Hash Tables

- Components
    - keys/elements
    - array
- HashCode
    - hashcode ideally creates a unique integer for each key based on the key's fields
        - hashcode is based on the fields of the object
        - two objects that are equal--have the same fields--produce the same hashcode
        - if overriding equals or hashCode, need to override the other as well
- hash function maps keys to array indices
        - hash function uses key's hashcode and the array length in computation, typically key.hashCode() % array.length
        - another hash function squares the hashCode and returns middle digits
        - another hash function multiplies the key's hashCode by a number less than 1 and returns the first few digits of the decimal
- Open-Address Hashing
    - dealing with collisions
        - collision: when there is already an item where a new element belongs
        - linear probing: search next indices one at a time until you find an empty space
            - causes clustering: when many items end up next to each other
            - avoid clustering with double hashing:
                - a second hash function determines how the next index is found
                - data.length and data.length-2 must be prime numbers
                    - If you want to know *why*, see Donald Knuth's The Art of Programming, Vol 3
                    - Donald Knuth is like a CS theory god
                - second hash function returns a number between 1 and data.length-2
        - quadratic probing: $h+1$, $h+2^2$, $h+3^2$, $h+4^2$, …
    - Worst-case runtime: O(n)
        - Have n collisions before finding the element or an available location
    - Typically adding/searching/removing takes O(1) with open address and a good hash function
    - What happens if we remove an element from the table?
        - What if we search for another element that had previously collided with that element?
        - One way to deal with this is a boolean array to record whether there used to be something there

        - Another way is to use “placeholders” or “zombies”--put a dummy element or a sentinel value in the spot to replace a removed item
            - When searching, pretend the zombie is an element
            - When adding, replace the zombie

- Chained Hashing
    - each array index has a linked list
    - if there is a collision, just add to the list
        - speeds up adding O(1)
        - searching and removing can still be slow, O(n)
            - If our hashCode and hash functions were really bad and everything ended up at the same index, we’d be searching through a linked list with n elements
- Efficiency
    - depends on the load factor, or a percentage of the array’s usage
    - To keep things efficient, we have a max capacity; in your homework this is ¾
        - as soon as your hashtable is 75% full, you make a bigger array and rehash--DO NOT COPY--the elements
            - the results of the hash function depend on the array length, so the hash values will no longer be the same!
    - depends on the type of hashing (open-address vs chained) and on the collision handling technique with open-address
    -