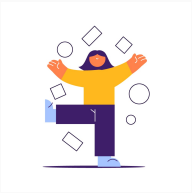
Lec 29: Hash Tables

May 4, 2026

Admin



- Remember: decide if you want to take the final during reading period, let me know by Wednesday
- This Wednesday's lab will be implementing hash tables
- Any questions?

Dictionaries and Maps

Dictionary data structure

- Store data associated with a set of *keys*
- Goal: for a given key, want to be able to look up the associated data (which we call a *value*)
- For example: let's say we have a list of words. We want to be able to look up the definition of any word.
  - keys are the words
  - definitions are the values
- For Google: given a keyword, find all websites that contain that keyword
- Given a course name, find the list of all students that are taking that course
- You may have seen dictionaries in Python before (uses `{ }` notation).

Dictionary methods

- method `contains(key)` returns a `boolean`
- method `getValue(key)` should get the value associated with a key
- Want to be able to update dictionary: `put(key, value)` adds a new value; `remove(key)` removes it
- Each key should appear once. (Why?)
    - Unambiguous lookup! If a query a key, I should know exactly what value I'm getting

A Dictionary Interface: Map

- Let's take a look at the `Map` interface

- (Map and Dictionary are synonyms. The interface in Java just happens to be called `Map`; the equivalent python data structure is a dictionary.)

Main Map Interface Methods

- `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

Simple Map Using an ArrayList

- Use an `Entry` class to hold each entry; need to hold a key and a value
- The dictionary consists of an `ArrayList` of these entries
- to add a new key/value pair, create a new `Entry` and call `add()`
  - Running time? $O(n)$ worst case if need to double in size; often $O(1)$
- the `contains()` and `getValue()` methods compare the keys of each entry
  - Running time? $O(n)$

```
1 public class Entry<K, V> {
2     private K key;
3     private V value;
4     //getters and setters (?), constructor
5 }
```

Can We Do Better?

- What data structure can potentially do better than $O(n)$ for both `put()` as well as `containsKey()` and `getValue()`?
- AVL tree can do all in $O(\log n)$.
- Note that the AVL tree requires that items can be ordered. So `K` needs to implement `Comparable<K>`, or we need a `Comparator<K,K>`.
- Can we do better than $O(\log n)$? Is it necessary to order the items to do better than $O(n)$?

Hash Codes and Hash Tables

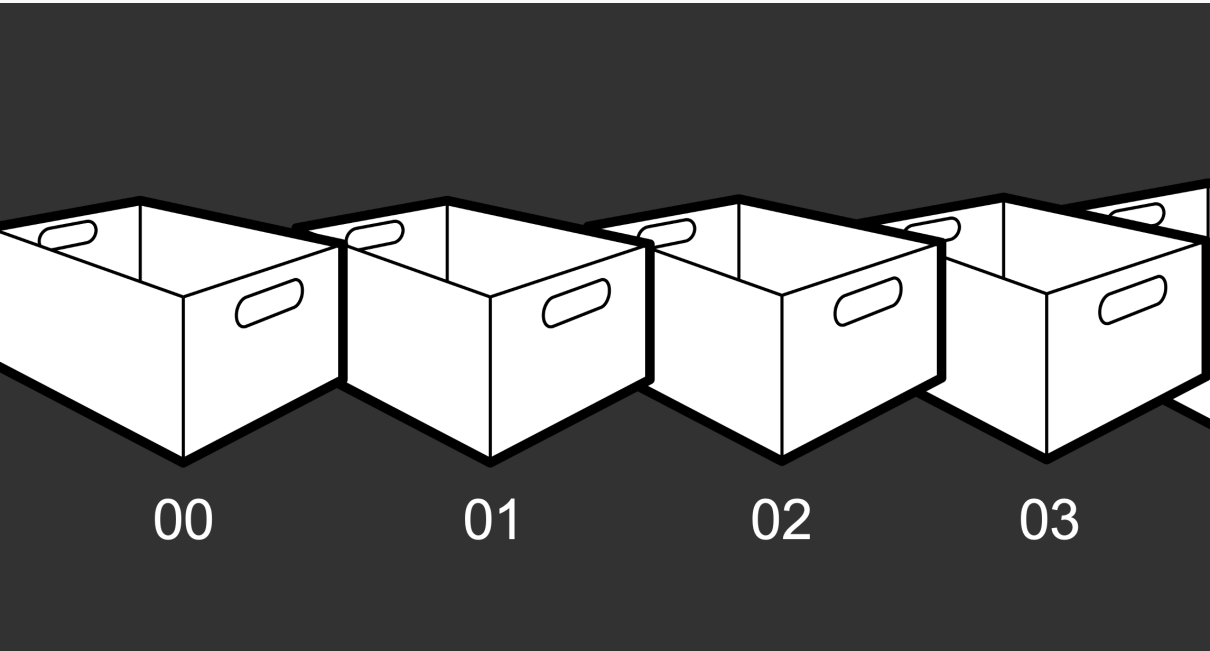
Hash Tables

- Hash tables can `add()`, `remove()`, `containsKey()` and `getValue()` in roughly $O(1)$ time!!!
- Later this week we'll see why $O(1)$ is a bit of a fuzzy claim
- Let's look at a real-world example to help us think through the hash table strategy

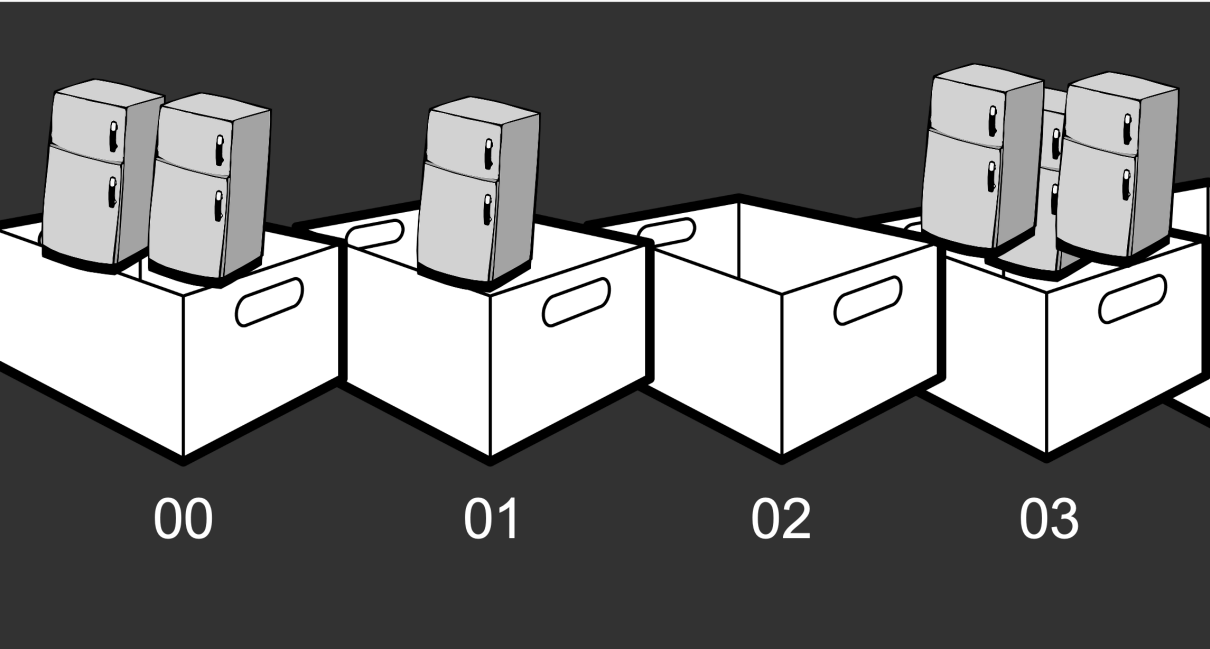
Example from Bailey

> *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.*

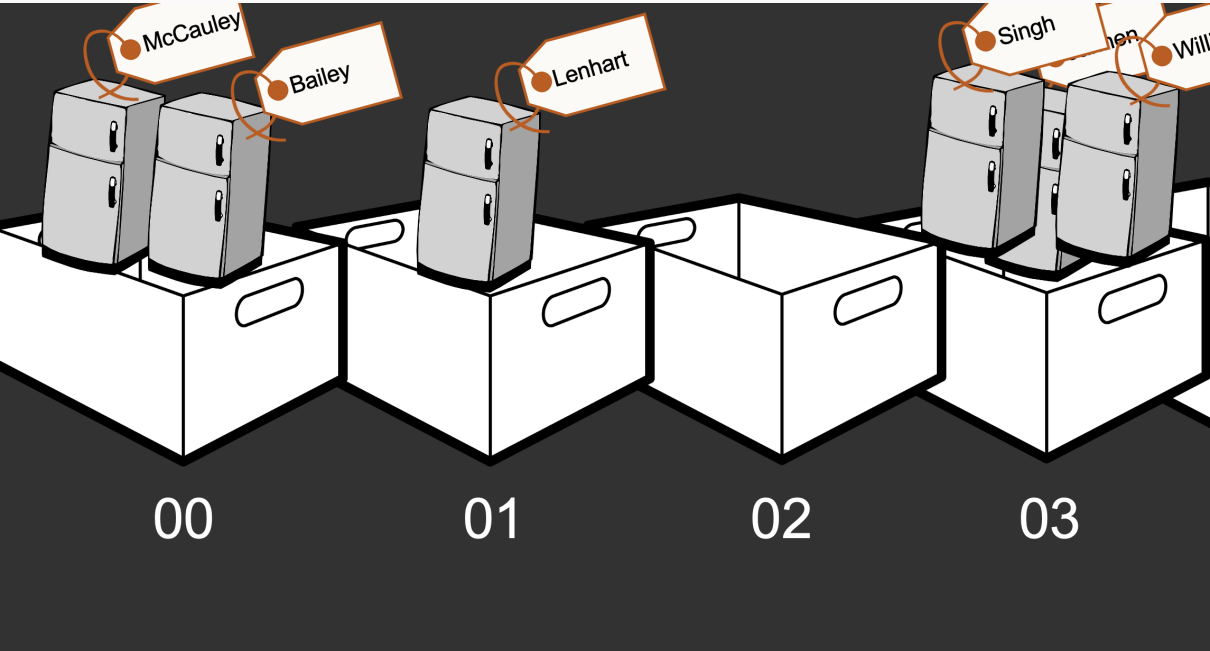
Example from Bailey



Example from Bailey



Example from Bailey



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?
- In pairs: What is the `Key`? What is the `Value`? Why do we assign to buckets based on phone number rather than name?

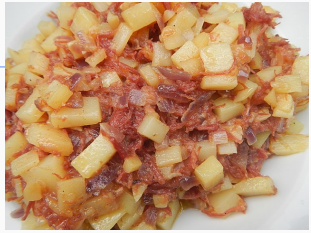
Hashing in a Nutshell

- Assign objects to bins based on key
- When searching for an 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 Hash Table

- How can we represent bins?
  - Slots in an array! (We'll talk about how to grow later.)
- How do we find a key's bin?
  - We use a *hash function* that converts keys (of type `K`) into `int`s
- In Java, all `Objects` have a method `public int hashCode()`
  - Just like `.toString()` and `.equals()`

`hashCode()` properties



- Return type `int`
- Hashing function is *one way:*
  - Can convert a key into a hashCode
  - May not be able to convert a `hashCode` into a key
- Hashing function is deterministic
  - Each time you call `hashCode()` on the *same* object (with the same data) you get the *same* output

Hash Code 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.

From the Oracle documentation on the `hashCode()` method

Implementing HashTable

- `hashCode()` allows us to jump directly to the bin for a particular object
- Problem 1: `hashCode()` yields an (arbitrary) int, but our array may be relatively small.
    - How do we convert `ints`—potentially, large ones—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!

Fitting Items Into Array

- Use mod (in Java: `%`) to map the object to an array index
- Something like:
- `array[o.hashCode() % array.length] = o;`
- That way, every object fits to some slot in our array using its hash code
- From example: “Last two digits” of phone number is the same as phone number `% 100`

Objects with the same hashcode

- If two objects map to the same slot in the array (after taking mod), it is called a *collision*
- Could be two objects with the same hashcode, or two objects with different hash codes that map to the same slot
- Can we guarantee that collisions can't happen?
  - No: unless our table is REALLY large, *some* pairs of objects will have a collision
- Instead: create a strategy for storing items that can handle collisions!

Hashing in a Nutshell

- `Map<K,V>` is an interface for storing key-value pairs; can be efficiently implemented using hashing
- Assign objects to “bins” based on key
- When searching for an 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

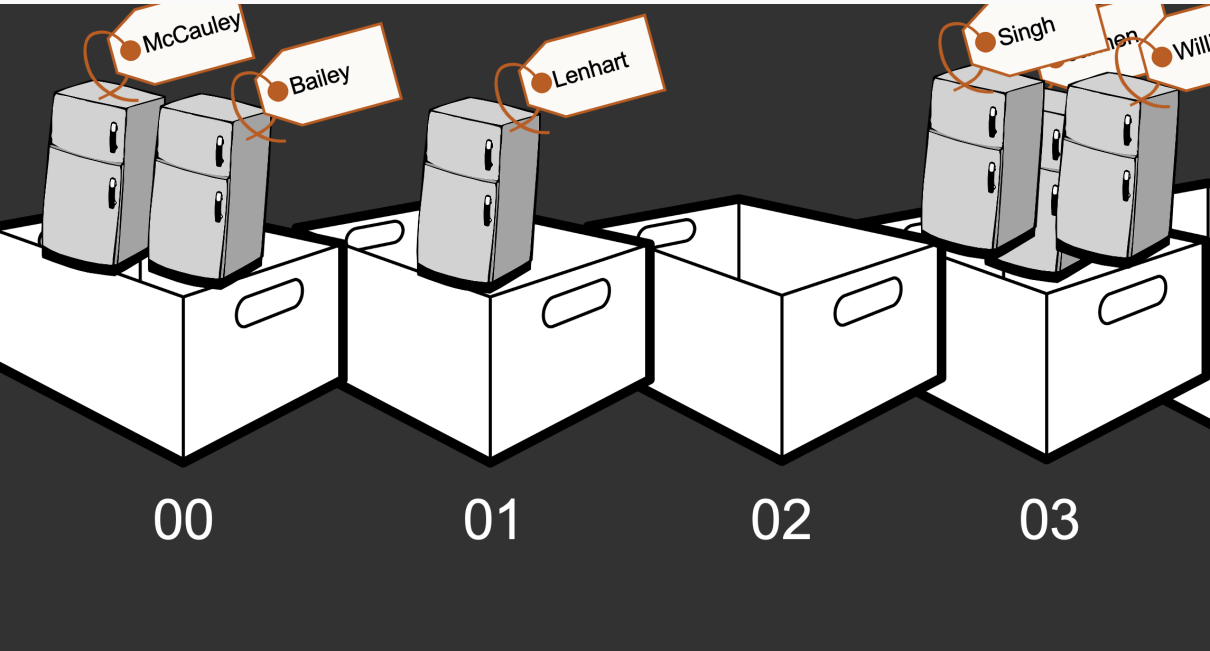
Hash Table Collisions

Navigating Hash Table Collisions

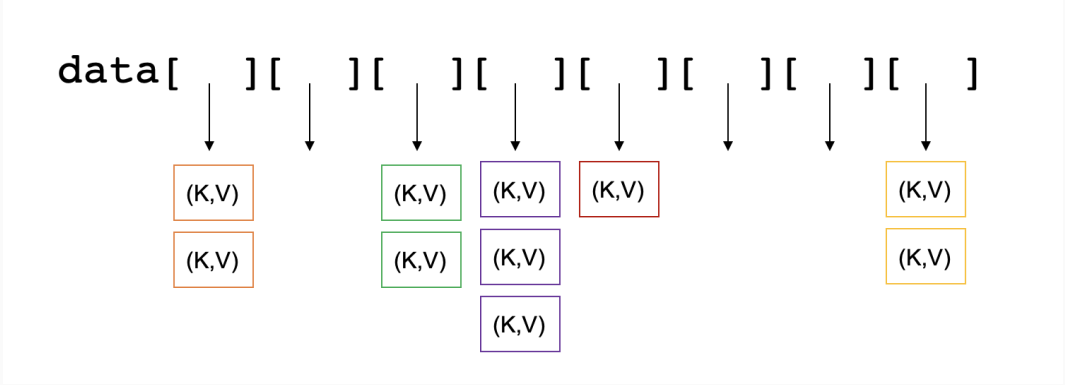
- *Problem*: collisions occur when two unique items are mapped to the same bin
  - This is a problem in arrays because we can only store one item per index
  - Collision management isn't just a performance issue, it is a *correctness issue*
- We'll discuss two strategies to resolve collisions:
  - External chaining
  - Linear probing (sometimes called open addressing)

External Chaining

Resolving Collisions in Practice?



External Chaining



- Idea: instead of keeping individual items in each bin, we store a *list* in each bin
- `get()`, `put()`, and `remove()` proceed in two steps: identify the bin using the hash code; then perform a linked list operation

Hash Table with External Chaining Practice

- Let's do the following operations for a hash table with chaining of length 6. Let's assume we are storing a `Map` from the ID of each student to the corresponding `Student` object. We'll assume the hash code of each ID is the ID itself.

- `put(27, new Student("Joyce"));`
- `put(38, new Student("Cora"));`
- `put(9, new Student("Kristie"));`
- `put(14, new Student("Jose"));`
- `put(7, new Student("Evan"));`
- `getValue(14);`

Downsides to External Chaining?

- Each slot in our array stores a list, even if the slot is empty
  - Consumes some extra space (but not much)
- Potentially poor locality
  - Not something we've talked about in the 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*
  - While array elements are contiguous (near), list elements may be scattered throughout memory (far)

Linear Probing

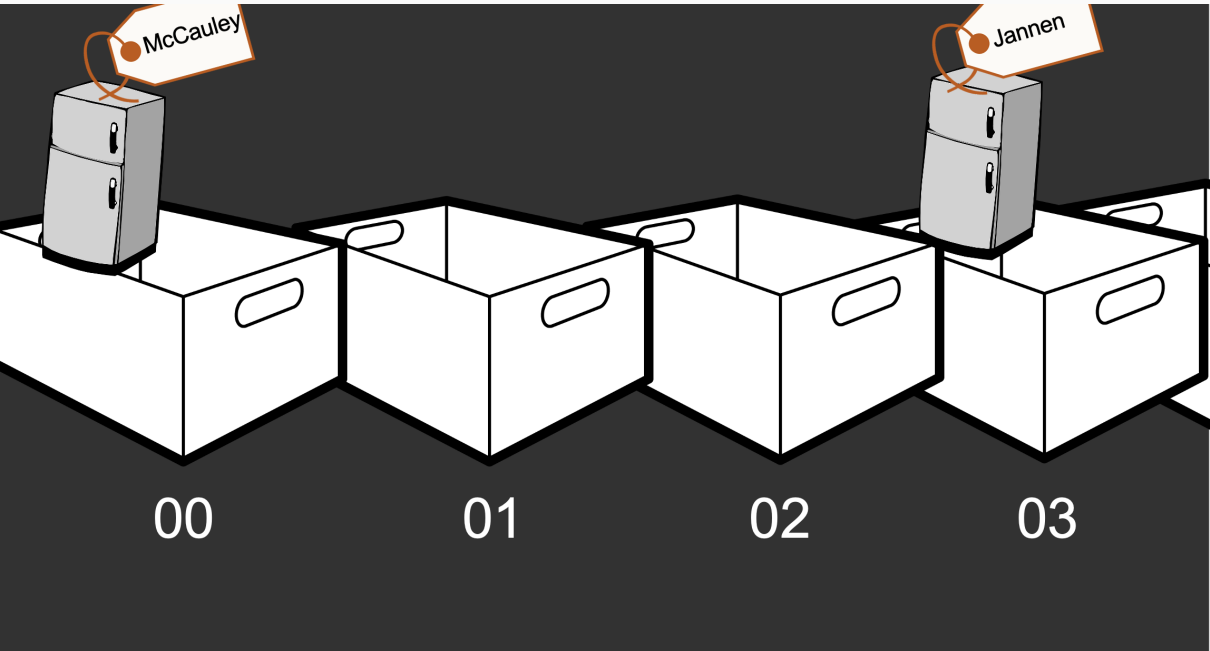
Rethinking Collisions

- Let's define an item's *canonical slot* as the place where the item begins (ignoring collisions)
- Something like the absolute value of the hash code modulo the table size
- If no two items have the same canonical slot, we're done!
- If multiple items do map to the same canonical slot, we need to figure out:
  - Which one goes in the canonical slot?
  - Among items that cannot be stored in their canonical slot, where should they go? How can we find them in the future?

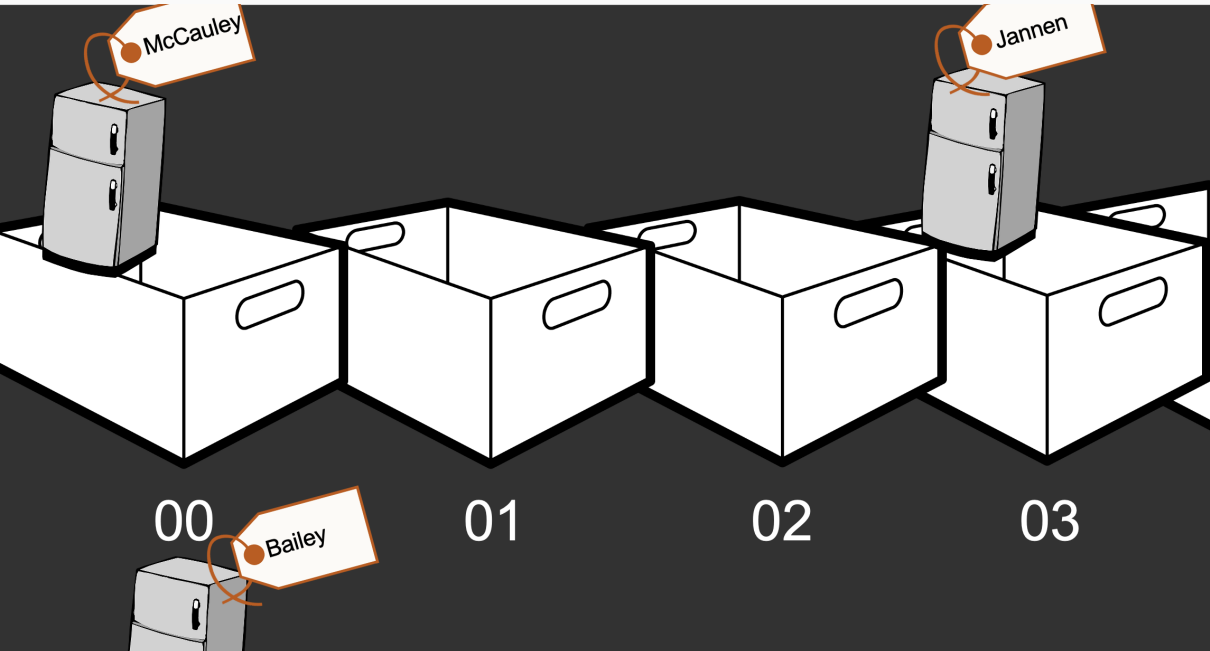
Linear Probing

- General idea: store each key-value pair in the first open slot on or after its canonical slot
- Insertion: if a collision occurs at the bin, just scan forward (linearly) until an empty slot is available; store the item there
  - We “wrap around” at the end of the array
  - Let’s call a contiguous region of full bins a *run*
- Lookup: to find a key-value pair, calculate the bin. Then, scan linearly until the item is found or you reach an empty slot

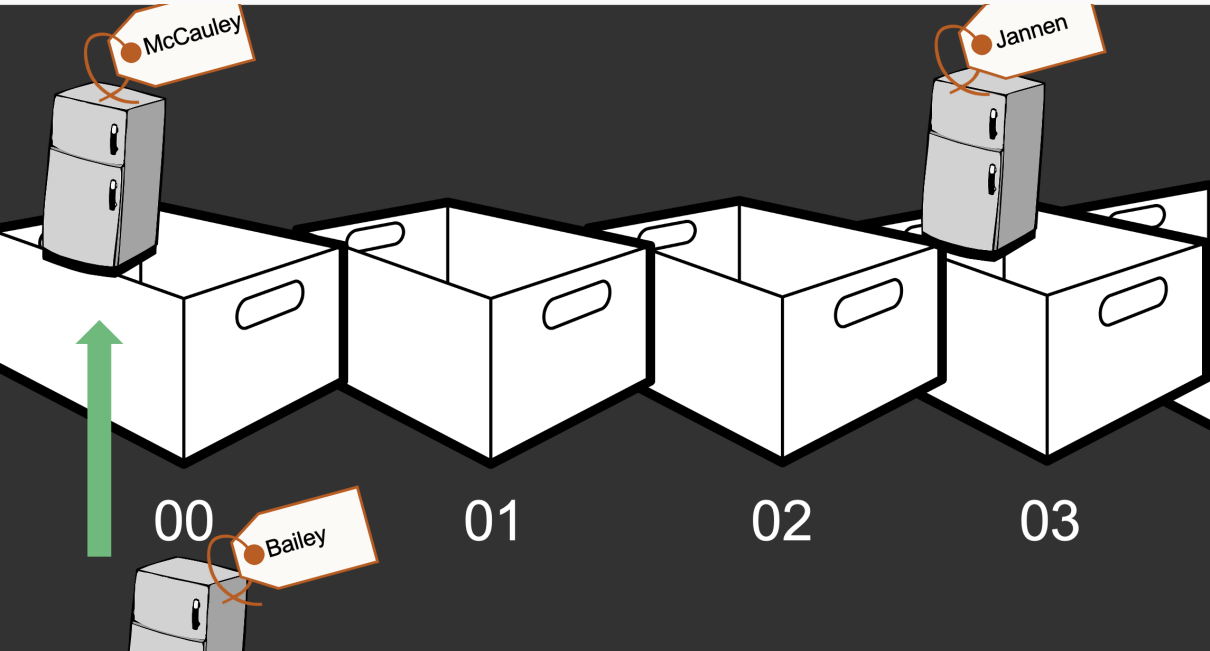
Linear Probing Example



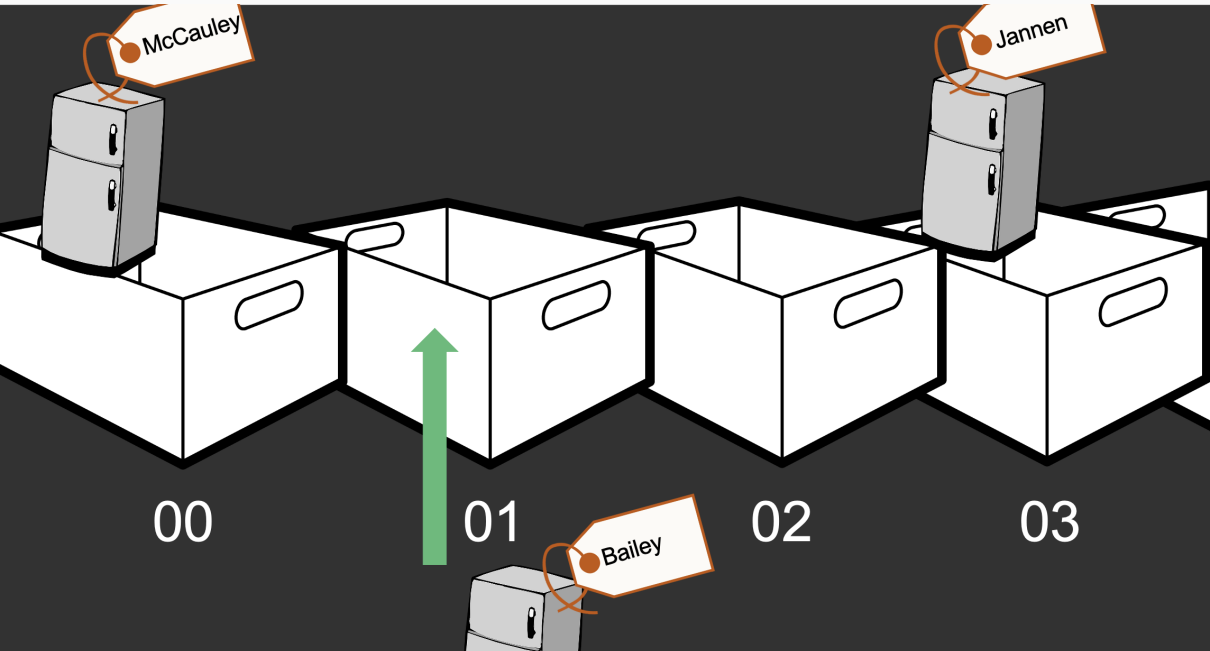
Linear Probing Example



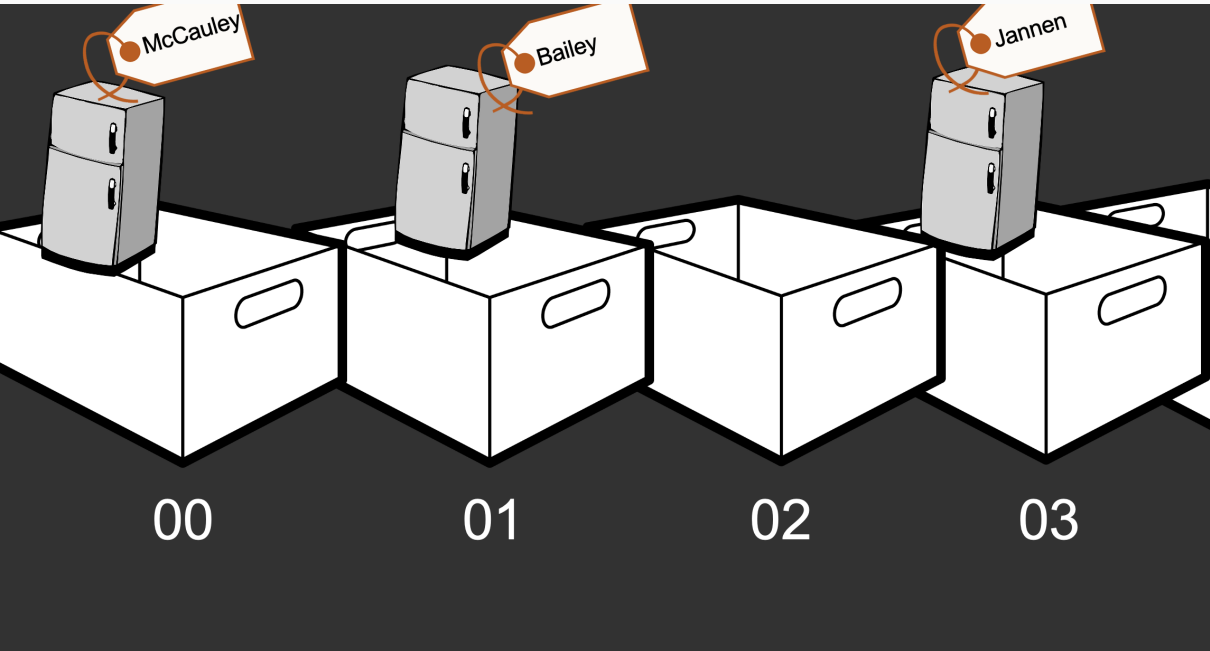
Linear Probing Example



Linear Probing Example



Linear Probing Example



Hash Table with Linear Probing Practice

- Let's do the following operations for a hash table with linear probing of length 6. Let's assume we are storing a `Map` from the ID of each student to the corresponding `Student` object. We'll assume the hash code of each ID is the ID itself.

- `put(27, new Student("Joyce"));`
- `put(38, new Student("Cora"));`
- `put(9, new Student("Kristie"));`
- `put(14, new Student("Jose"));`
- `put(7, new Student("Evan"));`
- `getValue(14);`
- `getValue(7);`

Tricky Part: Deletes

- How do we delete items using linear probing?
- Let’s try it on the board
- Problem: the “run” is broken up
- In pairs: how can we fix this?