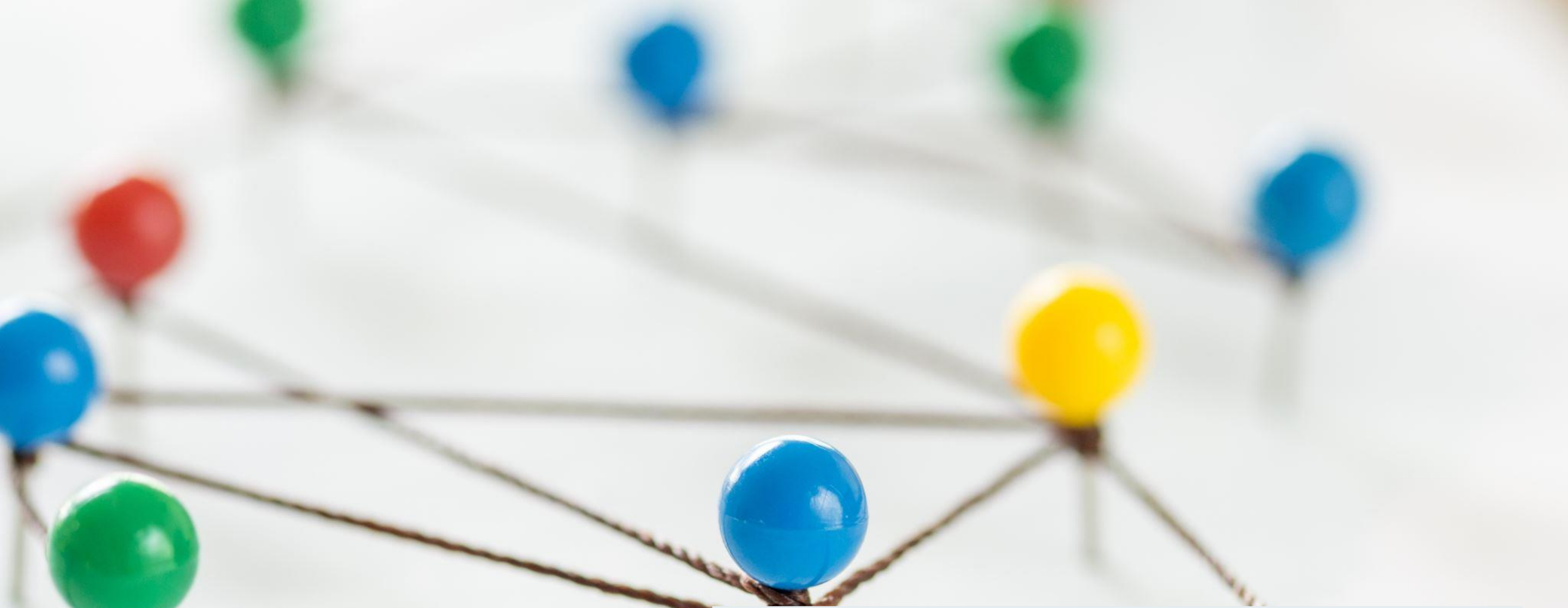
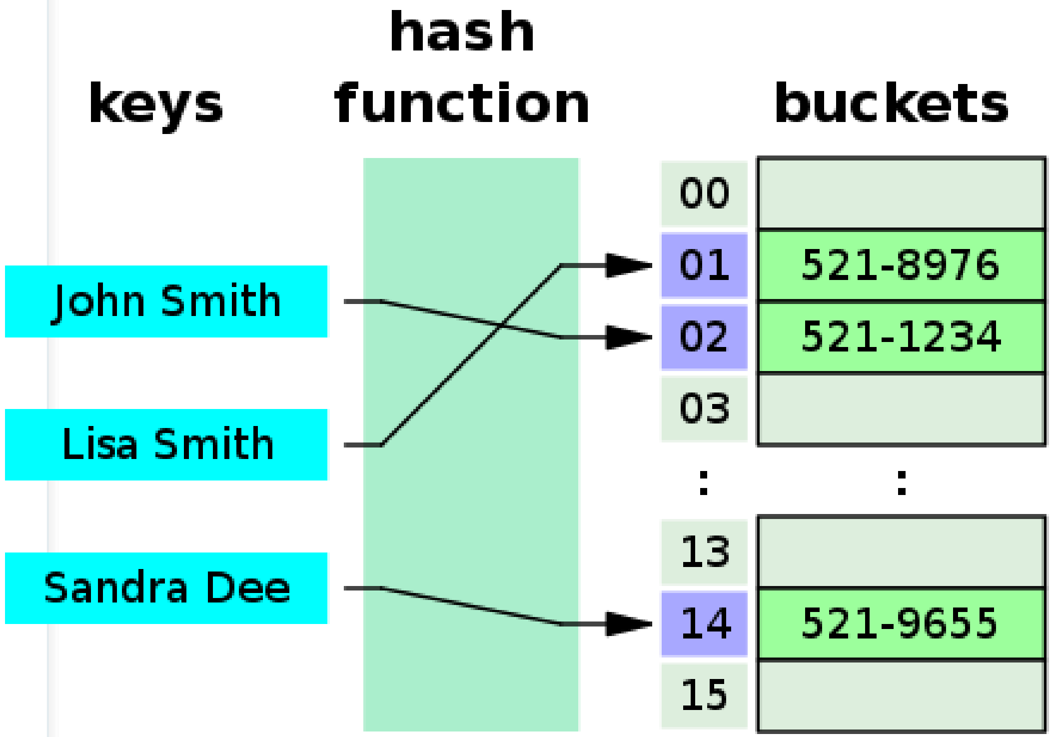


Hash Tables
Hash Maps

A powerful and versatile data structure!

A little bit of Background & some new Concepts

- In Computer Science, there is an ADT (Abstract Data Type [a math model of a type]) known as an **__Associative Array__** aka ***__Dictionary__*** type which maps keys to values as key-value pairs.
    - The Dictionary supports 3 fundamental operations:
        - Insertion
        - Removal/Deletion
        - Lookup/Find
- Invented in 1953 (thus a 'Boomer' type!)
- Works by storing key value pairs (behaves as a **__set__**).
- In practice, the set operation works through hashing.
- Hashing [*covered in CSC205*] is the process of mapping data of any type and size to a fixed-sized value representing the mapped data, usually in the form of an integer.
- In our case, we're going to use hashing to map data to fixed-sized values. These values are going to be used as array indexes.



Why Hash Tables?

Why not use an array or even a linked list [data structure that chains individual data]?

The Pros & Cons of Arrays:

- Fast when…
  - accessed by random integer index
  - appending (adding data at the end of the array)
  - overwriting data at an index.
- Slow when…
  - searching for data, prepending (inserting data at the beginning of the array) & when inserting data in the middle.
- Searches even faster when sorted.
- Slow when resizing [array is made larger and all data must be copied!]

Pros & Cons of Linked Lists:

- Fast when prepending and appending or deleting.
- Slow when searching and inserting when the position of a data isn’t known.
- Searches can be done faster if sorted as the data was inserted.

Best of both worlds come from the Hash Table

- Fast Insertion, Searching, & Deletion (Best Case Scenario)
- Not much slower in operation than an Array and Linked List (Worst Case Scenario)

First, we need a Hash Function!

- Can’t do a hash table without a decent hashing function!
- A good hash function must have a good distribution of the key’s values across the size of the hash table, so we reduce collisions as much as possible.
- Prime Numbers are key when it comes to good hashing. Numbers like 33, 37, 97. Sometimes BIG prime numbers work best, sometimes small.
  - ***Why***? Because they increase probability of having a unique hash.
- For this workshop, we’ll be using the **SDBM Hash Algorithm**. It’s also public domain.

Hash Function Code Check

```c
#include <stdio.h>

size_t string_hash(char const key[]) {
	size_t h = 0;
	for( size_t i=0; key[i] != '\0'; i++ ) {
		h = key[i] + (h << 6) + (h << 16) - h;
	}
	return h;
}

int main() {
	printf("%zu\n", string_hash("kevin"));
	printf("%zu\n", string_hash("ke"));
}
```

The Hash Function in Action

- Creates a unique hash per string key.
- Take the individual character’s ASCII Value.
    - ‘k’ = 107
- Add it with the previous hash value left shifted by 6.
    - 0 << 6 == 0
- Add again but with the ASCII value left shifted by 16.
    - 0 << 16 == 0
- Then subtract with the previous hash value itself.
    - 107 + 0 == 107
- Do this iteratively with the remaining letters!
    - ‘e’ = 101
    - 107 << 6 == 6848
    - 107 << 16 == 7012352
    - ‘e’ + 6848 + 7012352 - 107 == **7019194**
    - Try this with the string “ke”.
- sbdm(“kevin”) == **7629153830864703617**

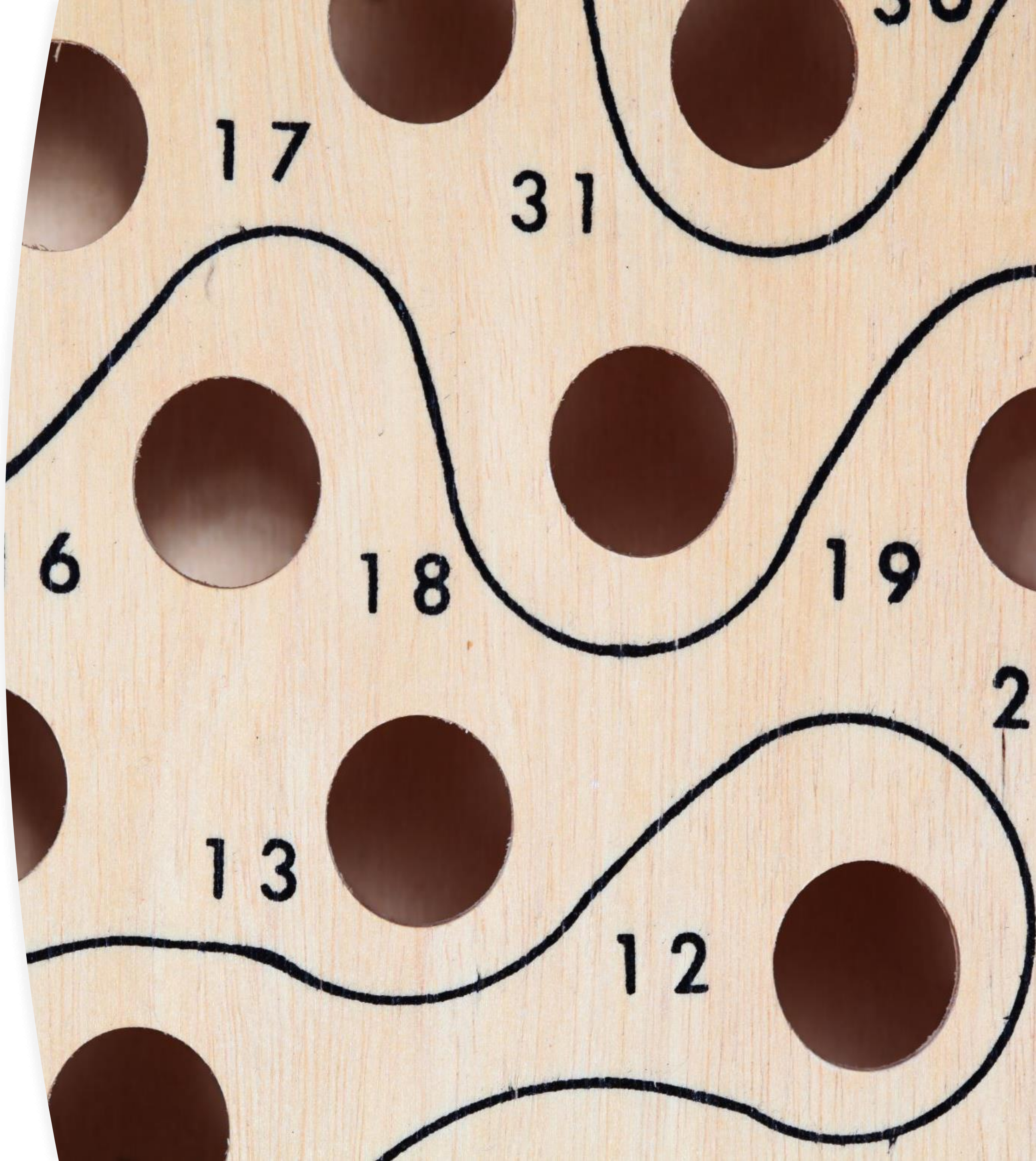


Table part of Hash Table!

- The ‘table’ part of the name implies one thing: Arrays.
- For our hash table, keeping it simple with 3 arrays and 2 integers.
- **3** arrays: to hold hashes, data, and buckets.
  - Hashes – self explanatory, this array holds the hashes of our string keys!
  - Data – also self explanatory, holds the data mapped to the hashed string keys.
  - Buckets – Holds the indexes of where our data is in terms of the index in the data array. Done by manipulating the hash value representing the string key.
- **2** integers: represent the capacity of our map, and how much data was stored in it (our length).
- 5 pieces of data total.
- For this workshop, arrays will be a fixed size of **32**.

Structure/Class Code Check Part I

```c
#include <stdio.h>

size_t string_hash(char const key[]) {
	size_t h = 0;
	for( size_t i=0; key[i] != 0; i++ ) {
		h = key[i] + (h << 6) + (h << 16) - h;
	}
	return h;
}

struct HashTable {
	int     data[32];
	size_t  hashes[32];
	ssize_t buckets[32];
	size_t  cap, len;
};
```

Structure/Class Code Check Part II

```c
/// constructor.
struct HashTable HashTableInit(void) {
	struct HashTable hashtable = {0};
	hashtable.cap = 32;
	hashtable.len = 0; /// length 0 means no data added.
	/// Initialize our arrays.
	for( size_t i=0; i < hashtable.cap; i++ ) {
		hashtable.buckets[i] = -1; /// -1 means empty spot.
		hashtable.hashes[i]  =  0;
		hashtable.data[i]    =  0;
	}
	return hashtable;
}

int main() {
	struct HashTable hashtable = HashTableInit();
	printf("%zu\n", string_hash("kevin"));
	printf("%zu\n", string_hash("ke"));
}
```

Hash Table Lookup Algorithm

- With a hash already computed, we use it to create a bucket index. The bucket index then gives us the value index from which we can retrieve our value/data.
- Clash of the Hash:
  - Data inserted in terms of hashes being made into bucket indexes, what if we tried to insert a value index into a bucket that isn't empty? → Hash Collision
- **How to deal with a Hash Collision?? Answer**: **<u>Linear Probing</u>**
  - Basically, iterate/loop the bucket until we find an empty spot.
  - Like when the Left Parking Lot is full, you have to look into the 2nd row to the right and so on until you find an empty spot.
  - In practice, takes advantage of CPU Cache memory to perform this faster than linked list iteration.
  - **Note: <u>Make sure the iterating does not exceed the end of our array</u>**
    - wrap the iterator around by setting the iterator to the start of the array!

Hash Table Lookup Code Check

```c
ssize_t HashTableFind(struct HashTable *hashtable, size_t hash, size_t *found_bucket_index) {
	size_t bucket_index = hash % hashtable->cap;
	ssize_t value_index = hashtable->buckets[bucket_index];
	if( value_index != -1 && hashtable->hashes[value_index]==hash ) {
		*found_bucket_index = bucket_index;
		return value_index;
	}
	
	/// Linear probing.
	size_t bucket_iterator = bucket_index + 1; /// original bucket index failed, so start on the next index.
	while( bucket_iterator != bucket_index ) {
		/// CAREFUL: What if the iterator goes to end of hash table arrays?
		/// Restart it at beginning of arrays then.
		if( bucket_iterator >= hashtable->cap ) {
			bucket_iterator = 0;
			continue;
		}
		
		ssize_t iterated_index = hashtable->buckets[bucket_iterator];
		
		/// make sure the hash is the hash we're looking for!
		if( iterated_index != -1 && hashtable->hashes[iterated_index]==hash ) {
			*found_bucket_index = iterated_index;
			return iterated_index;
		}
		
		/// didn't find it, search the next bucket index.
		bucket_iterator++;
	}
	return -1;
}
```

Hash Table Insert Algorithm

- **Important**:
  - check if the hash table is full before inserting the data!
    - Easy check, use an if-statement and make sure the hash table length is smaller than the hash table capacity.
  - check if we don’t already have the key in our map!
    - We have a Lookup/Find algorithm, we can use that!
- Create a bucket index out of the hash by modulo’ing it with the hash table’s current capacity.
- Check if the bucket index is empty.
  - If not, got to probe for an empty spot.
- Use the hash table’s current length to create a value index.
- Store the value & hash using the value index. Save the value index into the bucket index.

Hash Table Insertion Algorithm Code Check

```c
void HashTableInsert(struct HashTable *hashtable, char const key[], int value) {
	if( hashtable->len >= hashtable->cap ) {
		fprintf(stderr, "error, hash table is full: '%s'\n", key);
		return;
	}
	
	size_t hash = string_hash(key);
	size_t bucket_index = 0;
	if( HashTableFind(hashtable, hash, &bucket_index) != -1 ) {
		fprintf(stderr, "error, duplicate key: '%s'\n", key);
		return;
	}
	
	bucket_index = hash % hashtable->cap;
	/// If bucket index isn't empty, probe for an open spot.
	/// resolves hash collision! whoohoo.
	while( hashtable->buckets[bucket_index] != -1 ) {
		bucket_index++;
		if( bucket_index >= hashtable->cap ) {
			bucket_index = 0;
		}
	}
	ssize_t value_index = hashtable->len; /// use the current length as a value index.
	hashtable->data[value_index]     = value; /// save value.
	hashtable->hashes[value_index]   = hash;  /// save hash.
	hashtable->buckets[bucket_index] = value_index; /// save the value index into the bucket index.
	hashtable->len++; /// increase the current length.
}
```

Hash Table
Delete/Remove Algorithm

- Most simple algorithm of the three fundamental operations.
- Use the Find/Lookup algorithm to retrieve the value and bucket indexes.
- Use the value indexes to set the hash and data at the value index to 0.
- Use the bucket index to set the bucket to -1, thus marking it as empty.

Removal/Delete Code Check

```
void HashTableRemove(struct HashTable *hashtable, char const key[]) {
	size_t hash = string_hash(key);
	size_t bucket_index = 0;
	ssize_t value_index = HashTableFind(hashtable, hash, &
	bucket_index);
	if( value_index == -1 ) {
		fprintf(stderr, "error, no key exists: '%s'\n", key);
		return;
	}
	hashtable->buckets[bucket_index] = -1;
	hashtable->data[value_index]     =  0;
	hashtable->hashes[value_index]   =  0;
	hashtable->len--;
}
```

Testing

```c
int main() {
	struct HashTable hashtable = HashTableInit();
	HashTableInsert(&hashtable, "text here", 1337);
	
	size_t text_bucket_index = 0;
	ssize_t text_value_index = HashTableFind(&hashtable,
	string_hash("text here"), &text_bucket_index);
	
	printf("text here's value: %i\n", hashtable.data[
	text_value_index]);
	
	HashTableRemove(&hashtable, "text here");
}
```

Conclusion

What have we learned?

How to use a hash function to create array indexes.

Using multiple arrays in tandem to store, retrieve, and delete data.

How to resolve hashing collisions.

Next Week's Engineering Club Coding Workshop: