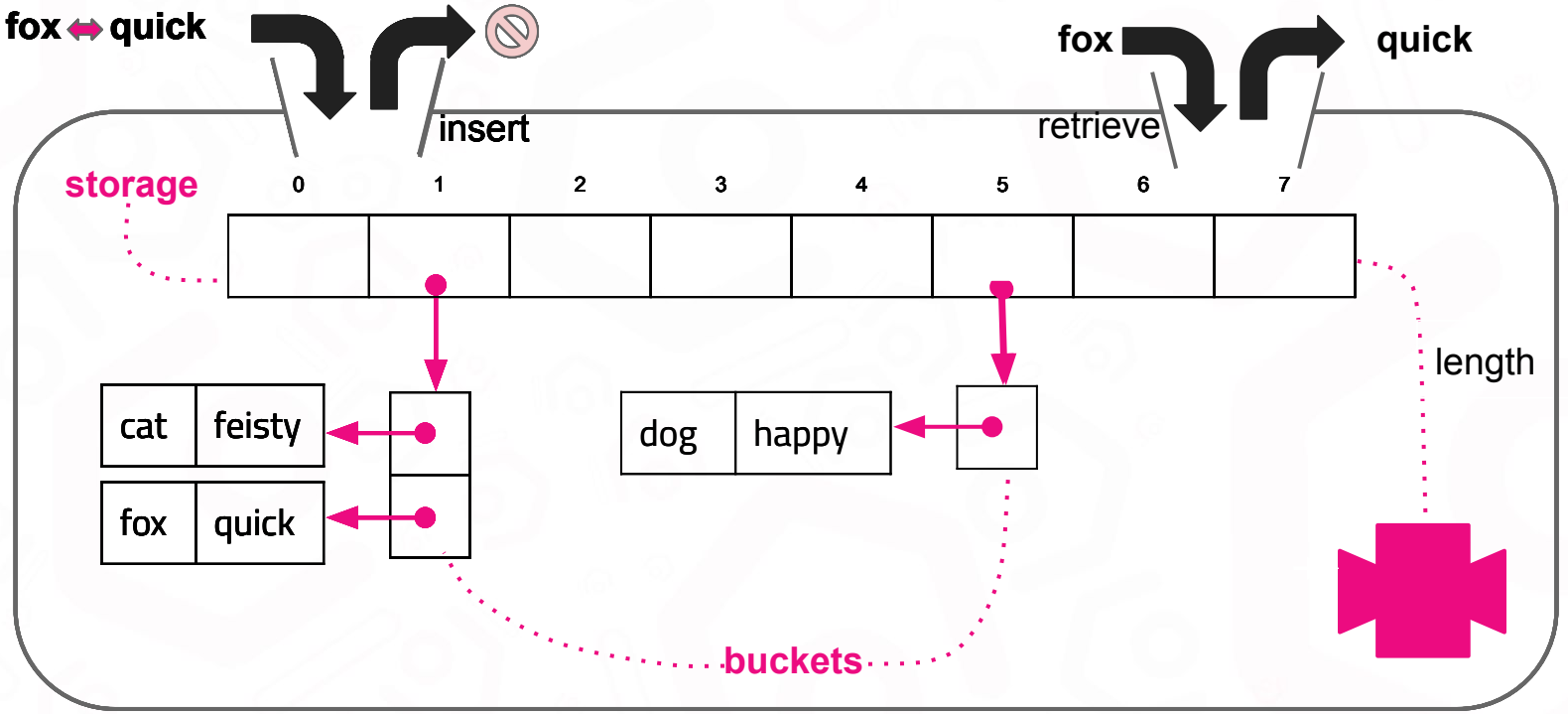
For completeness, let's update the item at index 5 to also store the key and value. As we just discovered every key/value pair must always be stored -- we'll never only store a value.



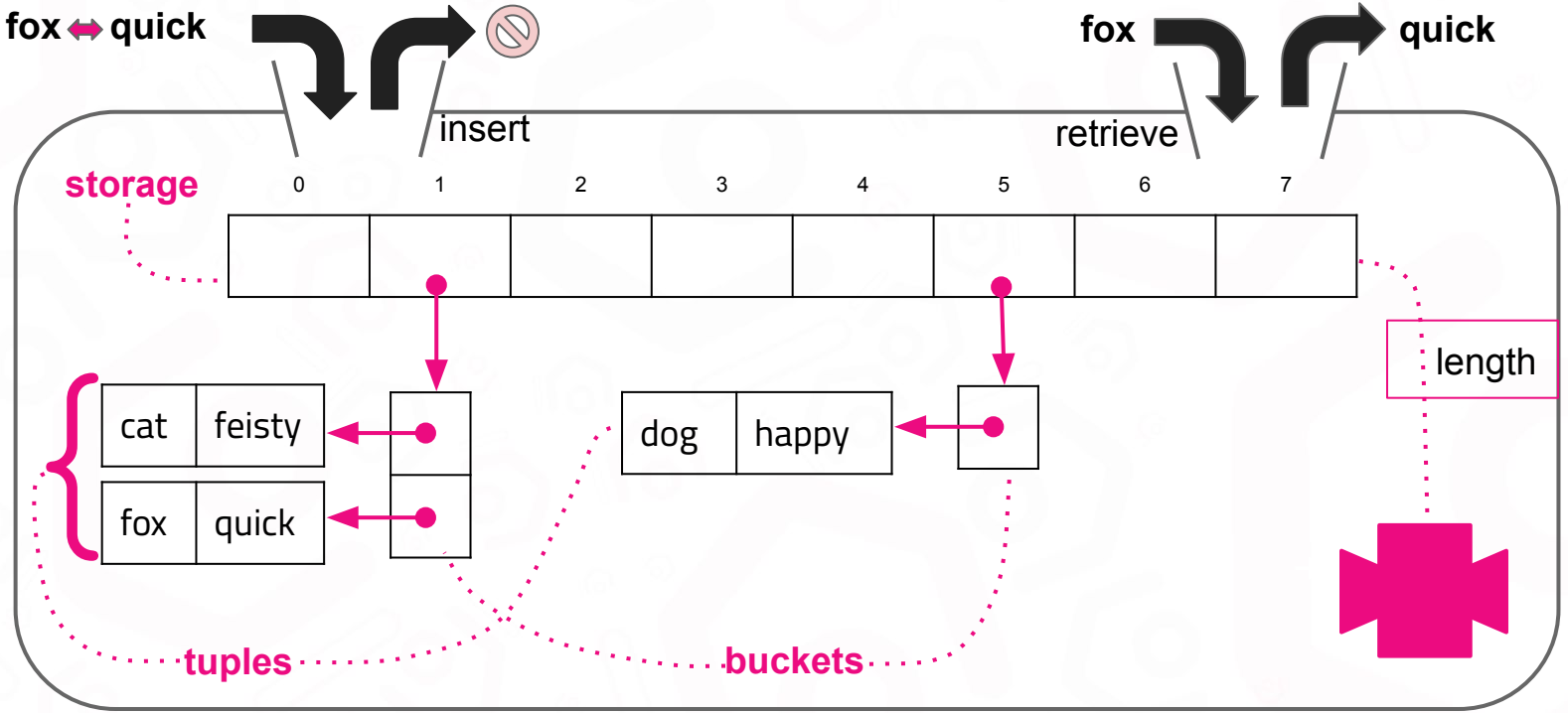
This is the complete implementation for a hash table. As a final step, let's add some labels to the arrays so we know what to call them.



The top-level array is called the storage array and is what our hashing function generates indexes for.



Any index that contains key/value pairs, will contain a bucket array.



Key/value pairs are stored in tuple arrays -- where index 0 is the key and index 1 is the value.

How do you maintain O(1) Time Complexity as Input Size Grows?

Hash Table Resizing

Hash tables operate **most effectively** when the ratio of tuple count to storage array length is **between 25% and 75%.** When the ratio is:

- > 75%, double the size of the storage array
- < 25%, half the size of the storage array

Resizing necessitates rehashing every key as it may end up in a different bucket. Remember, **the hashing function depends on the storage array size**!

Dirty Little Secrets

While most of the time insert & remove operations are O(1), the **worst case is O(n)**. This can occur for two reasons:

- When a hash table is **growing**, it resizes itself and every element must be rehashed
- It is possible for all keys to hash to the **same bucket**, which becomes a O(n) search for an item

Dirty Little Secrets

The quality of the hashing algorithm is a major factor in how well your tuples will be distributed within your buckets. An algorithm with more entropy will produce superior results.

CodingLab
Powered by RBK

That’s it