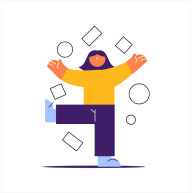
Lec 30: Hash Tables Continued

May 6, 2026

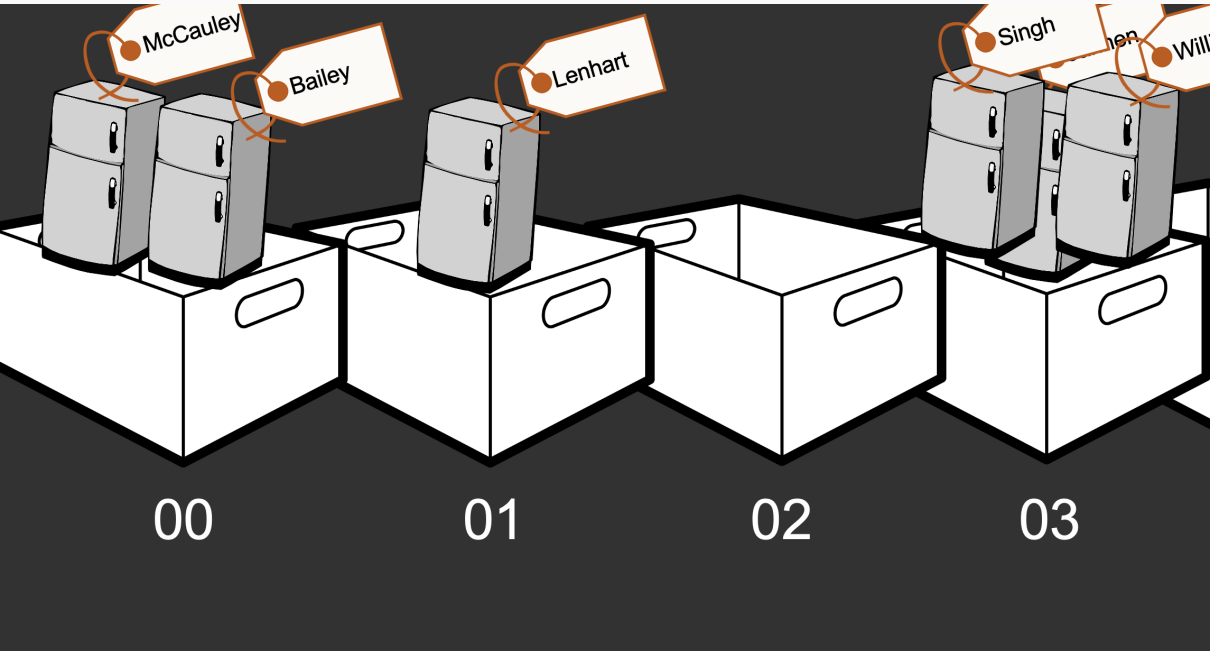
Admin



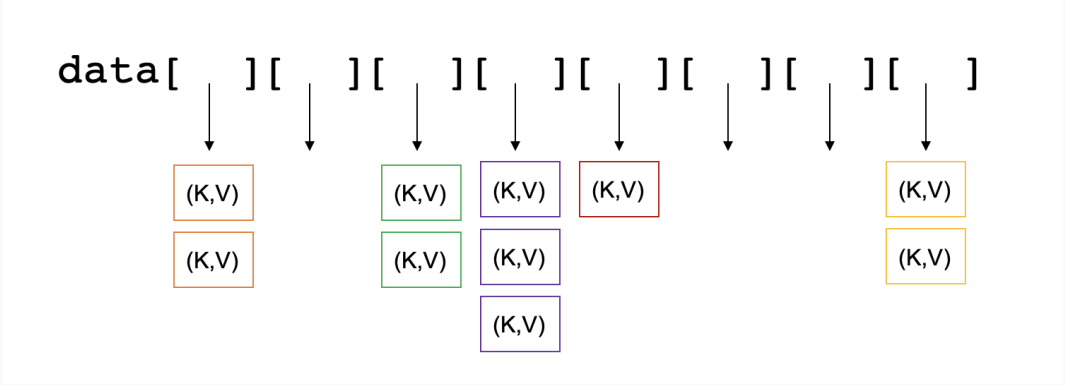
- Remember: decide if you want to take the final during reading period, let me know *today*!
- Today's lab is in-person; on implementing hash tables
- Any questions?

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)

Negative Hash Codes

Negative Hash Codes

- We said that we want to store key k in slot `k.hashCode() % arr.length`
- What happens if `k.hashCode()` is negative?
  - Answer: mod of a negative value is negative in Java. So: we get an *exception*
- How do we fix this? Let's come up with some ideas on the board.

Fixing Negative Hash Codes

- Option 1: use an if statement
  - Works well! Not often used (slightly slow)
- Option 2: Use `Math.abs()` to get the absolute value of the hash code
  - Does *not* work!
  - In Java, the absolute value of the smallest negative integer is negative
- Option 3: Use bit tricks to ensure the first bit of the integer is positive
  - `int slot = (s.hashCode() & 0x7FFFFFFF) % arraySize;`
  - We'll give this to you in the starter code for the lab

Linear Probing

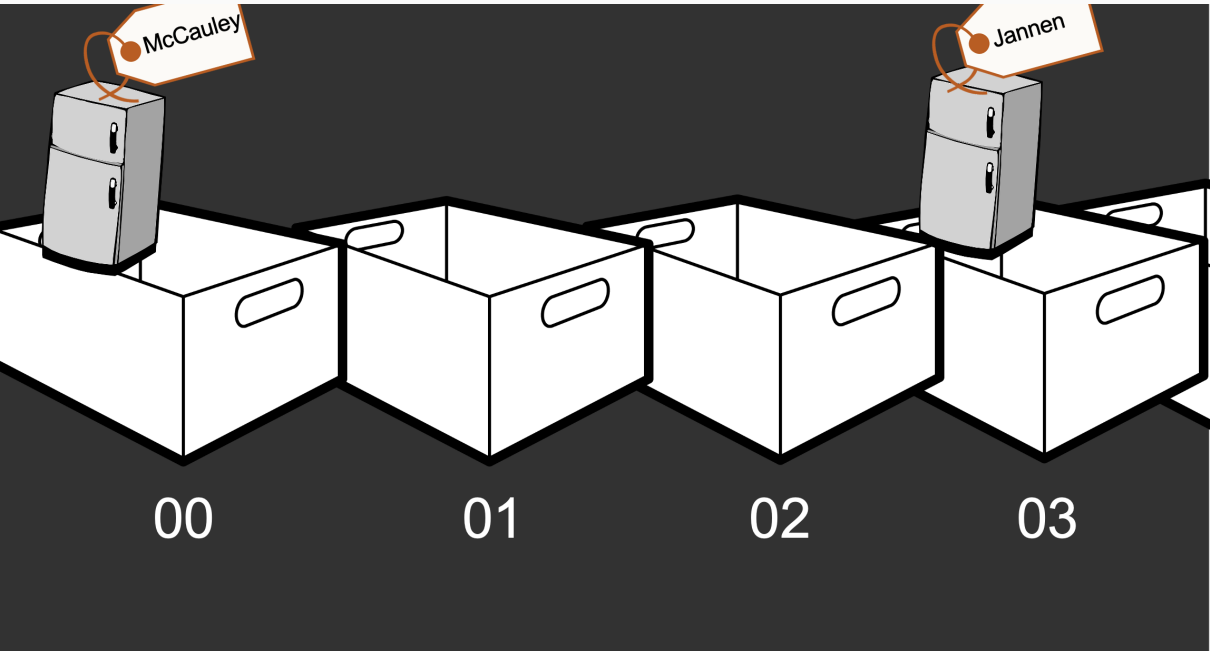
Rethinking Collisions

- 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?

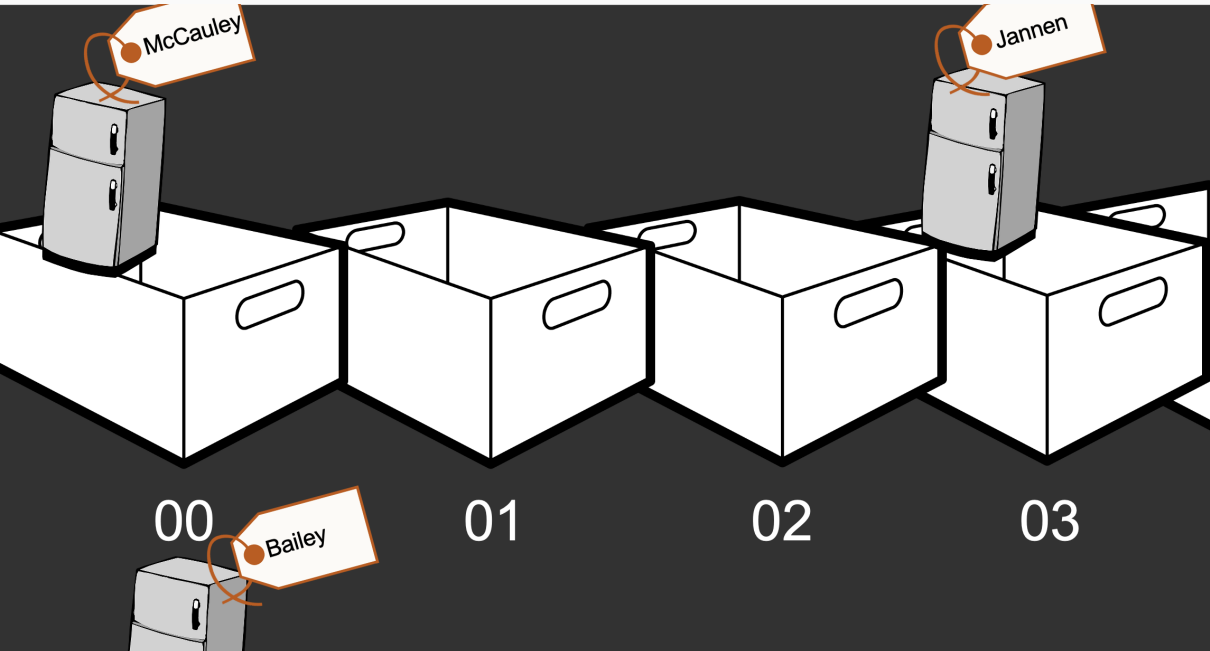
Linear Probing

- 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

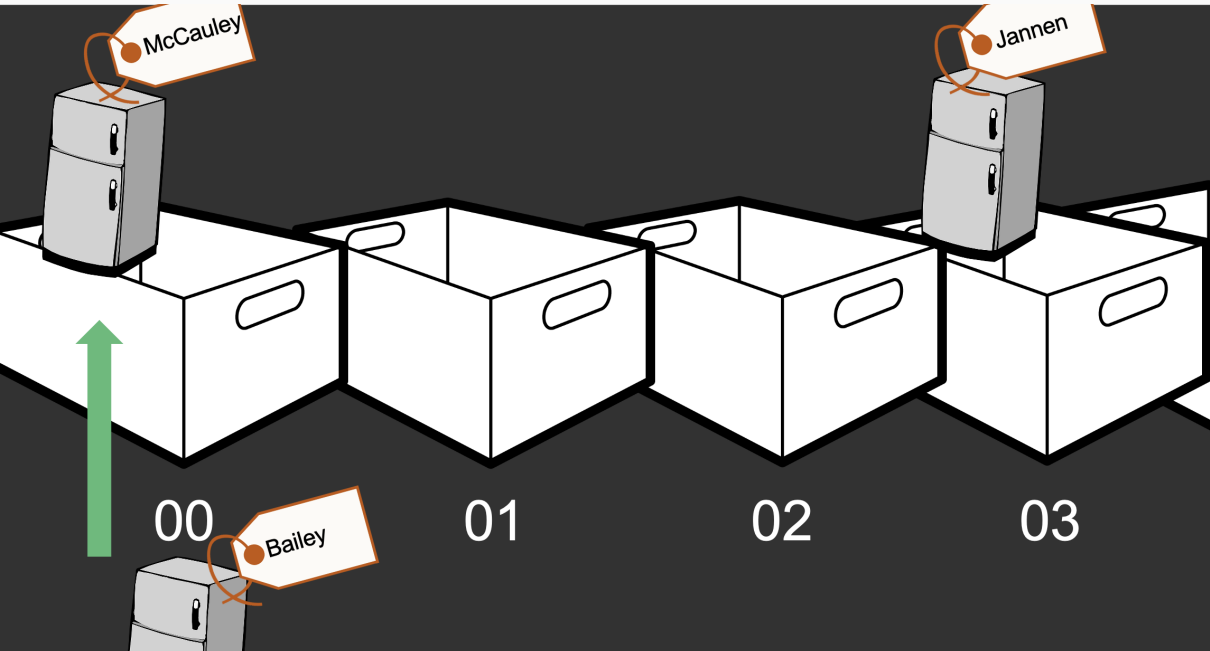
Linear Probing Example



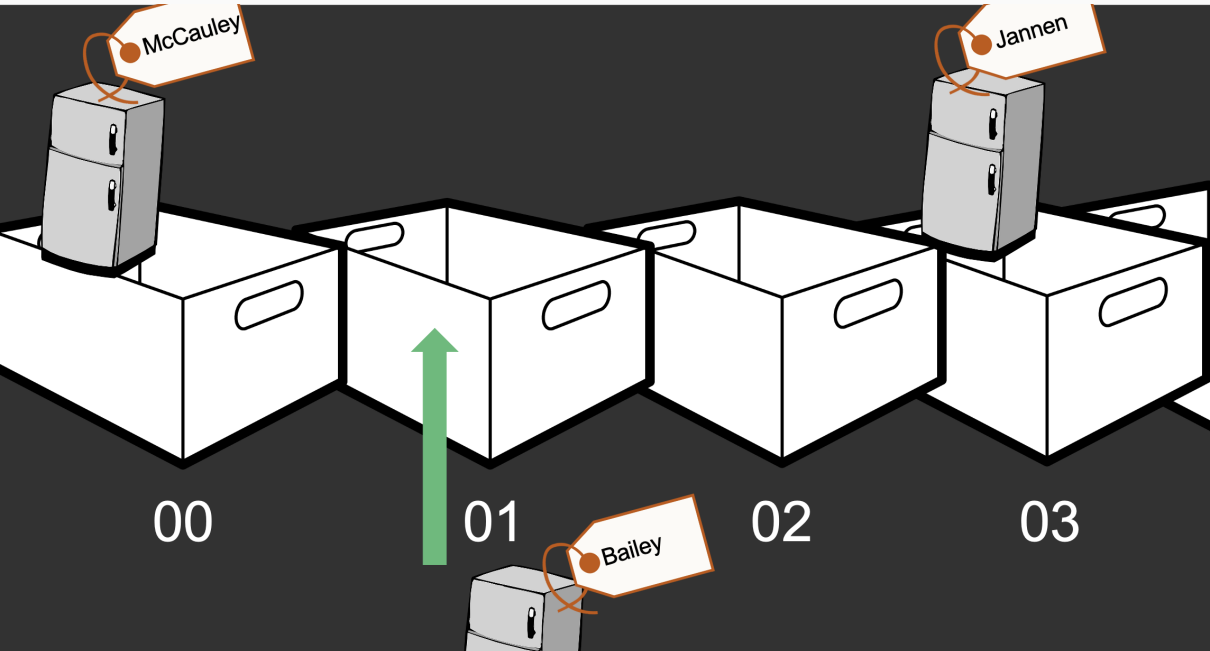
Linear Probing Example



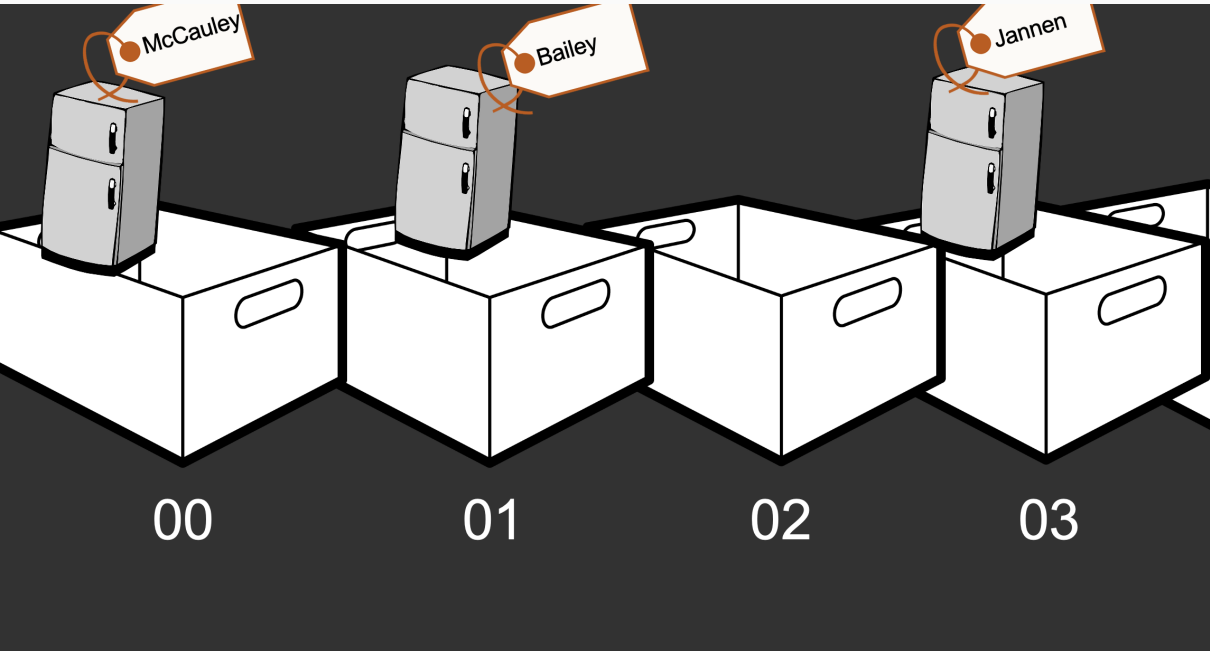
Linear Probing Example



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: `remove(9)`
- Problem: the "run" is broken up
- In pairs: how can we fix this?

Linear Probing Deletes

- Solution for today: 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!
- It *is* possible to move elements back to fix the run
  - Uses the hash table more efficiently
  - Finicky: need to ensure you don’t move any items before their canonical slot (may need to scan ahead and take future items)
  - Placeholders are fairly common in practice: save compute time and programmer effort. We’ll stick to placeholders in this course.

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)

Hashtable Size

Maintaining Hashtable Size

- Like ArrayList, we need to grow when we run out of space
- What do we mean by running out of space?
- We need to make a trade-off between *space* and performance:
  - We want our table size to be large to minimize collisions (and run/chain lengths): leads to good performance, *bad space*
  - We want our table size to be small to minimize wasted space (empty slots): leads to *good space*, bad performance
- Some flexibility (like with `ArrayLists`): we don't know the size up front

Load Factor

- Suppose a hash table with m slots stores n elements
- *Load factor* is a measure of how full the hash table is

$$\text{load factor} = \frac{\text{\# elements}}{\text{\# slots}} = \frac{n}{m}$$

- A smaller load factor means the hashtable is less full, which likely gives better performance

Using the Load Factor

- We can keep a running count of the table's elements so that we always know the load factor
- Given a hashtable's load factor, what should we do?
  - If the load factor is high (say $> .5$), we grow our table
- How to grow?
- `ArrayList`: `ensureCapacity()` allocates a new `Object` array, then copies elements over
- Does this work for hashtables?

Making Hashtables Larger

- Cannot just copy values! (why?)
- The canonical slot might change
- Example: suppose `key.hashCode() == 11`
- Then `11 % 8 == 3` but `11 % 16 == 11`
- How can we handle this?
- To grow our hashtable, we must recompute the canonical slot for each item, then reinsert the item into the new array

When to grow?

- Choose some load factor
- .50 and .66 are very popular; depends a bit on the use case
- Tradeoff between size and performance
- In the lab we'll use very full hash tables: .8. (It will still be quite fast.) Feel free to play around with this value once you're done with the lab.

Array Sizes

- Some people like using hash tables whose size is a prime
- Reason: remember that we use `% array.length` to calculate the canonical slot
- A prime size can help “spread out” the items
- Downside: need to find a prime size when doubling
- We won’t worry about this in this class; just a heads up. You’ll often see a hash table of size 997 or something—this is why.

Choosing Hash Functions

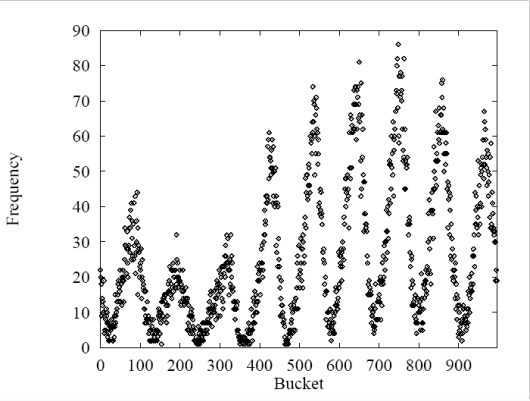
Good Hash Functions

- Good hash functions:
  - Are fast to compute
  - Uniformly distribute keys across the range
- Rules of thumb to make good hash functions?
  - Not really. We almost always have to test “goodness” empirically

Hashing Strings

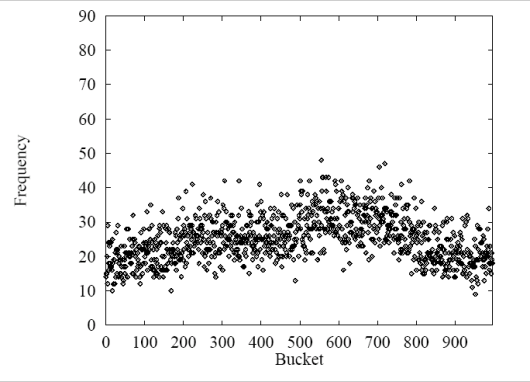
- What are some reasonable hash functions for `Strings`
- One idea: use the first character's Unicode value? (Every character is stores as a number in Java). Problems with this?
  - Can only return `0`–255
  - Not uniform (some letters far more common)
- Sum of the Unicode values of all characters?
  - Still not uniform! (We'll see in a second)
  - Doesn't work well for large hashtables
  - Not good at avoiding collisions: smile, limes, miles, and slime are all the same

Sum of Unicode Values



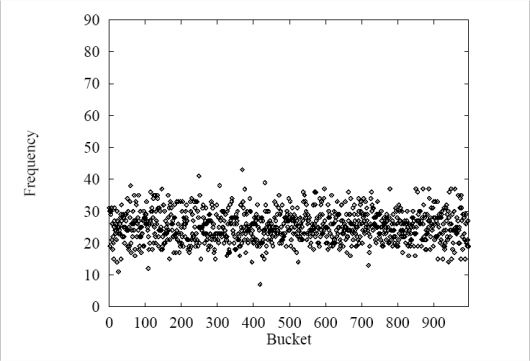
- This is the hash of all words in the UNIX spellchecking dictionary
  - *x*-axis is bucket; *y*-axis is number of words that hash to the bucket
- Uses 997 buckets
- Hash of a string *s*: $\sum_{i=0}^{s.length}$ `s.charAt(i)`

Sum of Unicode Values



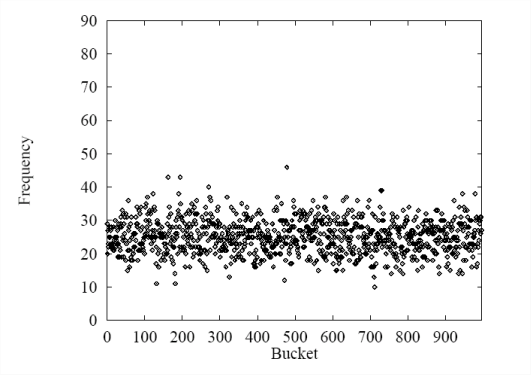
- Hash of a string s: $\sum_{i=0}^{s.length} 2^i \cdot$ `s.charAt(i)`
- Better! But still not great.

Sum of Unicode Values



- Hash of a string s: $\sum_{i=0}^{s.length} 256^i \cdot$ `s.charAt(i)`
- Really good! But do we need numbers as big as 256^i?

Sum of Unicode Values



- Hash of a string *s*: $\sum_{i=0}^{s.length} 31^i \cdot$ `s.charAt(i)`
- This is (essentially) what Java uses to hash strings!

Other Objects?

- Integers?
  - In Java: `i.hashCode()` is `i`
  - Could be terrible depending on your data
  - Might want to use another `hashCode()` method in that case
  - One popular one (has theoretical performance guarantees!):

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

- What about other classes?
- Write your own (probably similar) `hashCode()` methods. Test empirically to make sure elements are spread out

Hashtable Performance

Hashtable Performance

- Given the hash code of an object `o`, how long does `get(o)` take?
- *O*(run length) for linear probing; *O*(chain length) for external chaining
- How long does calculating a hash code take?
  - Can be long for, say, a long string.
  - *O*(1) in terms of the number of items in the hash table
  - Another example of being careful about how we're stating our running time. Usually: in terms of number of strings in the table. But do we care about the length of our strings?

Impact on Performance

- Let's say we have constant load factor
- Assume we have a good hash function
  - Spreads objects out "like random"
- Then an *average* bucket has constant chain length
- An *average* bucket is in a run of constant length
- (With overwhelming probability, never gets worse than $O(\log n)$ for any bucket)
- Usually we say we have $O(1)$ performance. True on average; the actual worst case might be a bit worse

Summary of Map Performance

	`put`	`get`	space
Unsorted ArrayList	$O(n)$	$O(n)$	$O(n)$
Unsorted LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log n)$	$O(n)$
Balanced BST (AVL Tree)	$O(\log n)$	$O(\log n)$	$O(n)$
Hashtable (average)	$O(1)$	$O(1)$	$O(n)$