Hash Functions

Step 1

We started our discussion of **Hashing** with the idea that, given some key *k*, we could come up with a function that could output an integer that "represents" *k*. Formally, we call this function a **hash function**, which we will denote as *h*(*k*) throughout this section.

Hopefully, our use of the word "represents" upset you a bit: what does the word "represent" actually mean mathematically? Formally, a **hash function** can be described by the following two properties, the first of which being a *requirement* of *all* valid **hash functions** and the second being a feature that would be "nice to have" if at all possible, but not necessary:

1. **Property of Equality:** Given two keys *k* and *l*, if *k* and *l* are equal, *h*(*k*) **must equal** *h*(*l*). In other words, if two keys are equal, they **must** have the same hash value
2. **Property of Inequality:** Given two keys *k* and *l*, if *k* and *l* are *not* equal, it would be nice if *h*(*k*) was not equal to *h*(*l*). In other words, if two keys are *not* equal, it would be nice (but not necessary!) for them to have different hash values

A **hash function** is formally called *perfect* (i.e., a **perfect hash function**) if it is guaranteed to return different hash values for different keys (in other words, if it is guaranteed to have *both* the **Property of Equality** *and* the **Property of Inequality**).

For our purposes with regard to Data Structures, a "good" **hash function** means that different keys will map to different indices in our array, making our lives easier. Formally, we say that a "good" **hash function** minimizes *collisions* in a **Hash Table**, even if collisions are still theoretically possible, which simply translates to faster performance. The mechanism by which collisions worsen the performance of a **Hash Table**, as well as how we deal with them, will be discussed extensively later in this chapter, but for now, trust us.

Step 2

In this text, we are introducing **hash functions** with the intent of using them to implement a **Hash Table**, but there are many real-world applications in which we use "good" **hash functions** outside the realm of Data Structures! One very practical example is checking the validity of a large file after you download it. Let's say you want to install a new Operating System on your computer, where the installation file for an Operating System can sometimes reach a few gigabytes in size, or let's say you want to stream some movie (legally, of course), or perhaps you want to download a video game to play (again, legally, of course). In all of these examples (which are quite different at a glance), the computational task is the same: there is a large file hosted on some remote server, and you want to download it to your computer. However, because network transfer protocols don't have many provisions for ensuring data integrity, packets can become corrupted, so larger files are more likely to become corrupted at some point in their transfer. As a result, you want some way to check your large file for validity before you try to use it.

Ideally, you might want to do a basic Unix command (e.g. `diff downloadedFile originalFile`) to check your downloaded file against the original for equality, but you can't check for equality unless you actually had the original file locally, which you don't! That was the whole reason why you were downloading it in the first place! Instead, many people who host these large files will *also* **hash** the file and post the hash value, which is just an integer (typically represented as a hexadecimal number) that you can easily copy (or just read) without any worries of corruption. You can then download the large file, compute a hash value from it, and just visually compare your file's hash value against the original one listed online:

- If your hash value is *different*, the file you downloaded does not match the original (because of the **Property of Equality**, which is a **must** for a **hash function** to be valid), so something must have gotten corrupted during the download process
- If your hash value *matches* the one online, even though there is *technically* still a chance that your file is different than the original (because the **Property of Inequality** is simply a "nice feature," but not a requirement), the **hash functions** used by these **hashing** tools are very good, so you can assume that the chance that your file being corrupted is negligible

If you are curious to learn more about using **hashing** to check a file for validity, be sure to check out the main **hash functions** used for this task today: MD5, SHA-1, and CRC (where CRC32 is typically the most popular out of the types of CRC).

Step 3

Let's imagine we are designing a **hash function** for keys that are ASCII characters (e.g. the `unsigned char` primitive data type of C++). Below is a C++ implementation of a **hash function** that is *valid*, because keys that are of equal value will have the same hash value, and *good* (*perfect*, actually), because keys of unequal value are guaranteed to have different hash values:

```
unsigned int hashValue(unsigned char key) {
    return (unsigned int)key; // cast key to unsigned int
}
```

Because each of the C++ primitive data types that are less than or equal to 4 bytes in size (`bool`, `char`, `unsigned char`, `int`, `unsigned int`, etc.) are stored in memory in a way that can be interpreted as an `unsigned int`, simply casting them as an `unsigned int` would result in a perfect **hashing**. Also, because primitive data types in most, if not all, languages can be cast to an integer in O(1) time, this perfect **hash function** has a worst-case time complexity of **O(1)**.

Let's imagine now that we are designing a **hash function** for keys that are strings. Below is a C++ implementation of a **hash function** that is *valid*, because keys that are of equal value will have the same hash value and keys with different values will *tend* to have different hash values:

```
unsigned int hashValue(string key) {
    unsigned int val = 0;
    for(int i = 0; i < key.length(); i++) {
        val += (unsigned int)(key[i]); // cast each character of key to unsigned int
    }
    return val;                        // return the sum over all characters in key
}
```

This hash function is pretty good because keys with different lengths or characters will probably return different hash values. If we didn't take into account all of the characters of the string, we would have more collisions than we would want. For example, if we wrote some **hash function** that only looks at the first character of a string, even though the **hash function** would be O(1), it would always return the same hash value for strings that had the same first character (e.g. "**H**ello" and "**H**ashing" have the same first character, so a **hash function** that only checks the first character of a string *must* return the same hash value for these two very different strings). As a result, for a string (or list, or collection, etc.) of length *k*, a good **hash function** must iterate over **all *k* characters**.

However, in the function above, even though we iterate over all *k* characters of a given string, we can still have a lot of collisions. For example, using the **hash function** above, because we simply add up the ASCII values of each character, the strings "Hello" and "eHlol" will have the same hash value because, even though their letters are jumbled, they contain the same exact letters, so the overall sum will be the same. Formally, if the arithmetic of our **hash function** is commutative, the order of the letters will not affect the results of the math, potentially resulting in unnecessary collisions.

Therefore, a good **hash function** for strings of length *k* (or generally, lists/containers/etc. with *k* elements) must have a time complexity of **O(*k*)** and must perform arithmetic that is **non-commutative**. If you are interested in learning about clever **hash functions** for strings, see this brief article by Ozan Yigit at York University.

Let's now imagine that we are designing a **hash function** for keys of some arbitrary datatype `Data` (it doesn't matter what `Data` is: the point we will make will be clear regardless of datatype). Below is a C++ implementation of a **hash function** that is *valid*, because keys that are of equal value will have the same hash value, but that is pretty terrible, because we will have a *lot* of collisions:

```
unsigned int hashValue(Data key) {
    return 0;
}
```

It should hopefully be obvious why this is such a bad **hash function**: no matter what *key* is, it will always have a hash value of 0. In other words, the hash value of *key* tells us absolutely nothing useful about *key* and can't really be used to do anything.

Let's again imagine that we are designing a **hash function** for keys of the same arbitrary datatype Data. Below is a C++ implementation of a **hash function** that is *not valid*, because keys that are of equal value will *not necessarily* have the same hash value:

```
unsigned int hashValue(Data key) {
    return (unsigned int)rand(); // return a random integer
}
```

Even if two keys are identical, if I compute their hash values separately, because my **hash function** simply generates a random number each time it's called, they will almost certainly have different hash values. Recall that, for a **hash function** to be *valid*, the **Property of Equality** *must* hold, but the fact that equal keys can have different hash values violates this property.

Step 4

EXERCISE BREAK: Which of the following **hash functions** are valid? (Select all that apply)

Note: `(unsigned int)time(NULL)` returns the current time.

To solve this problem please visit https://stepik.org/lesson/30914/step/4

Step 5

We've now discussed the properties of a **hash function** as well as what makes a given **hash function** good or bad. However, it is important to note that, for most things you will do in practice, excellent **hash functions** will have already been defined; you typically only have to worry about writing your own **hash function** if you design your own object.

Nevertheless, we can now assume that, given a datatype we wish to store, we have some (hopefully good) **hash function** that can return a hash value for an object of that type. We can now tie this back to our original idea of using hash values in order to determine indices to use to store elements in an array. Say I have an array of length *m* and I want to insert an object *key* into the array. I can now perform two **hashing** steps:

1. Call *h*(*key*), where *h* is the **hash function** for the type of object *key* is, and save the result as *hashValue*
2. Perform a second "**hashing**" by modding *hashValue* by *m* to get a valid index in the array (i.e., *index* = *hashValue* % *m*)

By performing these two steps, given an arbitrary object *key*, we are able to successfully compute a valid index in our array in which we can store *key*. Again, as we mentioned before, if our **hash function** is not *perfect* (i.e., if we sometimes hash *different* objects to the *same* hash value), or if two different hash values result in the same value when modded by *m*, we will run into *collisions*, which we will discuss extensively later in this chapter.

Step 6

EXERCISE BREAK: Say you have an array of length 5. Also, you have a primary **hash function** that simply returns the integer it was given (i.e., *h*(*k*) = *k*), and you have a secondary **hash function** that computes an index in your array by modding the result of the primary **hash function** by the length of the array (i.e., *index* = *H*(*k*) = *h*(*k*) % 5). For each of the following integers, compute the index of your array to which they **hash**.

For example, if you were given the number **8**, *h*(8) = 8, so ***index* = *H*(8) =** *h*(8) % 5 = 8 % 5 = **3**.

To solve this problem please visit https://stepik.org/lesson/30914/step/6

Step 7

Thus far, we have seen examples of **hash functions** that were good (or even *perfect*, like *h*(*k*) in the previous step). Nevertheless, picking an unfortunate size for our array or choosing a bad indexing **hash function** (like *H*(*k*) in the previous step) can still result in numerous collisions.

Also, we saw that, if a given object *key* has *k* elements (e.g. a string with *k* characters, or a list with *k* numbers), a good **hash function** for *key* will incorporate each of *key*'s *k* elements into *key*'s hash value, meaning a good **hash function** would be **O(*k*)**. Also, to avoid collisions that come about when comparing objects containing the same elements but in a different order, we need the arithmetic we perform to be **non-commutative**.

Thus far, we have discussed **hashing** and the motivation for **Hash Tables**, and we have defined what a **hash function** is as well as what makes one good or bad, but we still have yet to actually formally discuss **Hash Tables** themselves. In the next section, we will finally shed light on this seemingly mystical data structure that achieves O(1) average-case finds, insertions, and removals.