How Hash Tables Work

medium.com/@ebenwoodward/how-hash-tables-work-dcf61110ce45

As I've been deep diving into a self-taught journey of Data Structures and Algorithms, my studies have led me to the Hash Table data structure. Known in Ruby as a hash, in JS as an object, in Python as a dict(ionary), Java as HashMap, etc. This class is ready made in most OO languages, but there's a lot that's going behind the scenes.

Why are Hash Tables important?

Why would we want to store things inside of a hash/obj/dict/hashmap versus an ordinary array? What's the difference between storing something in an array in JS and implimenting .find() to search for it?

The reason why we don't do this is because searching inside of an array is quite costly. If you're searching for something at the end of an array, a .find() in its worst case, is O(n), meaning it has to go through every single item in the array before the function finds what you're looking for. And if you're trying to find something that's not there, all that work is for naught.

The same goes for removing something. You would have to search for the item, then remove it. Removing it is also costly in an array because that means the entire array needs to be reindex, basically creating a whole new array.

Where arrays do excel is in finding something at a given index. Arrays are indexed, so if you know exactly the index you're trying to shoot for, the search time is O(1). If we want search an array, arr, at index 3, its quite cheap. arr[3] is a great great way to go.

Hashes are arrays OF arrays

Hashes take advantage of the indexed nature of arrays. At their core, they are arrays. Well, arrays of arrays. And how I've written it, array of arrays of arrays.

When we make an object, let’s say

```
obj = {"apple" : "pie"}
```

and we then look for obj[“apple”], the object isn’t actually searching for the text “apple”. A hash table *hashes* a the keys that it’s given and converts it into a number, which is then used as an index. In reality our above object could look like this:

```
[undefined, undefined, undefined, [["apple", "pie"]], undefined]
```

Through a hashing method, “apple” is turned into an index number. In this case, its 3. An array of the key value pair is created, and pushed into an array at that index. If there is no array at that index, one is created and then the new key value array is passed in.

I wrote a simple hashing method and setting method (in JavaScript), just so you can get an idea of what’s happening.

```
class Hash {
   constructor(size=53){
      this.keyMap = new Array(size);
   }

_hash(val){
      let total = 0;
      let weirdPrime = 31;

      // I've read that if you multiply your value by a random
      // prime, the likely hood of getting a unique number is
      // increased

      for(let i = 0; i < Math.min(val.length, 100); i++){
         let char = val[i];
         let value = char.charCodeAt(0);
         total = (total * weirdPrime +  value) %
         this.keyMap.length;
      }
      return total;
   }
```

```
set(key, val){
        let idx = this._hash(key);
        let newArr = [key, val]
        if(!this.keyMap[idx]){
            this.keyMap[idx] = [];
        }

this.keyMap[idx].push(newArr);
        return this.keyMap
    }
}
```

As I've written it, when a new Hash instance is created, its default size is 53, meaning there are 53 options for an object's key to be hashed and indexed.

When we set a new key/value in an object, an index is generated by the _hash function. If there is nothing at that index, an array is generated and that value is pushed into it.

There is no Collusion!

The reason we generate an array and push our value is done to combat "collusion".

Collusion occurs when there are two keys that produce the same index after being hashed. Because i'm using the "separate chaining" method, if two keys have the same index they need to share! By creating an array and pushing the value into it, it allows for multiple items to be placed in the index spot.

There is a methodology called "linear probing" which means that, instead of sharing, the key will search for the next open index and store itself in that.

Probing aside, let's say we added another object to our array and it looked like this:

```
obj = {"apple":"pie", "peach":"cobbler"}
```

And our hashing method hashes both "apple" and "peach" and produces the same number, 3. Our hash would look like this, unmasked:

[undefined, undefined, undefined, [["apple", "pie"], ["peach","cobbler"]], undefined]

Searching

If we were to search the object like so — obj[“peach”] the hash class would break down “peach” into a number which it would use as an index. It then search for content at that index. If there is an array, it iterates through that array checking the first item of each subarray to see whether or not it matches the entered key, in this case “peach”. If there is a match, it would return the second item in that array.

A method to do just that is below:

```
get(key){
      let idx = this._hash(key);
      let keyMapIdx = this.keyMap[idx]
      if(!keyMapIdx){
         return undefined;
       } else if(keyMapIdx){
          for(let i = 0; i < keyMapIdx.length; i++){
             if(keyMapIdx[i][0] === key){
                return keyMapIdx[i][1]
             }
          }
       }
       return undefined
   }
```

And violá we have our searching method for our hash. A great, efficient way of storing all our data we need quick access to.