Implement the Map ADT

- ***The ADT:***
    - *You will need each item to have both a **key** and a **data** value, like the priority queue has both a data value and a priority.*
    - **insert(key, value)**
        - Adds an item to the map with this ***key*** and ***data*** value
    - **find(key)**
        - Returns the ***data*** value associated with that ***key***
    - **update(key, value)**
        - Sets a new value for the ***data*** associated with that ***key***
    - **remove(key)**
        - Removes the ***item*** associated with that ***key*** from the map
- Use any method you like, including built in python functionality
    - Doesn't matter if you order while adding and binary search while updating/finding/removing, or linear search while updating/finding/removing
        - Just get things to work consistently
    - Using a built in python dictionary or built in sort functionality is not really implementing it, so make sure you choose an implementation that is also good programming exercise. It's still OK to do it first using built in methods, just to get it working.
    - What is the *time-complexity* of the main operations in your implementation?
- Two recommended methods (implement both for practice):
    - Singly linked list
        - insert() can be a simple head insert
        - find/update can iterate through the list to find the key
        - remove can use recursion to remove the correct item from the list
    - Array or python list
        - insert can be implemented like a insert_ordered method
            - Ordered on key
            - Floats each value down the list
                - Shifts each value one back while searching for location
        - find/update/remove can use binary search to find the correct value
            - Remove then shifts values to finish
    - What is the time complexity of each operation?
- *Always add any operations that may help you to test the implementation, both in these class exercises and in programming assignments.*
    - For example:
        - ***__len__(self)***
        - ***__str__(self)***

Implement a hash function and test its distribution

- *The declaration*:
  - ***hash(some_value)***
  - Returns an integer
  - Always returns the same ***hash value*** for equal ***some_value***
- Do whatever you like to ***some_value*** to hash it, but bear in mind:
  - Computational complexity of a hash function can immensely impact the efficiency of the program functionality that uses it, i.e. a hash table data structure
  - Good distribution is achieved when hash values returned are dissimilar
    - *Seemingly random, but not actually random*
  - Values sent into hash functions are often similar or have patterns to them
    - *The hash function would do well to eliminate these patterns*
- *Try hashing an integer, a string, a pair of integers and a class with several variables*
  - In the class implement these functions for hash(my_class_instance) to work:
    - ***__hash__(self)***
    - ***__eq__(self, other)*** is also needed for use with hash tables
- **Testing the distribution of a hash function**
  - *You can try this testing method first with a random function to get it working*
  - Make an indexable list of a specific amount of integers.
    - *lis = [0] * lis_size*
  - Generate a considerable number of values to be hashed
    - Hash each value, then take the modulo of the hash value and the list size
      - *index = hash(some_value) % lis_size*
    - Raise the integer value in the list at that index by one
      - *lis[index] += 1*
  - Print the list to see how the values were distributed
  - Try this with different list sizes
    - Whole powers of 2 are common sizes of hash tables
      - 4, 8, 16, 32, 128, etc.
    - Whole tens are common in people's programming
      - 10, 20, 50, 100, 1000, etc.
    - Prime numbers can actually make the distribution better
      - 13, 17, 23, 53, 113
      - *You can't count on prime number list size though*
  - The modulo and list size are in fact the system's ***compression function***
  - *Here is the method that will be used to assess the distribution of hash functions:*
    - Find the min and max values in the distribution list and their difference
      - *difference = max - min*
    - Divide the difference by the max value to get a ratio
      - *ratio = difference / max*
    - A perfect distribution will yield a ***ratio*** of **zero**
    - *This will then be tested with a variety of list sizes and quantities of values*