CSCI-SHU 210 Data Structures

Homework Assignment10 Dictionaries and Hashing

1. Most frequent element in a python list.

Given an input python list, return the most frequent occurrence element in this python list.

For simplicity, you can assume there's only one most frequent element. No ties.

```
Input:
   L1: ['a', 'b', 'c', 'd', 1, 'a', 'b', 'b']
Output:
   'b'  (Because 'b' occurs 3 times)
```

```
Your task 1: implement function most_frequent(l1) in
most_frequent.py.

This function returns the most frequent element in l1.
```

Important:

- Required runtime: O(N) expected
- Use python dictionary to solve this problem.

2. Cuckoo hashing

The Cuckoo is a bird that does not make its own nest, rather it finds a nest of another bird and kicks the eggs out and puts its own eggs in.

The Cuckoo hashing is a different hash table implementation that guarantees O(1) search (Worst case O(1), wow!). If you remember, our previous hash table had O(1) expected search time, O(n) worst case search time.

Your task is to implement Cuckoo hashing hash table. It should work like a Map ADT. With internal implementation -- Cuckoo hashing.

```
Your task 4:
Implement the following functions in cuckoo_hashing.py:
def __getitem__(self,key):
def __setitem__(self,k,v):
def __delitem__(self,k):
def __len__(self):
def __contains__(self,key):
def __iter__(self):
def keys(self):
def values(self):
def items(self):
```

Important:

- Hash functions are already given and works. You can change them, but make sure your new one works.
- Define as many new functions as you like.
- Make sure you pass the test code.
- This is a Map ADT, it should have the same functionality as other Maps (Works like a dictionary).
- Implementation must be cuckoo hashing. You can't just use other implementation of Map ADT to pass the test code.

Ok, so let's talk about how cuckoo hashing works.

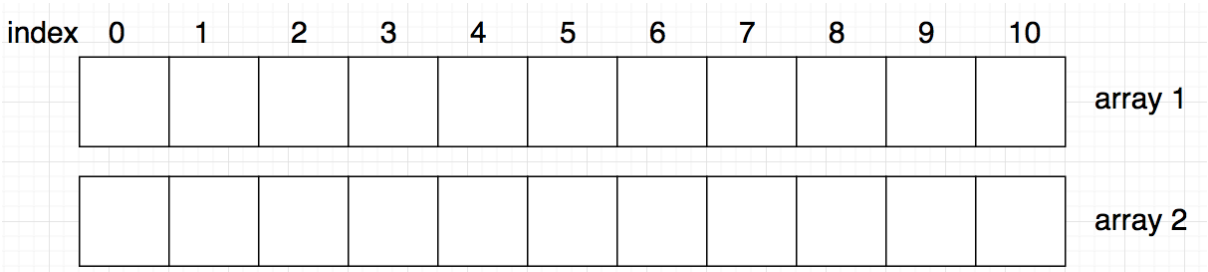
- Two (or more) python lists are used to store (key, value) pairs.
- Two (or more) hash functions are defined. First hash function calculates the corresponding index on the first python list, second hash function calculates the corresponding index on the second python list.
- When inserting a (key, value) pair (calling __setitem__),

  use the first hash function to calculate the index K1 on the first python list.
  use the second hash function to calculate the index K2 on the second python list.

  - If first list index K1 stores the key, we update the value. Done
  - If second list index K2 stores the key, we update the value. Done
  - Otherwise, start performing "kicking" operation.
    - Kicking means, move the previous (key, value) pair from the first list to the second list, or from the second list to the first list. The next corresponding index is calculated using hash function 1 or hash function 2. (Switch every time)
- When we perform kicking, we may encounter a kicking loop. If loop happens, we should resize the table. (Double the size of both python lists)

Cuckoo hashing example:

1) Starting point:

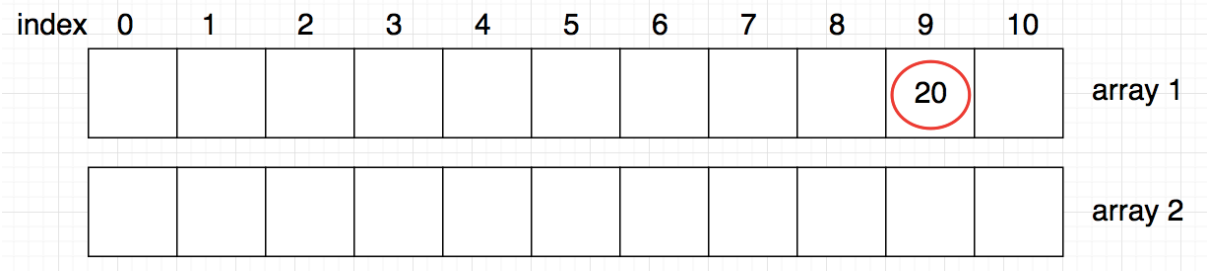


Insert those: 20, 50, 53, 75, 100, 67, 105, 3, 36, 39

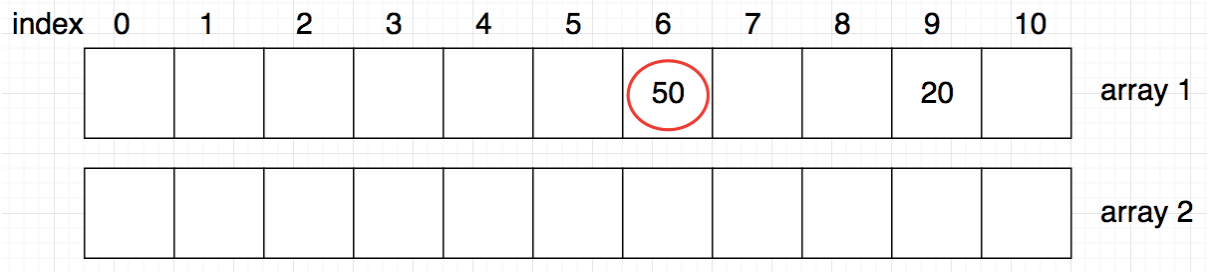
Hash function: h1(i) = i % 11

h2(i) = (i // 11) % 11

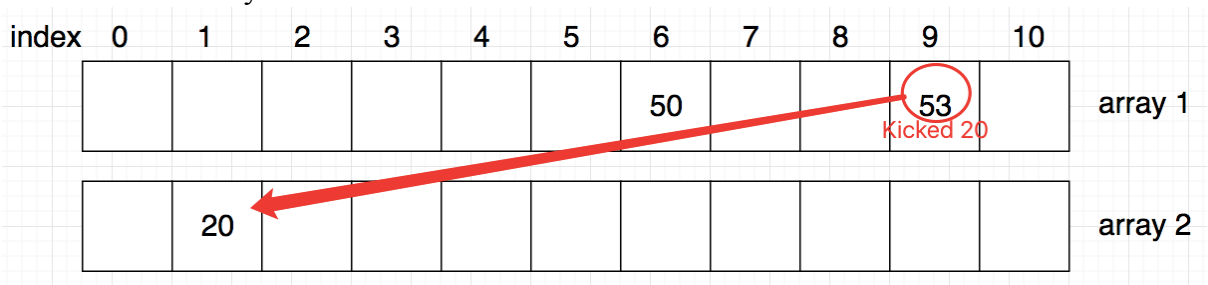
2) Try first hash function on 20, we get index 9. Index 9 on array 1 is not used. Great!



3) Try first hash function on 50, we get index 6. Index 6 on array 1 is not used. Great!

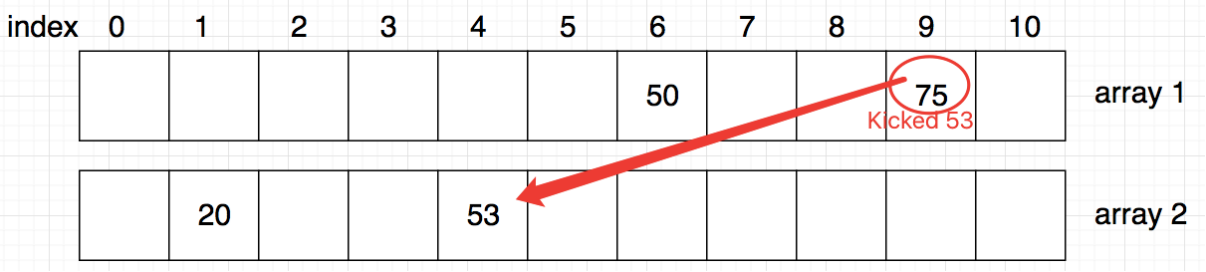


4) Try first hash function on 53, we get index 9. Index 9 on array 1 is used.
So we kick out what was there. Use second hash function on what was there. h2(20) = 1.
Index 1 on array 2 is not used. Great!

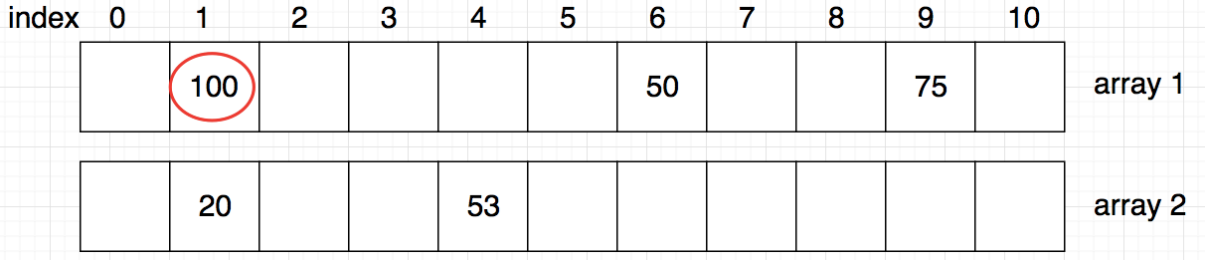


5) Try first hash function on 75, we get index 9. Index 9 on array 1 is used.

So we kick out 53. Try second hash function. h2(53) = 4. Index 4 on array 2 is not used. Great!

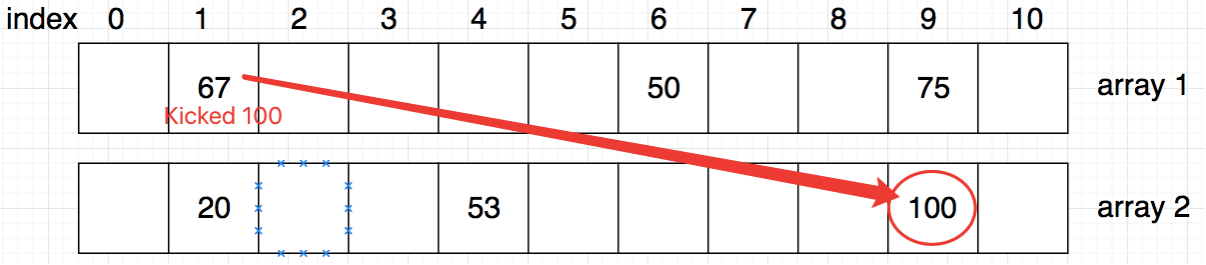


6) Try first hash function on 100, we get index 1. Index 1 on array 1 is not used. Great!



7) Try first hash function on 67, we get index 1. Index 1 on array 1 is used. Kick 100 out!

h2(100) = 9. Index 9 on array 2 is not used. Great!

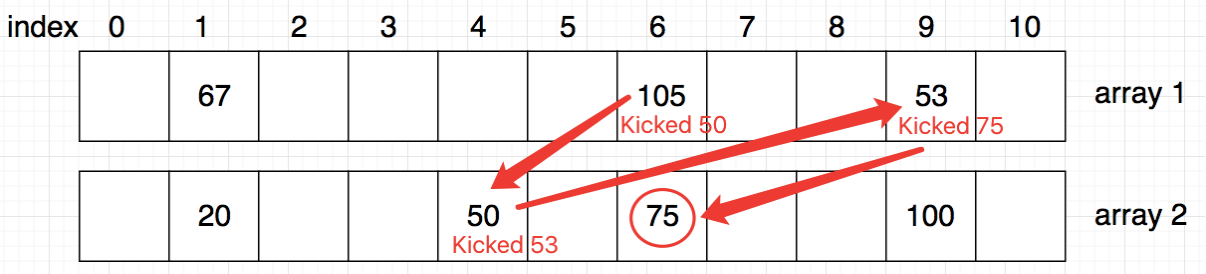


8) Try first hash function on 105. We get index 6. Index 6 on array 1 is used. Kick 50 out!

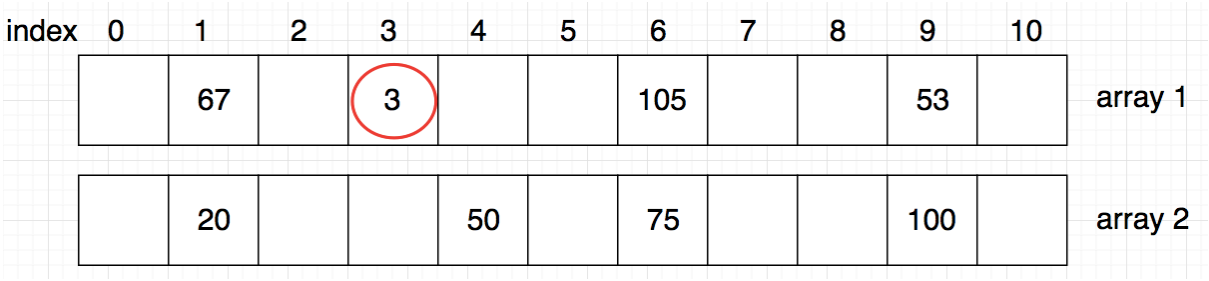
h2(50) = 4. Index 4 on array 2 is used. Kick 53 out!

h1(53) = 9. Index 9 on array 1 is used. Kick 75 out!

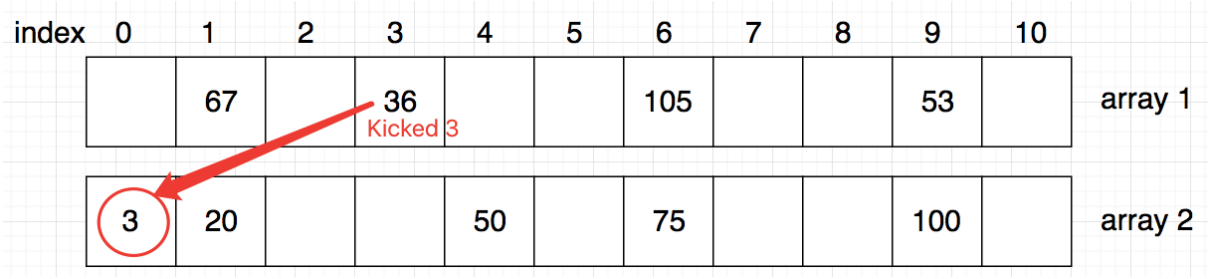
h2(75) = 6. Index 6 on array 2 is not used. Great!



9) Try first hash function on 3. We get index 3. Index 3 on array 1 is not used. Great!

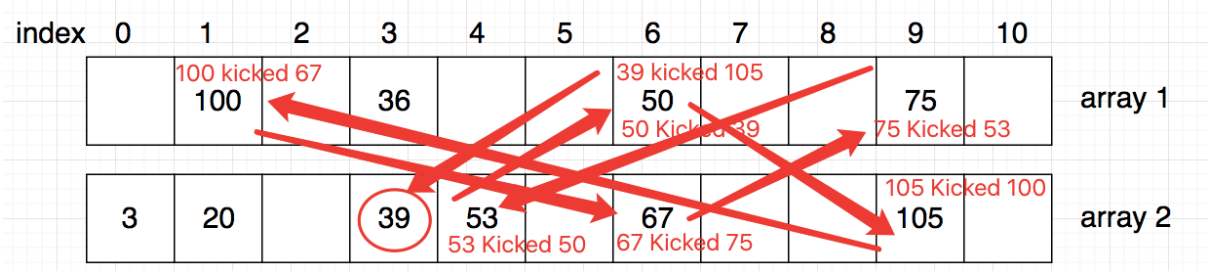


10) Try first hash function on 36. We get index 3. Index 3 on array 1 is used. Kick 3 out!
h2(3) = 0. Index 0 on array 2 is not used. Great!



11) Try first hash function on 39. We get index 6. Index 6 on array 1 is used. Kick 105 out!

Many kicks are performed here.
h2(105) = 9. Used
h1(100) = 1. Used
h2(67) = 6. Used
h1(75) = 9. Used
h2(53) = 4. Used
h1(50) = 6. Used
h2(39) = 3. Great!



Cycle

If you now wish to insert the element 6, then you get into a cycle. In the last row of the table we find the same initial situation as at the beginning again.

table 1	table 2
6 replaces 50 in cell 6	50 replaces 53 in cell 4
53 replaces 75 in cell 9	75 replaces 67 in cell 6
67 replaces 100 in cell 1	100 replaces 105 in cell 9
105 replaces 6 in cell 6	6 replaces 3 in cell 0
3 replaces 36 in cell 3	36 replaces 39 in cell 3
39 replaces 105 in cell 6	105 replaces 100 in cell 9
100 replaces 67 in cell 1	67 replaces 75 in cell 6
75 replaces 53 in cell 9	53 replaces 50 in cell 4
50 replaces 39 in cell 6	39 replaces 36 in cell 3
36 replaces 3 in cell 3	3 replaces 6 in cell 0
6 replaces 50 in cell 6	50 replaces 53 in cell 4

So, what should we do if we have a cycle?

In __setitem__ function, perform kicking operation maximum 2 * current_number_of_elements (2 * len(self)) times. If more than 2 * current_number_of_elements (2 * len(self)) times kicks were performed and we still cannot find an empty slot, double the size of the table.

The reason is, if we have tried 2 * current_number_of_elements times, we have tried both hash functions on each existing element. Thus, time to double the size!

Double the size of the table means:

- Double the size of both arrays
  (2 * maxsize – 1 would be even better, hash table prefers odd size)
- Get new random numbers (Optional)
- Move items from old table to the new double sized table using new hash function.
  (Well, not a new hash function, but the random number is changed.)