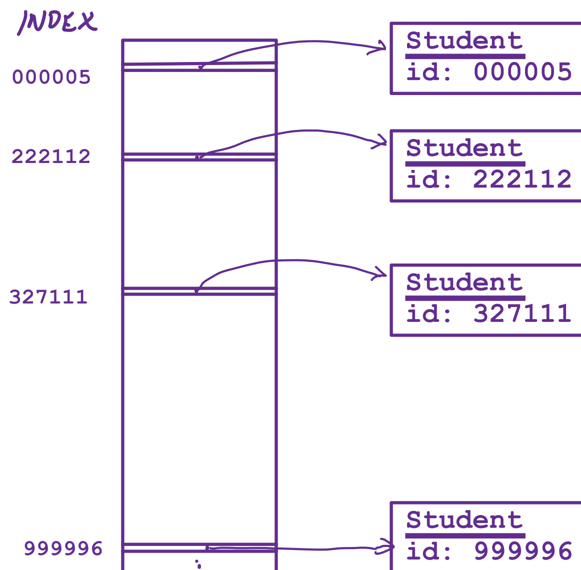
Hashmaps and how they work

Motivation: Suppose a University wants to store many Student objects and look them up by their ID number (eg. 1234567). Some options:

LINKED LIST

With a Linked-list structure (whether implemented via a Mutable/ImmutableList, Java LinkedList, etc.), we would need to search all of the nodes until we find one matching the student ID we want
=> Lookup has O(N) runtime

ARRAY




```
Would look up student like
this:

Student s = arr[id]
```

WITH ARRAY:
LOOKUP IS O(1)
=> CONSTANT

Idea: With an array-based structure (whether implemented as a plain array, or Java ArrayList), we *could* use the student's ID number as an index into a super-large array
=> Lookup time is constant, O(1)

However, there are some drawbacks....
- Wasting lots of memory: many array spaces would go unused!
- Key == array index => what if we needed to look up by some non-integer value (eg. the student's name?)
- What happens when students leave the University? Probably can't reuse ID numbers, so array would always grow in size!

How else could we leverage an array's constant-time access to look up objects?

Working with HashMaps

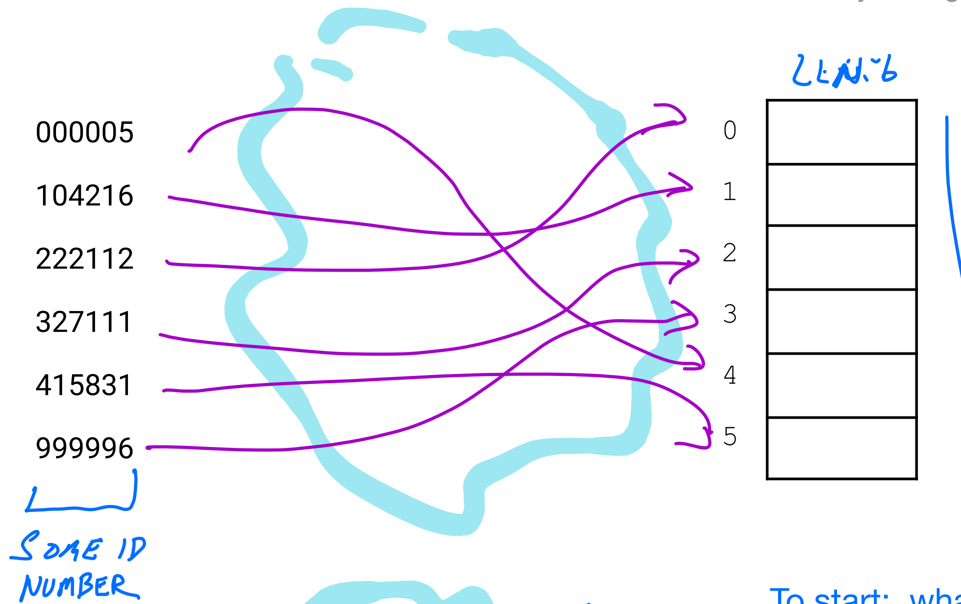
```
// Map lab times to room numbers
HashMap<String, String> labRooms = new HashMap<String, String>();

// Associate this key with this value
labRooms.put("Mon 4-6", "CIT219");
labRooms.put("Tue 6-8", "CIT501");

labRooms.get("Mon 4-6"); // Returns "CIT219"

// Changes the value mapped to this key
labRooms.put("Mon 4-6", "CIT444");  //
labRooms.get("Mon 4-6");

labRooms.get("Wed 8-10"); //

if(labRooms.containsKey("Mon 4-6")) {
   // . . .
}
```

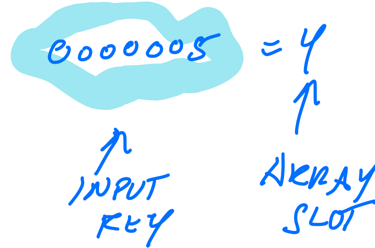
HashMaps, in practice
- Map a “key” to a “value” (HashMap<K, V>)
- Given key, hash map provides constant time (O(1)) access to lookup value
- There can be at most one value per unique key
- Key, Value can be any Java type

HashMap<String, List<String>> // Could have one key map to multiple things this way (still one object)

Speeding Up Access to ~~Accounts~~ Students

(For the remainder of these notes, we'll focus on how a hash map is implemented. As a programmer using hash maps, it's not necessary to understand these details—but we can learn a lot about data structures by seeing how hash maps work!)





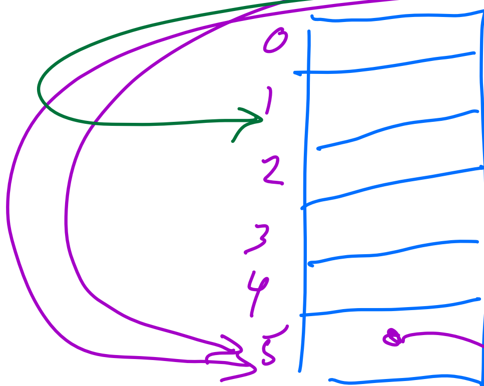
To start: what if we had some mathematical function that could turn account numbers (an integer) into an array slot?

We can do this using modulo with the size of the array—

Modulo (%): remainder when you do division

```
0000005 % 6 = 5
0000019 % 6 = 1
0000011 % 6 = 5
```

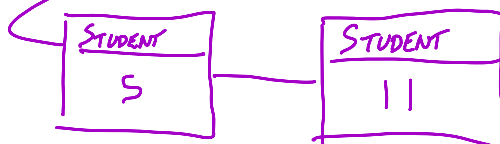
```
AcctNum % Array Size
=> returns number in range 0..(array size - 1)
```



=> However, with modulo it's possible to have multiple keys map to the same slot!!!

So how does it work?

- **Internally, HashMap is based on an array**
- **Keys are mapped to slots using %**
- **Each slot contains a linked list of entries that mapped to that slot**



What would it look like to implement get()?
(Partially)

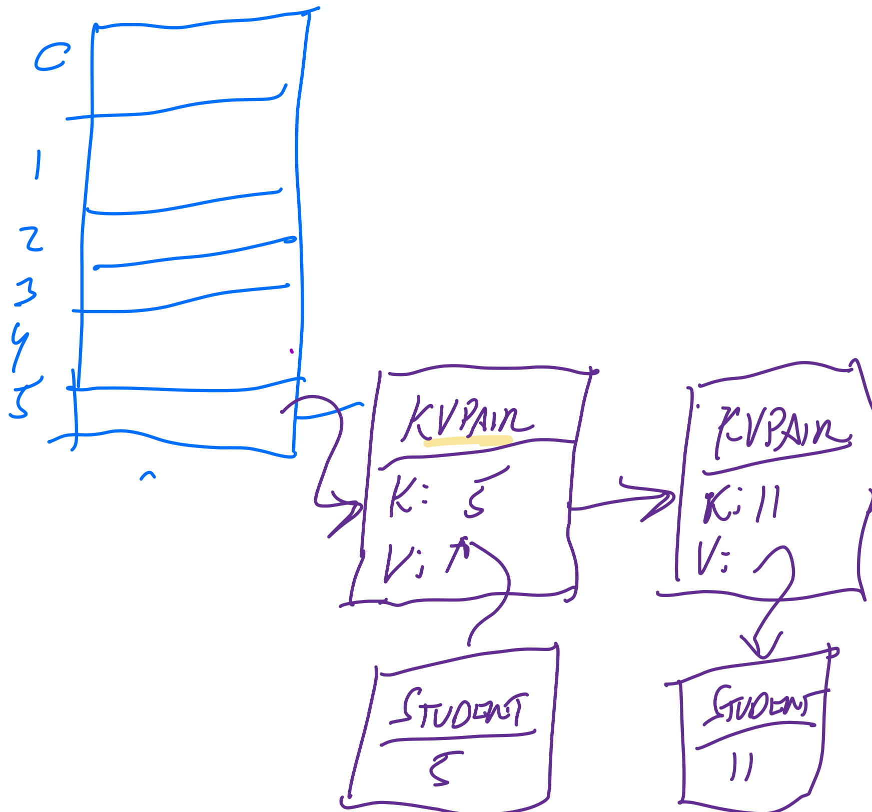
```
Get(K)
  slot = k % (size of array)
  LinkedList l = array[slot];

  for(Account a : l) {
      if a.idNum == k
         return a
  }
```

① FIND SLOT

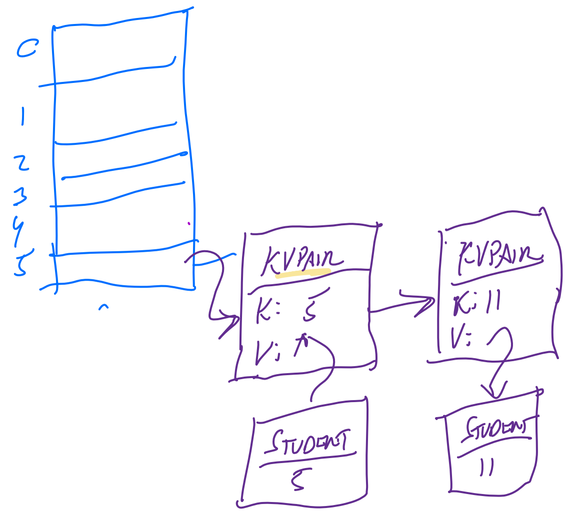
② SEARCH LINKED LIST FOR.... THE KEY?

Problem: what goes in the linked list?
We need to know if the item in the list matches the key k, so we need to store both the key (which tells us what item it is) and the value (the thing we want to look up) in the hashmap!

Need to keep track of both key and value in linked list
=> LinkedList contains Key Value pairs (KVPair)
eg. LinkedList<KVPair>[]



Q: Why does the KVPair need to store the key? Can't we figure out the student ID number from the Student object?

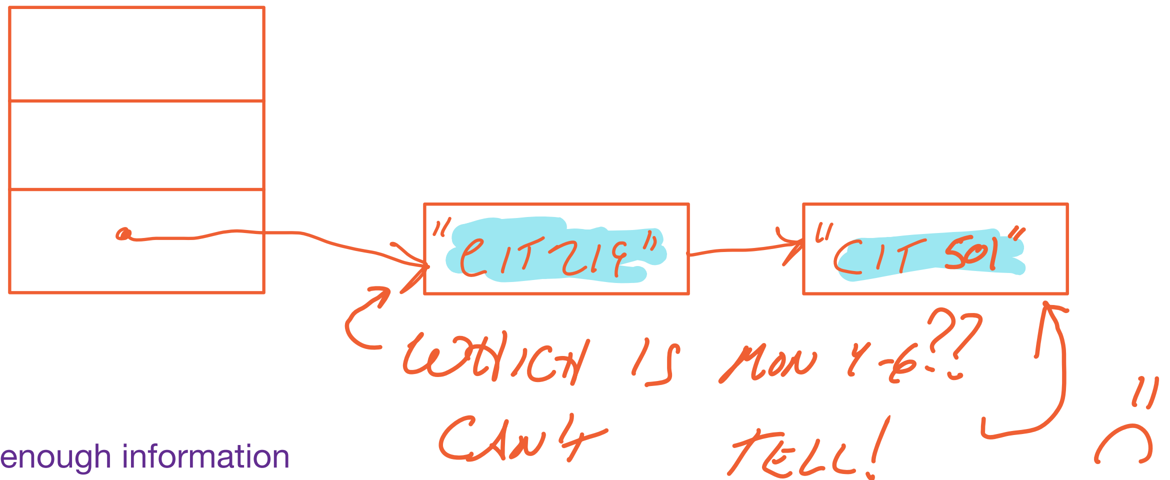


This approach may work in this example. However, the key and value could be any Java object, and they might not relate to each other, so, when implementing a *generic* hash table, we can't make assumptions that the values will have this info.

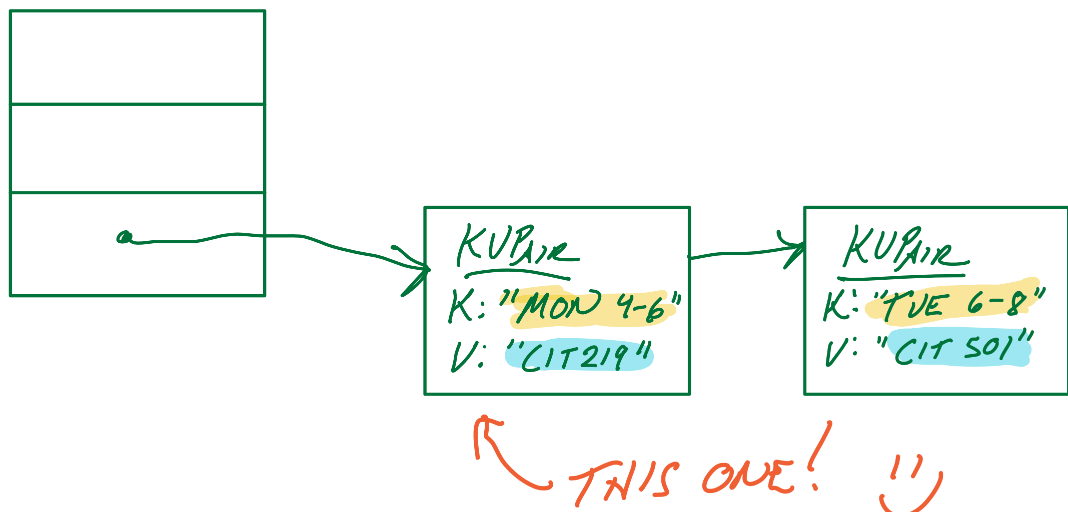
To see this, consider the earlier example of mapping lab times ("Mon 4-6") to rooms (eg. "CIT 444"):

```
// Associate this key with this value
labRooms.put("Mon 4-6", "CIT219");
labRooms.put("Tue 6-8", "CIT501");
```

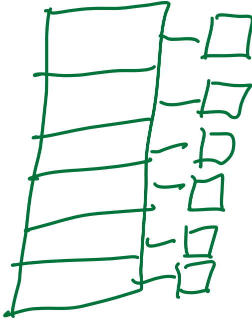
Suppose we call labRooms.get("Mon 4-6")
If we stored just values in the hashmap (ie, no KVPairs)....

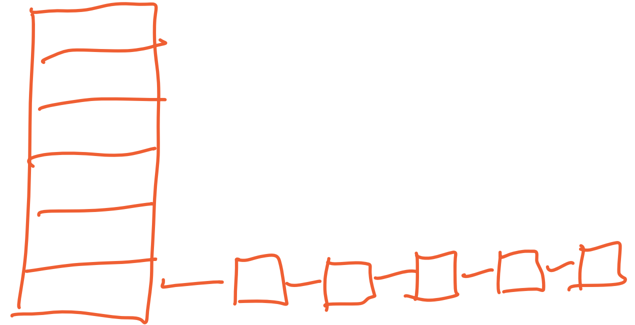


With KVPairs, we have enough information to tell which value maps to which key:



What about runtime?





Each element in its own array slot,
no wasted (empty) array slots

Lots of elements in one array slot
(long linked list => long search time)
Many wasted array slots

Ideally, want lists to be small so search is fast

Things that we can control to help this happen:
- Initial array size (in practice, a prime number)
- If/when you resize the map (75% full)
- Hashing function (math)

Need: a way to turn an arbitrary object (String, Course, Account, whatever) into an integer
=> integer, can do % => get to a slot

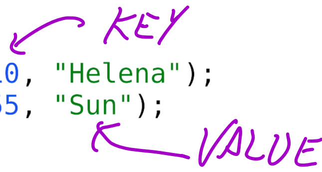
How to handle keys that aren't integers?
Every object has a function called hashCode()

```
public int hashCode() {
}
```

EXAMPLE FOR WORKING W/ HASHMAPS

(Additional notes page from a previous semester)

```java
HashMap<Integer, String> offices = new HashMap<Integer, String>();
```




```java
offices.put(210, "Helena");
offices.put(255, "Sun");
```

VALUE

Programmer perspective:

- Each key can only map to one value in the HashMap
- For all operations (get, put, containsKey, …), Java calls hashCode() on the key to get an integer value (the "hash code")—if keys have the same hash code, they will map to the same value
- Java has already has a hashCode() for built-in types (Integer, String, …)
  If you are making your own class, you should write your own hashCode() method (just like equals())

IMPLEMENTATION PERSPECTIVE

Example: what if we want to add some elements:
put(250, "A");
put(255, "B");
put(230, "C");

What happens inside the hash table?
(ie, hidden from the programmer)

INSIDE A HASH TABLE: INSERTING VALUES (PUT)

EXAMPLE FROM FULL NOTES

```java
public void insert(K key, V value) {
  // hash the key and apply compression
  int index = key.hashCode() % size;
  // store the value under the key's index
  this.contents[index].addFirst(value);
}
```

PUT

① Use hashCode() to get a unique hash value for this key. hashCode() always returns an int.
(If the key is an Integer, hashCode() just returns the integer itself—like you see here.)

② Since there are more possible hash values than array slots, we "compress" the hash value to pick or a slot for it in the array.
Usually we use modulo, ie: hash % size

Compression means that more than one key may occupy to the same array slot, even when their hashCodes are different values

③ Why does this work? Each array slot contains a list or (key, value) pairs that were mapped to that slot.

To add an element to the hash map, put() (also called insert()) adds the new element to this list.

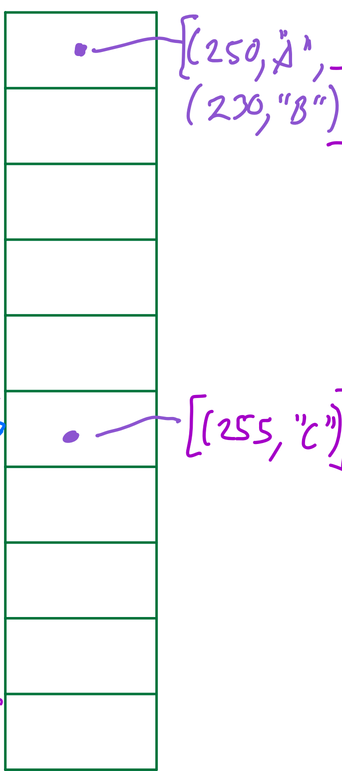
KEY | HASH CODE | ARRAY SLOT (INDEX)

250 → 250 % 10 = 0

230 → 230 % 10 = 0

255 → 255 % 10 = 5

Modulo (%) is the remainder after doing division:
150 % 10 => 0
11 % 10 => 1
52 % 10 => 2
9999 % 10 => 9



For an example with get(), see the full typed notes

```java
public interface IDictionary<K, V> {
  public V lookup(K key) throws KeyNotFoundException;
  public V update(K key, V value) throws KeyNotFoundException;
  public void insert(K key, V value) throws KeyAlreadyExistsException;
  public V delete(K key) throws KeyNotFoundException;
}

public class Chaining<K, V> implements IDictionary<K, V> {

   private static class KVPair<K, V> {
       public K key;
       public V value;
   }

   public Chaining(int size) { . . . }

   private KVPair<K, V> findKVPair(K key) throws KeyNotFoundException {
       . . .
   }

   public V lookup(K key) throws KeyNotFoundException {
       KVPair<K, V> pair = findKVPair(key);
       return pair.value;
   }

   public V update(K key, V value) throws KeyNotFoundException {
       KVPair<K, V> pair = findKVPair(key);
       V oldValue = pair.value;
       pair.value = value;
       return oldValue;
   }

   public void insert(K key, V value) throws KeyAlreadyExistsException {
      . . .
   }

   public V delete(K key) throws KeyNotFoundException { . . . }
   public boolean equals(Object ht) { . . . }
   public String toString() { . . . }
}
```