CodingLab
Powered by RBK

Junior Phase

Hash tables

An introduction to hash tables

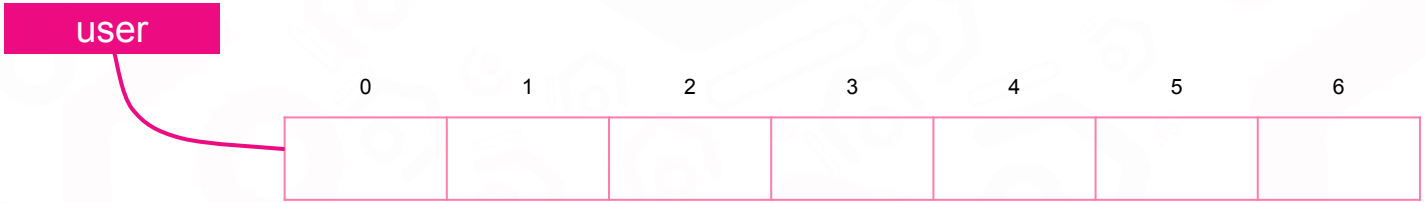
Duration: 30 minutes :
Q&A: 5 minutes by the end of the lecture

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.

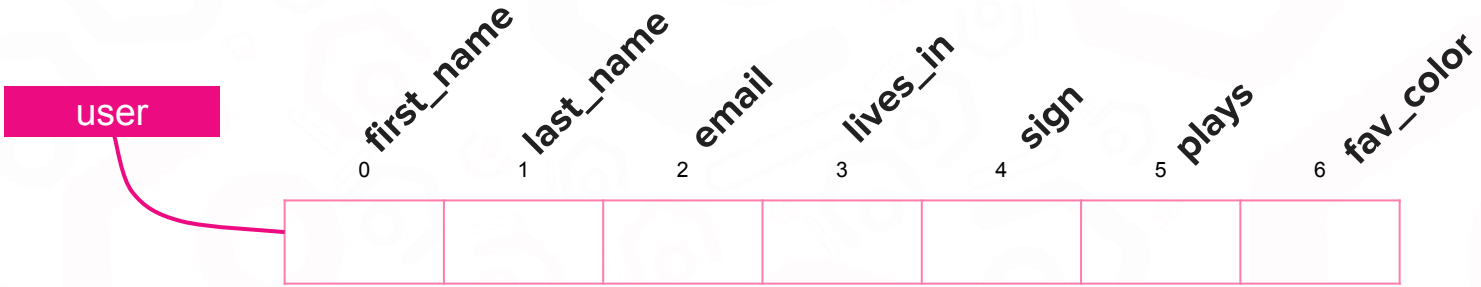
A Hash Table has other names too, such as an Associative Array, Dictionary, Map, or Hash.

Data Association - The Problem



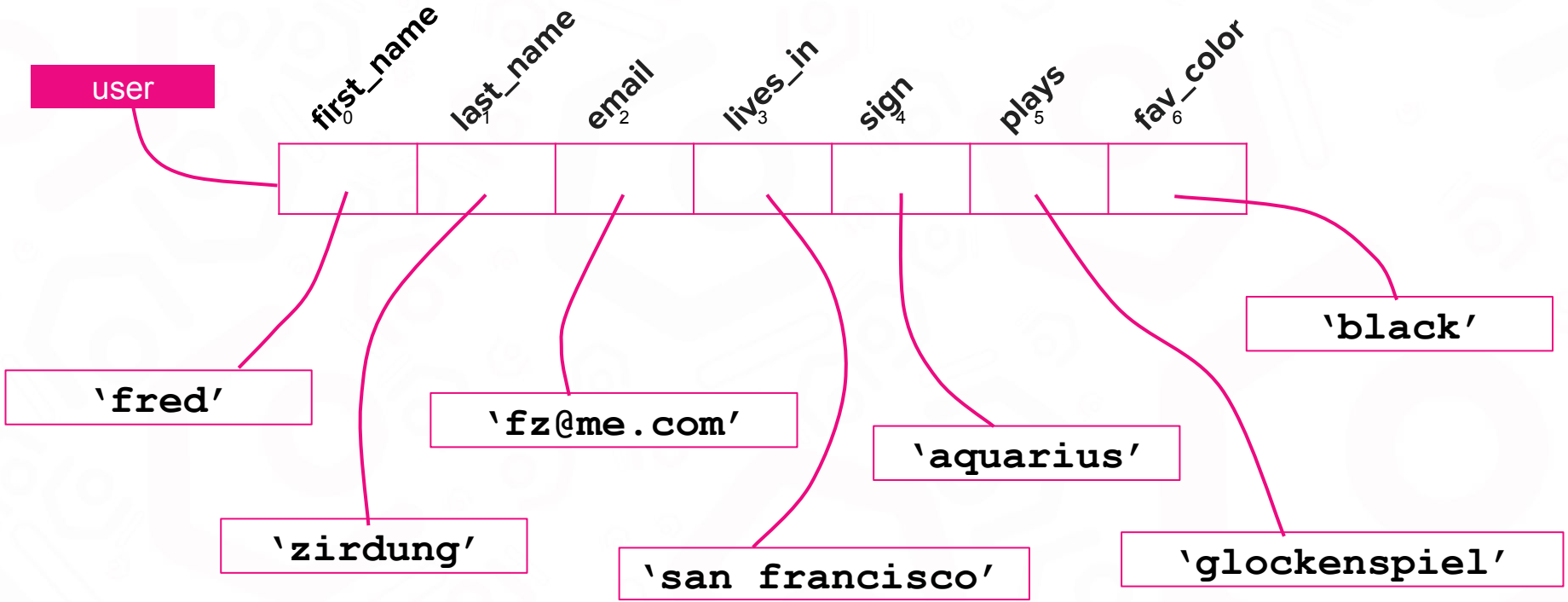
In order to understand the importance of associative data, let's start by creating a data structure to store user information. For this task, we'll use the most basic of data structures - an array.

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.

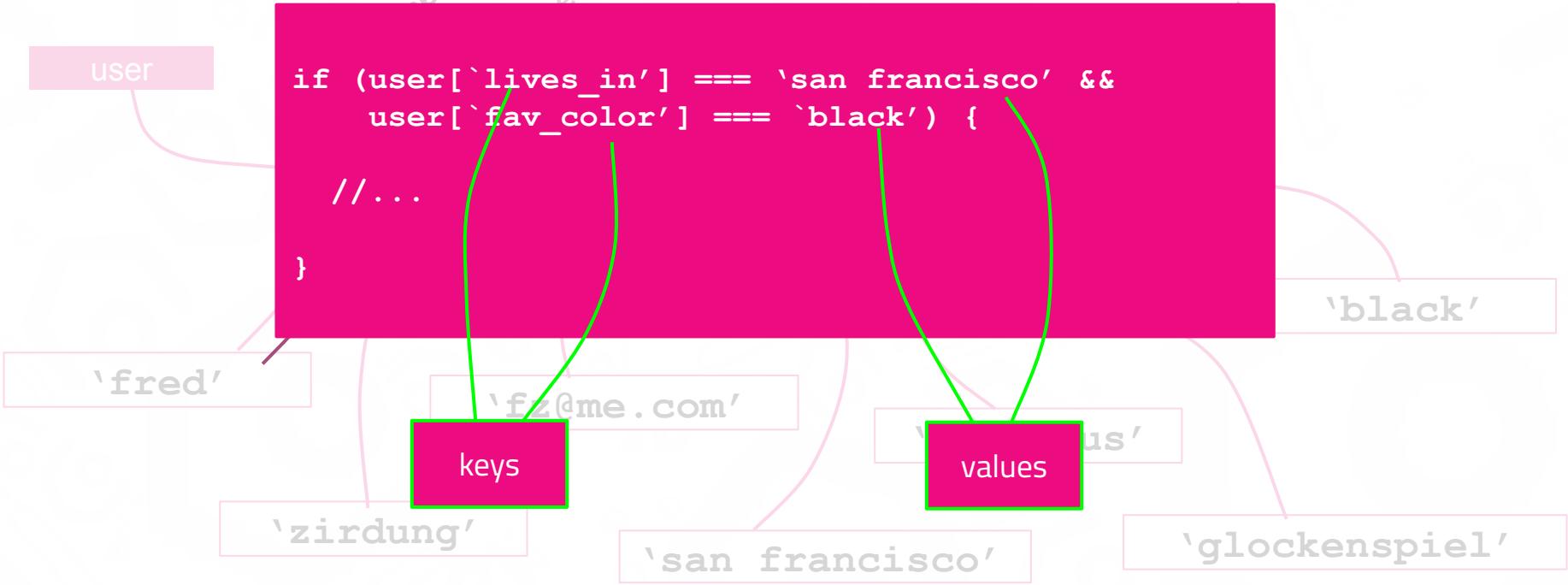
Data Association - The Problem

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

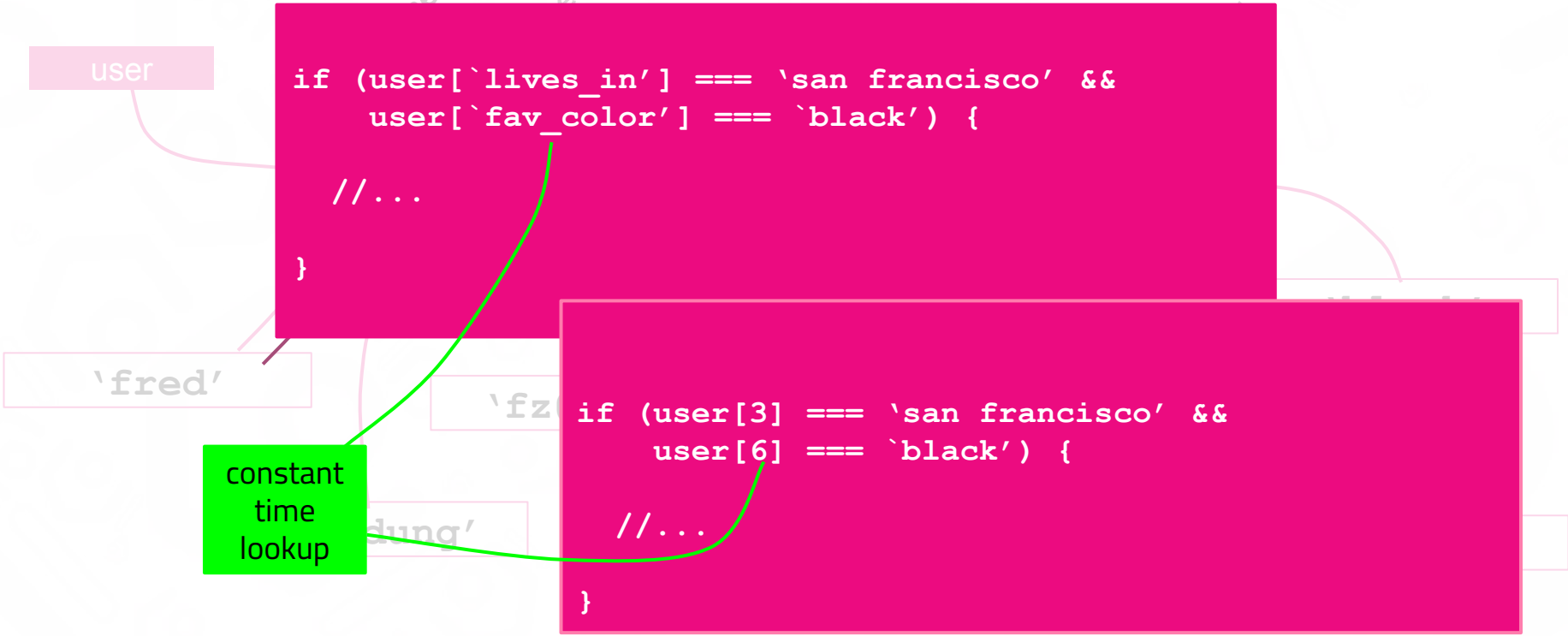
```
if (user[3] === 'san francisco' &&
    user[6] === `black') {

  //...

}
```

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.

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.