Maps

The last few lectures, we have looked at ways of organizing a set of elements that are comparable. Today we will begin looking a different way of organizing things which doesn't require that they are comparable. We will mainly be looking at *maps*.

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.

Maps as (key,value) pairs

You are also 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.

Maps are defined as follows. 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 `contains(key)` or `contains(value)` as well but the above are the main ones.

Map data structures

How can we represent a map using data structures that we have seen in the course? 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`. We could use an arraylist or a linked list of entries, for example. This would mean that the operations defined above would be slow in the worst case, however, namely the worst case would be $\Theta(n)$ if the map has n entries.

What if we made a stronger assumption, namely what if the keys of map were comparable? In this case, to organize the entries of the map, we could use a sorted arraylist or a binary search tree. If we used a sorted arraylist, then we could find a key in $O(\log n)$ steps, where n is the number of pairs in the map. Each slot of the sorted arraylist would hold a map entry. We would use binary search to find the entry, based on the key. However, with a sorted array it can be relatively slow to `put` or `remove` a (key,value) pair, namely in the worst case it is $\Theta(n)$.

We could instead use a binary search tree (BST) to store the (k, v) pairs, namely we index by comparing keys. Although the binary search trees we have covered have $\Theta(n)$ worst case behavior, I have told you that there are fancier versions of binary search trees that are balanced that allow adding, removing, finding in $O(\log n)$ time. You'll learn about these in COMP 251.

What about a heap? A heap would not be an appropriate data structure for a map because it only allows you to efficiently get or remove the minimum element. However, for a general `get` operation, a heap would be $\Theta(n)$, 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.

Another special case to consider is that the keys `K` are positive integers in a small range. In this case, we could just use an array and have the key be an index into the array and the value be the thing stored in 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.

For the rest of today, we consider the case that the keys map to a set of integers, without requiring these integers to be a small range. Next lecture we will address the problem of mapping to a small range of integers and treating these integers as an index into an array.

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). This location is called an *address*. In particular, each Java object has a unique 32 bit number associated with it which by default is the number returned when the object calls `hashCode()` method. (What exactly this 32 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 non-overlapping in memory, it follows that they have different addresses. In particular, when the object's `hashCode()` is the object's address, the expression "`obj1 == obj2`" means 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.

Example: String.hashCode()

The above discussion about the `hashCode()` method only considers the default implementation. Some classes override this default `hashCode()` method. For example, each String object is also located somewhere in memory but the String class uses a different `hashCode()` method.

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,

$$h(\text{"}eat\text{"}) = 101 * 31^2 + 97 * 31 + 116$$

and

$$h(\text{"}ate\text{"}) = 97 * 31^2 + 116 * 31 + 101$$

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

[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.

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

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 of Horner's rule for the case of hashcodes for a `String` object with four characters:

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

For a `String` object of arbitary length, Horner's rule is the following algorithm :

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

In the lecture, I ran out of time and did not present Horner's rule. But I would like you to see it and understand it since it is a simple idea and worth knowing.