Last lecture I introduced the idea of maps and how they might be used for efficient access to some collection. The general definition of a map is a set of (key, value) pairs, so we have a mapping from a set of keys to a set of values. The idea is that we would have a data structure that would efficiently organize the keys and, for each key, the data structure would have a reference variable that would point to the key's value.

The terms "key" and "value" are old terms – older than object oriented programming, for example – and you should not attach too much significance to their meaning. For now, I suggest you think of a mapping in an object oriented way: the keys are objects of some type `K` and the values are objects of some type `V`. Sometimes the key is just a field of the value object. For example, the value class might be `Employee` and the key class might be the `Employee`'s field `socialInsuranceNumber`.

If the keys are comparable, then we can organize them using a binary search tree and access each value in $O(\lg n)$ time[1] In many situations, however, you don't want to wait $O(\lg n)$ time to access an item. Rather, you want $O(1)$ access time. How can we achieve this?

As we mentioned at the end of last lecture, if the keys are positive integers then we can use the keys as indices into an array whose size is at least as big as the largest key. This provides $O(1)$ access time to the values, namely the time it takes to access an array element plus the time to follow the reference stored there. The reference in the array would be `null` if the corresponding key was not in the map. Notice that accessing an array element just requires that you compute the address of that array element in memory and then you access the element at that memory location. Accessing an element at some memory address takes constant time. (See page 1 of lecture 8, if you are still uncomfortable with this.)

What if the largest possible key is too large a number for us to make an array of that size e.g. the case of 9 digit social insurance numbers? Or what if our keys are not integers? For example, what if the keys are strings? There are ways to define a unique integer for each string (e.g. consider numbers with base 2^{16} , where each character in the string is coded with 16 bits!). But this would define an array that is too large. What do we do in these cases ?

The solution is this problem is to use a much smaller array for our key-to-value map, and to allow many (keys,values) to be associated to the same array element. For example, we could store a *linked list* of (key,reference-to-value) pairs at each array element, rather than just a single reference. Let's now examine this solution in more detail.

Hashing

Define a *hash function*,

$$h : \{keys\} \rightarrow \{0, 1, 2, m-1\}$$

where m is some positive integer. Typically m is smaller than the number of keys that we might be considering. e.g. if the keys are possible social insurance numbers (10^9 of them), then we might let me be 1000. It is obviously possible for two keys to map to the same integer.

Next, define an array called a *hash table*, which we will use to hold references to the *values* of a map. The elements of a hash table are also called *buckets* or *slots*. The indices $0, \ldots, m-1$ of the hash table are often called *hash values*, that is, the values of the hash function.

[1]The search in a binary tree is $O(\lg n)$ provided we maintain the tree so that its height doesn't grow too large. You will have to wait for COMP 251 to see algorithms for rebalancing binary search trees.

If the keys are not integers, then the hash function needs to specify an integer for each key. This integer is called the *hash code* for that key. The hash function is then defined by two steps, called *hash coding* (keys to hash codes) and *compression* (hash codes to hash values), respectively:

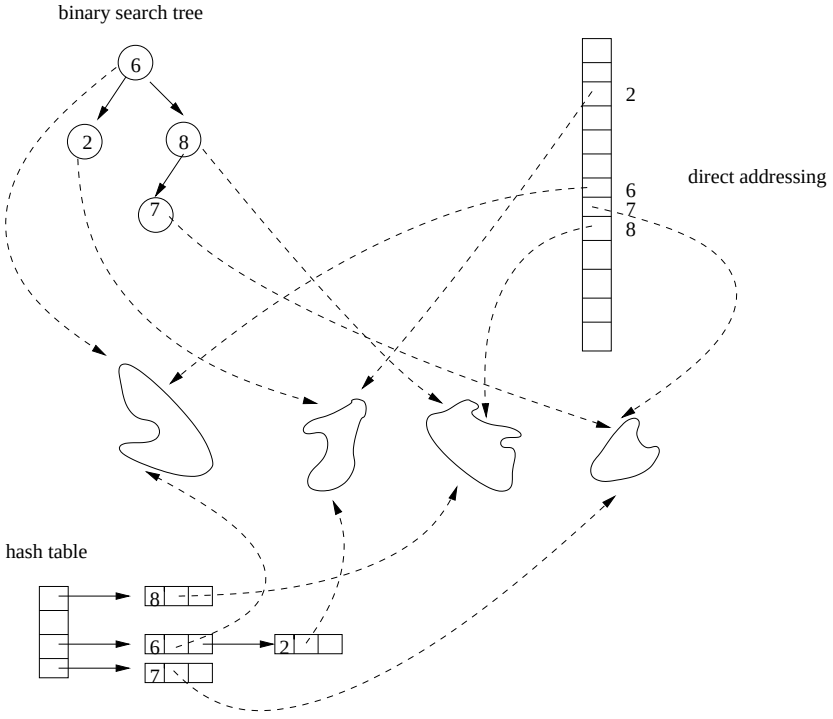
$$h : \{keys\} \to \{hash\ codes\} \to \{hash\ values\},$$

i.e. the values $\{0, 1, \ldots, m-1\}$ are called the *hash values*. Note that these "hash values" should not be confused with the "values" of the map (which are the objects that we are looking up).

Collisions, and separate chaining

It can happen that two keys hash to the same value. Thi is called a *collision.* How can we resolve collisions? One way is called *separate chaining.* Rather than keeping a single reference variable at each array position, we keep a linked list. Each node in the linked list has a key, a reference to a value, and a reference to the next node in the list.

The figure below illustrates the different ways we have been discussing for implementing maps. The keys are $\{2, 6, 7, 8\}$. We have organized them in three ways: a binary search tree, a large array, and a hash table. For the hash table, $h(2) = h(6) = 2, h(8) = 0, h(7) = 3$. Notice that in each of the three cases, the data structure storing the key has a reference variable that points to one of the four values (objects).



In the worst case that all the elements in the collection hash to the same location in the array, then we have one linked list. This is clearly undesirable for large collections since we want to access values as quickly as possible, in particular, in $O(1)$ time. To avoid having such long lists, we would like to choose a hash function so that there is roughly an equal chance of mapping to any of the hash values. (The word hash means to "chop/mix up".)

Hash functions

Assume the keys themselves are either non-negative integers i.e. they belong to $\{0, 1, 2, \dots\}$, or each key has a hash code. How can we define a mapping from the keys to the buckets $\{0, 1, \dots, m-1\}$?

division method

One common method is to define:

$$h(k) = \text{k mod m}$$

Often one takes m to be a prime number, though this is not necessary. This method is relatively easy to compute. Note that in the example above used this method (with $m = 4$).

multiplication method (no, you don't need to memorize this one)

Suppose that the *hashcode* (or key) is a b bit positive integer. Take a constant A which is also a b bit integer, i.e. between 0 and $2^b - 1$. Then Ak is at most $2^{2b} - 1$. We can then define the hash function by pulling out a particular set of n bits from this product and considering the binary number defined by these bits, e.g.:

$$h(k) = (Ak \bmod 2^b)/2^{n-b}.$$

This would give a particular set of n bits from Ak – namely bits $b-n, b-n+1, \dots, b-1$ – and you would use these bits to define a binary number in $0, 1, \dots, 2^n - 1$.

hashing strings (no, you don' t need to memorize this one either)

Suppose `s` is a string. Define its hash code:

$$h(\mathtt{s}) = \sum_{i=0}^{n-1} \mathtt{s}[i] p^{n-1-i}$$

where $s[i]$ is the 16-bit unicode value of the character at position i in the string, and p is some positive integer. To be concrete, take the case $p = 31$. Notice that $31 = 2^5 - 1$ and so $31^n < (2^5)^n = 2^{5n}$, and so it is at most a $5n + 16$ bit number.

```
                    ****************     s[3] x 31^0
                ************************     s[2] x 31^1
            ****************************     s[1] x 31^2
+       ********************************     s[0] x 31^3
        --------------------------------
      ************************************     hashCode( s )
```

You could then use this hashcode to define a hash function that maps strings to positions in a hashtable. Then you need to define a function (such as using the division method above) that maps arbitrary positive integers to values $0, 1, \dots, m-1$ namely the indices of your hashtable. *This is how Java defines hashcodes on strings. More on this next lecture.*