CSCI 136
Data Structures & Advanced Programming

The Map Interface

Video Outline

- Maps!
  - The `Map<K,V>` interface
  - A sample implementation
    - `List<Association<K,V>>`
  - Explore concepts in `structure5.Hashtable`
    - Hash functions
    - Collisions

Map Interface

Key methods for the interface `Map<K, V>`

- `int size()` – returns number of entries in map
- `boolean isEmpty()` – true if there are no entries
- `void clear()` – remove all entries from map
- `boolean containsKey(K key)` – true if key exists in map
- `boolean containsValue(V val)` – true if `val` exists at least once in map
- `V get(K key)` – get value associated with key
- `V put(K key, V val)` – insert mapping from key to val, returns value replaced (old value) or null
- `V remove(K key)` – remove mapping from key to val

Map Interface

Big picture: A `Map` stores a set of key-value pairs that we can *insert/update* and *query*

- We've already explored the notion of key value pairs with the `Association<K,V>` "container class"
- A set is an unordered collection of unique items (i.e., no duplicates)
  - By unordered, we don't mean that we are *prohibited* from ordering items in our data structure; we just aren't required to maintain any particular structure from an external user's perspective

Map Interface

Other methods for Map<K,V>:

- `void putAll(Map<K,V> other)` – puts all key-value pairs from an existing Map into the current map
- `Set<K> keySet()` – return set of keys in map
- `Structure<V> valueSet()` – return collection of values
- `Set<Association<K,V>> entrySet()` – return set of key-value pairs from map
- `boolean equals()` – true if two maps are entrywise equal
- `int hashCode()` – returns hash code associated with values in map (stay tuned…)

A Dictionary Using Map<K,V>

```
public class Dictionary {

    public static void main(String args[]) {
          Map<String, String> dict = new Hashtable<String, String>();
          …
         dict.put(word, def);
          …
          System.out.println("Def: " + dict.get(word));
     }

}
```

What’s missing from the Map API that a physical dictionary lets us do efficiently?

`successor(key), predecessor(key)`

Simple Implementation: MapList

- Uses a `SinglyLinkedList` of `Associations` as underlying data structure

- How would we implement `get(K key)`?
- How would we implement `put(K key, V val)`?

MapList.java

```java
public class MapList<K, V> implements Map<K, V>{

    //instance variable to store all key-value pairs
    SinglyLinkedList<Association<K,V>> data;

    public V put (K key, V value) {
        Association<K,V> temp =
                        new Association<K, V> (key, value);
        // recall: association equals() just compares keys
       // remove old K-V pair if one exists (no dups allowed!)
        Association<K,V> result = data.remove(temp);

        data.addFirst(temp);
        if (result == null)
            return null;
        else
            return result.getValue();
    }
}
```

Simple Map Implementation

- What is `MapList`'s running time for:
  - `containsKey(K key)?`
  - `containsValue(V val)?`

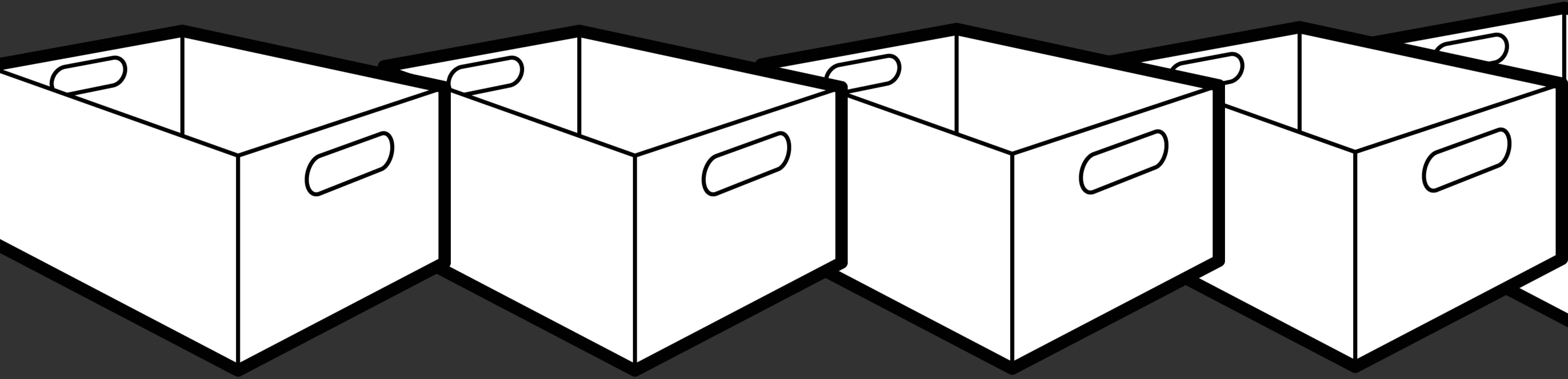
- O(n) time :(
- We want O(1)!

Search/Locate Revisited

- How long does it take to search for objects in (unsorted) Vectors and Lists?
  - O(n) on average
- How about in BSTs?
  - O(log n)
- Can this be improved?
  - Hashtables can locate objects in *roughly* O(1) time!
    - (in future videos we will cover the reasons that O(1) performance is a fuzzy claim)
- Let’s look at a real-world example to help us think through the strategy

Example from Bailey

> “We head to a local appliance store to pick up a new freezer. When we arrive, the clerk asks us for the last two digits of our home telephone number! Only then does the clerk ask for our last name. Armed with that information, the clerk walks directly to a bin in a warehouse of hundreds of appliances and comes back with the freezer in tow.”

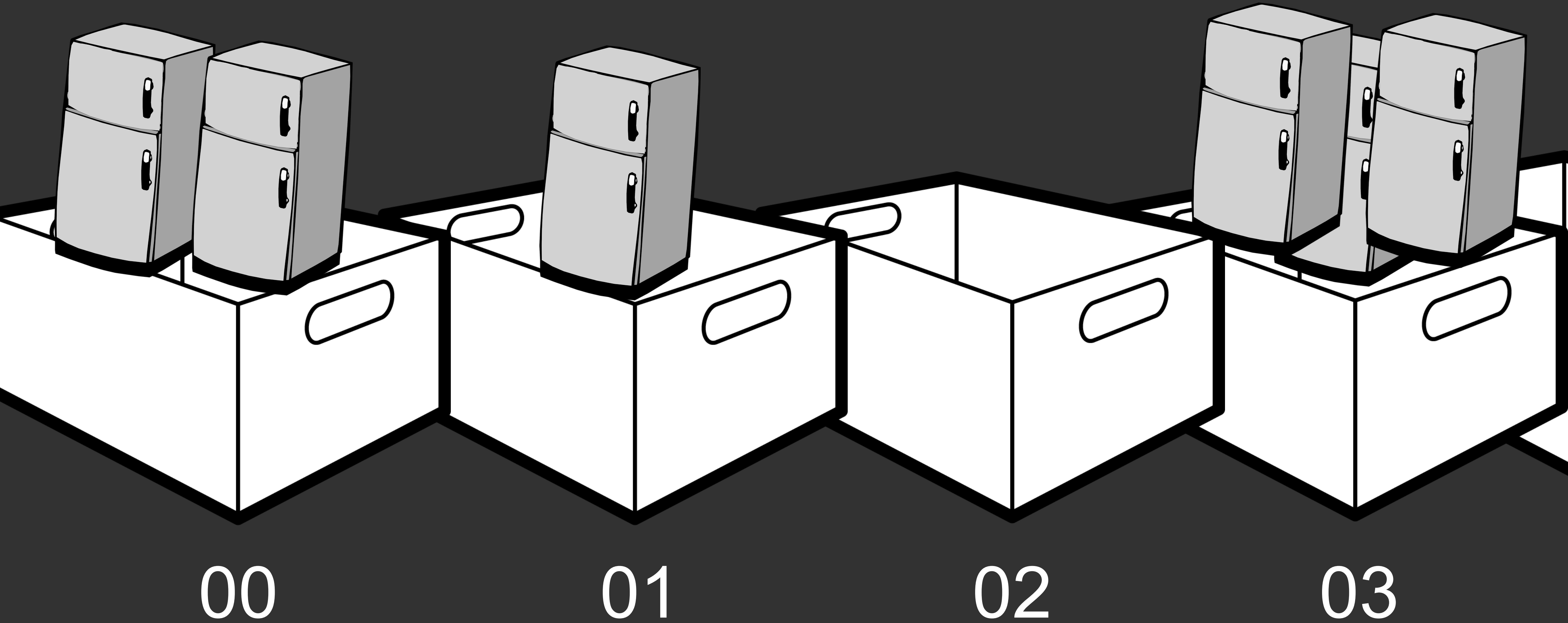


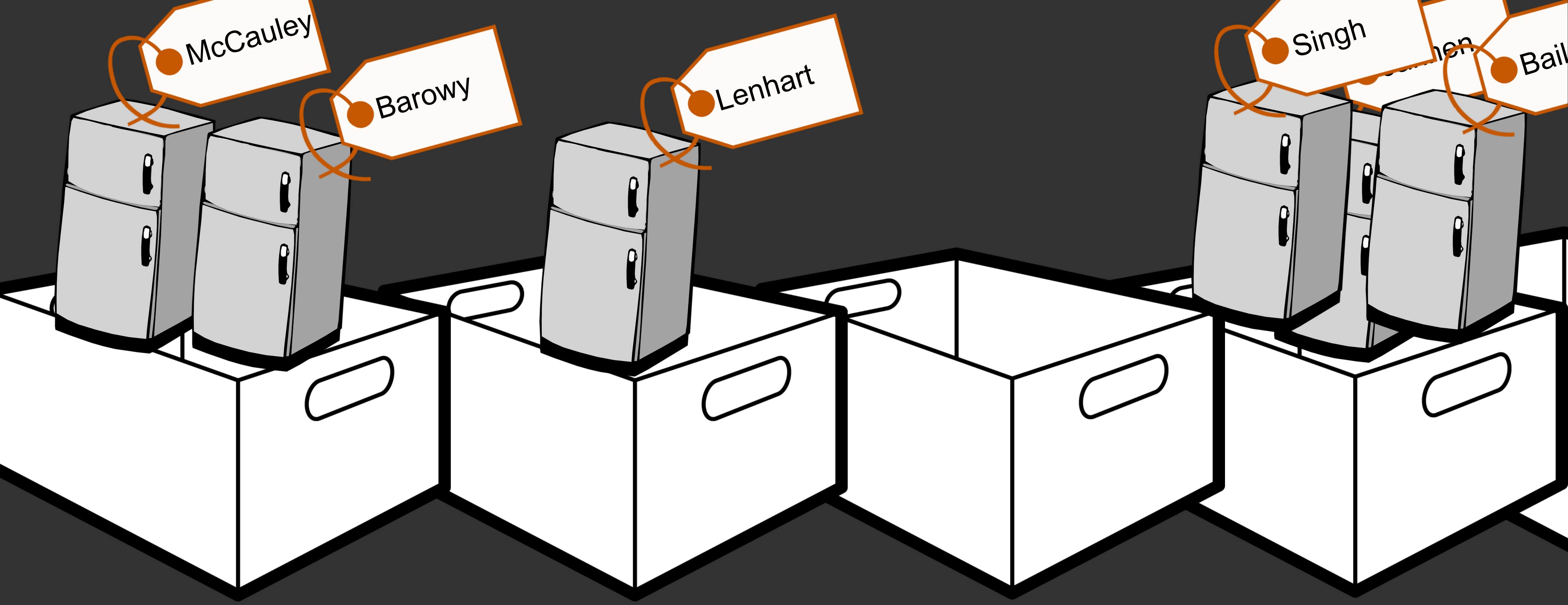
00

01

02

03


00
01
02
03



00 01 02 03

Example from Bailey

> “We head to a local appliance store to pick up a new freezer. When we arrive, the clerk asks us for the last two digits of our home telephone number! Only then does the clerk ask for our last name. Armed with that information, the clerk walks directly to a bin in a warehouse of hundreds of appliances and comes back with the freezer in tow.”

- How does this relate to the Map interface?
  - What is Key? What is Value?
  - Why those choices?
    - Are names evenly distributed?
    - Are the last 2 phone digits evenly distributed?

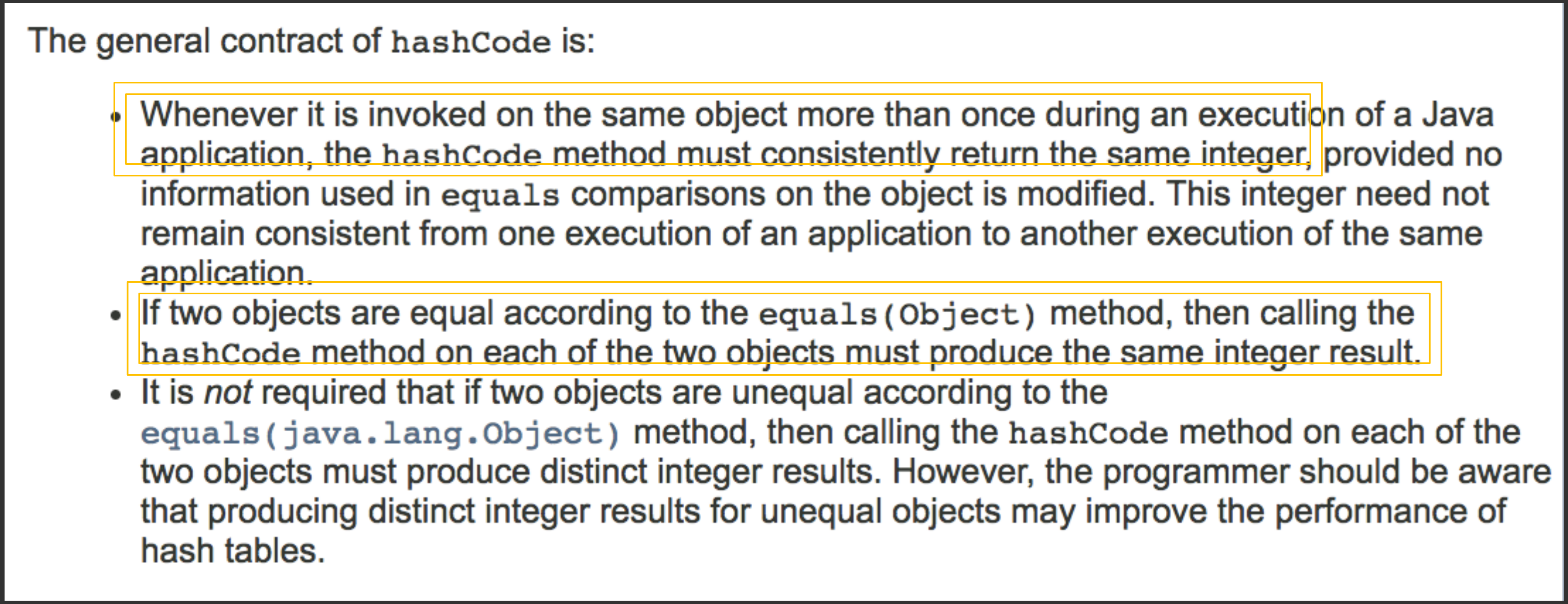
Hashing in a Nutshell

- Assign objects to “bins” based on key
- When searching for object, jump directly to the appropriate bin (and ignore the rest)
- If there are multiple objects assigned to the target bin, then search for the right object
- Important Insight: Hashing works best when objects are evenly distributed among bins
  - Phone numbers are randomly assigned, last names are not!

Implementing a HashTable

- How can we represent bins?
  - Slots in array (how to grow later)
  - How do we find a key’s bin?
  - We use a *hash function* that converts keys into integers
  - In Java, all Objects have
    `public int hashCode()`
    - Hashing function is one way:
      - Can convert a key –> hashCode
      - *Cannot* convert a hashCode –> key
    - Hashing function is deterministic

hashCode() rules


The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()

Implementing HashTable

- So we have a "magic" method that lets us jump directly to an object's bin: `obj.hashCode()`. How do we add Associations to the array?
  - Problem 1: `hashCode()` yields an `int`, but our array may be relatively small.
    - How do we convert arbitrary `int`s to array locations?
  - Problem 2: We can represent 2^{32} unique `int`, but there may be infinitely many values that an object can take on (e.g., String).
    - By the pigeonhole principle, some Strings will have to "share" a hashcode!

Implementing HashTable

- So we have a "magic" function that lets us jump directly to an object's bin: `obj.hashCode()`.
  How do we add Associations to the array?
  - We can use mod (%) to map an into to an array index
    - `array[o.hashCode() % array.length] = o;`

- Problem 2: If two objects have the same "spot", the above expression overwrites
  - This is called a collision

Navigating HashTable Collisions

- Collisions make life hard
- Collisions are possible if:
  - two unique objects map to the same hashCode
  - two unique hashCodes map to the same array index
- We either need to guarantee that collisions can’t happen (which we can’t do) OR build a strategy that can tolerate them
  - We may need to sacrifice some performance to guarantee correctness

Navigating HashTable Collisions

- In a subsequent video, we’ll discuss two approaches to handling collisions
  - Linear probing (sometimes called open addressing)
  - External chaining

- Both strategies work and both are used in practice. External chaining is “easier to implement”, but linear probing can have better “cache locality”. Understanding both techniques is important.

Map Intro Summary

- Map<K,V> defines an interface for storing and querying individual key-value pairs.
- If we implement the interface using Lists that hold associations (Vector, SLL, etc), inserting/querying items is O(n)
- If we instead use binary search trees, operations are O(logn)
- Hashtables promise O(1)-“ish” operations, but we need to manage collisions