Recall the scenario from last lecture: suppose we have a map, that is, a set of ordered pairs $\{(k, v)\}$. We want a data structure such that, given a key k, we can quickly access the associated value v. If the keys were integers in a small range, say $0, 1, \ldots, m-1$, then we could just use an array of size m and the keys could indices into the array. The locations k in the array would correspond to the (integer) keys in the map and the slots in the array would hold references to the corresponding values v.

In most situations, the keys are not integers in some small range, but rather they are in a large range, or the keys are not integers at all – they may be strings, or something else. In the more general case, we define a function – called a *hash function* – that maps the keys to the range $0, 1, \ldots, m-1$. Then, we put the two maps together: the hash function maps from keys to a small range of integer values, and from this small range of integer values to the corresponding values v. Let's now say more about how we make hash functions.

Hash function: hash code followed by compression

Given a space of keys K, a *hash function* is a mapping:

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

where m is some positive integer. That is, for each key $k \in K$, the hash function specifies some integer $h(k)$ between 0 and $m-1$. The $h(k)$ values in $0, \ldots, m-1$ are called *hash values*.

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 $\{0, 1, \ldots, m-1\}$. The first map is called a *hash code*, 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 $\{0, 1, \ldots, m-1\}$.

A typical compression function is the "mod" function. For example, suppose i is the hash code for some key k. Then, the hash value is i mod m. (Often one takes m to be a prime number, though this is not necessary.) The mod function can be defined for negative numbers, such it returns a value in 0 to $m-1$. However, the Java mod function doesn't work like that. e.g. In Java, the expression "-4 mod 3" evaluates to -1 (rather than 2, which is what one would expect after learning about "modulo arithmetic"). To define a compression function using mod in Java, we need to be a bit more careful since we want the result to be in $\{0, 1, \ldots, m-1\}$.

The Java `hashCode()` method returns an `int`, and `int` values can be either positive or negative, specifically, they are in $\{-2^{31}, \ldots -1, 0, 1, \ldots 2^{31}-1\}$. (You will learn how this works in COMP 273.) The compression function that is used in Java is

$$| \texttt{hashCode}() \mid \text{ mod } m,$$

i.e. gives a number in $\{0, 1, \ldots, m-1\}$, as desired.

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

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

where $\circ$ denotes the composition of two functions, and

$$\text{hash code} \quad : \quad \text{keys K} \to \text{integers}$$

$$\text{compression} \quad : \quad \text{integers} \rightarrow \{0, .., \text{m} - 1\}$$

and so

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

i.e. the set of hash values is $\{0, 1, \ldots, m-1\}$. Note that a hash function is itself a map!

It can happen that two keys k_1 and k_2 have the same hash value. This is called a *collision*. There are two ways a collision can happen. First, two keys might have the same hash code. Second, two keys might have different hash codes, but these two different hash codes map (compress) to the same hash value. In either case we say that a *collision* has occured. We will deal with collisions next.

Hash map (or Hash table)

Let's return to our original problem, in which we have a set of keys of type K and values of type V and we wish to represent a map M which is set of ordered pairs $\{(k, v)\}$, namely some subset of all possible ordered pairs `K` $\times$ `V`.

[ASIDE: Keep in mind there are a few different maps being used here, and in particular the word "value" is being used in two different ways. The values $v \in$ `V` of the map we are ultimately trying to represent are not the same thing as the "hash values" $h(k)$ which are integers in $0, 1, \ldots, m-1$. Values $v \in$ `V` might be `Employee` records, or entries in an telephone book, for example, whereas hash values are indices in $\{0, 1, \ldots, m-1\}$.]

To represent the (k, v) pairs in our map, we use an array. The number of slots m in the array is typically bigger than the number of (k, v) pairs in the map.

As we discussed above, we say that a collision occurs when two keys map to the same hash value. Since the hash values are indices in the array, a collision leads to two keys mapping to the same slot in the array. For example, if $m = 5$ then hash codes 34327 and 83322 produce a collision since 34327 mod 5 $=$ 2, and 83322 mod 5 $=$ 2, and in particular both map to index 2 in the array.

To allow for collisions, we use a linked list of pairs (k, v) at each slot of our hash table array. These linked lists are called *buckets*.[1] Note that we need to store a linked list of pairs (k, v), not just values v. The reason is that 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. We use the hash function to map the key k to a location (bucket) in the hash table. We then try to find the corresponding value v in the list. We examine each entry (k, v) and check if the search key equals the key in that entry.

Good vs. bad hash functions

Linked lists are used to deal with collisions. We don't want collisions to happen, but they do happen sometimes. A good hash function will distribute the key/value pairs over the buckets such that, if possible, there is at most one pair per bucket. This is only possible if the number of entries in the hash table is no greater than the number of buckets. We define the *load factor* of a hash table to be the number of entries (k, v) in the table divided by the number of slots in the table (m). A load factor that is slightly less than one is recommended for good performance.

[1] Storing a linked list of (k, v) pairs in each hash bucket is called *chaining*.

Having a load factor less than one does not guarentee good performance, however. In the worst case that all they keys in the collection hash to the same bucket, then we have one linked list only and access is $O(n)$ where n is the number of entries. This is undesirable obviously. To avoid having such long lists, we want to choose a hash function so that there are few, if any, collisions.

The word *hash* means to "chop/mix up".[2] We are free to choose whatever hash function we want and we are free to choose the size m of the array. So, it isn't difficult to choose a hash function that performs well in practice, that is, a hash function that keeps the list lengths short. In this sense, one typically regards hash tables as giving $O(1)$ access. To prove that the performance of hash tables really is this good, one needs to do some math that is beyond COMP 250.

As an example of a good versus bad hash function, consider McGill student ID's which are 9 digits. Many start with the digits 260. If we were to have a hash table of size say $m = 100,000$, then it would be bad to use the first five digits as the hash function since most students IDs would map to one of those two buckets. Instead, using the last five digits of the ID would be better.

Java `HashMap` and `HashSet`

In Java, the `HashMap<K,V>` class implements the hash map that we have been describing. The `hashCode()` method for the key class `K` is composed with the "mod m" (and absolute value) compression function where m is the capacity of an underlying array, and it is an array of linked lists. The linked lists hold `(K,V)` pairs. Have a look at the Java API to see some of the methods and their signatures: `put` , `get`, `remove`, `size`, `containsKey`, `containsValue`, and think of how these might be implemented.

In Java, the default maximum load factor for the hash table is than 0.75 and there is a default array capacity as well. The `HashMap` constructor allows you specify the initial capacity of the array, and you can also specify the maximum load factor. If you try to add a new entry – using the `put()` method – to a hash table that would make the load factor go above 0.75 (or the value you specify), then a new hash table is generated, namely there is larger number m of slots and the (key,value) pairs are remapped to the new underlying hash table. This happens "under the hood", similar to what happens with `ArrayList` when the underlying array fills up and so the elements needs to be remapped to a larger underlying array.

Java also has a `HashSet<T>` class. This is similar to a HashMap except that it only holds objects of type `<T>`, not key/value pairs. It uses the `hashCode()` method of the type `T`. Why would this class be useful? Sometimes you want to keep track of a set of elements and you just want to ask questions such as, "does some element e belong to my set, or not?" You can add elements to a set, remove elements from a set, compute intersections of sets or unions of sets." What is nice about the `HashSet` class is that it give you quick access to elements. Unlike a list, which requires $\Theta(n)$ operations to check if an element is present in the worst case, a hash set allows you to check in time $O(1)$ – assuming a good hash function.

[2] It should not be confused with the # symbol which is often the "hash" symbol i.e. hash tag on Twitter, or with other meanings of the word hash that you might have in mind.

Cryptographic hashing

Hash functions comes up in many problems in computer science, not just in hash table data structures. Another common use is in password authentication. For example, when you log in to a website, you typically provide a username or password. On a banking site, you might provide your ATM bank card number or your credit card number, again along with a password. What happens when you do so?

You might think it works like this. The web server has a file that lists all the user names and corresponding passwords (or perhaps a hash map of key-value pairs, namely usernames and passwords, respectively). When you log in, the web server takes your user name, goes to the file, retrieves the password stored there for that username, and compares it with the password that you entered. This is *not* how it works however. The reason is that this method would not be secure. If someone (an employee, or an external hacker) were to break in an steal the file then he would have access to all the passwords and would be able to access everyone's data.

Instead, the way it typically works is that the webserver stores the user name and the *hashed* passwords. Then, when a user logs in, the web server takes the password that the user enters, hashes it (that is, applies the hash function), throws away the password entered by the user, and compares the hashed password to the hashed password that is stored in the file.

Why is then any better? First, notice that if a hacker could access the hashed password, this itself would not be good enough to log in, since the process of logging in requires entering a password, not a hashed password. Second and more interesting, you might wonder if it is possible to compute an *inverse hash map*: given a hashed password, can we compute the original password or perhaps some other password that is hashmapped to the given hashed password. If such a computation were feasible, then this inverse hashmap could be used again to hack in to a user's account.

However, "cryptographic" hashing functions are designed so that such a computation is *not* feasible. These hashing functions have a mixing property that two strings that differ only slightly will map to completely different hash values, and given a hash value, one can say almost nothing about a keyword that could produce that hash function.

A few final observations. First, a cryptographic hashing function does not need to be secret. Indeed there are standard cryptographic hashing functions. One example is MD5 `http://www.miraclesalad.com/webtools/md5.php`. This maps any string to a sequence of 128 bits.[3]

Second, cryptographic hashing is not the same as encryption. With the latter, one tries to encode a string so that it is not possible for someone who is not allowed to know the string cannot decrypt the coded string and have the original one again. However, it *should* be possible for someone who *is* allowed to know the original string to be able to "decrypt" the encrypted string to get the original message back. So, the main dfference is that with encryption it *is* possible to invert the "hash" function. (It isn't called hashing, when one is doing encryption, but the idea is similar.) You will learn about encryption/decryption if you take MATH 240 Discrete Structures.

[3] If you check out that link, you'll see that the hash values are represented not as 128 bits there, but rather as 32 *hexadecimal* digits. I haven't discussed hexadecimal in the course, but the idea is simple: it is just a code for 4-bit strings. The digits 0 to 9 are used for 0000 to 1001 in the obvious way, namely binary coding. The letters 'a' to 'f' are used for the other five combinations of bits, which correspond to binary codes for numbers 10 to 15).

