CSCI241 Fall 2020: Lab 5
Due Sunday, November 8th at 9:59pm

Overview

In this lab, you'll learn a little more about hash functions, including properties they must have, properties that are nice to have, and how hashing is done in Java.

Goals

This lab presents new material that was not discussed in lecture but you may be asked about on quizzes and/or exams. Here are the goals for this lab:

- Know what properties a function must have in order to be a **valid** hash function, and why.
- Know what properties are **nice** to have in a hash function, and why.
- Understand the contract of the java `Object` class's `equals` and `hashCode` methods.

1 Definitions

For purposes of this lab, a hash function is a function that maps a value to an integer. Note that we've excluded the modulus from this definition: someone (e.g., you) implementing hash table can take the integer returned by a hash function and calculate the modulus with their table size. This definition matches the way Java defines its `hashCode` method.

2 Hash Functions: Necessary Properties

To be a **valid** hash function, the hash function should satisfy the following properties:

1. Calling the hash function on the same value twice must return the same value.
2. Calling the hash function on two equal values must return the same value.
3. It is **not** necessarily the case that calling the hash function on two different objects returns **different** hash values. In other words, hash **collisions** are possible.

The reason for the first necessary property is simple: if you use a hash value to decide where (e.g., in which bucket) to store a value, then you need to be able to get back there again to retrieve it.

The second property is important for maintaining a well-defined set or mapping. In a Set, duplicate values are not allowed; in a Map, duplicate keys are not allowed. If a value is added to a HashSet, you need the hash function to send you to the correct bucket where its already-existing duplicate would exist, if it does, or you risk adding a duplicate value; similar reasoning applies to HashMaps with duplicate keys.

It is not reasonable to insist that different values have different hash values. There are only about 4.3 billion `int` values, and many types may have more (indeed, infinite) possible values. For example, there are about 8 billion strings containing 5 lower-case letters. Clearly, we can't assign a unique integer to every possible String.

3 Hash Functions: Nice Properties

In addition to the necessary properties above, it is **nice** if a hash function also has the following properties:

1. Fast (e.g., constant time).
2. As much as possible, different values have different hash values.

Speed is important because all of the runtime benefits of hashing hinge on the efficiency of finding the correct bucket where a value must be stored.

Although we can't count on it, a nice well-behaved hash function does a good job of spreading values out among many buckets, thereby minimizing the number of collisions. As we've seen in lecture, collisions are the cause of all the worst-case inefficiencies of hash tables. Fewer collisions means better practical runtime.

4 `hashCode` and `equals`

Hashing in java abides by the above principles. The Object class, from which all reference types inherit, has methods `equals` and `hashCode`. The `equals` method is Java's way of determining what is meant by "the same value"; the `hashCode` method implements a hash function for any type.

Read the specification for the `equals` method, then read the spec for the `hashCode` method. Notice that the `hashCode`'s contract closely resembles the rules outlined in Section 1.

The key thing to keep in mind when writing classes in Java is that **if your class overrides `equals`**, then there's a good chance that your `hashCode` method (still the version inherited from `Object`) violates the contract by allowing two values that are equal according to `equals` to hash to different values. If this is the case, your class needs to also override `hashCode` in order to avoid breaking the contract of the `hashCode` method's contract.

5 Questions

Submit your typed answers to the following questions to the Lab 5 assignment on Canvas in pdf format.

1. (5 pts) Suppose you are implementing a hash table. For some reason, you will never need to hash anything that's not an integer value. You plan make sure the size of your table is always a prime number. The functions below do not include the modulus; your hash table implementation will take the modulus of the result to compute the correct bucket index. Consider the following candidate hash functions:

   (a) $h(x) = x$
   (b) $h(x) = 2x$
   (c) $h(x) = x * x$
   (d) $h(x) = x\%10$
   (e) $h(x) =$`x * random.nextInt()` (assume `random` is an instance of `java.util.Random`

   For each hash function above, answer the following three questions:

   i. Say whether it is a valid hash function (i.e., it satisfies all the necessary properties)
   ii. Say whether it has one or more of the desirable (nice) hash function properties. Explain your answer.
   iii. Do any of your answers in (ii) change if the size of your table might be even? If so, explain.

2. (1 pt) What value does a class's `hashCode` method return if it's not overridden? In other words, what does the java Object class's hashCode method return? Hint: take a look at the documentation for Object's hashCode method.

3. (4 pts) You are a member of a team of software developers implementing a Course Management System called Linen. You are working on implementing a Student class to store information about each student:

```
public class Student {
    private String firstName;
    private String lastName;
    private int wNum; // unique W#
    private Date lastLogin; // when the student last logged in

    /** Returns whether this is the same student as other */
    @Override
    public boolean equals(Student other) {
      return wNum == other.wNum;
    }

    public int hashCode() {
      // code here
    }
}
```

You have just written the `equals` method to override `Object`'s equals method and you're contemplating overriding the `hashCode` method.

(a) Do you even need to override `hashCode`? Why or why not?

(b) Assume that W#s (stored in `wNum`) are unique, so any two student objects with the same `wNum` represent the same student and will also have the same values in the name fields. Consider each line of code below as a possible implementation of `hashCode` (to replace the "code here" comment in the class above). Which of the following implementations would constitute a *valid* hash function? Explain.

i. `return firstName.hashCode() + lastName.hashCode();`

ii. `return wNum + lastLogin.hashCode();`

iii. `return wNum;`

iv. `return 0;`

(c) Which of the above is the *best* hash function? Explain.