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)

How to Build A Hash Map?

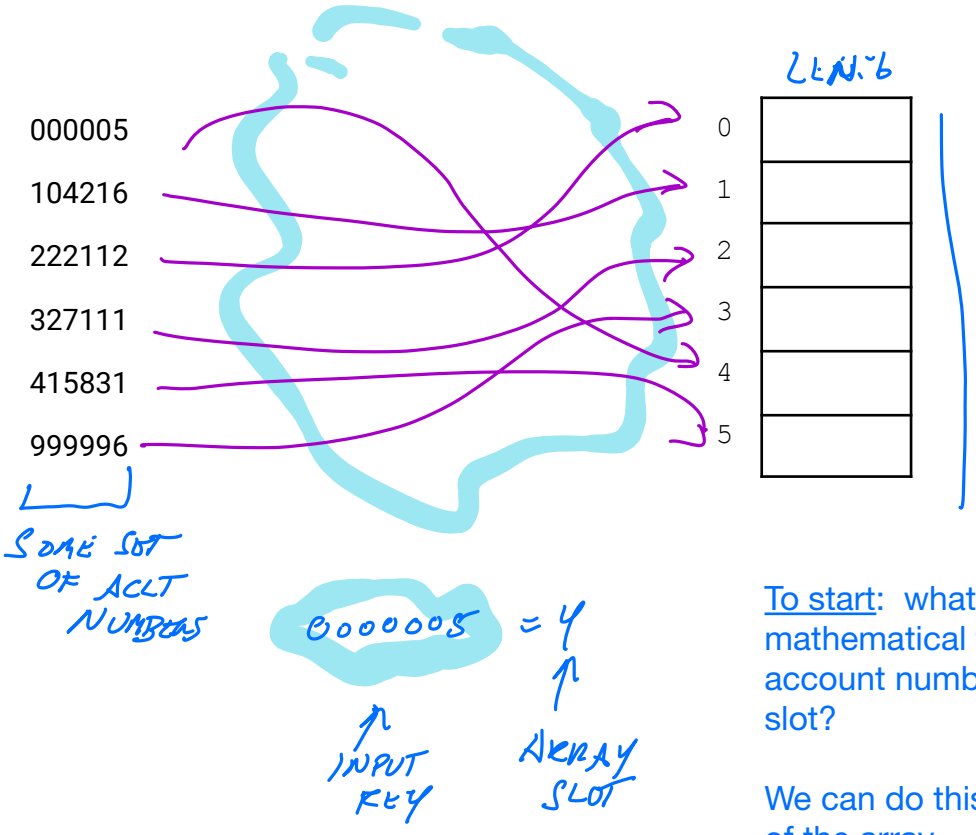
(For the remainder of these notes, we'll focus on how a hash map is implemented. As a programmer using hash maps, it's not necessary to understand these details—but we can learn a lot about data structures by seeing how hash maps work!)

Consider the array-based version we were discussing for account numbers: the account number serves as an array index to get an Account object

=> This has constant time lookup, BUT
- Waste memory when removing accounts
- Need array as big as highest account number
- "Key" == array index…
  what about things that aren't integers?

=> This approach gives us the performance we want, but has a lot of setbacks. How can we do better?

Speeding Up Access to Accounts



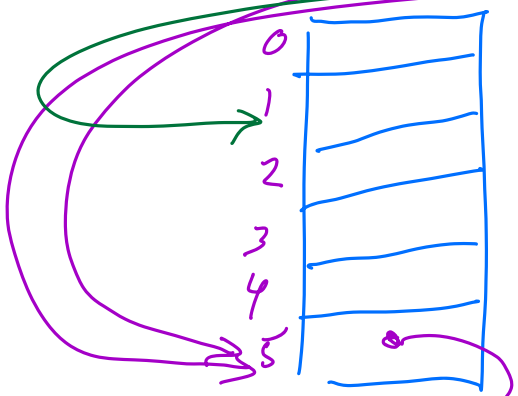
To start: what if we had some mathematical function that could turn account numbers (an integer) into an array slot?

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

Modulo (%): remainder when you do division

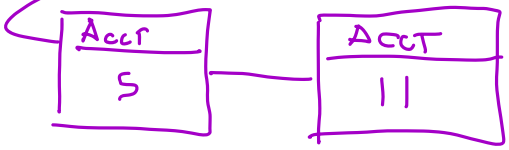
```
0000005 % 6 = 5
0000019 % 6 = 1
0000011 % 6 = 5
```

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)

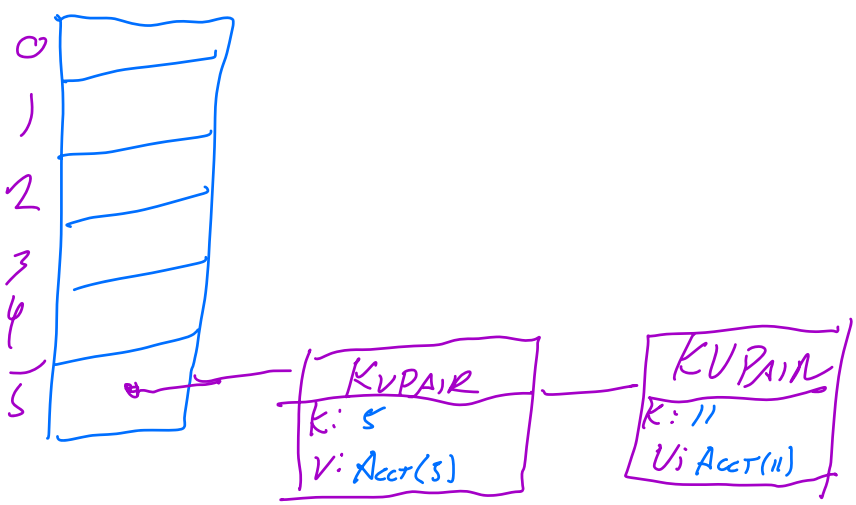
```
Get(K)
  slot = k % (size of array)
  LinkedList l = array[slot];

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

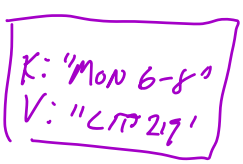
1) FIND SLOT

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!

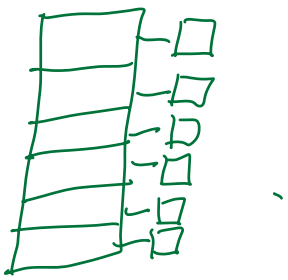
Need to keep track of both key and value in linked list
=> LinkedList contains Key Value pairs (KVPair)





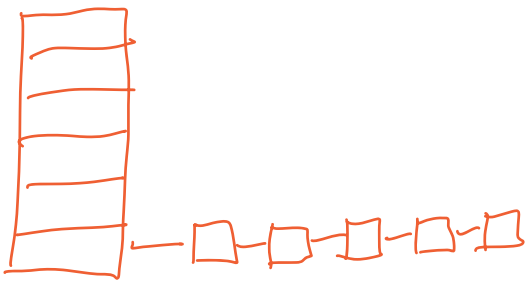
What about runtime?

MOST OPTIMISTIC CASE



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

MOST PESSIMISTIC CASE



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() {
}
```

```
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() { . . . }
}
```