

## Maps

We have seen many data structures for making a collection of objects (lists, trees, etc) and operations such as finding, adding and removing objects from these collections. We saw that when the objects were comparable i.e. so that an ordering was defined, we could use sorted arrays or binary search trees for fast indexing.

There are various ways to define a comparison relation between a given set of objects. Objects often have several pieces of data that are associated with them. The most obvious are the data items that are stored or referenced by the fields. If one is comparing the objects based on one of these fields, the field is called the *key*. For example, take the entries in a telephone book or a personal address book. Each entry has a name that you use to index an entry. This is the key.

The key is used to access all the information about the object, or to put it more simple, it is used to access the object itself. When discussing maps in general, one refers to the *value*. For the address book example, the value is an address and phone number and maybe email, etc.

Note that there is nothing inherently different about the type of information used in the key vs. value. The value might contain data that could be used as a key, even though it is *not* being used as the key. For example, we could organize our address book with the phone numbers as the key, and use the phone number to look up the names of the person with this particular phone number. This is what “Caller ID” does. Someone calls you and caller ID uses the phone number to lookup the person’s name.

[ASIDE: for those of you with a stronger math background, here is a more formal definition of what a map is. Suppose we have two sets: there is a set of keys  $K$ , and there is a set of values  $V$ . Define a *map* to be a set of ordered pairs

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

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. In that case, we would say that the mapping is not 1-1 (i.e. non-injective).]

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

Think of an object oriented language like Java and imagine how you might represent a map using a data structure. We will 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  $\{(k, v)\}$ , such that  $k$  is an object of type  $K$  and  $v$  is an object of type  $V$ .

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.) 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 to add or remove a  $(k, v)$  pair takes  $O(n)$  on average. To get around this bad average 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 on average.<sup>1</sup>

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) and the values be `Employee` objects (which hold varies pieces of information about a person working in a company). Since social insurance numbers have 9 digits, you could in principle define an array of size  $10^9$  of type `Employee`. That is, you would use someone's social insurance number to index directly into the array, and access the address of an `Employee` object associated with that social insurance number (if indeed there was an `Employee` object with that social insurance number. Otherwise, the reference would be null and the `find` call would return null). See the sketch in the slides. This scheme for representing a map is called *direct addressing*. You provide the key, and in time  $O(1)$  – namely an array access – you have the reference to the `Employee` object with that social insurance number.

Direct addressing works fine when the number of possible integers keys is relatively small, but there is a problem 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 `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. As discussed earlier in the course, memory addresses in 32-bit machines are numbers in the range  $0$  to  $2^{32} - 1$ .

If we consider some set of objects together with their addresses, then we have a set of  $\{(address, object)\}$  pairs. Since addresses are numbers, they are comparable and so if we have a reference to an object (namely we have the address of the object) we can get the object by indexing to that position in memory. Alternatively, we could build a data structure (a binary search tree, or a sorted array, or something else) that would use the memory address as the key, and this would allow us to index the object from this data structure.

---

<sup>1</sup>This idea was only sketched out in the BST lecture.

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  $s[0]s[1] \dots s[n-1]$  with  $s.length$  characters. A mapping from strings to integers could be defined using base  $2^8$ , as follows:

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

For example, the string "Mike" would be mapped to the integer

$$77 * 256^3 + 105 * 256 * 256^2 + 107 * 256^1 + 101$$

since the ASCII codes of the characters 'M', 'i', 'k', 'e' are 77, 105, 107, and 101, respectively. See <http://www.asciitable.com>. Note that we use the 0 to  $length - i - 1$  counting in the index of the summation, rather than 0 to  $i$ . This preserves the lexicographic ordering of strings of a given length. That is, if  $s1 < s2$  then  $m(s1) < m(s2)$  where  $m(s)$  is the mapped (integer) value.

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:

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

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  $2^{16}$ !

[ASIDE: see [http://en.wikipedia.org/wiki/List\\_of\\_Unicode\\_characters](http://en.wikipedia.org/wiki/List_of_Unicode_characters) for a list of the unicode characters.]