Chapter 12

Hashing

So far, we have explored a pair of implementations for our map ADT. Association lists gave us $\mathcal{O}(N)$ lookup, and tree maps gave us $\mathcal{O}(\log N)$ lookup. Can we do better than this? Recall that an integer map is a special case of a map where the keys are draw from a finite set of integers. In this situation, we can use an array to efficiently implement the map ADT where keys are indices and values are elements of the array.

In this reading, we'll breaking apart the assumption of the integer map to generalize the structure to handle a wider variety of domains. By doing so, we'll derive a data structure called a *hash map* which is a very efficient implementation of the map ADT. Along the way, we'll also develop a technique called *hashing* which is one of the most important techniques in computer science with applications towards fingerprinting, compression, and cryptography, among others.

12.1 Revisiting Integer Maps

The first assumption we made when using an integer map was that the space of keys was finite and known. Recall that our running example of an integer map is an inventory where we map product IDs (pids) to inventory sizes. For this example, we assumed that our pids were drawn from the range 0–999. What happens if we don't know what values the space of keys ranges over? For example, we may want to support a number of products beyond 1000.

This seems to pose a problem for our integer map strategy. By definition, an array can only hold a fixed number of elements. With a variable number of possible keys, it seems like we cannot use an array to hold them. This seems like a job for an array list which can hold a variable number of elements, but a deeper problem remains. Consider the following map where we only hold an entry for key value 10000 (and we use `null` as our value indicating that this key has no entry in our map):

```
[null, null, null, ..., v]
                        ^

                    index 10000
```

Because we map keys directly onto indices, our array list would need to contain entries for 0–9999 even though only one entry is in the map for key 10000. This is an extreme waste of space—the size of the backing array or array list must be as large as the largest key that we need to support. However, if the space of keys that the map actually needs to store is *sparse*, *i.e.*, very few keys are required from the range, then we end up wasting lots of space. Ideally, we'd like to avoid using a variable-size structure and instead

use a fixed-size structure so that our space usage is not proportional to the size of the key space that we might support (which is very large in practice).

How can we support a variable number of keys using a fixed-sized array? One trick we can try is using the modulo operator to fit the keys into the available space of the array. Suppose that we only allocate an array of size 10 to hold the entries of our map. Then we can simply mod our key by 10 to get an index that fits in the array, *e.g.*,

- The values $0, 10, 20, 30, \ldots$, all map to index 0.
- The values $1, 11, 21, 31, \ldots$, all map to index 1.
- ...
- The values $9, 19, 29, 39, \ldots$, all map to index 9.

If our map contains entries for keys 232, 11196, 555, and 8, then our backing array would look like this:

```
[null, null, v1, null, null, v2, v3, null, v4, null]
```

Where `v1` is the value associated with key 232, `v2` is the value associated with key 555, `v3` is the value associated with key 11196, and `v4` is the value associated with key 8. Accessing the entry for key `k` means that we simply look up index `k % 10` in the array.

Of course, the issue here is that if also want to store an entry for key 1992, we have a problem because there is an entry in index 2 already for key 232. This is known as a *collision*—two keys want to use the same position in the array. There are two primary methods of resolving collisions: *probing* and *chaining*.

12.1.1 Probing

When a key's *preferred index* has already been taken by another key, we can *probe* the array for another position (presumably empty) to store the key. Probing means searching the array in a systematic manner starting from a key's preferred index to find an open position to store its value. A simple probing strategy is to simply search successive elements in the array until we find an empty spot, a technique called *linear probing*.

To do this, we must augment what our arrays carry as values. Rather than carrying the values of our map directly, each array index will carry both a key and its corresponding value. This is because with our probing strategy, a cell may no longer correspond to a key's preferred index—some other key might have already taken that spot. By carrying the key, we can ensure that we found the appropriate value by comparing the key we found in the array to our target key.

In the example above, consider starting with an array of five elements. For simplicity's sake, we'll just note the key in our diagram, but keep in mind that we are storing an entire key-value pair in each array slot.

```
[null, null, k1, null, null ,k2 ,k3, null, k4, k5]
```

Suppose that we are adding the key `k6` that has a value of 2. The key `k1` is already in that slot, so we search to the right until we find an empty position and place `k6` there.

```
[null, null, k1, k6, null, k2, k3, null, k4, k5]
```

When we go to look up `k6` in this map, we start at index 2 and scan to the right until we find the entry for `k6`. Now, what happens if we add a key `k7` that has id 8? Index 8 in the array is already taken by `k4`, so we search index 9. However that is taken by `k5`. We proceed by *wrapping around* the array and starting our first at the index 0. This index is empty, so we place `k7` there:

```
[k7, null, k1, k6, null, k2, k3, null, k4, k5]
```

In general, during `put` and `lookup`, our linear search *wraps around* the array, terminating when we reach the preferred index of the key in question.

Using a probing scheme, we achieve $\mathcal{O}(1)$ complexity for our fundamental map operations—`put` and `lookup`—as long as there are no collisions with our keys. However, if there are collisions, we'll need to search part of the array to find our key. In the worst case, all keys have the same preferred index, so we'll need to perform a linear scan of the array which takes $\mathcal{O}(n)$ time. Note that discussing the "average case" here is very difficult because it depends on the likelihood of collisions with our keys which is dependent entirely on the specific keys we add into the map.

12.1.2 Chaining

With probing, we maximize the space in our backing array by checking successive indices for open positions. When our map is *sparse*, this is fine because there's plenty of empty spaces between elements for collisions to be placed. However, if the map is *dense*, *i.e.*, contains many elements, then probing will need to traverse significant portions of the array to find the key of interest. Rather than doing this, we can elect to store all of the keys of a particular id at their associated index. To do this, we store a *list* of key-value pairs at each index of the array rather than a single such pair. For example, we may have the following map:

```
[ ][ ][ ][ ][ ]
 |  |  |  |  |
 |  |  |  |  |--> [k7]
 |  |  |  |--> [k5, k6]
 |  |  --> []
 |  --> [k4]
 -->[k1, k2, k3]
```

Keys `k1`, `k2`, and `k3` all have id 0, so they map onto index 0 in the array. Index 0 contains a list that holds these three key-value pairs. In contrast, there are no stored keys with id 2, so index 2 holds an empty list.

This method of collision resolution is called *chaining*, named as such because these lists look like chains hung off of each array index. Alternatively, we can call each index a *bucket* and the effect of chaining is to store all the key-value pairs of a certain id in their corresponding bucket. In essence, these lists function like the association lists we studied earlier except that all the keys contained in a list have the same id.

By doing this, we no longer have to perform a linear scan of the array. Instead, we perform a linear scan of a bucket's association list. A bucket fills up only with colliding keys, so the amount of keys that we need to traverse is proportional to the number of keys that collide for a particular bucket. In the best case, a bucket contains exactly one key-value pair corresponding to the desired key (in the absence of collisions) for $\mathcal{O}(1)$ lookup. In the worst case, everything has the same key, so a single bucket contains all the entries in the map. In this situation, lookup takes $\mathcal{O}(n)$ time. Like probing the average case is dependent on the nature of the collisions of the keys which depends on the specific keys stored in the map.

At first glance, it seems like we do not have resizing issues with chaining like we do with probing. With lists holding each bucket, we never run out of room for the keys. However, because our arrays have finite size, if we store many more keys that we have buckets, the pigeonhole principle tells us we will have collisions. In other words, eventually our map will look as follows:

```
[ ][ ][ ][ ][ ]
 |  |  |  |  |--> [ ... ]
 |  |  |  |--> [ ... ]
 |  |  |--> [ ... ]
 |  |--> [ ... ]
 |-->[ ... ]
```

where each of the buckets contain many keys. In this situation, it is prudent to *resize* the array, creating more buckets and subsequently more opportunities to spread out keys among the different buckets.

Like probing, we simply cannot create an array of double the size and copy over the elements blindly—we will not respect the new preferred indices of the keys if we do this. Instead, we must create an array with double the size and then *rehash* the key-value pairs back into the newly created array.

When we choose to resize the array with chaining is different than probing. With probing, we can simply resize the array when it is full. With chaining, we never truly "run out of space"; instead, we must choose an appropriate *load factor* that consists of the ratio of the size of the number of entries n to the number of buckets k: n/k. A load factor of 0 indicates that the map is empty. In contrast, a load factor of 1 indicates the map is "full" in the sense that adding any more elements guarantees a collision. With a low load factor, the backing array is sparsely populated meaning that we are likely wasting space with empty array indices without positively affecting our lookup times. A high load factor means that there will likely be many collisions pushing our lookup time to $\mathcal{O}(N)$.

12.1.3 Removal

One final operation that we have yet to discuss is *removal* of key-value pairs. In the case of chaining, removal amounts to removing a key-value pair from an association list which is easy to do. Like an array list, we can choose to leave the backing array sparsely populated or *contract* it to increase its load factor, saving space in the process.

With probing, removal becomes a more nuanced affair. Consider the following array:

```
[k1, k2, k3, k4]
```

Where k2 and k3 both have id 1, and suppose we now remove k2. Because an array must contain a value in each of its indices, we can elect to null out index 1.

```
[k1, null, k3, k4]
```

However, what happens when we lookup `k3` now? We'll start at its preferred index, 1, and note that no element is there. We could pass over index 1 and find k3 at index 2. But now consider this scenario where the map is empty:

```
[null, null, null, null]
```

Here, when we look up at index 1, we note that no element is there. Rather than skipping over to the next element, we want to terminate the search *right now* so we don't spend $\mathcal{O}(n)$ time trying to find an element in this empty map.

Thus, we need to differentiate between an *empty index due to a deletion* and an *empty index due to no key-value pairs being added there yet.* In the former case, we want to move onto the next index while performing lookup; in the latter case, we do not. To obtain this behavior in Java, we may use the Java 8 `Optional<T>` class. Instances of `Optional<T>` either contain a value of type T or nothing. We can use this class as follows:

- A `null` value in the array means there is no corresponding entry in the map.
- An empty `Optional<T>` value corresponds to a deleted value.
- A non-empty `Optional<T>` value corresponds to an actual entry in the map.

12.2 Transforming Values to Integers

Finally, we need to lift the restriction on our integer maps that our keys must be values. To do this, we develop the fundamental idea of *hashing*, transforming an arbitrary value into an integer. The resulting data structure that we obtain, the *hash map*, gives us the constant-time map operations that were seeking over any data type for which we can write a *hash function.*

12.2.1 The Hash Function

A hash function is simply a function $h : T \longrightarrow \texttt{int}$ that converts a value of type T into an integer with some restrictions. To see what these are, let's consider writing a hash function for type char (so $T = \texttt{char}$). Here is a simple example of a hash function:

$$h(c) = 0$$

which transforms every character c into the integer 0. This is not a very good example of a hash function because it maps every character into the same index! For example, if we used a chaining scheme and our keys were characters, using this hash function to convert characters into indices nets us the following structure:

```
[ ][ ][ ][ ][ ]
 |
 |--> [c1, c2, c3, ...]
```

Where all the keys sit in the zero bucket! We say that this hash function does not *distribute* its keys among the space of possible indices effectively.

On the other hand, the following hash function:

$$h(c) = \text{rand}(0, 100)$$

effectively distributes its keys—it indeed chooses a random index in the range 0 through 100 on each invocation. However, the resulting hash value is not *consistent* meaning that two separate calls to $h(c)$ will likely result in two different indices. This makes lookup with such a hash function impossible!

In summary, we need a hash function that:

1. Distributes its keys as evenly as possible among the space of possible integers.
2. Consistently assigns keys so that if $k1 = k2$ then $h(k1) = h(k2)$.

Note that the second requirement says that equal keys produce equal *hash values*. The converse is not necessarily true! A valid hash function may have $h(k1) = h(k2)$ but $h1 \neq h2$. This is precisely a *key collision* that we discussed previously and have methods for resolving after-the-fact. The first hash function we examined—the constant hash function—is valid, but it produces too many collisions.

For characters in Java, the hash function is quite simple: take the character value of the `char`!

```
public static int hashChar(char ch) {
    return (int) ch;
}
```

Note that the way that we represent characters in a computer program is by assigning a unique integer to every character (known as the Unicode character encoding standard). Thus, we can simply use this integer as an index into our backing array for keys that are characters.

The function above is certainly consistent as it always returns the same value for, *e.g.*, `'a'`. It is *very* well distributed because by definition of the Unicode standard, every character has a unique integer value. We call such a hash function that provably never produces any collisions between elements a *perfect hash function*.

12.2.2 Hash Functions for Primitives

We can apply similar logic to the different primitive types of Java in order to obtain hash functions for each. Recall that the primitives types in Java are:

- `int`
- `char`
- `boolean`
- `float`
- `long`
- `double`

For integers, the hash function can simply be the identity function:

$$h_{\texttt{int}}(n) = n$$

As discussed, the hash function for characters simply takes their character value as the hash:

$$h_{\texttt{char}}(c) = (int)\ c$$

For booleans, there are only two possible values—`true` and `false`—we can simply assign two integers to them.

$$h_{\texttt{bool}}(b) = \text{if } b \text{ then } 1 \text{ else } 0$$

Floats are trickier to hash. Recall that floats and doubles are floating-point numbers—decimals of finite length—represented using the IEEE floating point standard. As a first attempt, we might try to simply truncate the float to an integer via a cast. However, for certain sets of keys, this is a very bad hash function. For example, if all our keys are drawn from the rationals in the range $(0, 1.0)$, then this function will hash them all of them to the zero index in our map.

A better hash function comes from the realization that a float is the same size as an integer—32 bits. Because of this, there is a one-to-one correspondence between floats and integers (although it is not a natural correspondence because a float is represented differently than an integer). In Java we can take advantage of this correspondence by using the static method `Float.floatToIntBits` to convert a float into an integer by reinterpreting the 32 bits of the float as an integer. For the float 3.14 this yields the integer 1078523331. In contrast, if we simply truncate-cast the float to an int, we receive 3.

Like `char`, this hash function for `floats` is also perfect because it maps every possible `float` to a unique integer. In contrast, we cannot do the same thing for longs. This is because a `long` is 64 bits whereas an `int` is 32 bits. Because there are many more `long`s than `int`s, there will be collisions with any hash function that we can design.

There are a variety of possible hash functions we can use, *e.g.*,

- $h_1(f)$ = `(int) (f \ Integer.MAX_INT)`: mod the `long` by the maximum integer.
- $h_2(f)$ = `(int) (f >>> 32)`: use the most significant 32 bits of the `long`.
- $h_3(f)$ = `(int) (f << 32 >>> 32)`: use the least significant 32 bits of the `long`—shift the least significant bits to the top, then shift back.

Each of these hash functions are consistent, so they are valid hash functions. However, there are potential problems with each:

- With h_1, the modulo operator is a costly operation which we would like to avoid with our hash function if possible.
- With h_2, Shifting is cheap, however, we ignore the bottom 32 bits of the `long`. If all our keys only differed in the bottom 32 bits, then this hash function would send every key to the same hash value, zero.
- Likewise with h_3, if the all our keys differed in the top 32 bits, then this hash function would also send every key to the same hash value.

To avoid these difficulties, Joshua Bloch in his book, *Effective Java*, recommends the following expression for its hash function:

$$h(f) = \texttt{(int) (f ^ (f >>> 32))}$$

This is a variation of using the most significant bits of the `long` that takes into account the least significant bits as well. It does this by performing a bitwise or with the top 32 bits (the result of the shift) and the bottom 32 bits. This is also what the Java standard library performs for its hash function for `long`s. Again, note that this is not a perfect hash function, but it performs well in practice.

Finally, to hash a `double`, we can simply combine our strategies for a `float` and `long`:

1. Convert the `double` to a `long` by reinterpreting its 64 bits (using `Double.doubleToLongBits`).
2. Perform the `long` hash function described above to produce an `int`.

12.2.3 Hash Functions for Objects

Of course, we are unlikely to need to only produce hash values for primitives. More likely, we will need to hash object values of a particular class type that we have designed that has arbitrary fields. In Java, the `Object` class provides a method:

```
/** @return a hash value appropriate for this object */
public int hashCode() { /* ... */ }
```

That any class can override to provide appropriate hashing behavior for their instances. On top of requiring that an implementor's `hashCode` function must return an integer, Java requires several other properties of such an implementation:

- Like the hash functions discussed previously, `hashCode` must be *consistent*—it must report the same hash value for any particular object during the execution of the program.
- If two objects are deemed equal (via the `equals()` method), then their hash values must be identical.

Like hash functions, `hashCode` is free to return the same value for un-equal objects—this is a hash collision and ideally avoided as much as possible.

The default implementation of `hashCode()` on most versions of Java returns the *memory address* of the object. Note that this is a unique representation of the object as only one object can sit at that address in memory. This is also the default implementation of `equals()`—using *object identity* as the definition of equality. However, we frequently want to override the definition of equality as *structural* over the fields of the object. Likewise, to meet condition (2) described above, we must implement `hashCode` to perform similarly. This is an important enough rule to codify below:

> If a subclass overrides the `equals` method, then it must also override the `hashCode` as well.

As an example, consider the humble Point class:

```
public class Point {
    private int x;
    private int y;
    public Point (int x, int y) { this.x = x; this.y = y; }
    public boolean equals(Object o) {
        if (o instanceof Point) {
            Point rhs = (Point) o;
            return x == rhs.x && y == rhs.y;
        } else {
            return false;
        }
    }
}
```

Equality over points is defined to be equality over their x- and y-coordinates. How do we override `hashCode` to mimic this behavior? There are two integer fields for Point—how do we combine them into a single hash value? One solution is to simply add the two fields together:

```
public int hashCode() {
    return x + y;
}
```

However, this means that the two points $(1, 0)$ and $(0, 1)$ will hash to the same value: 1. We would like to avoid *permutation collisions* of this nature since they are common for a variety of objects.

Joshua Bloch recommends the following general strategy for implementing hashCode for classes taking into account their fields:

```
public int hashCode() {
    int result = /* a random non-zero integer */;
    for (/* each field f in this object */) {
        int c = /* the result of hashing the field f */;
        result = 31 * result + c;
    }
    return result;
}
```

Start with a random non-zero integer. Then for each field of the object, add its hash code to the result after multiplying the current value of result by 31. This approach is essentially taking the *linear combination* of the hash codes of the fields of the object with the exception of multiplying the accumulated value by 31. According to Bloch, the value 31 is chosen because:

1. 31 is an odd prime. An odd prime is chosen to minimize the probability that the multiplication does not result in the hash value and the number of buckets having a similar prime factorization. If this is the case, then the hash values will collide much more frequently.
2. 31 is specifically chosen because the compiler can turn operations over this number into series of shifts and subtractions, specifically `31 * i == (i << 5) - i`. Although, any odd prime of sufficient size is fine.

For our point class above, we would adopt this strategy into its `hashCode` method as follows:

```
public int hashCode() {
    int result  = 1091;
    result = 31 * result + x;
    result = 31 * result + y;
    return result;
}
```