20.1 Hash Table Recap, Default Hash Function

> "The whole point is that we have a bunch of lists that are all short."
> - Professor Hug.

Let's continue understanding Hashing. We've now seen implementations for sets and maps.

1. Red-Black Based Tree Approach: TreeSet/TreeMap

- requires the items to be comparable (the notion of less or greater than)
- logarithmic time complexity

2. HashTable based approach: HashSet/HashMap

- constant time operations if the hashCode spreads the item nicely (few collisions)

Recall that with a hash table, the idea is that for any piece of data, like a String (or any other type of object) we want to store in our hash table, we need to turn this into a number called a hash code. So a string like "Mihir" can be converted to -2101281024.

> ***NOTE:*** Fun Fact: And that integer is anything between about negative two billion and about two billion, the space of all Java Integers. This range yields about 4 billion integers or 2^32 integers. If you are interested in why this is the case, please take CS 61C!

Once we have our number, we want to convert this number to our bucket number (i.e. which of the many linked lists I want to add this new entry to). In the first example, we will use theMath.floorMod(x, 4), since the length of the underlying buckets array has length 4.

If we take the converted hash code for "Mihir", -2101281024, and then mod this by 4, we get 0. This reduces our hash code, -2101281024 to a valid index, 0. This means that we would use the 0th bucket to place our data, "Mihir", in our LinkedList at the 0th bucket.

> You can essentially think of Java HashTables as just an array of LinkedLists categorized under bucket labels and hopefully have better performance by utilizing these LinkedLists.

But how many LinkedLists/Buckets should we use? This is an important question, because as we insert more and more items into our buckets, the length of the LinkedLists will inevitably grow, which in turn compromises the runtime for the hash table's operations. For example, let's say we inserted strings like "Mihir", "Mirchandani", "loves", and "61B", all of which yielded hash codes that were divisible by 4. In this case, all strings would be put into the 0th index bucket and all strings would be inserted into the same LinkedList. What happens to the runtime for a search operation? Well, we would have to search through an entire LinkedList to look up our data! This runs in linear time and is too slow for HashMap's famous title of holding fast lookup times. Last time, we saw that we can have a variable number of LinkedLists. The idea is that we resize that array of linked lists whenever the load factor (N/M, where N is the number of elements in our table and M is the number of buckets) exceeds some constant. Java picks 0.75 as we'll see later. This prevents collisions from happening too frequently. So long as our items are spread out nicely, between the buckets, the LinkedLists at each bucket for the most part have a very small size, which means we'll on average get constant run time! Example: If the HashTable has load factor 3, and our hashCode() function spreads out the entries evenly, we are going to end up with just 3 items per bucket, and our search operation takes *O(1)* time.

Comparing Data Structure Run Times!

Data Structure	contains(x)	add(x)
Bushy BSTs	O(log N)	O(log N)
Separate Chaining Hash Table with NO resizing	O(N)	O(N)
Separate Chaining Hash Table with resizing	O(1)	O(1)

20.2 Distribution By Other Hash Functions

> "HashMaps/Tables have fast lookup times, but behind that"superpower" is a hash function."

Suppose our hashCode() implementation simply returns 0.

```
@Override
public int hashCode() {
    return 0;
}
```

> What distribution do we expect?

We would expect all of the items in our Hash Table to be in bucket 0. As we discussed in the previous section, our Hash Table would place all elements in the 0th bucket because the hashCode tells it to. In the 0th bucket, there will be a LinkedList of all elements from the data yielding a very inefficient linear lookup time compared to the constant time we are expecting. No matter what key we provide, our hashCode always tells the HashMap to only add to the 0th bucket which is why we get this long LinkedList. So what do we do to make sure we get constant lookup time? We use a better hash function!

In order to get a more even distribution, what we can do is something to what we tried in the previous section where we utilize modulo. Let's say that we define the size of our Hash Table to have 4 buckets. This means it has 4 corresponding LinkedLists and 4 bucket indices labeled {0, 1, 2, 3}. The modding is not required in our hashCode() function as it is being done for us in the hash table to guarantee we can add to that bucket. As we said the hash code could really be any integer in the range of 4 billion unique values!

By using modulo, we ensure that our hashcode yields a number that can be represented as an index and clearly identifies which LinkedList to add to. Additionally, when adding a series of numbers at once, we see that we get an even distribution of numbers in our LinkedList yielding a constant lookup time.

```
@Override
public int hashCode() {
    return num;
}
```

This hash function should yield a much more even distribution! Objects with different num values will now be more spread out across the buckets instead of all living in the 0th bucket. If our class does not explicitly override the hashCode() function, Java will use the default implementation, which returns the object's address in memory as its hash code!

Let's discuss if the default hashCode function is a good hashCode function! It actually is a good spread as it relies on the fact that different objects will live in different places in the memory, and the memory address is effectively random. We will get a good distribution, since objects are basically assigned random indices to insert into the hash table.

Basic rule (also definition of deterministic property of a valid hashcode): If two objects are equal, they must have the same hash code so the hash table can find it.

19.2 Hash Code

The core mechanism of hashing!

We face the problem that not every object in Java can easily be converted to a number. However, the key idea behind hashing is the transformation of any object into a numeric representation. The key is to have a hashing function transform our keys into different values, and convert that number into an index to then access the array.

We achieve this through our own implementation of a hashCode() function, with a return value of an int type. This int type is our hash value.

The built-in String class in Java, for example, might have the following code block:

```
public class String {
    public int hashCode() {
        //implementation here
    }
}
```

Based on this example, we can call `key.hashCode()` to generate an integer hash code for a String instance called key.

Memory Inefficiency in Hash Codes

An issue mentioned earlier is memory inefficiency: for a small range of hash values, we can get away with an array that individuates each hash value. That is, every index in the array would represent a unique hash value. This works well if our indices are small and close to zero. But remember that Java's 32-bit integer type can support numbers anywhere between -2,147,483,648 and 2,147,483,647. Now, most of the time, our data won't use anywhere near that many values. But even if we only wanted to support special characters, our array would still need to be 1,112,064 elements long! Instead, we'll slightly modify our indexing strategy. Let's say we only want to support an array of length 10 so as to avoid allocating excessive amounts of memory. How can we turn a number that is potentially millions or billions large into a value between 0 and 9, inclusive?

Wrapping!

The modulus operator (%) allows us to achieve this. Review: The result of the modulo operator is like a remainder in fractional division. For example, 65 % 10 returns 5 because after dividing 65 by 10, we are left with a remainder of 5. Thus as other examples, 3 % 10 = 3, 20 % 10 = 0, and 19 % 10 = 9. For an intuitive understanding of this, think about how we used the modulo operator in Project 1: Deques to ensure you avoided IndexOutOfBounds Exceptions and could accurately index into your deque via the concept of "wrapping around"

the array. Returning back to our original problem, we want to be able to convert any number to a value between 0 and 9, inclusive. Given our discussion on the modulo operator, we can see that any number mod 10 will return an integer value between 0 and 9. This is what we need to index into an array of size 10! More generally, we can locate the correct index for any key with the following:

```
import java.lang.Math;

Math.floorMod(key.hashCode(), array.length)
```

where `array` is the underlying array representing our hash table.

Quick Note: In Java, the Math.floorMod function will perform the modulus operation while correctly accounting for negative integers, whereas % does not.