Pre-Class Work 17: Hashing

Hashing [Background Reading]

□ Motivation

By now, you have interacted with data structures such as `HashMap`s and `HashSet`s, which are examples of **hash tables**. You have been told that these data structures can efficiently store and retrieve data but why is that the case? Today, we will be exploring a technique called **hashing** which is what **hash tables** use to implement fast data retrieval and efficient data storage.

What is Hashing? #□

The term **hashing** refers to a mathematical function that transforms data (such as a `String` or any object) into a number called a **hash value**. The **hash value** is used to put the data into the corresponding index in the **hash table**.

A **hash table** is an array that stores your elements through **hashing**.

One of the key properties of a **hash function** is that it should be one-way, which means that it is easy to compute the hash value for any input, but it should be difficult to generate the original input from the **hash value**.

This property makes **hash functions** useful for password storage, as the actual password can be **hashed** and stored instead of the plaintext password, making it more difficult for an attacker to obtain the original password.

> i It's worth noting that while **hash functions** are commonly used for password storage, other techniques are used in conjunction with **hashing**. For the purposes of this course, we won't go into the details of these other techniques such as salting. Just keep in mind that password security is an important consideration and there are multiple techniques that need to be deployed to help protect user data.

To get the **hash value** of an object in Java, you can call the `hashCode()` method. For example:

```
public class Client {
    public static void main(String[] args) {
        String word = "hi";
        System.out.println(word.hashCode());
    }
}
```

Here, we see that the word "hi" has a **hash value** of 3329. When we call `hashCode()`, that is the equivalent of putting the word "hi" into a mathematical function to compute the **hash value**:
$hashFunction(”hi”) = 3329$

□ Main Points

- **Hashing** is a mathematical function that transforms data into a **hash value**.
- Each **hash value** will be placed in a corresponding index in the **hash table**, which is an array that store our elements through **hashing**.
- A **hash function** is one way: it is easy to turn our original input into a **hash value**, but very difficult to convert our **hash value** back into our original input.
- **Hash functions** are great for storing passwords so we can store our **hashed** password instead of our actual one, making it harder for hackers to get our password.
- We can use the `hashCode()` method to get the **hash value** of an object in Java.

hashCode() [Background Reading]

In Java, `hashCode()` is a method which returns an integer value, known as the **hash value**. The **hash value** is used for **hashing** and indexing objects in data structures like **hash tables**, which allow fast retrieval and insertion of objects based on their **hash values**. Suppose we had the following class:

```
public class Car {
    private String make;
    private String model;
    private int year;

    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    public String getMake() {
        return this.make;
    }

    public String getModel() {
        return this.model;
    }

    public int getYear() {
        return this.year;
    }

    @Override
    public int hashCode() {
        // TODO
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) {
            return true;
        } else if (o instanceof Car) {
            Car other = (Car) o;
            return this.make.equals(other.make) &&
                    this.model.equals(other.model) &&
                    this.year == other.year;
        } else {
            return false;
        }
    }
}
```

Implement `hashCode()`

First, let's implement `hashCode()`. Here is an initial implementation:

```
@Override
public int hashCode() {
    return this.make.length() * 2;
}
```

With this **hash function**, we take the length of the name of the make, multiply it by 2, and then return that value. This leads us to an interesting observation which is that two objects can produce the same **hash value**.

Is this a bad thing? Not necessarily. In Java's `String` class, you might encounter two `String`s that are not equal, yet have the same **hash value**:

```
public class Client {
    public static void main(String[] args) {
        String word1 = "Siblings";
        String word2 = "Teheran";
        System.out.println(word1.hashCode());
        System.out.println(word2.hashCode());
        System.out.println(word1.equals(word2));
    }
}
```

When the `hashCode()` method returns the same **hash value** for different objects, this results in **hash collision**. In other words, this means that multiple objects are going to the same **hash table** index.

While **hash collisions** happen, we want to minimize the amount of times it occurs in our **hash table**. With our current **hash function**, any make with the same length will result in the same **hash value**. For example, "Honda" and "Tesla" would result in a **hash collision**.

Recall that the **hash value** is used to put the object at a specific index in the **hash table**. In this course, we will not discuss collision resolution techniques but just understand that when multiple objects continuously map to the same index in the **hash table**, it can slow down the performance of the **hash table**. Thus, let's revise the **hash function**.

A general rule of thumb when creating **hash functions** is to use prime numbers in order to reduce the likelihood of a **hash collision** as prime numbers reduce the number patterns in a **hash value**. Here is better implementation of `hashCode()`:

```
@Override
public int hashCode() {
    int prime = 31;
```

```
    return this.make.length() * 31 * this.year;
}
```

We can go a step even further and use the `hashCode()` method inside the `String` class!

```
@Override
public int hashCode() {
    return Math.abs(31 * this.make.hashCode() * this.model.hashCode() *
        this.color.hashCode() * this.year);
}
```

> **NOTE:** We are using Math.abs() because some **hash values** can go over the integer limit in Java. Negative **hash values** are perfectly legal but since we want to use the **hash values** for indexing, we will need them to be positive.

Equality with Hashing

One important invariant that the `hashCode()` method imposes is that if two objects are equal, they must always produce the same **hash value**. Therefore, when you are creating your **hash function**, you cannot include any element of randomness. To put it more clearly:

- Two objects can have the same **hash value** and not equal.
- Two objects that are equal must have the same **hash value**.

Main Points

- In this lesson, we worked on implementing our own `hashCode()` method. We wanted to do this because **hash tables** allow fast retrieval and insertion of objects based on their **hash values**.
- Two `String`s that are not equal may have the same **hash value** in Java.
- A **hash collision** occurs when the `hashCode()` method returns the same **hash value** for different objects and multiple objects go into the same **hash table** index.
- We want to minimize how many **hash collisions** occur in our **hash table**. One way of doing this is to use *prime numbers* when creating our **hash function**, since prime numbers reduce the number patterns in a **hash value**.
- When we create our **hash function**, we must ensure that two objects that are equal must have the same **hash value**.
    - But two objects can also have the same **hash value** and not be equal.

Implementing a Hash table [Background Reading]

Now that we've implemented the `hashCode()`, let's build a **hash table**!

Suppose that we had a garage for `Car` objects called `Garage`. The `Garage` class keeps track of the `parkingLot` which represents the parking lot and the `size` which represents the number of cars in the `Garage`. In our `Garage`, we will assign each `Car` a parking lot number, depending on their **hash value**:

```
public class Garage {
    public static final int PARKING_SPOTS = 100;
    private Car[] parkingLot;
    private int size;

    // Constructs a new, empty set with the given underlying capacity
    public Garage() {
        parkingLot = new Car[PARKING_SPOTS];
        size = 0;
    }

    // Returns true if this set contains the specified car and false otherwise
    public boolean contains(Car car) {
        int index = car.hashCode() % PARKING_SPOTS;
        return parkingLot[index] != null;
    }

    // Adds the specified car to this garage if it is not already present.
    // Returns true if the car was not already present and false otherwise
    public boolean add(Car car) {
        if (contains(car)) {
            return false;
        }
        int index = car.hashCode() % PARKING_SPOTS;
        parkingLot[index] = car;
        size++;
        return true;
    }

    // Removes the specified car from this garage if it is present.
    // Returns true if the car was present and false otherwise
    public boolean remove(Car car) {
        if (!contains(car)) {
            return false;
        }
        int index = car.hashCode() % PARKING_SPOTS;
        parkingLot[index] = null;
        size--;
        return true;
    }
```

```
    // Returns the number of car in this garage
    public int size() {
        return this.size;
    }

    // Returns true if this garage has no cars
    public boolean isEmpty() {
        return this.size == 0;
    }
}
```

One thing to note is that in our `Garage`, if two `Car` objects have the same **hash value**, they will map to the same parking lot. Again, we won't be covering any collision resolution techniques. In our current `Garage` implementation, we will give the most recent `Car` object the parking spot.

You often hear that **hash tables** provide fast look-up! To fully visualize the power behind **hashing**, let's see how Java's `ArrayList` compares with our **hash table**, a `Garage`.

□ Main Points

- We created a `Garage` class that keeps track of the number of cars in our garage and the parking lot of each car in our garage.
- We assigned each `Car` a parking lot based on their **hash value**.
- To do this, we established the index of each car by using the line `int index = car.hashCode() % PARKING_SPOTS;` and then we put each car into that corresponding index in the array that holds our parking lots.
- If two `Car` objects had the same **hash value**, they mapped to the same parking lot.
- We will explore how Java's `ArrayList` compares with our **hash table**, a `Garage`, in the next slide!

ArrayList vs. Garage [Programming Example]

In `HashingTest`, we are calling `add()`, `remove()`, and `contains()` on 20 `Car` objects on Java's `ArrayList` and our `Garage` to see how long it takes to complete. Press the "Run" button a couple of times to see the performance difference!

NOTE: The performance different between the two will be minor but it should be visually noticeable. You could imagine that if this program scaled to a larger amount of `Car` objects, the difference would be much greater than it currently is.