An Introduction to Hashing

"The woods are lovely, dark, and deep, / But I have promises to keep, / and miles to go before I sleep, / and miles to go before I sleep". -Robert Frost.

Why do we need hashing?

I think one of the best ways to illustrate the purpose and importance of hashing is to show how other methods of storing data have shortcomings.

Think of an array of ten elements, from 0-9. Searching for an element in this array is pretty easy! – it may look something like this:

```
int[] nums = new int[] {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

public boolean search(int find) {
   for (int x : nums) {
      if (find == x) {
         return true;
      }
   }
   return false;
}
```

This method of searching works well on small sets of data, because there are not that many numbers to check to know whether or not your array has the value you're looking for.

However, consider if the nums variable had a lot more data in it – for example, the social security numbers of every citizen in the United States. That is almost 450 million numbers. How long would it take to find something in that array?

The answer is that it could *possibly* take very long. Maybe you get lucky and it's the first number you check! However, in a worst-case scenario, the number you are looking for isn't in the array, meaning you would check all 450 million of those numbers. Your computer could do it – it just may start reciting the Robert Frost quote about halfway through.

Let's try to take this example and build a general case scenario that we can apply to any data set that is stored in an array. For any array that has some amount of elements in it, let's call that N, we can say that the time that it takes to do a search operation is $\Theta(N)$[1].

What does this mean? Well, this theta bound says that as the amount of elements in the array becomes infinitely large, the time it takes to do a search operation scales linearly to the amount of elements. If it said N_2, that would mean that as the amount of elements in the array becomes infinitely large, the time it takes to do a search operation scales quadratically or exponentially to the amount of elements.

In an intuitive sense, this means if I have 100,000 elements, I might have to check 100,000 elements to do a search. If I have 250,000, I might have to check 250,000 elements to do a search. The amount of elements you need to check grows larger, in this case, in a constant one-to-one ratio to the overall amount of elements.

We've seen that small searches on small sets of data are fast and easy. What if there was a way to *keep* that speed on a large set of data?

The Idea of Hashing

One way of keeping a small search speed on a large set of data is to divide the data into different buckets, each with a small number of elements. Let's assume we keep the number of elements in each bucket close to equal. (If the idea of splitting data into buckets isn't clear, check out the visuals later in this document).

Then, for any given number I want to find, if I know what bucket it is in, I only have to search the small amount of numbers in that bucket to tell whether or not it's there!

Think about how much of a massive improvement this is. Let's say for our Social Security example, each of the 450 million numbers was in a bucket of 5 elements, and you knew which bucket to look in to check for the number you are looking for. You would have to check, at most, 5 elements, to tell if an element is in an array of 450 million numbers. That's a 99.999998% improvement. Woah.

If we are thinking about time complexity, the amount of time it takes to do a search operation using hashing is $\Theta(1)$, or constant time. This is because, no matter what number I'm

[1] For an intuitive explanation of Big-O, Omega, and Theta bounds, see my guide on that topic! I reiterate some information here for your understanding nonetheless.

looking for, I only have to search a constant number of elements (in this case, 5) to do a search operation.

But remember, all this amazing performance only happens *if I know what bucket to be looking in*. If I have 450 million numbers and each bucket has 5 numbers, that means there's 80 million buckets. How do you know which one to look in, and without doing a whole lot of computation?

Welcome to the Hash Function

A hash function is a function that takes in some value and tells you what bucket it corresponds to.

For example, assume we have a set of N = 400 random numbers, and you want to make sure you search no more than 4 numbers on every search. That means you would have 400/4 = 100 buckets. A simple hash function for this set of data could be *h(k) = k%100*, where *k* is a number in the set. This reads as: the bucket you should look in is the bucket numbered as the remainder of your number divided by 100.

So, the number 434 would go in bucket number 34 (since 434%100 = 34), the number 534 would go in the same bucket (since 534%100 = 34), but 122 would go in a different bucket (since 122%100 = 22). These numbers – 34 and 22 – are what we would call *hash codes*.

The natural question, then, is what hash function do you use? You actually get to design whatever hash function you want!

However, for hashing to provide you the improved performance that you want, your hash function should fulfill two principles.

Ideally, a hash function is:

a) Valid. For a hash function to be valid, it should return the same *hash code* (in this case, the bucket number) for inputs that are considered equal. Simply put, this means that the number 434 should always return 34 from the hash function described above. This is important because if the hash function returns something different each time for the same number, you wouldn't receive the correct bucket number to look in when doing a search!

b) Good. This means that your hash function must be valid and should generate a varied and even distribution of hash codes. Why is this important? Well, you want to minimize the amount of inputs that return the same code from the hash function, because this means that those inputs would be in the same bucket. If your hash function, for example, is h(k) = 5 (a valid but not a good hash function) then all your elements would be in the fifth bucket, and you would always have to look in the fifth bucket for every search (making the search equal in performance to an array!) Recall that keeping a small amount of items in each bucket is what leads to the great time performance of hashing.

A hash function that returns one unique and consistent hash code for every input that it is given is what we call a *perfect* hash function. In this case, the hash code acts as a unique identifier for the number (like your Social Security number is a unique identifier for you).

Example Hashing Function

Assume we are dealing with Strings in place of numbers. Recall Strings are just linked lines of text, much like the sentences in this document.

Java's hash code for Strings looks something like this:

For a string s, let h(s) yield $S_0 \bullet 31_{n-1} + S_1 \bullet 31_{n-2} + \ldots + S_n \bullet 31_0$.

Looks complicated! In short form, Java takes the numeric value of each character in the string and multiplies it by 31 raised to some power to make the hash code. 31 is a pretty random prime number and it isn't used for any particularly profound reason except that it gives a nice even distribution of hash codes. Plus the operation with 31 can be simply written using bit-shift operators. Read more on that here if you are interested.

An example hash function, then, would be to take that hash code, and return the remainder of the hash code divided by the amount of buckets to get the bucket number.

If items that are not the same object are never considered equal, you can use Java's built in hash code, the identity hash function, by calling Object.hashcode(). This will return the memory address that the object is stored in. However, if you override the .equals() method for any class, make sure to override the .hashcode() method so that it will return equal hash codes for equal objects.

Implementing Hashing

Design Overview

Let's think about some of the design choices we have to make when using hashing to store our data.

Let's assume we have N elements we want to store, and we want to have no more than L elements per bucket, with L being a relatively small number so that our search times remain fast. The amount of buckets, what I will call M, is then N elements / L elements per bucket = M.

Therefore, our hashing function *h(k)* should give us bucket numbers between 0 (since we start counting from 0 in Java) and M, the total amount of buckets. So, all outputs of our hashing function should be in the range [0, M]. One common and easy method to achieve this is to take the remainder of a hashcode and M.

Resizing and Amortized Time

Recall that hashing as a technique only yields superior performance as long as the number of elements in each bucket is relatively low, on average. What happens when we add lots of elements, and so our buckets begin to fill up and hold more elements than we want?

For example, let's say we have 100 buckets, and we want to search no more than 10 elements per bucket. Initially, we store 1000 elements into the buckets, so we average 10 elements per bucket. However, we end up having more data added later, perhaps an additional 100000 elements. This would lead to our buckets having the original 10 elements plus an additional 1000 elements per bucket, or 1010 elements per bucket! This would significantly slow down our search times. How can we design hashing to deal with when this happens?

A common way to do so is to keep track of something called the load factor, or the average amount of elements per bucket. This can be easily computed if you keep track of the total amount of elements and the total amount of buckets. Once our load factor is greater than a certain number – in the previous example, 10 – we can *resize* our amount of buckets so that each bucket still holds a small amount of elements, therefore preserving our Θ(1) search time!

How can we do this? Well, one way to do it would be to increase your number of buckets by some multiple – like 2x or 3x the current size. This is simple enough;

```
List[] buckets = new ArrayList[oldbuckets.length * 2];
```

(I chose to represent the hashed data by a List array – you can do it in different ways ☺)

You would then have to recalculate the bucket numbers of all your data. Why is this so? A simple explanation is that your hash function *h(k)* only yielded bucket numbers from 0 to M, but now there are buckets all the way from 0 to 2M. When you recalculate and place the data using a hash function that factors in the new size, it will spread out the data more, achieving the lower load factor that you want!

Something to notice is that I have said that insertion, search, and deletion times should all be constant time or Θ(1). But when we resize, which would happen when we add an element that causes the load factor to go above what we want, we have to double our bucket size and recalculate the bucket numbers of all the data, which is a Θ(N) operation (N being the total amount of data in all our buckets).

Did I lie or something? No – our operations are still constant time, but they're *amortized constant time*. The word amortized comes from the Latin/French root *mort* or *mors* meaning 'to die' (if you've ever read Le Morte d'Arthur this might sound familiar).

How I like to think about this is our constant time performance "dies to live another day". Every so often, you may need to resize, which is a Θ(N) operation, but this is pretty infrequent, and the majority of the time you will have a constant time operation. Let's say you can add 100 elements, all in constant time, before you have to resize. You resize, which then buys you another 100 elements of constant time addition before you have to resize again. You have made 200 constant time operations, and 1 Θ(N) operation. So, *on average,* hashing leads to a constant time performance, only occasionally needing to do an expensive computing operation.

Open Address Hashing

Let's get to some actual ways of implementing hashing! A common technique is to use 'open address hashing'. Let's see what this looks like visually. The below box represents a one dimensional array that is empty. Each space represents a bucket that can hold one element.

Let's use a hashing function *h(k) = k % m* , returning the remainder of *k* and the amount of buckets *m*.

Let's add the numbers 12, 14, 16, 17, and 9.

		12		14		16	17		9

So far, so good! But what happens when the hash function returns a bucket number that already has a number in it? Let's say, 22, which would collide with 12.

		12	22	14		16	17		9

Open addressing would look at bucket number 2, see 12, and say "oh whoops, this is full, so why don't I go to the next spot and see if that's empty?" Search and deletion with open addressing work the same way – they go to where the hash function tells them to go, and they check the next position if it's not there, and the next position after that, etc!

In this case, 22 was put right next to 12. This is called a linear probe. There are also quadratic probes, which put colliding elements at positions $h(k) + 1_2$, $h(k) + 2_2$, $h(k) + 3_2$, $h(k) + 4_2$ and so on, wrapping back around when necessary (meaning if the position is greater than the size of the array, you just keep counting from the front. For example, if I had to go to position 12 on an array of 10 elements, I'd count the ten, then wrap back around to the front and go to position 2).

What this means is that your program will first check what your hashing function tells you, and if that's full, you'd check the position your hashing function gave you plus 1. If that's full, you'd check the position your hashing function gave you plus 4. If that's full, you'd check the position your hashing function gave you plus 9. And so on.

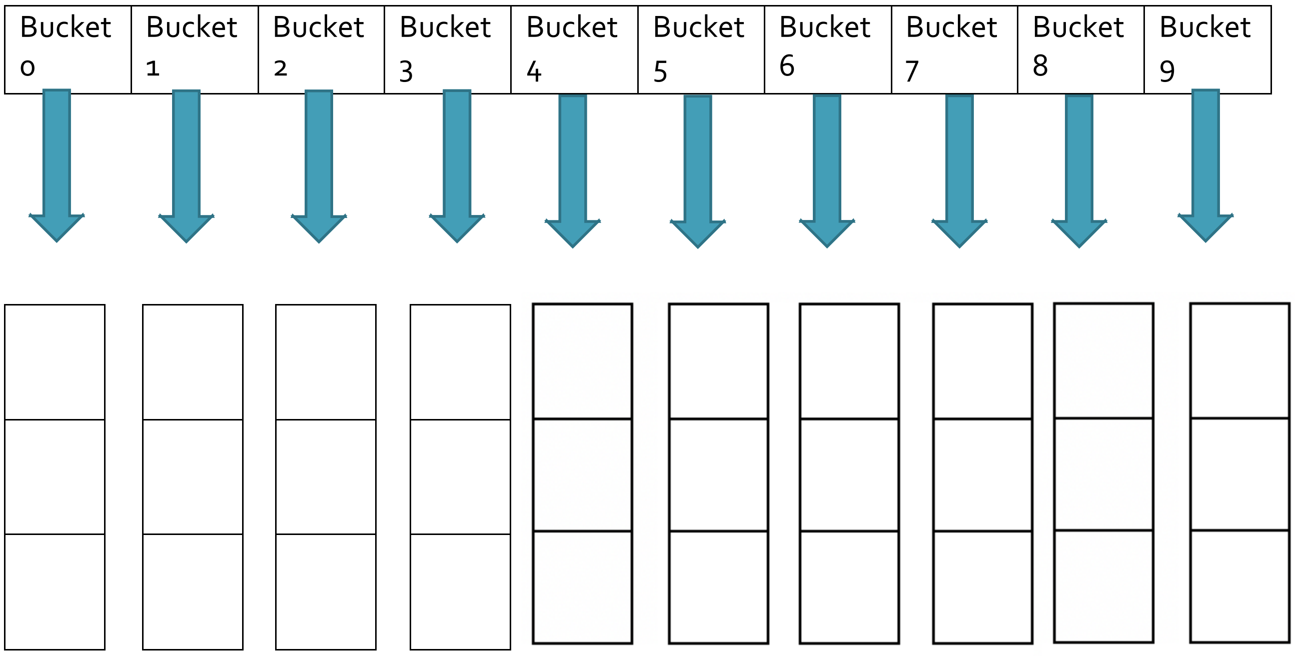
You can even have a second hash function, $s(k)$, and do a double hashing probe, like

$$h(k) + s(k),\ h(k) + 2 \bullet s(k),\ h(k) + 3 \bullet s(k)\ldots$$

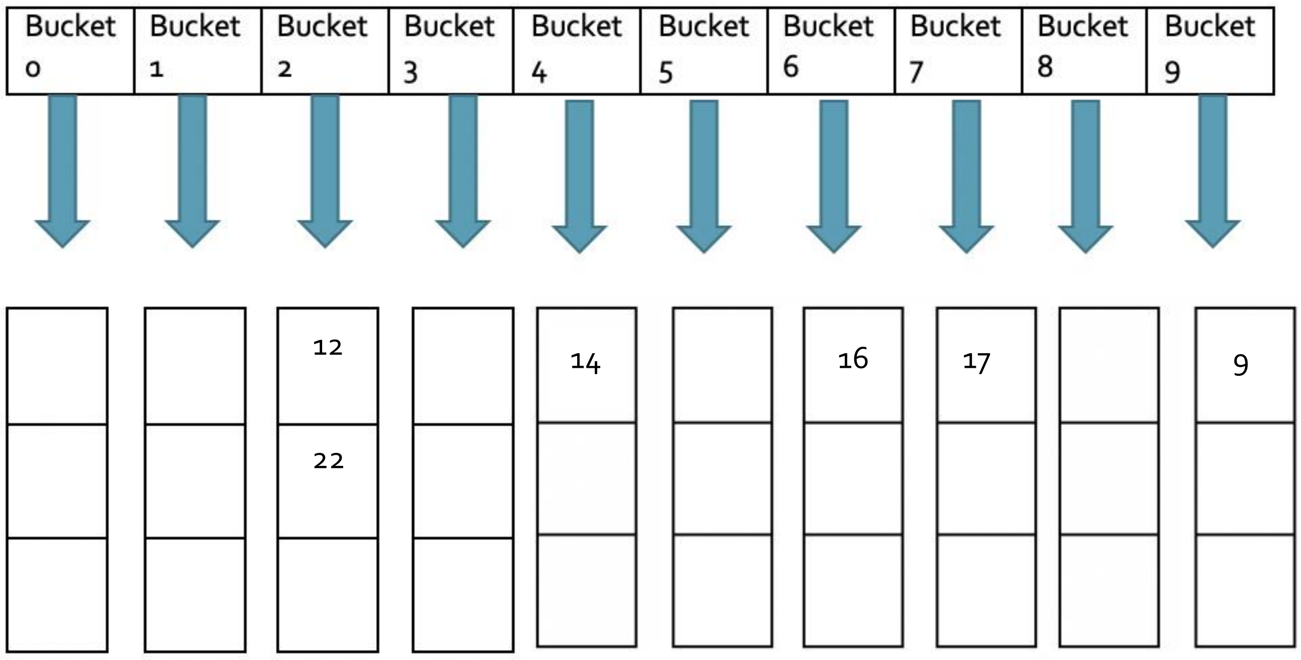
This technique is complicated, can get slow, and is just not intuitive in my opinion. I far prefer external chaining!

External Chaining Hashing

External chaining hashing is where each position in the array represents a bucket number, and each bucket can store more than one element.



Using the same hash function (*h(k)* = k % m) and numbers as before (12, 14, 16, 17, 9, 22), let's see what the data looks like.



We call this technique *external chaining* because each bucket number is "chained" (or in technical terms, has a pointer) to the actual data in each bucket.

As you can see, each bucket has a certain number of elements (0, up to 2 elements). If I search for the number 22, I get the number 2 from my hashing function. Looking in bucket 2, I only have to look at 2 elements (12 and 22) to tell whether or not it's there in the data. If I'm looking for the number 41, my hashing function would return the bucket number 1. Searching in bucket 1, which is empty, you see that there are no numbers there, which means that the data does not contain the number.

Our load factor here is 6 elements / 9 buckets or 0.66, a pretty low number, meaning that our searches, insertions, and deletions should be really fast. We maintain constant time since, on average, we check no more than a *constant number* of elements (here, no more than potentially 3 elements).

Think about ways to represent this data structure – what decisions you might have to make surrounding resizing, what the data structure of the buckets should be, what the data structure of the bucket numbers should be!

Conclusion

I hope this guide provided a comprehensive overview to hashing, its purposes, and some ways of implementing it. As always feel free to email me at adityavarma@berkeley.edu if you see any necessary corrections on this information.

References

https://inst.eecs.berkeley.edu/~cs61b/sp20/materials/lectures/lect24.pdf

http://www.cse.yorku.ca/~oz/hash.html

https://www.cs.hmc.edu/~geoff/classes/hmc.cs070.200101/homework10/hashfuncs.html

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Hashing/hashing.html