

Today we will look at a particular kind of map, sometimes called a *hash map*, which is very commonly used in computer science. Hash maps are great because they allow us to access values in $O(1)$ time. For large maps, this is faster than the $O(\log n)$ steps of binary search.

Hashing

Given a space of keys K , define a *hash function* to be a mapping:

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

where m is some positive integer. For each key $k \in K$, the hash function specifies some integer $h(k)$ between 0 and $m - 1$. Typically m is smaller than the number of keys in K . For example, if the keys are possible social insurance numbers (10^9 of them), but we are only concerned with a few hundred people and their social insurance numbers, then we might use as hash function where m is 1000, rather than 10^9 . Note that the hash function h is a mapping that is defined on the entire set K and not just on a subset of K . As such, it can happen (obviously) that two keys in K to map to the same integer.

It is very common to design hash functions by writing them as a composition of two maps. The first map takes keys K to a large set of integers. The second map takes the large set of integers to a small set of integers $\{0, 1, \dots, m - 1\}$. (The reasons for this will be clear by the end of the lecture.) The first mapping is called *hash coding* and the integer chosen for a key k is called the *hash code* for that key. The second mapping is called *compression*. Compression maps the hash codes to *hash values*, namely in $\{0, 1, \dots, m - 1\}$.

A typical compression function is the “mod” function. e.g. suppose i is the hash code for some key k (or maybe the keys K are already integers). Then, the hash value is $i \bmod m$. Often one takes m to be a prime number, though this is not necessary.

To summarize, we have that a hash function is typically composed of two functions:

$$\text{hash function } h : \text{compression} \circ \text{hash code}$$

where

$$\begin{aligned} \text{hash code} & : \text{keys} \rightarrow \text{integers} \\ \text{compression} & : \text{integers} \rightarrow \{0, \dots, m - 1\} \end{aligned}$$

and so

$$h : \{\text{keys}\} \rightarrow \{\text{hash values}\},$$

i.e. the set of hash values is $\{0, 1, \dots, m - 1\}$.

Hash map

Let’s return to the problem we discussed last class in which we have a keys K and values V and we wish to represent a map M which contains some set of ordered pairs $\{(k, v)\}$. See definition of map from last lecture. We now wish to define a hash function that will let us compute this map.

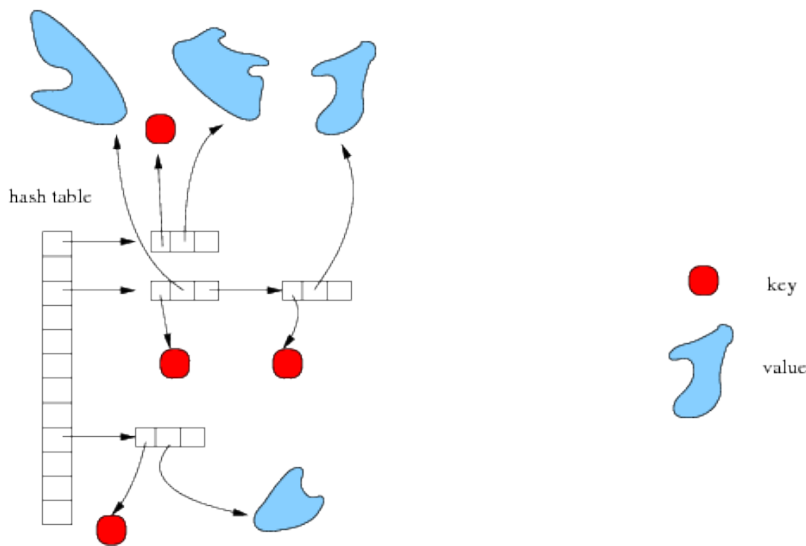
[ASIDE: The “values” $v \in V$ of the map M should not be confused with the “hash values” which are integers in $0, 1, \dots, m - 1$. The values $v \in V$ might be **Employee** records, or entries in an telephone book, for example, whereas the hash values are in $\{0, 1, \dots, m - 1\}$.

Define an array called a *hash table*, which will hold the (k, v) pairs. This is like the direct mapping array we discussed last lecture, except that it has far fewer slots than the number of keys in our key space K , namely the number of slots in the array is typically just a bit bigger than the number of (k, v) pairs.

Although the number of slots m is (typically) bigger than the number of pairs in the map, it could still happen that there are two keys k_1 and k_2 that hash to the same hash value, that is, there is an index i in the array such that $h(k) = i$ for more than one key k in the map. When this happens, we say that a *collision* has occurred. This can happen either if two keys have the same hash code or if two keys have a different hash code but the compression function maps them to the same slot e.g. $7 \bmod 5 = 2$, and $22 \bmod 5 = 2$.

To allow for collisions, we can use linked list of pairs (k, v) at each slot of our hash table array. The entries of a hash table, which now can hold multiple (k, v) pairs, are called *buckets* or *slots*.¹ Storing a list of (k, v) pairs in each hash bucket is called *chaining*.

Note that we need to store the key k because when use a key k to try to access a value v , we need to know which of the v 's stored in a bucket corresponds to which k . When we use a (search) key k to find a value v , we match the search key k with the key in the bucket. For example, with social insurance numbers (keys) and employee records (values), there may be multiple employee records stored in each bucket and we need to be able to check which of these corresponds to the given social insurance number.



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 undesirable since the whole point here is to represent a map so that we can access values as quickly as possible. To avoid having such long lists, we 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”.) *If we assume that we are free to choose whatever hash function we want, we can guarantee that the linked lists have a worst case constant.* In this sense, we say that *hash tables give $O(1)$ access.*

¹I prefer to use the word “slot” for the position in the array and “bucket” for the position in the array together with the entries i.e. list. Other people use the two words interchangeably. Its not a big deal.

More terminology..

Is it possible to design hash functions that avoid collisions entirely? For this to be possible in any given situation, it is necessary (but not sufficient) that the number of buckets of the hash table be greater than or equal to the number of key-value pairs in the map. Hash functions that are designed to avoid collisions are called *perfect hash functions*.

Another term that comes up often is the *load factor* of a hash table. This is the ratio of the number of (k, v) pairs currently in the table to the number of slots in the table (m). In Java, for example, the `HashMap` and `HashSet` classes implement hash functions, and the load factor for the hash table is never more than $0.75m$. If you try to add a new entry to a hash table that would make the load factor go above 0.75, then the `put` method would call a (private) method that generates a new hash table with a larger number of slots.

hash codes for strings [Modified April 22]

Suppose s is a string, composed out of unicode characters (16 bits each). A hash code for strings can be defined by:

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

where x is some positive integer. In the last lecture, I gave you similar formulas for the case $x = 2^{16}$ with $x = 2^8$.

For Java strings, the hash code is defined by setting $x = 31$. Check it out, [http://java.sun.com/javase/6/docs/api/java/lang/String.html#hashCode\(\)](http://java.sun.com/javase/6/docs/api/java/lang/String.html#hashCode())

Another alternative is to let $x = 1$, in which case the above formula reduces to

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

where $s[i]$ is still the 16-bit unicode value of the character at position i in the string. The hash code will be an integer at most $s.length * 2^{16}$. In this above example, two strings consisting of the same set of characters, such as “eat” and “ate” will have the same hash code. (Similarly, “hello” and “lleoh” would have the same hash code.)

Hash maps in Java

The Java API has a `Map<K, V>` interface. (Check it out, and note that only Java 1.6 or later has the generic version.) If you use any map that implements this interface, then:

- The key class `K` cannot be a primitive type. So if you want to use integers or characters, then for `K` you need to use the Java classes `Integer` or `Character`, respectively.
- You don't need to declare a class of ordered pairs entries. This is done for you automatically. The `Map` interface expects the implementing class to have a *nested class* which itself implements another interface namely `Map.Entry<K, V>`.