1 Hash functions

Intuition: The reason we sometimes prefer ArrayLists over LinkedLists is because we want to have to ability to index into an array in constant time. Using the property I can create sets with constant time containment lookup. Say I want to store a set of integers that I know are bounded by [0, 5). Instead of having to iterate through my entire set structure to check if integer i is in my set, I can instead map value `array[i] = true` if i is in my set, and `false` otherwise.

The set $\{0, 2, 3\} \Rightarrow$ `array` $=$

0	1	2	3	4
`true`	`false`	`true`	`true`	`false`

Extending this idea: The idea proposed above is perfect if we only want to store integers in our set. While useful in some situations, not very extensible for many other uses. In comes the `hash` function. We define the hash function as a function that takes any structure (Object in java, but the hashing concept extends to any programming language) and mapping that object to an integer. So for the above example, `hash(i) = i`. A simple hash function, but it has one key property we must maintain:

1. Given two integers, `a` and `b`, if `a = b` then it must be the case that `hash(a) = hash(b)`
2. So in Java, if `a.equals(b)` (or `b.equals(a)`) then `a.hashCode() == b.hashCode()` must be true

This means, if we are going to implement `hashCode()` for an object, we better implement `equals()`.

2 Array implementation and load factor

Now that we've defined our hash function and some of it's properties we will define how the hash function is used to create our awesome data structure. `int`s in java are 32 bits, meaning there are 2^{32} possible hash values. While it's great that there are this many possible hash values, we don't want to keep an array of size 2^{32} (it would be 4GB). Instead we leverage the mod operator to make our hash function "wrap around" the size of the table. If we have a table size of `size`, then to find the index to place an element into the table, we compute `hash(x) % size` and this will give our index into the hashtable. An example using my previous hash function `hash(i) = i`, with an array of size 7:

Let's insert $x = 4, y = 16, z = 31, a = 111$,

`hash(x) % size = 4 % 7 = 4`
`hash(y) % size = 16 % 7 = 2`
`hash(z) % size = 31 % 7 = 3`
`hash(a) % size = 111 % 7 = 6`

0	1	2	3	4	5	6
		y	z	x		a

How do we find a value? We do the same operation! Lets see if 16 is in the table: `hash(16) % size = 16 % size = 2`, look at index 2, heyo, 16 is there! Now lets see if 23 is in our table. `hash(23)`

`% size = 2`. Ok, there is a value at 2, but it's not 23, so 23 is not in our table (now we see why implementing an equals function is important. If we just checked there was a value at the index, we verify that 23 is in the table (when it clearly is not)).

What happens when two hashes collide? Since there are only 7 table slots, and presumably there are infinite possible elements to add, at some point the hashes will collide. See the next section for dealing with collisions. Before collisions, we will talk about the load factor of a table. We define load factor as:

$$\lambda = \frac{\text{num elements}}{\text{size of table}}$$

For our table above, $\lambda = \frac{4}{7}$. Load factor will come up when we talk about resizing the underlying array to accommodate more elements, so store this concept away.

3 Collisions and probing methods

Collisions happen all the time in hash tables and like all things in computer science, there are multiple ways to handle the collisions (with trade offs). The simplest way to resolve collisions, is to move to the next index in the array until we find an empty slot to put the item, the pseudocode with `array` as the underlying array and `x` as the element we are trying to add:

```
int idx = hash(x) % array.length;
while (array[idx] is not empty) {
    idx = (idx + 1) % array.length
}
array[idx] = x
```

This method is called **linear probing**. There are a couple things to notice about the code above:

1. When we are advancing the index, we need to again mod by the table size. Say we have a table of size 5, and element in index 4, and another item to add with hash 4. If we try to probe without moding by table size, we will run off the end of the table. Instead we want to "wrap around" to index 0.
2. The method will only work if there is *at least* one empty slot in the underlying array. If the array is completely full, this probing method will run infinitely. Recall load factor λ from last section. Linear probing will only work strictly when $\lambda < 1$.

This method is simple to implement but it has some drawbacks, depending on the elements inserted. A case can arise where large contiguous chunks of the underlying array can become full. Linear probing becomes more expensive as large contiguous blocks are full, because we have to probe all the way to the end of the block to insert and to check for containment.

Another method, that spreads out elements instead of sticking them in one large block is called **quadratic probing**. Instead of going to the next index in the table, we quadratically increase our index:

```
int hash_x = hash(x)
int i = 1;
int idx = hash(x) % array.length
while (array[idx] is not empty) {
    idx = (hash_x + i * i) % array.length
    i++
}
array[idx] = x
```

This will yield idx values of the form:

i	idx (remember to mod by table size!)
0	`hash(x)` + 0^2
1	`hash(x)` + 1^2
2	`hash(x)` + 2^2
3	`hash(x)` + 3^2
⋮	⋮
i	`hash(x)` + i^2

This alleviates the problem that arose above where contiguous blocks of the table would be full of elements with different hashes (cool). But, there is unfortunately a way for quadratic to fail to find an empty slot even if there are empty slots (not as cool). Just know that as long as you keep your table size prime, and the load factor $\lambda < \frac{1}{2}$ quadratic probing will find a slot.

We can extend the idea of quadratic probing by supplying a different function to add by. Notice above, the function to probe with is $f(i) = i^2$. This is nice because it distributes the elements, but what if we have two elements map to the same bucket? They will both follow the same path. Instead, we would like to "randomize" the paths different elements take. What if we made the iterations of the probing specific to the element. This idea is called **double hashing**. Instead of using $f(i) = i^2$, we will use $f(i) = i \times h'(x)$, where $h'(x)$ is a different hash function than $hash(x)$. If $h'(x) = hash(x)$ then we are stuck with the same problem where if two elements map to the same index in the table, they will take the same path when we probe. The pseudocode below shows how we would implement double hashing:

```
int hash_x = hash(x)
int hprime_x = h'(x)
int i = 1
int idx = hash_x % array.length
while (array[idx] is not empty) {
    idx = hash_x + i * hprime_x
    i++
}
array[idx] = x
```

One thing to note, if $h'(x)$ is 0, then we will infinite loop when `array[hash(x) % array.length]` is not empty, so avoid that if using double hashing.

4 Separate chaining

So far we have talked about ways to store just the raw data in an array. When collisions arose, we picked a probing strategy to resolve collisions. What if, instead, we stored a "bucket" of all the items that index in our array. This is **separate chaining**. At each index we now store a data structure that contains all the elements that hashed to that bucket. Note, we can put any data structure here such as an `ArrayList`, `LinkedList`, `BinaryTree`, or `AVLTree`, etc. To check for containment of an element:

```
contains(x) {
    list = array[hash(x) % array.length]
    return list.contains(x)
}
```

A similar method can be written to insert and delete.

Like many other data structures, we have proposed multiple ways to implement the Hashtable, especially with respect to collision resolution. It is up to you, the implementer, to choose the best conflict resolution for the problem you are trying to solve.

5 Resizing the table (Rehashing)

In the above sections I have alluded to not letting your load factor get too large. In order to combat too large of a load factor, at some point we are going to have to resize the underlying table to accommodate more elements. We call this rehashing. It would be nice if we could just increase the table size and copy over the elements, but a problem arises. Lets say we double the underlying array size (in practice, you would probably like to find another prime number about 2 times the previous table size, but this, again, is up to you). It will likely be the case that `hash(x) % array.length != hash(x) % 2 * array.length`:

```
new_array = new array of size array.length * 2
for (every element e in array) {
    idx = hash(x) % new_array.length
    insert e at new_array[idx] with probing or chaining
}
array = new_array
```

It should be clear now that we call this procedure rehashing because we have to recalculate the hash and reinsert into the new array.

What about the run time? If at some point, we have to resize our table, then it must be the case that one of our operations takes $\mathcal{O}(n)$ runtime. But not every call to insert (hint... Amortization). Recall that amortization is:

$$\frac{\text{sum of runtimes of operations}}{\text{number of operations}}$$

Assuming we double the size of the underlying array:

$$\frac{\mathcal{O}(n) + n\mathcal{O}(1)}{1+n} = \mathcal{O}(1)$$

6 More hashing (if you didn't get enough)

If this got you stoked on hashing, And you want to go through some hashtable proofs, I would suggest reading this pdf. It goes into a little more math about hash tables like bounding runtimes. Would not recommend reading if you do not have a solid math background: HERE