Hashing

Instructors: Sam McCauley and Dan Barowy

April 27, 2022

Admin

- Any questions?

Hash Functions

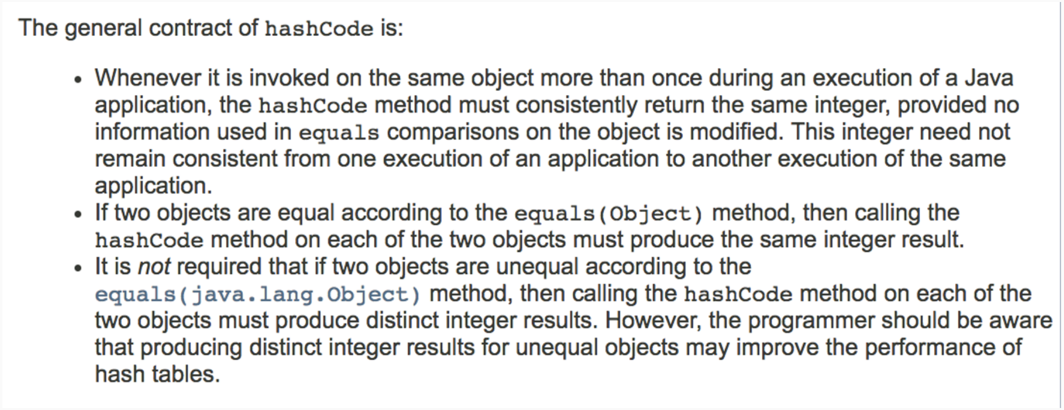
Hashing in a Nutshell

- `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

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 `Object`s have a method `public int hashCode()`

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.

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

Implementing HashTable

- `hashCode()` allows us to jump directly to the bin for a particular key object
- How do we add `Associations` to our array?
- Problem 1: `hashCode()` yields an int, but our array may be relatively small.
    - How do we convert arbitrary ints 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;`
- Is this always legal? Hint: no! But why not?
    - Mod of a negative int is negative!
- Instead: `array[Math.abs(o.hashCode() % array.length)] = o;`
- That way, every object fits to some slot in our array using its hash code
- This is called a *hash table*

Objects with the same hashcode

- If two objects map to the same slot in the array (after taking mod of the hashcode), 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: for any hash code we write, *some* pairs of objects will have a collision
- Instead: create a strategy for storing items (extending the "mod" idea above) that can handle collisions!

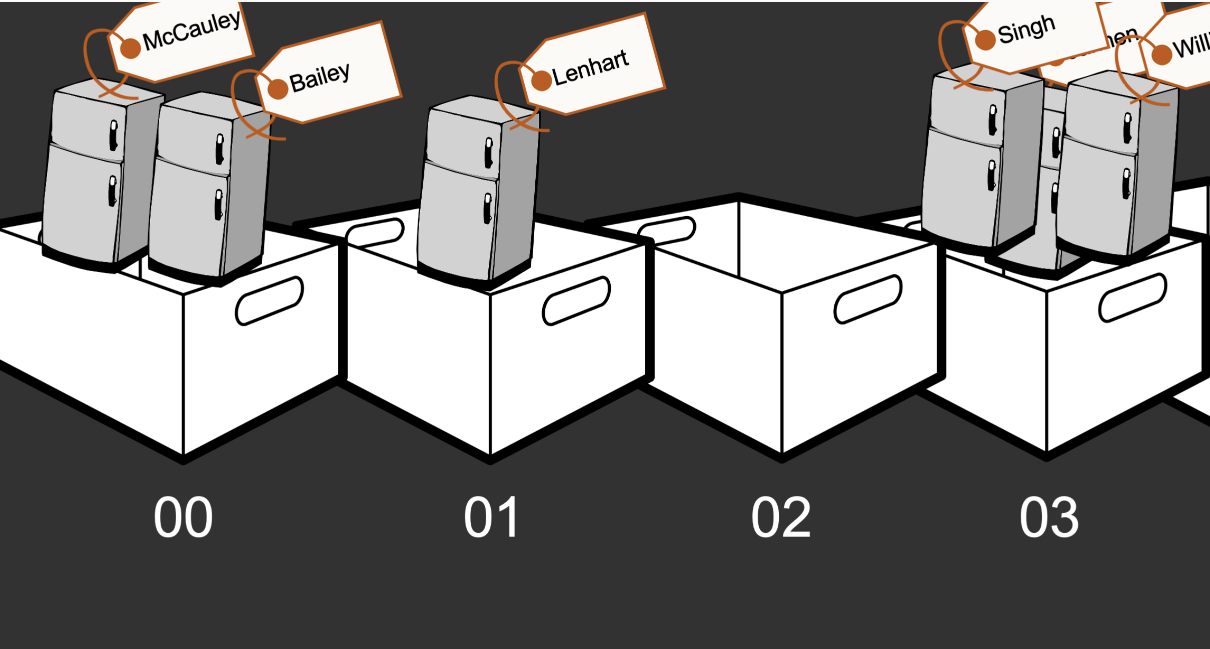
Hash Table Collisions

Navigating Hash Table Collisions

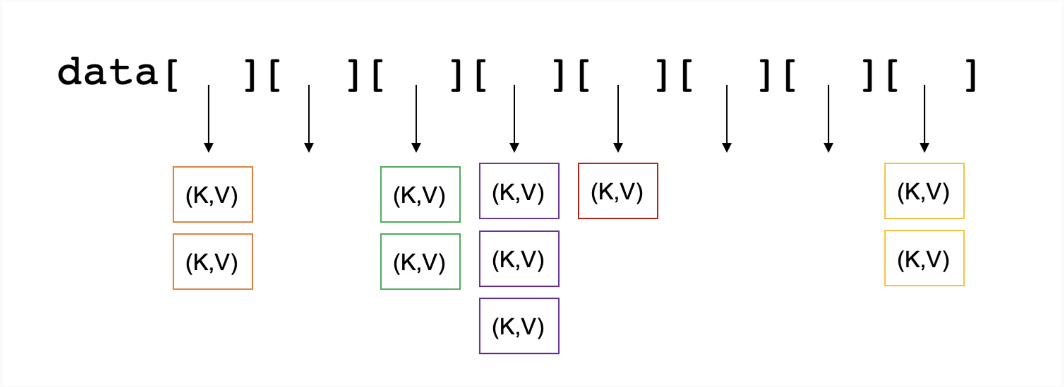
- *Problem*: collisions occur when two unique items are mapped to the same bin
    - This is a problem in arrays because we can only store one item per index
    - Collision management isn’t just a performance issue, it is a *correctness issue*
- We’ll discuss two strategies to resolve collisions:
    - 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 Implementation With 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 With External Chaining

```java
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 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

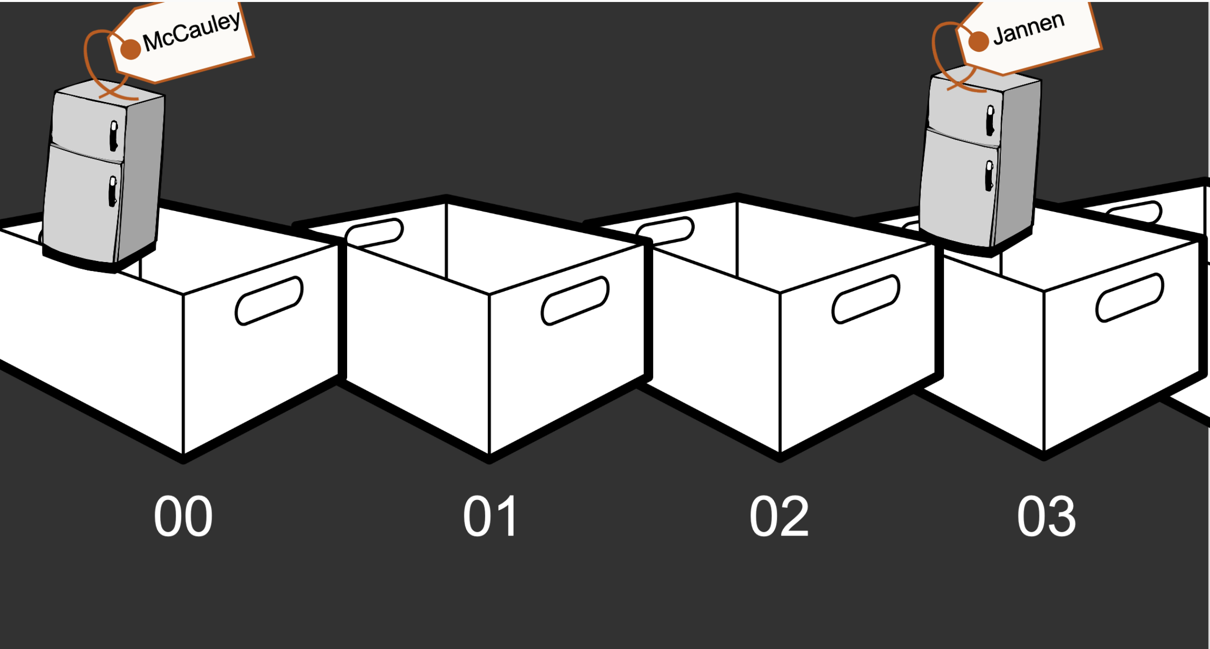
Rethinking Collisions

- 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?

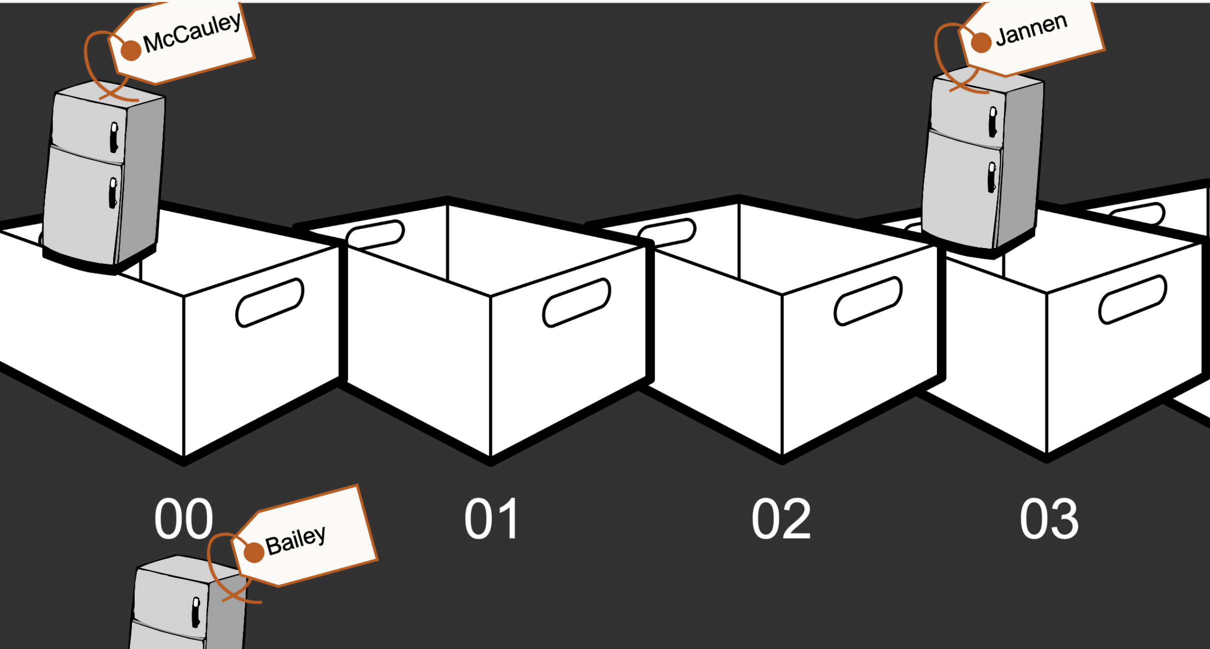
Linear Probing

- 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 the end of the run.

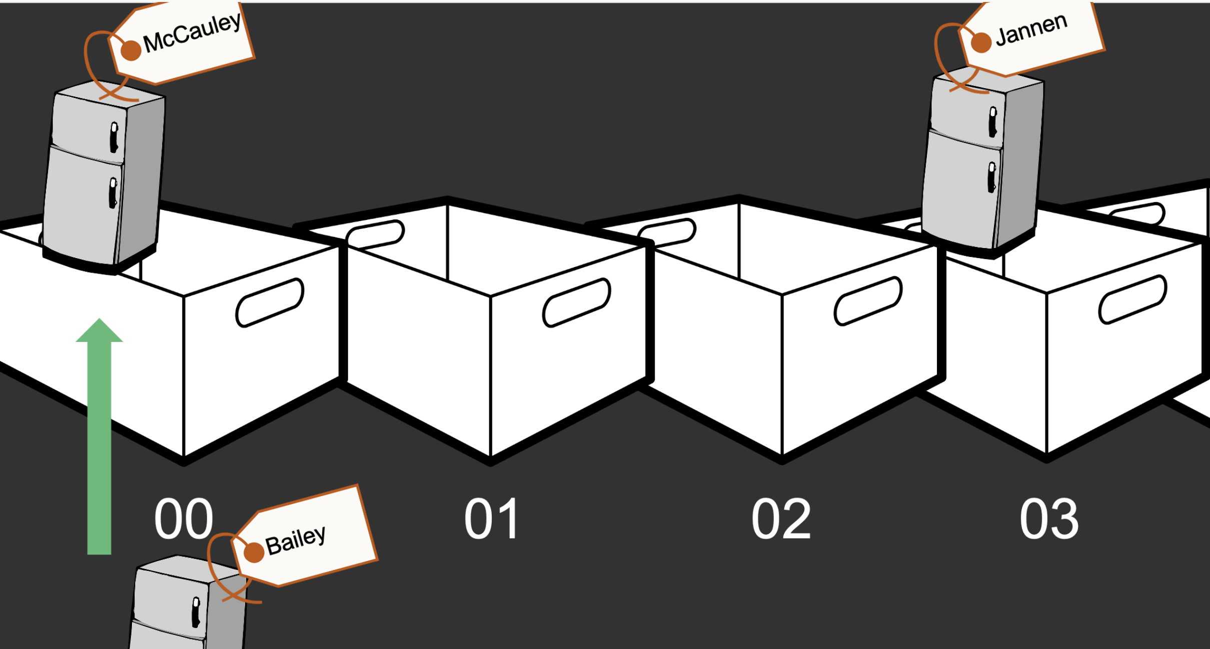
Linear Probing Example



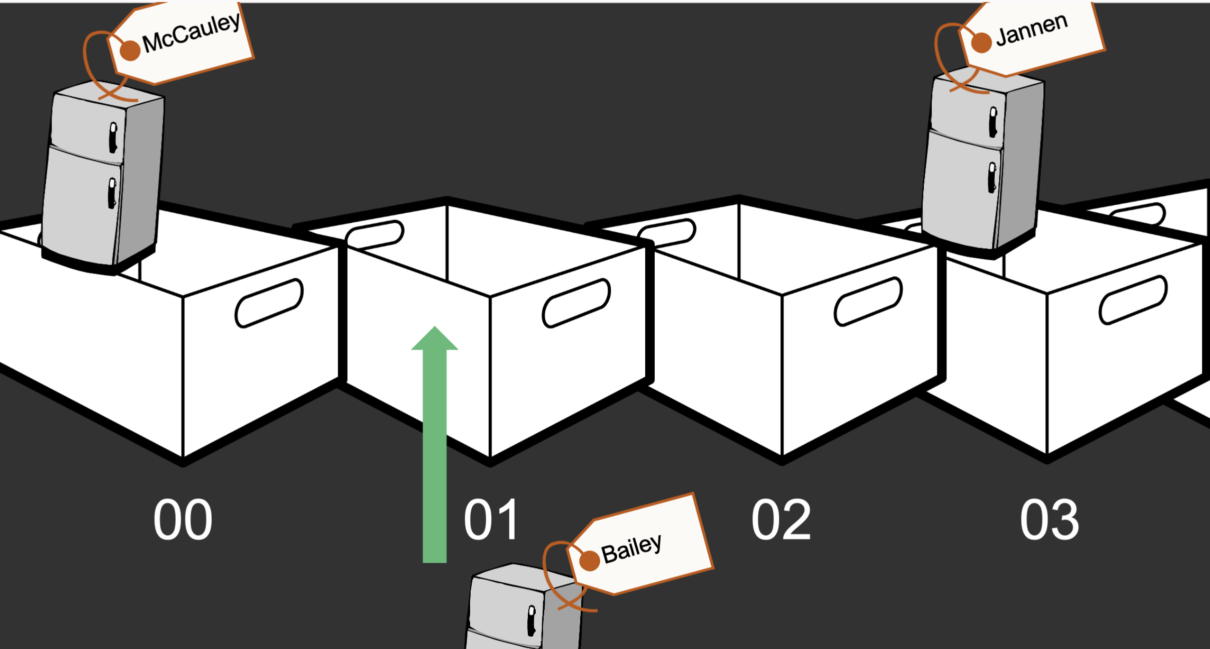
Linear Probing Example



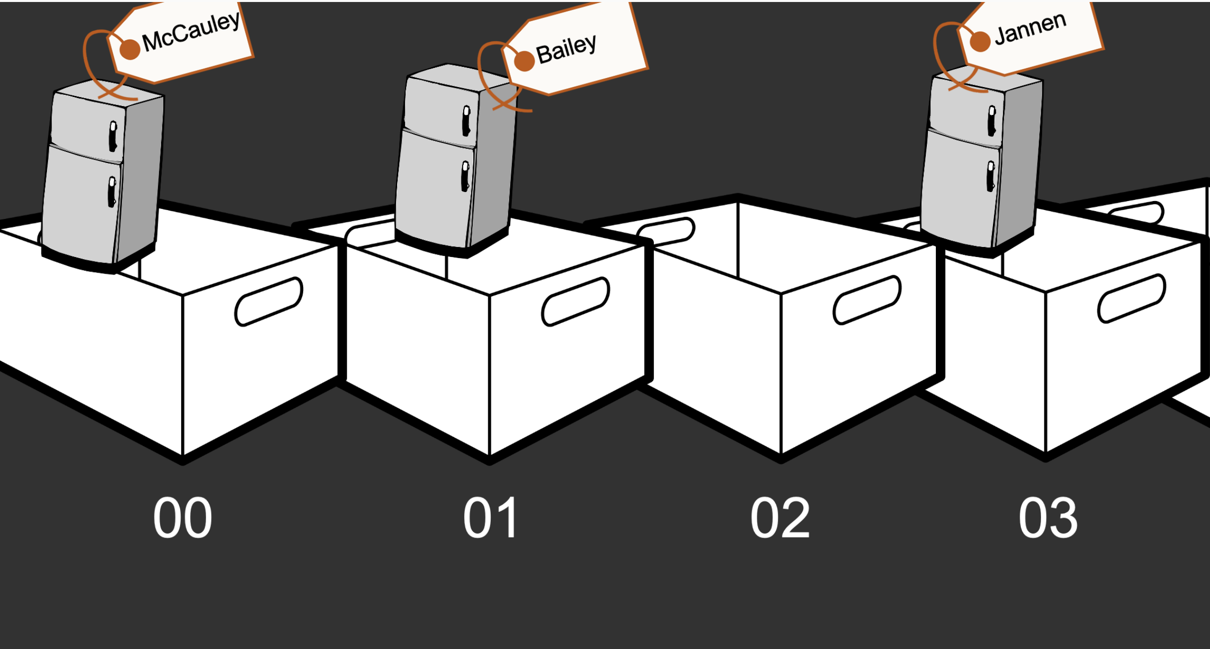
Linear Probing Example



Linear Probing Example



Linear Probing Example



First Attempt: `put(K)`

```java
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)`

```java
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;
   }
}
```

Tricky Part: Deletes

- Let’s look at `NaiveProbing.java`
- Simple (not very good) hash function: index of first letter of word
- Initial array size = 8
- Add “atlanta” to the hash table, then “detroit,” then “queens”
- What happens if we remove “atlanta” and then look up “queens?”
    - Our run was broken up!
    - Now `get()` won’t work correctly

Linear Probing Deletes

- 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 was created with 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 an insertion, we treat it as an open slot
        - Must still scan the whole run to make sure the key isn’t present later on

Implementation

- Let’s look at `HashAssociation.java`

- Finally, `Hashtable.java`

Linear Probing Observations

- Code is more complicated than in external chaining, but still manageable
- The length of a run dictates the performance
- Removing elements does not shrink the run–it defers the work to other operations
    - Keeping runs small is important, so we may want to reconsider some design decisions if we expect a lot of deletions

Linear Probing Observations

- Downsides of linear probing?
- What if the array is almost full?
    - *Very* long runs
- Does external chaining avoid this problem?
    - Short answer: yes
    - Only scan through collisions, not the entire run
    - Never scans more items than linear probing!
    - But: worse cache behavior (locality)

Performance: Linear Probing vs Chaining

- What is the performance of `put(K,V)`?
    - Linear probing: $O(1 + \textit{run}\text{ length})$
    - External Chaining: $O(1 + \textit{chain}\text{ length})$
- Same for `get(K)`, `remove(K)`
- So: how do we control the length of a run/length of a chain?
- Related: how do we actually choose a hash function?

Choosing a Hash Function

Hash Function Goals

- Must be deterministic (and consistent), return an int

- But: want to “spread out” items

Hash function for an integer

- How can we hash an `int` (or an `Integer`)?
- One idea: `hashCode()` can just return the number itself
    - Is this deterministic?
    - Does it spread out values?
- Despite the fact that it does not spread values out very well at all, Java actually uses this as the `hashCode()` method for `Integer`
- What are some upsides of this approach?
    - Fast, simple

Better Hash Functions for Integers

- Lots of hash functions out there for an integer x
- One popular one: choose random large numbers a and b, and a large random prime p. Define the hash function to be

$$h(x) = ax + b \% p$$

- Gives actual theoretical bounds!
    - (Outside the scope of the course:) for two different given x, y, the probability that $h(x) = h(y)$ is $1/p$.

How Can we hash Strings?

- We saw a very bad method that just looks at the first letter
- Each letter can easily be mapped to a number
- How can we take other letters into account? How can we broaden the number of possible hash values? Any ideas?
- One idea: add up all the values of the letters
- Downsides of this?
    - Still relatively few hash values
    - Anagrams always collide! Not ideal

Java’s hash function for strings

- Same idea as above
- But: multiply ith letter from the right by 31^i.
- Fast, and works fairly well in practice
- Let’s look briefly at the code

Choosing a Hash Function

- Somewhat serious research area
- Tradeoffs: how complicated it is, how well it performs in theory, how well it performs in practice
- What you’ll probably do with them: look up a decent one and use it

Friday

- Performance
- Dealing with hash table sizes
- Growing and shrinking hash tables
- Intro to Graphs!