Today we will look at a technique called hashing which is very commonly used in computer science. We will concentrate on hash maps. At the end of the lecture, I will mention a specific case of this, which in Java are known as hash sets. (In the lecture itself, I did not get to hash sets. I have included that material here and on the slide, in case you are interested.)

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 access the associated value \( v \). The main idea is as follows. If the keys are integers in a small range, say \( 0, 1, \ldots, m - 1 \), then we can just use an array of size \( m \) and the keys simply would index the array. The locations \( k \) in the array that are keys in the map would hold references to their corresponding values \( v \). This was the “direct addressing” or “direct mapping” approach I mentioned last lecture. (See the slides for an illustration.) If the keys are more general, which is usually the case, then we will define another function – called a hash function – which maps the keys to the range \( 0, 1, \ldots, m - 1 \). Then, we put the two maps together. This gives us a mapping from the keys \( k \) to their corresponding values \( v \). The result of this lecture will elaborate on this idea.

**Hash function: hash code, compression**

Given a space of keys \( K \), define a hash function to be a mapping:

\[
    h : K \rightarrow \{0, 1, 2, m - 1\}
\]

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

Typically \( m \) is smaller than the number of keys in \( K \). For example, if the key space is all possible social insurance numbers (\( 10^9 \) of them), but our map only has a few hundred keys, 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 typically happens that a huge number of keys in the key space \( K \) will map to the same hash value. This is not a problem since the keys in a given map will typically be a small subset of the space of possible keys. What’s important is that, for the keys in the map, we would like it to be rare that two keys 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 \( \{0, 1, \ldots, m - 1\} \). 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 \( \{0, 1, \ldots, m - 1\} \).

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.

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 \( \circ \) is the composition of two functions i.e. \( f \circ g(x) \equiv f(g(x)) \), and

\[
    \text{hash code : keys } \rightarrow \text{ integers}
\]
compression : integers \rightarrow \{0, \ldots, m - 1\}

and so

\[ h : \{\text{keys}\} \rightarrow \{\text{hash values}\}, \]

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

Also note it can happen that two keys \(k_1\) and \(k_2\) map to the same hash value. Indeed this can happen in two ways. First, two keys might have the same hash code. Second, two keys might have different hash codes, but these two different hash codes map to the same hash value. In either case we say that a collision has occurred. We will discuss collisions below.

**Example: hash codes for strings**

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

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

where \(x\) is some positive integer. In the last lecture, we looked at the case \(x = 2^{16}\) with \(x = 2^8\).

For Java strings, the hash code is defined by setting \(x = 31\). That is, String.hashCode() method computes the above sum. A few questions about this came up during in the lecture:

- **Q:** Does using \(x = 31\) guarantee that two different strings will return different hash codes?
  
  **A:** No, it doesn’t. To see why not, consider strings of length 2. If we are using 16 bit chars (unicode, as in Java), then there are \(2^{16} \times 2^{16} = 2^{32}\) possible strings of length 2. However, the number of hash codes defined by all strings of length 2 is much less than this. Any such hash code is of the form \(s[0] \cdot 31 + s[1]\), where \(s[0]\) and \(s[1]\) are 16 bit numbers. Since \(31 < 2^5\), we can observe that any hash code \(s[0] \cdot 31 + s[1]\) must be a number less than \(2^6 \cdot 2^{16} = 2^{22}\) which is much less than \(2^{32}\). But we cannot uniquely encode \(2^{32}\) things (possible strings of 2 unicode characters) with with \(16 + 6 = 22\) bits. It has to be the case that many different strings map to each such hash code.

- **Q:** What is the return type of the Java hashCode() method?
  
  **A:** If you look it up, you’ll find that the return type is int. How can we reconcile this with the above polynomial, which obviously can compute numbers that are very large and cannot be represented with only 32 bits, i.e. the int type. The answer is that String.hashCode() uses the above sum \(\text{mod} \ 2^{32}\), i.e. it uses the last 32 bits of the number defined above as the return value of hashCode(). This is implicitly stated in the Java API where it says that this method computes the above polynomial “using int arithmetic”.

An alternative hash code for strings is to use the above polynomial but let \(x = 1\), in which case the above formula reduces to

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

In this example, two strings consisting of the same set of characters, such as “eat” and “ate” or “hello” and “lleoh” would have the same hash code. A hash function based on this hash code would thus produce a collision.
Hash map: hash function, collisions

Let’s return to our original problem, in which we have spaces of keys $K$ and values $V$ 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: Note that the “values” $v \in V$ of a map $M$ 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 $\{0, 1, \ldots, m - 1\}$.

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. Here, the number of slots in the array is $m$. This number is typically a bit bigger than the number of $(k, v)$ pairs.

As we discussed above, we say that a collision occurs when two keys map to the same hash value. For example, if $m = 5$ then hash codes 7 and 22 produce a collision since $7 \mod 5 = 2$, and $22 \mod 5 = 2$.

To allow for collisions, we can use linked list of pairs $(k, v)$ at each slot of our hash table array. These linked lists are sometimes called buckets. Storing a linked list of $(k, v)$ pairs in each hash bucket is called chaining.

Note that we need to store both the key $k$ and the value $v$, 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 $k$. We use the hash function to map the key $k$ to a location/slot/bucket in the hash table. We then try to find the corresponding value $v$ in the bucket/list. The bucket contains a linked list of $(k, v)$ pairs. We examine each pair and try to match the search key $k$ with the key in each pair. 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 the keys of each one to see which, if any, corresponds to the given key (social insurance number).

---

1I mention these terms mainly in case you want to look them up and you need an index term.
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 and access is $O(n)$ where $n$ is the number of (key, value) pairs. 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 are few, if any, collisions. If we are free to choose whatever hash function we want and we are free to choose the size $m$ of the array, we can guarantee that the linked lists have a worst case constant. In this sense, we say that hash tables give $O(1)$ access.

Is it possible to design hash functions that avoid collisions entirely? (Such hash functions are called perfect hash functions.) For this to be possible in any given situation, it is necessary that the number of buckets of the hash table be greater than or equal to the number of key-value pairs in the map. We define the load factor of a hash table to be the ratio of the number of $(k, v)$ pairs currently in the table to the number of slots in the table ($m$). So a perfect hash function will need a load factor that is at most 1.

Java HashMap

In Java, the `HashMap<K,V>` class implements the sort of hash map that we have been describing. The `hashCode()` method for the key class $K$ is composed with the “mod $m$” compression function where $m$ is the capacity of an underlying 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 load factor for the hash table is never more than 0.75. 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, 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 (recall lecture 6).

`hashCode()` and `equals()`

For any class, the `hashCode()` and `equals()` method should be related as follows: if $k1.equals(k2)$ is true, then $k1.hashCode() == k2.hashCode()$ should be true. I did not give a good example of this in class, so let me do so here. Suppose our keys are user names in some database and we want the user names to not be case sensitive i.e. we want to Chang123 and Chang123 and CHANG123, etc to be “equal”. If these keys had different hash codes, then they might map to different buckets. So if we put the pair (Chang123, value) into the map and then we tried to get using the key CHANG123, we would fail to find the key and hence fail to find the value. Obviously this is not the behavior we want.

What about the converse of the above rule? If $k1.hashCode() == k2.hashCode()$ is true, then should we require that $k1.equals(k2)$ returns true? No, we shouldn’t require this. For example, two different strings which we don’t necessarily want to consider equal could have the same hash code, as we discussed earlier. In this case, we would get a collision. We would prefer not to have

---

2The word hash means to “chop/mix up”. It should not be confused with the # symbol which is often the “hash” symbol i.e. hash tag on Twitter.
collisions, since it means we have to search through a linked list. But allow for collisions as the price to pay for generally quick access.

Bottom line: if you write a class which uses the `hashCode()` method and you want to override either the `Object` class’s `hashCode()` or `equals()` methods, then you should ensure this: if `k1.equals(k2)` returns true or false, then `k1.hashCode() == k2.hashCode()` should return true or false, respectively. This may require that you override both of them.

**Java HashSet<K> class**

The lecture ended here, and I did not have time to discuss a second interesting Java class that uses hashing. I am including a brief discussion below, and I strongly encourage you to read it. It is *not* on the final exam, but it is interesting and it will help you understand better how hashing relates to previous data structures we have seen in the course.

`HashSet<T>` is a class that stores a set of objects of some generic type `T`. Think of it as an alternative to `LinkedList<T>` or `ArrayList<T>`, for certain situations. `HashSet<T>` uses a hash table, and as its hash function it uses the `hashCode()` method for type `T`, together with the “mod m” compression function where again `m` is the capacity of the underlying array. Each bucket in the array holds a list of objects of types `T`. Think of a `HashSet<T>` as a `HashMap<T,V>` without the values `V`!

`HashSet` does not implement the `List` interface, but rather it implements the `Set` interface. (Check it out.) For example, suppose you simply want to keep track of a set of `Strings`. Suppose you don’t need the strings to be ordered, i.e. you won’t be asking for the i-th string in a lexicographic order (as you might for a sorted list), and you won’t be asking for the string that was added last (as you would for a stack), and you won’t be asking for the string that was added first (as you would for a queue). Suppose you simply want to keep track of a set of strings, and be able to ask questions like “is string x a member of this set?” or perform operations like adding a string to the set or deleting a string from the set. In this case, using a hash table would work great since it would allow you to answer these questions in time $O(1)$. The cost of doing so, of course, is that you are limited in the operations you can do.