19.2 Hash Code

The core mechanism of hashing!

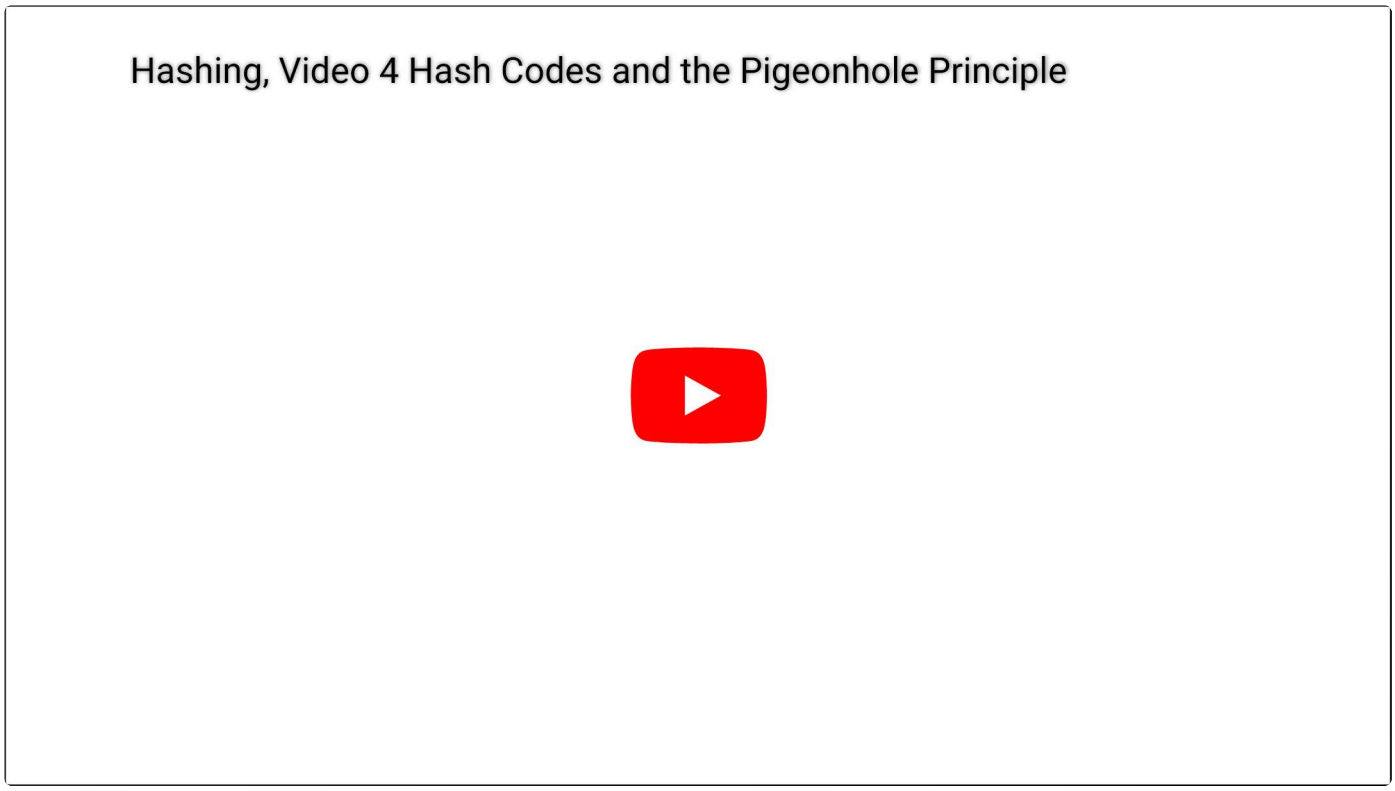
We face the problem that not every object in Java can easily be converted to a number. However, the key idea behind hashing is the transformation of any object into a numeric representation. The key is to have a hashing function transform our keys into different values, and convert that number into an index to then access the array.

We achieve this through our own implementation of a `hashCode()` function, with a return value of an `int` type. This `int` type is our *hash value*.

The built-in String class in Java, for example, might have the following code block:

```
public class String {
    public int hashCode() {
        //implementation here
    }
}
```

Based on this example, we can call `key.hashCode()` to generate an integer hash code for a `String` instance called `key`.



Professor Hug's Lecture on Hash Codes

Memory Inefficiency in Hash Codes

An issue mentioned earlier is memory inefficiency: for a small range of hash values, we can get away with an array that individuates each hash value. That is, every index in the array would represent a unique hash value. This works well if our indices are small and close to zero. But remember that Java's 32-bit integer type can support numbers anywhere between -2,147,483,648 and 2,147,483,647. Now, most of the time, our data won't use anywhere near that many values. But even if we only wanted to support special characters, our array would still need to be 1,112,064 elements long!

Instead, we'll slightly modify our indexing strategy. Let's say we only want to support an array of length 10 so as to avoid allocating excessive amounts of memory. How can we turn a number that is potentially millions or billions large into a value between 0 and 9, inclusive?

Wrapping!

The **modulus operator (%)** allows us to achieve this.

Review: The result of the modulo operator is like a remainder in fractional division. For example, `65 % 10` returns `5` because after dividing 65 by 10, we are left with a remainder of 5. Thus as other examples, `3 % 10 = 3`, `20 % 10 = 0`, and `19 % 10 = 9`. For an intuitive understanding of this, think about how we used the modulo operator in Project 1: Deques to ensure you avoided `IndexOutOfBounds` Exceptions and could accurately index into your deque via the concept of "wrapping around" the array.

Returning back to our original problem, we want to be able to convert any number to a value between 0 and 9, inclusive. Given our discussion on the modulo operator, we can see that any number mod 10 will return an integer value between 0 and 9. This is what we need to index into an array of size 10!

More generally, we can locate the correct index for any key with the following:

```
Math.floorMod(key.hashCode(), array.length)
```

where `array` is the underlying array representing our hash table.

Quick Note: In Java, the `Math.floorMod` function will perform the modulus operation while correctly accounting for negative integers, whereas `%` does not.

Previous
19.1.3 A third attempt: DataIndexedStringSet

Next
19.3 "Valid" & "Good" Hashcodes

Last updated 1 year ago