Maps

Today we will begin looking at collections of data that are called maps. 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.

For a more formal 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 – also called entries –

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

such that, for any key $k$ in our set of possible keys, there is at most one value $v$ such that $(k, v)$ is in the map. It is possible to have two keys map to the same value.

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

$$\{(3, \text{cat}), (18, \text{dog}), (35446, \text{blablabla}), (5, \text{dog}), \}$$

whereas the following is not a map,

$$\{(3, \text{cat}), (18, \text{dog}), (35446, \text{blablabla}), (5, \text{dog}), (3, \text{fish})\}$$

because the key 3 has two values associated with it.

Data structures for maps

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.

In general, if the keys of map are comparable, then we can use one of the data structures we discussed earlier (sorted array, binary search tree) to organize the pairs $\{(k, v)\}$ so that we can quickly access a pair by its key $k$. (See slides for an illustration of the ideas in this paragraph.) 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 (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. (You will learn in COMP 251 how to maintain a binary search tree so that it is balanced.)

Next lecture, we will discuss another scheme for representing maps, called hashing. Before I can explain hashing, though, I need to introduce a more few concepts.
Direct addressing

Suppose the keys \( K \) are positive integers. For example, the keys might be social insurance numbers (9 digit integers – in Canada anyhow) and the values be \texttt{Employee} objects (which hold varies pieces of information about a person working in a company). Social insurance numbers have 9 digits, so you could in principle define an array of size \( 10^9 \) of type \texttt{Employee}. That is, you would use someone’s social insurance number to index directly into the array, and access the address of an \texttt{Employee} object associated with that social insurance number (if indeed there was an an \texttt{Employee} object with that social insurance number. Otherwise, the reference would be null and the \texttt{find} call would return null). See the sketch in the slides. This scheme for representing a map is called \	extit{direct addressing}. You provide the key, and in time \( O(1) \) – namely an array access – you have the reference to the \texttt{Employee} object with that social insurance number.

Direct addressing works fine when the number of possible integers keys is relatively small, but not when the number of possible keys is large. For example, a typical company will not have anywhere near \( 10^9 \) employees, so an array of size \( 10^9 \) holds references to \texttt{Employee} objects would be mostly empty, i.e. contain nulls. This is clearly a waste of memory. Next class, we will see how to deal with this problem.

Other examples of maps

(address, object)

When a Java programming is running, every object must sit somewhere in memory and thus has a (starting) memory address. For example, in many computer operating systems, addresses are either 32 bit numbers or 64 bit numbers. Another example is that in many implementations of the Java Virtual Machine, objects have a unique 24 bit number which is determined (in an unspecified way) by the object’s address in the JVM, such that different objects have different numbers. This is the number returned by the \texttt{Object.hashCode()} method.

A set of objects together with their addresses, give us a set of \{\texttt{(address,object)}\} pairs. This mapping from addresses to objects is exactly what the Java Virtual Machines does when it runs your program. Reference variables hold object addresses (keys) and the JVM takes an address and looks up the object (value) at that address in memory.

What’s new here is that we are thinking for the first time of using the object’s address as a key for comparisons. Indeed, if you think about it, that’s not so different from what that a social insurance number (or phone number) does. Its just a number that is used to index other information. The number itself has no special meaning.

(string, big integer)

Another interesting map is from the set of strings to the set of integers, that is, for each string we define some integer. For example, consider strings made up of ASCII characters which are 8 bits (one byte) each. Let \( s \) be a string \( s[0]s[1]\ldots s[n-1] \) with \( s.length \) characters. A mapping from (ASCII) strings to integers could be defined using base \( 2^8 \), as follows:

\[
\text{mapping } m : s \rightarrow \sum_{i=0}^{s.length-1} s[i] \cdot (2^8)i .
\]
If the characters that make up a string were in UNICODE (16 bits, i.e. 2 bytes each) then for any string we could define a positive integer:

\[ s \rightarrow \sum_{i=0}^{s.length-1} s[i] \cdot (2^{16})^i \]

where \( s[i] \) represents a number from 0 to \( 2^{16} - 1 \). So the integer defined by string \( s \) is being expressed as a number with base \( 2^{16} \). That is, not base 2, not base 10, not base 16 (like hexadecimal), not base \( 2^{8} = 256 \) like in the string example above, but rather base \( 2^{16} \).

Notice that this mapping would not produce preserve the lexicographic ordering of strings of a given length, however, since strings \( s_1 \) and \( s_2 \) should be compared first by their \( s_1[0] \) vs \( s_2[0] \) characters but that is not what would happen here. Instead, one could change the definition slightly to:

\[ mapping \ m : s \rightarrow \sum_{i=0}^{s.length-1} s[i] \cdot (2^{16})^{length-i-1} \]

This now would preserves the lexicographic ordering of strings of a given length. That is, if \( s_1 < s_2 \) then \( m(s_1) < m(s_2) \) where \( m(s) \) is the mapped (integer) value.

**SEE THE SLIDES FOR AN EXAMPLE – “hello” vs. “prime”.

The same argument holds for base 256 (ASCII), of course. If we define the mapping

\[ mapping \ m : s \rightarrow \sum_{i=0}^{s.length-1} s[i] \cdot (2^{8})^{length-i-1} \]

then the string of my first name, “Mike”, would be mapped to the integer

\[ 77 \cdot 256^3 + 105 \cdot 256^2 + 107 \cdot 256^1 + 101 \]

since the ASCII codes of the characters 'M', 'i', 'k', 'e' are 77, 105, 107, and 101, respectively. See [http://www.asciiart.com](http://www.asciiart.com).

Does this given a unique mapping from the set of all strings to the set of positive integers? Not quite. The ASCII or UNICODE value of 0 stands for the symbol null – indeed, when you have a null reference in your program, it means that the value stored in that variable has all 0 bits. Thus “Mike” would have the same encoding as “Mike” with an arbitrary number of null characters appended to the end of it. Other than this ambiguity, no two strings map to the same value (and for every value, there is a unique string – up to the ambiguity of trailing null characters).