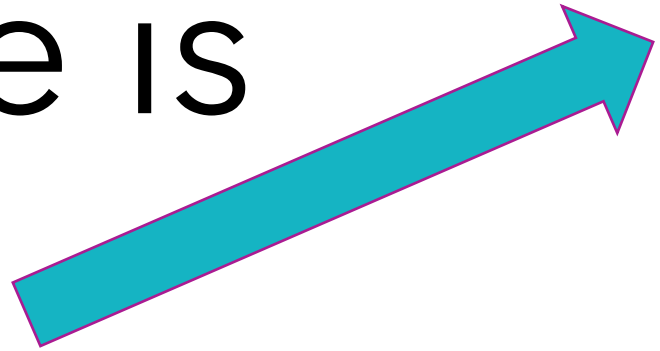
Hash Tables

Hash tables are a data structure used to store data in the form of {key:value} pairs.

The function that fills up the hash table is like this…



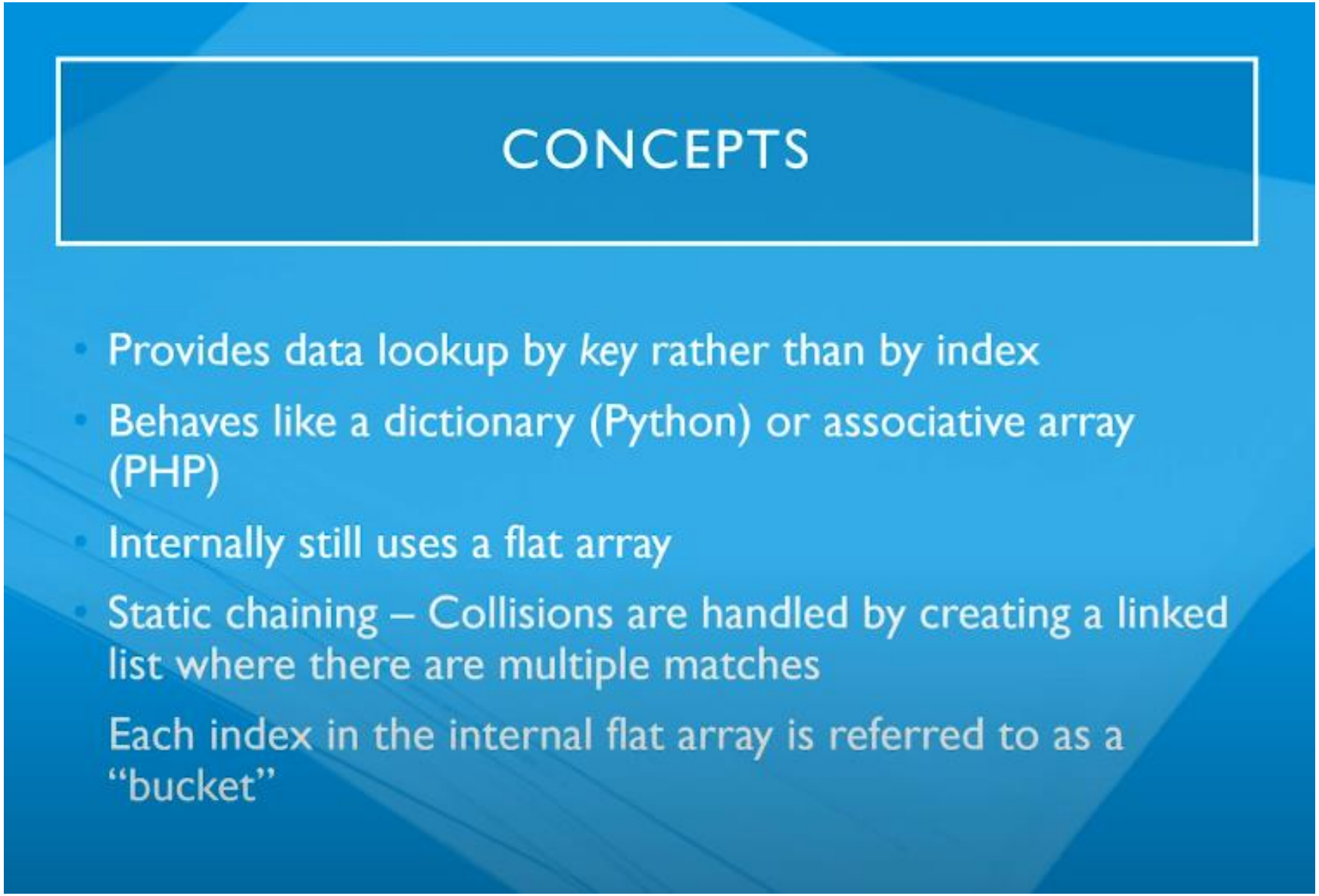
{ key : value }

Hash the (key)

Creates the index position

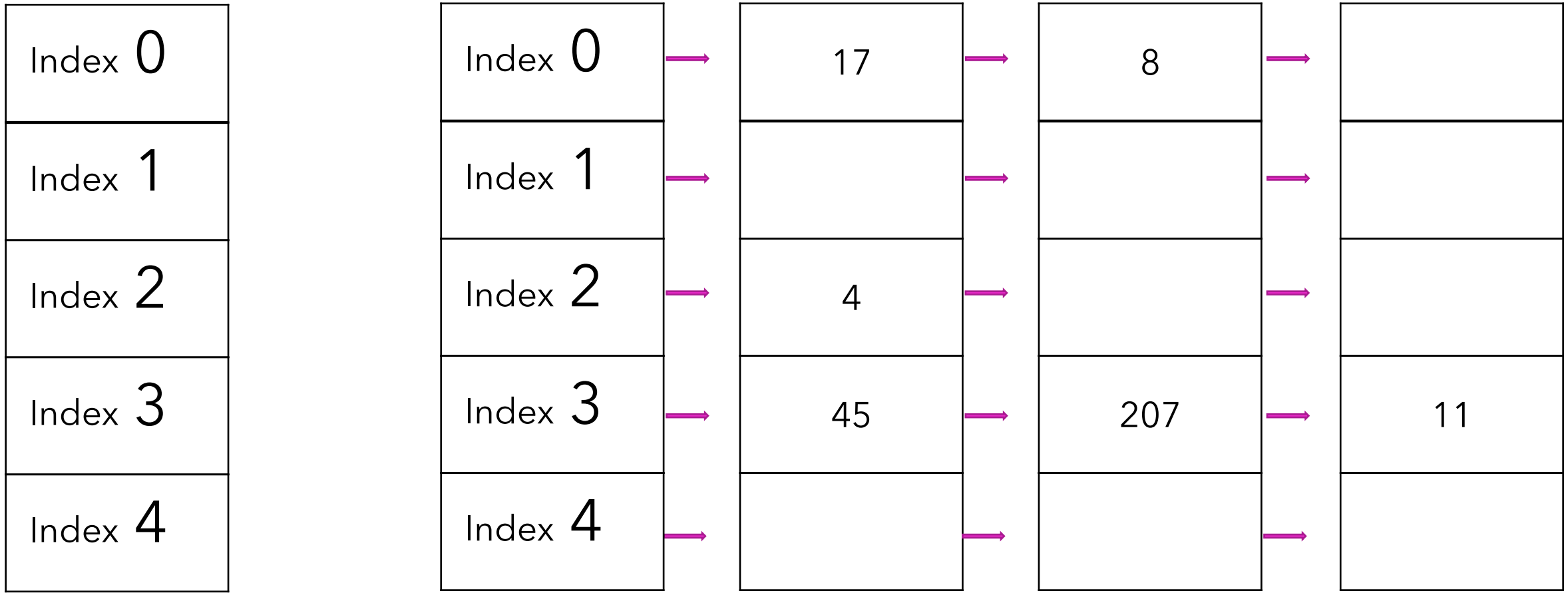
(some where between zero and the length of the hashtable)

What are our main concepts?



What If more than one key hashes to the same index?

Chaining - using a linked list to avoid collisions



We are given this code that will establish the data's ability to link if they hash to the same index

```
class HashTableEntry:
    """
    Linked List hash table key/value pair
    """
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.next = None
```

KEY

VALUE

NEXT

Let's start building our HashTable class!

- What needs to be in our __init__?
- Think about where items will be stored, how much can be stored there and keeping track of how close we are to the capacity.

```
class HashTable:
    """
    A hash table that with `capacity` buckets
    that accepts string keys

    Implement this.
    """

    def __init__(self, capacity):
        # Your code here
```

Code so far...

```
def __init__(self, capacity):

    self.capacity = capacity  # Number of buckets in the hash table
    self.storage = [None] * capacity
    self.item_count = 0
```

Next up… get_num_slots

```
def get_num_slots(self):
    """
    Return the length of the list you're using to hold the hash
    table data. (Not the number of items stored in the hash table,
    but the number of slots in the main list.)
```

What will this code look like?

We are returning the length of storage

```
def __init__(self, capacity):

    self.capacity = capacity  # Number of buckets in the hash table
    self.storage = [None] * capacity
    self.item_count = 0
```

```
def get_num_slots(self):
    return len(self.storage)
```

Next up… get_load_factor

- What is the load factor?

It is the **size**/quantity/item_count

Divided by

The **capacity**

```
def get_load_factor(self):
    """
    Return the load factor for this hash table.
```

How many items are being stored ÷ the capacity of the table

Code for get_load_factor

```
def __init__(self, capacity):

    self.capacity = capacity  # Number of buckets in the hash table
    self.storage = [None] * capacity
    self.item_count = 0
```

```
def get_load_factor(self):
    """
    return self.item_count / self.capacity
```

Now onto the Hashing!!

- Any function that maps arbitrarily long data to something of a set length in a non-reversible collision resistant way…
- So f(x) *maps* to y….. f(x) ----> y
- The length of ‘y’ is constant….. len(y)
- Need to be collision resistant …..
- Ideally… depending on the complexity of the hash, you should be able to assume that if the hashes are not the same, the contents are not the same either.

Lets look up the options provided in the project

2. Implement a good hashing function.

   Recommend either of:

   - DJB2
   - FNV-1 (64-bit)

   You are allowed to Google for these hashing functions and implement from psuedocode.

Lets move forward with DJB2

- Djb2, is based on integer arithmetic using the string values. It's an interesting case, because no one is entirely sure of exactly why it works better than other functions… but, we know that in practice, it does the trick! It was invented by a guy named Dan Bernstein.
  - Here's djb2 in Python:

```
def djb2(self, key):
    str_key = str(key).encode()
    hash_value = 5381
    for b in str_key:
        hash_value = ((hash_value << 5) + hash_value) + b
    hash_value &= 0xffffffff
return hash_value
```

Lets breakdown this hash function…

```
def djb2(self, key):

    str_key = str(key).encode()

    hash_value = 5381

    for b in str_key:
        hash_value = ((hash_value << 5) + hash_value) + b

    hash_value &= 0xffffffff

    return hash_value
```

Stringify the key

Encode the stringified key to get bytes

Start from this larger, prime number.

preform a bit shift

sum up the value

Only keep 32 bits

BONUS INFO!!
32 bit of information but it is how your processor treats. However, in short **0XFFFFFFFF** is (2^32)-1
i.e 4294967295 in decimal notation

4. **Bitwise operators:** Bitwise operators acts on bits and performs bit by bit operation.

OPERATOR	DESCRIPTION	SYNTAX
&	Bitwise AND	x & y
|	Bitwise OR	x | y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

We will figure out our hash_index

```
def hash_index(self, key):
    """
    Take an arbitrary key and return a valid integer index
    between within the storage capacity of the hash table.
    """
```

```
return self.djb2(key) % self.capacity
```

Continue on your own to figure out the rest!

```
def put(self, key, value):
```

```
def delete(self, key):
```

```
def get(self, key):
```

```
def resize(self, new_capacity):
```

Good luck to you all!!!