Maps

Today we will begin looking at a way to organize data which is called a *map*. You are familiar with maps already. In high school math and in Calculus and linear algebra, you have seen functions that go from (say) $\Re^n$ to $\Re^m$. In COMP 250, we have seen functions $t(n)$ that describe how the number of operations performed by some algorithm depends on the input size n.

In general, a map is a set of ordered pairs $\{(x, f(x))\}$ where x belongs to some set called the domain of the map, and $f(x)$ belongs to a set called the co-domain. The word range is used specifically for the set $\{f(x) : (x, f(x)) \text{ is in the map}\}$. That is, some values in the co-domain might not be reached by the map.

Example: Object.hashCode()

Let's jump right in and consider a map that you may be less comfortable with at this point but which is hugely useful when programming in Java. When a Java programming is running, every object is located somewhere in the memory of the Java Virtual Machine (JVM) and we call this location the address of the object. In particular, each Java object has a unique 24 bit number associated with it which by default is the number returned when the object calls `hashCode()` method. (What exactly this 24 bit number means may depend on the implementation of the JVM. We will just assume that the number is the object's (starting) address in the JVM.) Since different objects must be located at different memory address locations, we should understand "`obj1 == obj2`" to mean the same thing as "`obj1.hashCode() == obj2.hashCode()`". The statement can be either true or false, depending on whether the variables `obj1` and `obj2` reference the same object or not.

ASIDE: As we will see a few lectures from now, the above discussion about the `hashCode()` method only is about the default implementation, namely the `Object.hashCode()` method. Some classes override this default `hashCode()` method. This is the case in the next example we consider.

Example: String.hashCode()

Each string is an object and is also located somewhere in memory. The String class uses a different `hashCode()` method, however. (Let's not concern ourselves with the reason why, for now.)

In Java, strings are composed out of a particular subset of the unicode characters, namely each character in Java has a 16 bit code. It has a minimum value of 0 and maximum value of $2^{16} - 1$. The characters include those of most written languages. For details, see:

`https://en.wikipedia.org/wiki/Plane_(Unicode)#Basic_Multilingual_Plane`

When one is writing in English, the characters that one uses also belong in particular to the ASCII set. ASCII characters are encoded with only 8 bits each. The ASCII characters are the first 256 (2^8) characters of unicode.

Anyhow, back to the task at hand which is to define the `hashCode()` method of the `String` class. Let's first consider a simple map from `String` to positive integers.

Suppose s is a string which consists of characters $s[0]s[1]...s[s.length-1]$. Consider the function

$$h(s) = \sum_{i=0}^{s.length-1} s[i]$$

which is just the sum of the codes of the individual characters. Notice that two strings that consist of the same letters but in different order would have the same $h()$ value. For example, "eat", "ate", "tea" all would have the same code. Since the codes of `a, e, t` are 97, 101, and 116, the code of each of these strings would be 97+101+116.

In Java, the `String.hashCode()` method is defined[1] :

$$h(s) = \sum_{i=0}^{s.length-1} s[i]\ x^{s.length-i-1}$$

where $x = 31$. So, for example,

$$h("eat") = 101 * 31^2 + 97 * 31 + 116$$

and

$$h("ate") = 97 * 31^2 + 116 * 31 + 101$$

Thus, here we have an example of how strings that have the same letters do not have the same hashCode.

What about if two strings have the same hashcode? Can we infer that the two strings are equal? No, we cannot. I will include this as an exercise so that you can see why.

ASIDE: Horner's rule for efficiently evaluating polynomials

Suppose you wish to evaluate a polynomial

$$h(s) = \sum_{k=0}^{N} a_k\ x^k$$

It should be obvious that you don't want to separately calculate $x^2, x^3, x^4, ..., x^N$ since there would be redundancies in doing so. We only should use $O(N)$ multiplications, not $O(N^2)$.

Horner's Rule describes how to do so. The following example gives the idea for $N = 3, x = 31$:

$$s[0] * 31^2 + s[1] * 31 + s[0] = ((s[0] * 31^1 + s[1]) * 31 + s[2]) * 31 + s[3]$$

One uses the following:

```
h = 0
for  (i = 0;  i < s.length;  i++)
    h = h*31 +  s[i]
```

[1]You might wonder why Java uses the value $x = 31$. Why not some other value? There are explanations given on the website stackoverflow, but I am not going to repeat them here. Other values would work fine too.

Maps as (key,value) pairs

You are familiar with the idea of maps in your daily life. You might have an address book which you use to look up addresses, telephone numbers, or emails. You index this information with a name. So the mapping is from name to address/phone/email. A related example is "Caller ID" on your phone. Someone calls from a phone number (the index) and the phone tells you the name of the person. Many other traditional examples are social security number or health care number or student number for the index, which maps to a person's employment record, health file, or student record, respectively.

We will use the following definition of a map. Suppose we have two sets: a set of keys K, and a set of values V. A *map* is a set of ordered pairs

$$M = \{(k, v) : k \in K, v \in V\} \subseteq K \times V,$$

The pairs are called *entries* in the map. A map cannot just be any set of (key, value) pairs. Rather, for any key $k \in K$, there is *at most* one value v such that (k, v) is in the map. We allow two different keys to map to the same value, but we do not allow one key to map to more than one value. Also note that not all elements of K need to be keys in the map.

For example, let K be the set of integers and let V be the set of strings. Then the following is a map:

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{meatball}), (5, \texttt{dog}), \}$$

whereas the following is not a map,

$$\{(3, \texttt{cat}), (18, \texttt{dog}), (35446, \texttt{meatball}), (5, \texttt{dog}), (3, \texttt{fish})\}$$

because the key 3 has two values associated with it.

Map ADT

The basic operations that we perform on a map are:

- put(key, value) – this adds a new entry to the map
- get(key) – this gets the entry (key, value), and it would return null if the given key did not have an entry in the map.
- remove(key) – this removes the entry (key, value) – this might return the value, or else return null if the key wasn't present

There are other methods such as How can we represent a map using data structures that we have seen in the course? For example, we might have a key type `K` and a value type `V` and we would like our map data structure to hold a set of object pairs $\{\ (k, v)\ \}$, where k is of type `K` and v is of type `V`. (In the context of Java programming, we would have to decide whether these are primitive types or reference types, but that is not the point now.)

Map data structures

We can use many different data structures to represent maps. We could use an arraylist or linked list of entries, for example. This would mean that the operations defined above were slow, however, namely $O(n)$ if the map has n entries.

What if we made a stronger assumption, namely what if the keys of map were comparable? (Recall that by "comparable" we mean that $k_1 < k_2$ or $k_1 = k_2$ or $k_1 > k_2$ for any two keys k_1 and k_2.) In this case, we could use a sorted array or a binary search tree) to organize the entries of the map. If we use a sorted array, then we can find a key in $O(\log n)$ steps, where n is the number of pairs in the map. Once we have found the entry with that key, we can find the key's associated value v in $O(1)$ steps, since a reference to the value is stored together with the key, that is, (k,v) is stored as a pair. However, with a sorted array it is relatively slow to `add` or `remove` a (key,value) pair, namely $O(n)$. To get around this worst case behavior, we could instead use a binary search tree (BST) to store the (k, v) pairs, namely we index by comparing keys. The reason is that with a BST you can index with $O(\log n)$ steps. (I will remind you again that, although the binary search trees we have covered have $O(n)$ worst case behavior, you will learn in COMP 251 that it is possible to have balanced= binary search trees that allow adding, removing, finding in $O(\log n)$ time.)

What about a heap? A heap would be an appropriate data structure for a map if keys were comparable and if were often trying to access the minimum key. For a general `get` operation though, a heap would be $O(n)$ since one would have to traverse the entire heap. Note that the heap is represented using an array, so this would just be a loop through the elements of the array. (Note that this corresponds to breadth first search of a the heap.)

While sometimes keys are comparable, often this is too strong an assumption. For example, with student ID's, we can compare the values of the IDs but the ID values themselves are not meaningful. (We would never say, for example, "give me all students whose ID's are between values X and Y", or give me the students with the next ID.) Another example is if we have a hashCode() for some class of objects, since again we would have a number and we could order objects based on the `hashCode()`. However, the `hashCode()` doesn't isn't meaningful. Of course, we *could* define an ordering based on these numbers and we could define a data structure such as a BST that is based on the ordering. But, it turns out there is a better way to go, called `hashMaps`. I'll discuss it more next lecture.

To give you the basic idea, let's suppose the keys `K` are positive integers. In this case, we could just use an array such that the key is used as an index into the array. Note that this typically will *not* be an arraylist, since there may be gaps in the array between entries. Using an array would give us access to map entries in $O(1)$ time. However, this would only work well if the integer values are within a small range. For example, if the keys were social insurance numbers (in Canada), which have 9 digits, we would need to define an array of size 10^9 which is not feasible since this is too big.

Despite this being infeasbile, let's make sure we understand the main idea here. Suppose we are a company and we want to keep track of employee records. We use social insurance number as a key for accessing a record. Then we could use a (unfeasibly large) array whose entries would be of type `Employee`. That is, you would use someone's social insurance number to index directly into the array, and that indexed array slot would hold a reference to an `Employee` object associated with that social insurance number. Of course this would only retrieve a record if there were an an `Employee` with that social insurance number; otherwise the reference would be null and the `find` call would return null).