**Exercises for lesson 2: Abstract datatypes, sets, maps and hashing**

**Exercise 2.1**

Discuss the difference between data structures and abstract datatypes, and find examples of both.

Abstract datatypes: don’t know the implementation, mathematical model

Data structures: implementation, physical memory layout

**Exercise 2.2**

Discuss what is meant by the union and intersection of two sets.

Union: everything

Intersection: only the common stuff

**Exercise 2.3**

State whether the following are sets:

{3,5,7,1,9} Yes

{3,5,6,5,1} No

{1,2,3,4,5} Yes

**Exercise 2.4**

Which of the following are maps?

A) { (J, Jane), (B, Bill), (S, Sam), (B1, Bob), (B, Bill) } No (duplicate B)

B) { (J, Jane), (B, Bill), (S, Sam), (B1, Bob), (B2, Bill) } Yes

C) { (J, Jane), (B, Bill), (S, Sam), (B1, Bob), (J, Jane) } No (duplicate J)

D) { (S, Sam), (B, Bill), (S, Sam), (B1, Bob), (B2, Bill) } No (duplicate S)

**Exercise 2.5**

Given input {4371, 1323, 6173, 4199, 4344, 9679, 1989} and a hash function , show the tables for a hash table size 10:

a. Separate chaining hash table.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 4371 |  | 1323 -> 6173 | 4344 |  |  |  |  | 4199 -> 9679 -> 1989 |

b. Hash table using linear probing.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9679 | 4371 | 1989 | 1323 | 6173 | 4344 |  |  |  | 4199 |

c. Hash table using quadratic probing.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 9679 | 4371 |  | 1323 | 6173 | 4344 |  |  | 1989 | 4199 |

d. Hash table with second hash function .

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 4371 |  | 1323 | 6173 | 9679 |  | 4344 |  | 4199 |

Cannot put 1989.

**Exercise 2.6**

The contract between equals() and hashCode()

In Java, all classes inherit from the Object class. Object defines many methods, and among those are equals() and hashCode().

Let us take a look at the Camera class. It has two fields, id and model. In our understanding, if you make two instances of Camera with the same parameters, they should be the same. Try to create two such objects, and then print the result of using equals() to compare them.

As you probably noticed, the two Camera objects are not the same, according to the equals() method inherited from the Object class. If we want to fix that, we need to override the equals() method.

There are some requirements that must be fulfilled. You can find them in the [Java documentation](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)), but really they are common sense.

Try the following code. Does it fix the issue? Does it fulfill the requirements?

@Override  
public boolean equals(Object o) {  
 if (!(o instanceof Camera))  
 return false;  
 Camera other = (Camera)o;  
 boolean equalModels = (this.model == null && other.getModel() == null)  
 ||  
 (this.model != null && this.model.equals(other.getModel()));  
 boolean equalIds = (this.id == null && other.getId() == null)  
 ||  
 (this.id != null && this.id.equals(other.getId()));  
 return equalIds && equalModels;  
}

*Aside:*

*You also need to have equals() in mind with inheritance. Consider a CellPhone class. One might consider a CellPhone a Camera with extra functionality, and have CellPhone extend Camera. You would then override the equals() method, and this method would always return false when compared to a Camera, even if the Camera and the CellPhone had the same id and the same model. However, if those two fields indeed had the same values, Cameras equals() method would return true, since all its fields match, and the CellPhone object actually is an instance of Camera, because of the inheritance rules!*

*Once again, the compostion over inheritance principle is useful here: Don’t have CellPhone extend Camera; instead, CellPhone could have a field of type Camera.*

Now, on to hashCode(). This method also has some requirements. They can be found in the docs as well, but here is the important one:

* If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result.

Since we have changed how equals() works, most likely hashCode() will now violate this principle.

Try to make a HashMap with Camera as the key and perhaps Integer as the value, and put one of the instances of Camera you used earlier. Then use get with the other instance as the key. You will see that you will receive null instead of the Integer. That is because the two instances return different hash codes.

So you will need to override hashCode(). Remember the requirements. It would be best if the outcomes looked random, so that two similar objects wouldn’t have similar hash codes - but of course, they cannot be random. It is customary to use the hash codes of all of the object’s fields, and then do some multiplication or other mathematical operations on them, but there are no hard rules. Try to come up with a good scheme, and test it!