CSCI 136
Data Structures &
Advanced Programming

Hashtables & Collisions

Video Outline

- Hashtables
  - Recap “big picture”
- Collision resolution strategies
  - External chaining
  - Linear probing/open addressing

Hash Table Implementation

General idea: Use an array to represent "bins"

- `V get(K key)`:
  - use key's hashcode to identify bin (% array length)
  - Search bin for item with matching key
- `V put(K key, V val)`:
  - use key's hashcode to identify bin
  - Search bin for item with matching key:
    - If a match exists, replace old value with `val`
    - If no match exists, add new (key,value) pair

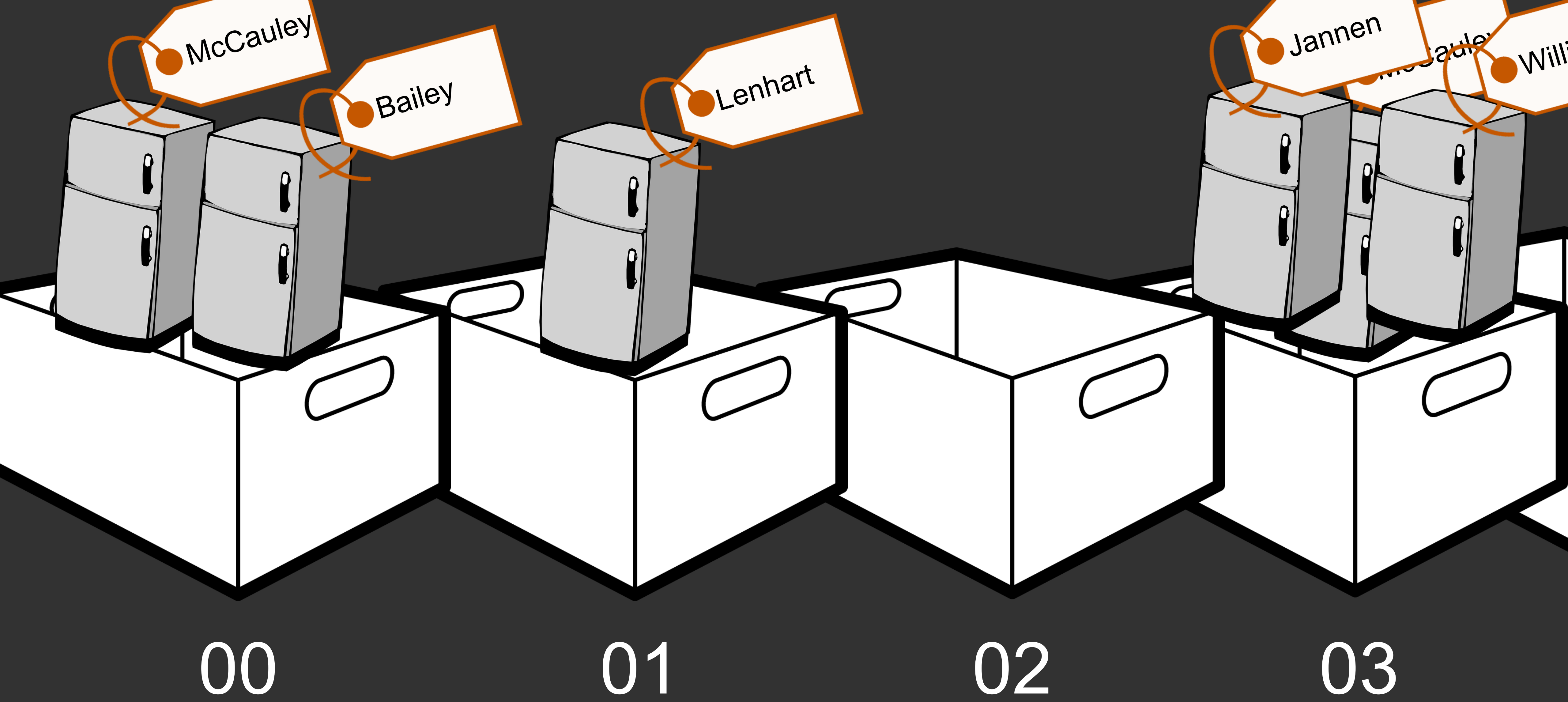
Notes on hashCode()

- We can use mod (%) to map an into to an array index
  - `array[o.hashCode() % array.length] = o;`
- What does a hashCode() return?
  - 32 bit integer
  - Can be negative!
  - This gives an array out of bounds exception
- Let's do the following:
  - `array[Math.abs(o.hashCode() % array.length)] = o;`

Navigating HashTable Collisions

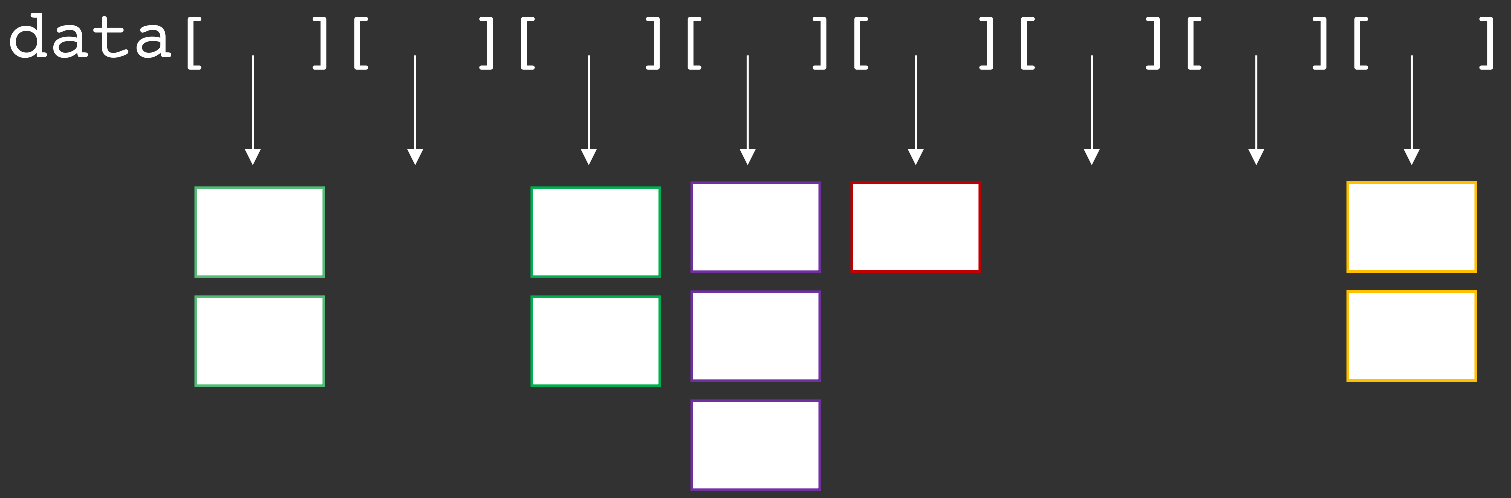
- Problem: *collisions* occur when two unique items are mapped to the same bin
  - This is a problem in arrays, because we can only store one item per index
  - Thus, collision management isn't *just* a performance issue, it is a *correctness* issue

- We'll discuss two strategies to resolve collisions
  - External chaining
  - Linear probing (sometimes called open addressing)


McCauley
Bailey
Lenhart
Jannen
00
01
02
03

Idea 1: External Chaining

- Idea: Instead of mapping individual items to bins, we store a *list* in each bin



- `get()`, `put()`, and `remove()`, then, need to (a) identify the bin, then (b) check bin's list

Hash Table Implementation w/ External Chaining

```
public V get(K key) {
    int bin = Math.abs(key.hashCode() % table.length);
    // search for value in bin
    Association<K, V> temp = new Association<K,V>(key);
    Association<K, V> ret = table[bin].remove(temp);

    if (ret != null) { // if found, return value
        // restore value to bin so don't modify table
        table[bin].add(ret);
        // return the value we found
        return ret.getValue();
    }
    return null;
}
```

Hash Table Implementation w/ External Chaining

```
public V put(K key, V val) {
    int bin = Math.abs(key.hashCode() % table.length);
    // search for old value in bin and remove if found
    Association<K, V> toAdd = new Association<>(key, val);
    Association<K, V> old = table[bin].remove(toAdd);

    // add our new K,V pair
    table[bin].add(toAdd);

    if (old != null) {
        // if old value found, return val we're replacing
        return old.getValue();
    }
    // not found, return null
    return null;
}
```

Downsides to External Chaining

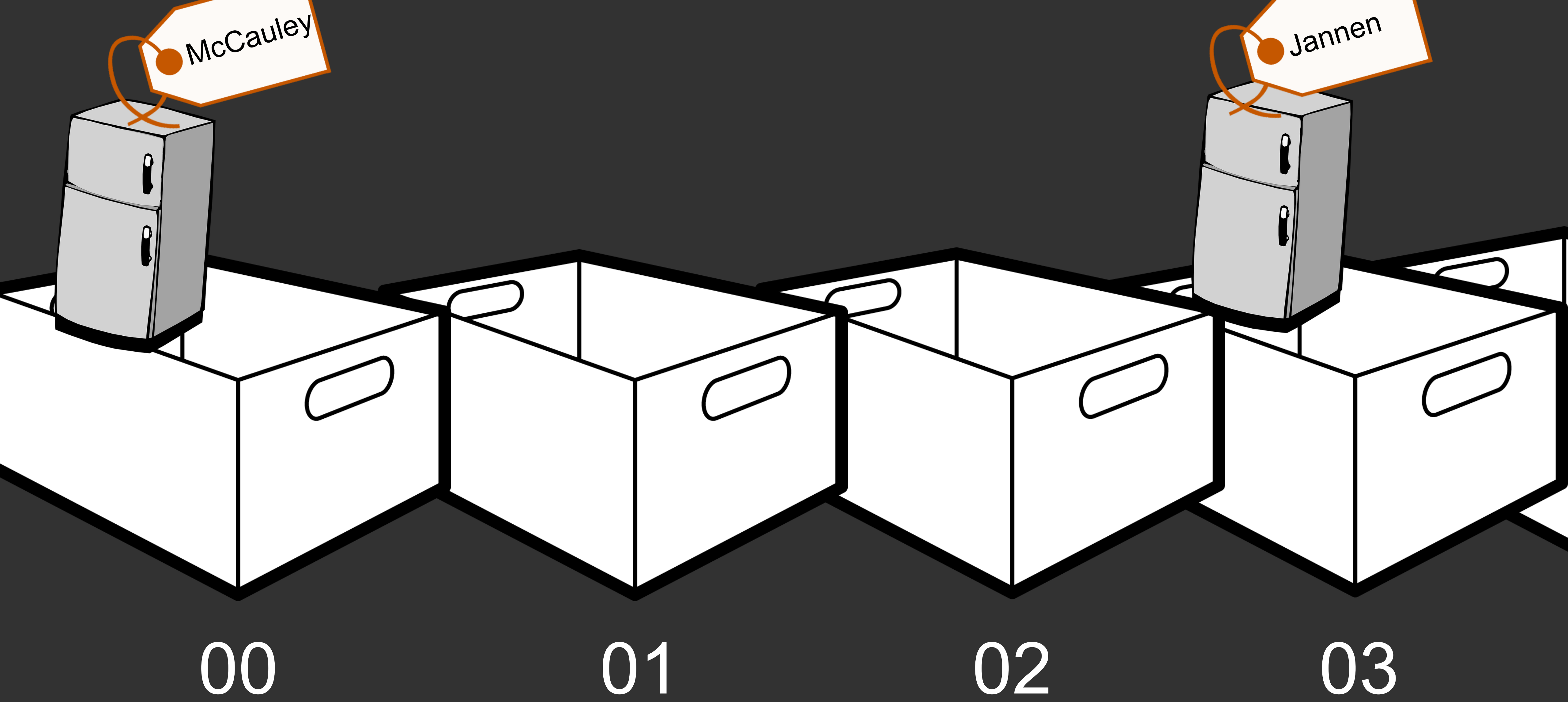
- Each slot in our Hashtable's array stores a list, even if the slot is empty
  - This consumes extra space
- Potentially poor *locality*
  - Not something we've talked about so far in this course, but a general rule of thumb: it is faster to access things that are near to each other than it is to access things that are far away.
    - Array elements are always contiguous (near)
    - List elements may be scattered throughout mem (far)

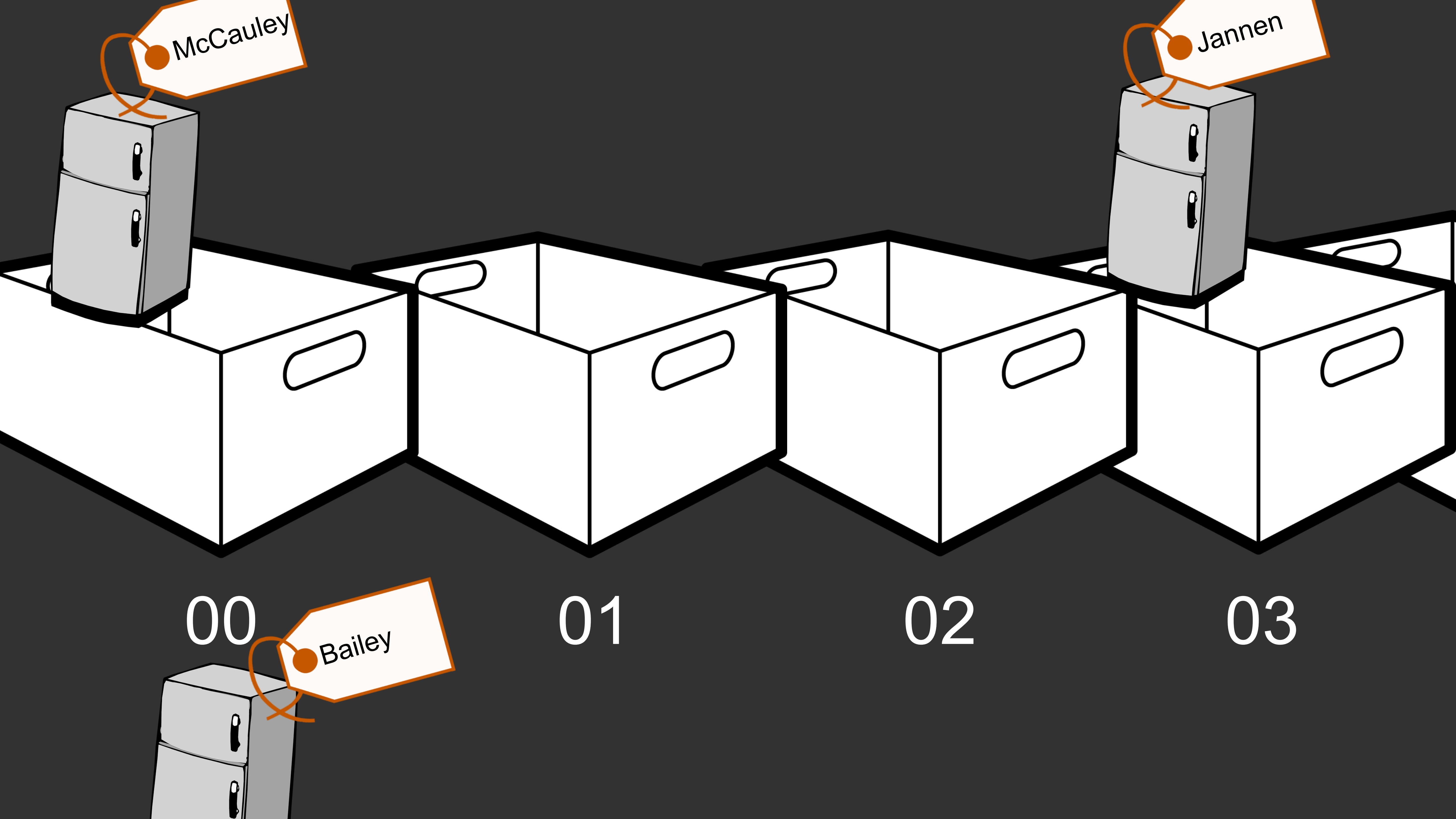
Rethinking Collisions

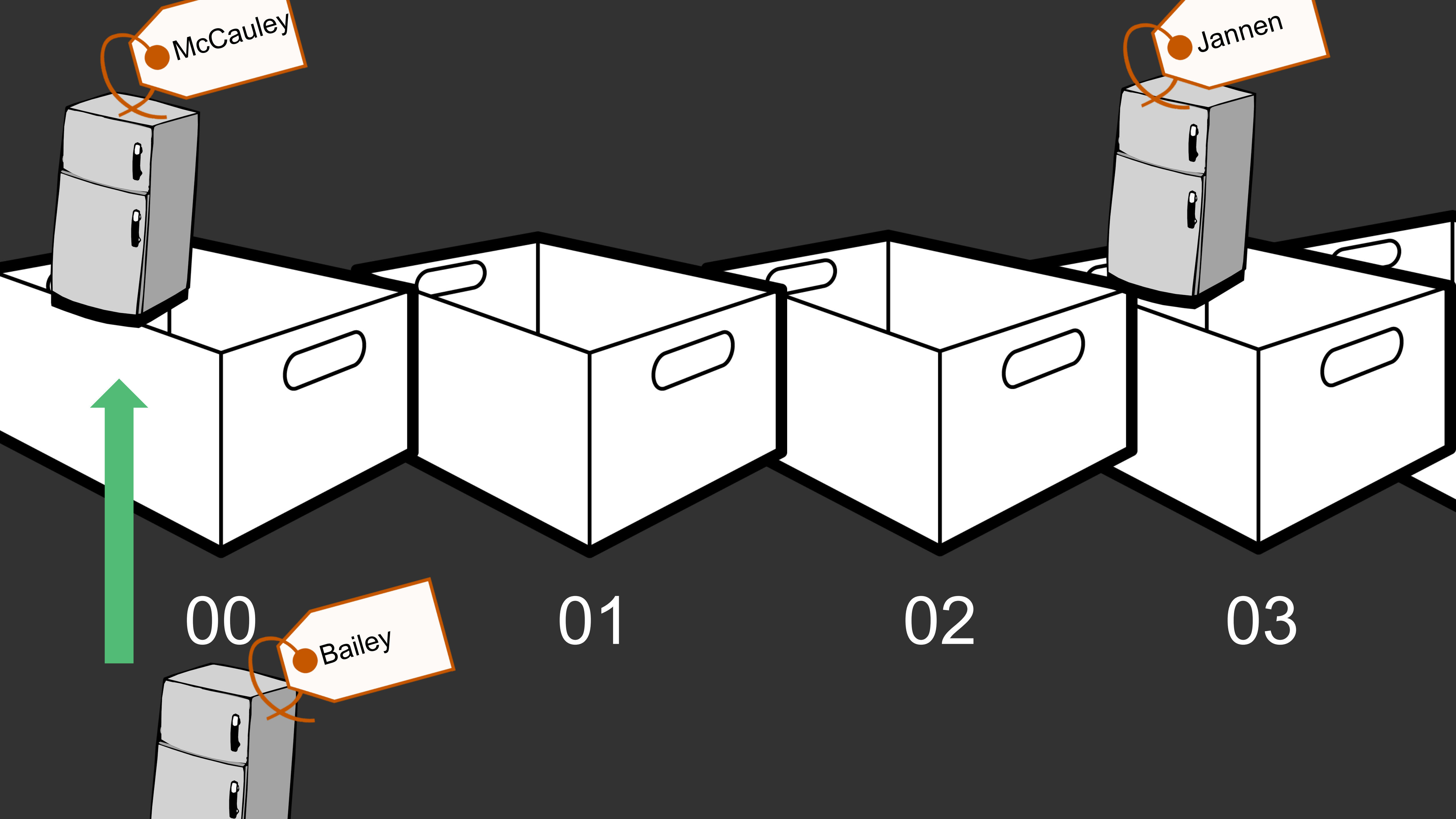
- Let's define an item's canonical slot as the place where the item belongs *ignoring collisions*
  - If no two items map to the same canonical slot, we don't have any problems
  - If multiple items do map to the same canonical slot, we need to figure out:
    - Among the set of colliding items, which one belongs in the canonical slot
    - Where do the "losing" items belong so that we still can find them in the future?

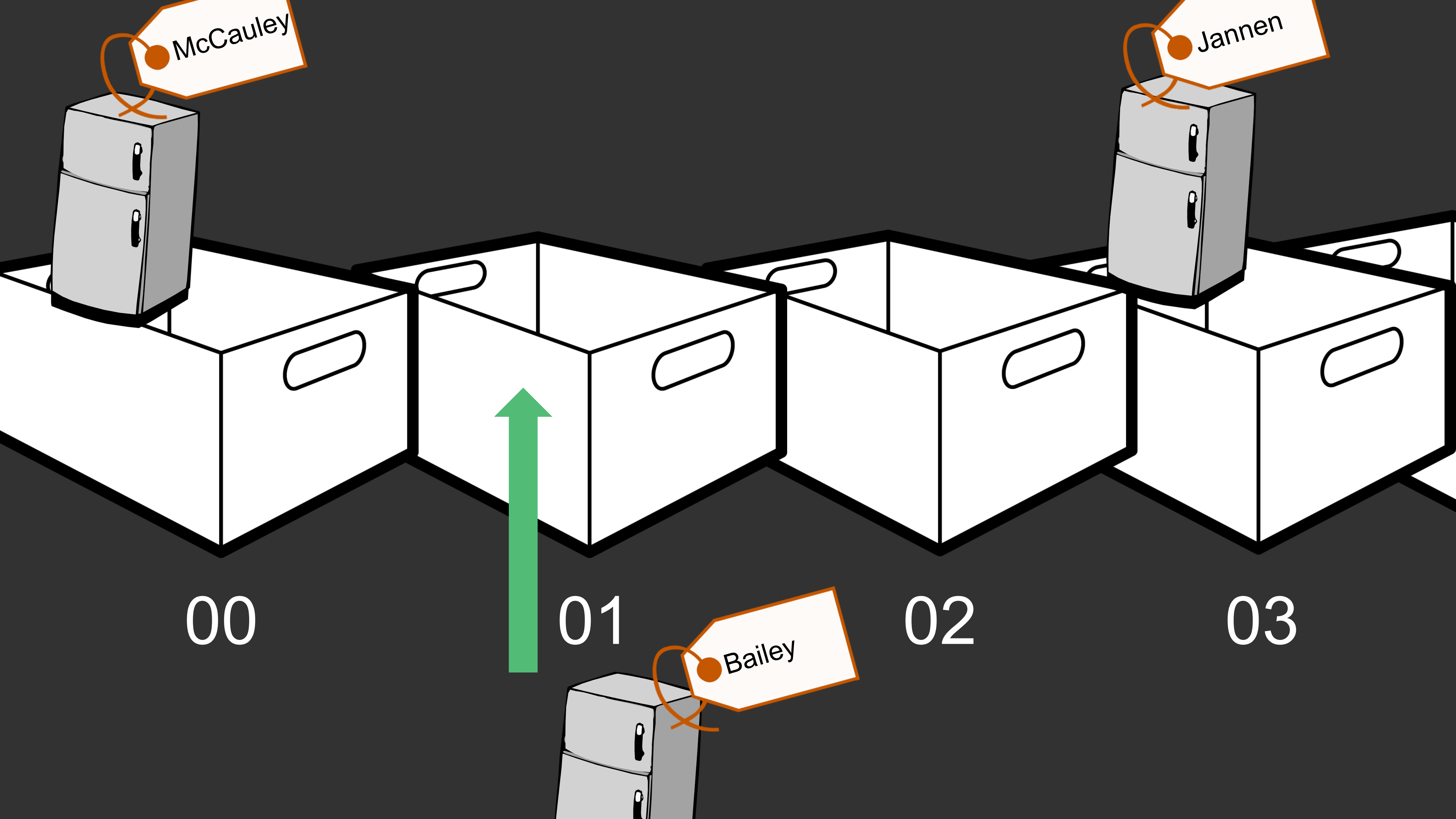
Linear Probing

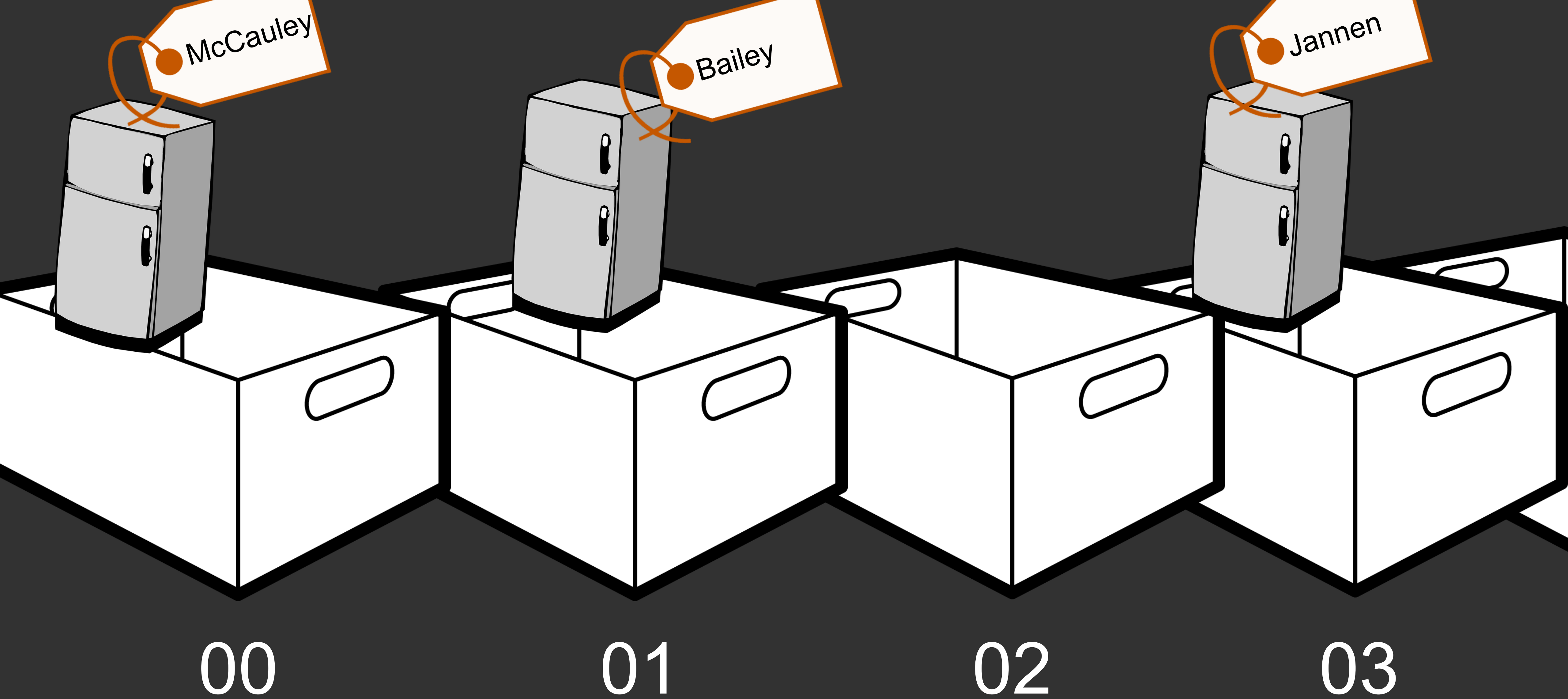
- General idea: store each key-value pair in the first open slot on or after its canonical slot
- Insertion: If a collision occurs at a given bin, just scan forward (linearly) until an empty slot is available, and store it there
  - We "wrap around" at the end of the array
  - We will call a contiguous region of full bins a *run*
- Lookup: To find a KV-pair, scan linearly through the run until you find it or reach the end of the run

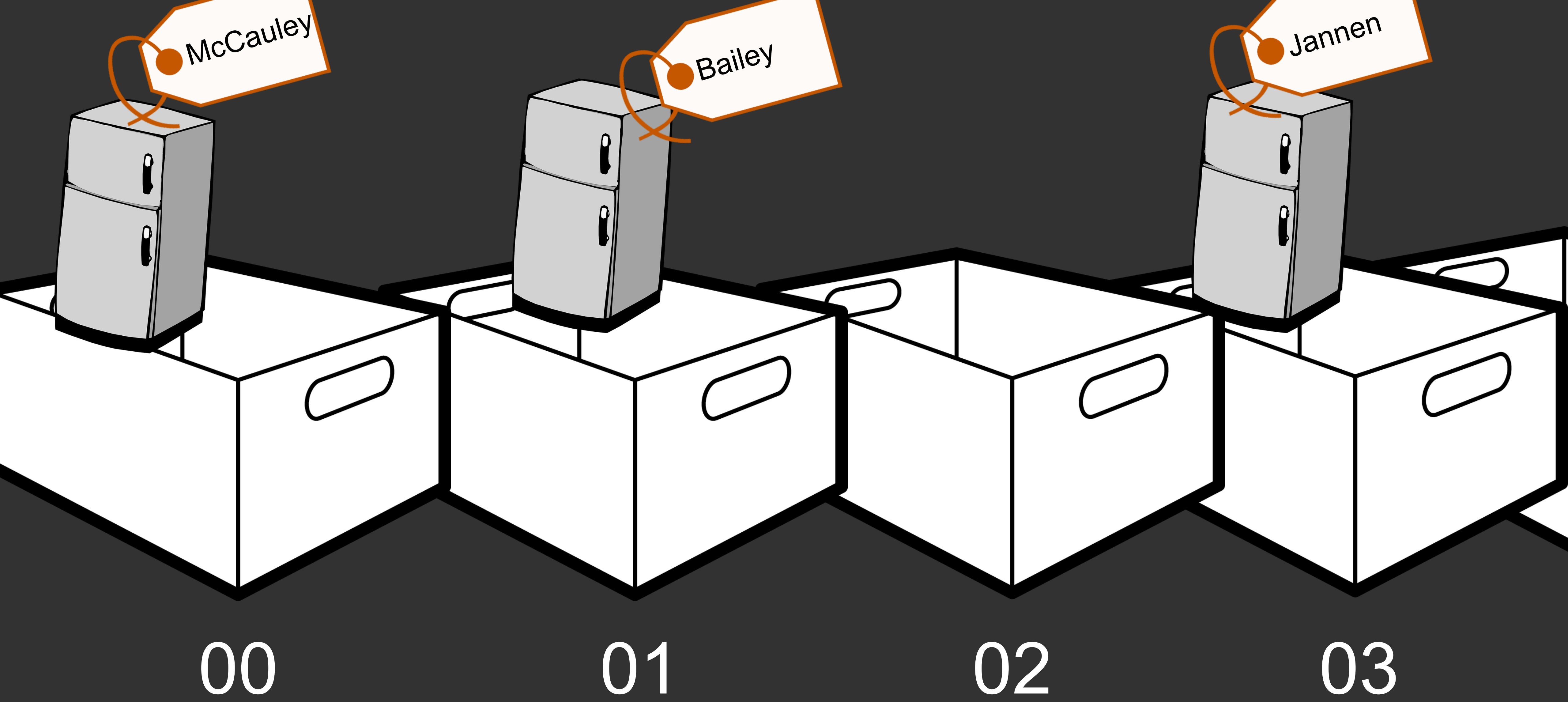

McCauley
Jannen
00
01
02
03


McCauley
Jannen
Bailey
00
01
02
03


McCauley
Jannen
Bailey
00
01
02
03


McCauley
Jannen
00
01
02
03
Bailey


McCauley
Bailey
Jannen
00
01
02
03



Bailey’s phone number ended in 00, but his freezer winds up in 01

First Attempt: put(K)

```
public V put (K key, V value) {
   int bin = Math.abs(key.hashCode() % data.length);
   while (true) {
      Association<K,V> slot = (Association<K,V>) data[bin];
      if (slot == null) { // Found an empty bin!
         data[bin] = new Association<K,V>(key,value);
         return null;
      }
      if (slot.getKey().equals(key)) { // already exists!
         V old = slot.getValue();
         slot.setValue(value);
         return old;
      }
      // Bin filled. Check the next bin…
      bin = (bin + 1) % data.length;
   }
}
```

First Attempt: get(K)

```
public V get (K key) {
   int bin = Math.abs(key.hashCode() % data.length);
   while (true) {
      Association<K,V> slot = (Association<K,V>) data[bin];
      if (slot == null) // Found an empty bin. End of the run
         return null;

      if (slot.getKey().equals(key))
         return slot.getValue();

      bin = (bin + 1) % data.length;
   }
}
```

Linear Probing Gotchas

- Let’s look at NaiveProbing.java
  - We specify a dummy hash function: index of first letter of word
  - Initial array size = 8
  - Add “atlanta” to hash table
  - Add “detroit”
  - Add “queens”
- What happens when we remove “atlanta”, and then lookup “queens”?
  - Our *run* was broken up!
  - Now get() won’t work correctly

Linear Probing Challenge

- When we delete an element from a run, we create a “hole”
  - Challenge: How do we tell if the run has ended, or if the hole is from a deletion?
  - Solution: Insert a “placeholder”
    - If we see the placeholder during a lookup, we treat it as a collision, and keep scanning until we find a true hole
    - If we see the placeholder during insertion, we treat it as an open spot
      - (We must still scan the whole run to see if our key is present)

HashAssociation.java

```
public class HashAssociation<K,V> extends Association<K,V>
{
    protected boolean reserved;
    /*...*/
    public boolean reserved()
    {
        return reserved;
    }

    public void reserve()
    {
        Assert.pre(!reserved,"HashAssociation reserved twice.");
        reserved = true;
    }
```

Hashtable.java

```
protected int locate(K key) {
        // initial hash code
        int hash = Math.abs(key.hashCode() % data.size());
        // keep track of first unused slot, in case we need it
        int reservedSlot = -1;
        boolean foundReserved = false;
        while (data.get(hash) != null) {
                // loop until end of run OR find target key
                if (data.get(hash).reserved()) {
                        // remember reserved slot if we fail to locate value
                        if (!foundReserved) {
                                reservedSlot = hash;
                                foundReserved = true;
                        }
                } else {
                        // value located? return the index in table
                        if (key.equals(data.get(hash).getKey())) return hash;
                }
                hash = (1+hash)%data.size();
        }
        // return first empty slot we encountered
        if (!foundReserved)
                return hash;
        else
                return reservedSlot;
}
```

Hashtable.java

```java
public V get(K key) {
        // find bin where key lives (after resolving collisions)
        int hash = locate(key);

        // if the key is not found, the resulting location
        // is either null or “RESERVED”
        if (data.get(hash) == null || data.get(hash).reserved())
                return null;

        // key was found, so return associated
        return data.get(hash).getValue();
}
```

Hashtable.java

```
public V remove(K key) {
        // find bin where key lives (after resolving collisions)
        int hash = locate(key);

        // if the key is not found, the resulting location
        // is either null or “RESERVED”
        if (data.get(hash) == null ||
            data.get(hash).reserved())
               return null;

        // key was found, so remove, then return old value
        count--;
        V oldValue = data.get(hash).getValue();
        data.get(hash).reserve();
        return oldValue;
}
```

Linear Probing Observations

- Code becomes more complicated, but manageable
- The length of a run dictates the performance
- Reserving elements does not “shrink” the run–it defers the work to other operations
  - Keeping our runs small is important, so we may want to reexamine design decisions if we expect a lot of deletions

Linear Probing Observations

- Downsides of linear probing?
  - What if array is almost full?
    - Loooong runs for every lookup…
    - Items out of place if we don't re-index after removing (placeholders are correct, but they defer work)
- Does external chaining avoid these problems?
  - Recall, *External chaining* "groups" objects with the same hash value together in same bin in a `Collection` (usually a SLL)
    - Only scan collisions, not an entire run
    - Never scans more items than linear probing
    - Worse cache behavior (locality)

Summary: Probing vs. Chaining

What is the performance of:

- `put(K, V)`
  - LP: O(1 + run length)
  - EC: O(1 + chain length)
- `get(K)`
  - LP: O(1 + run length)
  - EC: O(1 + chain length)
- `remove(K)`
  - LP: O(1 + run length)
  - EC: O(1 + chain length)
- Parting Question: how do we control cluster/chain length?