- This lab will cover Hash Maps.
- You may want to refer to the text and your lecture notes during lab as you solve the problems.
- When approaching the problems, think before you code. It is good practice and generally helpful to lay out possible solutions for yourself.
- You should write test code to try out your solutions.
- You must stay for the duration of the lab. If you finish early, you can help other students to complete the lab. If you don't finish by the end of the lab, it is recommended that you complete the lab on your own time.
- Your TAs are available to answer your questions in lab, during office hours, and on Piazza.

Vitamins (maximum 30 minutes)

---

1. Given a bucket array of size **N = 7**, perform the following operation using separate chaining. Draw the abstract representation of the bucket array on a piece of paper. Trace all the changes. Use **Multiply-Add-Divide (MAD) for compression method**. The MAD method maps an integer i as follows:

$$[(ai + b) mod\ p] mod\ N$$

Where, a, b and p are specific to the instance of the hash table object: p is a prime number bigger than U; a is a number within the range of [1, p-1]; b is a number within the range of [0, p-1].

For this problem, assume p = 107, a = 1, b = 2. The items don't have any values associated with them, meaning they are just integers. You will have to rehash when the number of items, n is greater than the total capacity N, i.e. when n>N

   a. Insert 37
   b. Insert 47
   c. Insert 51
   d. Delete 37
   e. Insert 65
   f. Insert 104
   g. Insert 8
   h. Insert 5
   i. Insert 10
   j. Insert 7
   k. Delete 8

2. Perform the above operations **(a) through (f)** using open addressing with Linear Probing for collision resolution. Use the **Division method** for compression. The division method simply performs the modulo operation for a given integer i:

$$i \bmod N$$

Here, the N is the capacity of the table. Assume the hash code just simply returns the number instead of doing any calculation on it. The items don't have any values associated with them, meaning they are just integers. Draw out the bucket array on a piece of paper. Trace all changes. You will have to rehash when the load factor of the hash table is **greater than 0.5**. The load factor of a hash table is calculated as follows:

$$Load\ Factor\ =\ \frac{\#\ of\ items\ in\ the\ table}{Total\ Capacity\ of\ the\ table}$$

3. A hash table of length 10 uses open addressing with hash function h(k)=k mod 10, and linear probing. After inserting 6 values into an empty hash table, the table is as shown below.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

(A) 46, 42, 34, 52, 23, 33
(B) 34, 42, 23, 52, 33, 46
(C) 46, 34, 42, 23, 52, 33
(D) 42, 46, 33, 23, 34, 52

4. Given a hash table T with 25 slots that stores 2000 elements, the load factor α for T is _____.

Coding

In this section, it is strongly recommended that you solve the problem on paper before writing code.

1. Implement the function:

   ```
   def most_frequent(lst)
   ```

   This function is given lst, a list of numbers, and returns the number that appears most frequently.

   For example, if `lst=[5,9,2,9,0,5,9,7]`, the call `most_frequent(lst)` should return 9, since it appears more than any other number. Give an implementation for `most_frequent` that optimizes the **average-case** runtime.

   Note: You can assume that the most frequent number in `lst` is unique.

   Determine the **worst-case** runtime of your function.

   What data structure could be used which would improve the **worst-case** runtime?

2. Open Addressing with Linear Probing

   Implement a class `OpenAddressingHashMap` that uses linear probing for collision resolution. Use python's builtin hash function `hash()` to obtain the hash code. Use the Divide Method for compression.

   The table should resize whenever the **Load Factor is greater than 0.5 or less than 0.125**. In order to resize you will double (or halve) the capacity of the table and, consequently **rehash** every single existing element in that new table.

   In order to help with search, we need to mark each index of the table with its state: empty or formally occupied. Use a string (or any variable of your choice) to serve this purpose.

```
class OpenAddressingHashMap:
      class Item:
         def __init__(self, key, value=None):
             self.key = key
             self.value = value

      def __init__(self):
      '''Put any useful member variable here. Suggestions: Capacity N,
size of the hash table n'''

      def __len__(self):
      '''Returns the number of elements in the map'''

      def is_empty(self):
      '''Returns whether the map is empty or not'''

      def __getitem__(self, k):
      '''Finds the key k in the table. And returns the value associated
with key k. If key not found, raises a KeyError'''

      def __setitem__(self, k, v):
      '''Finds the key k in the table. If found the already existing
value at that key is replaced by v. If not, the item (k,v) is
inserted.'''

      def __delitem__(self, k):
      '''Finds the key k in index j and marks it as available
(vacated)'''
```

```
        def __iter__(self):
        '''Iterates through the bucket array'''

        def rehash(self, new_size):
        '''Updates the capacity of the bucket array; reinserts
    everything'''
```

Extra: One of the disadvantages of Linear Probing is that it can result in clustering in which keys form continuous blocks in the array. Finding a particular key in this case requires iterating over all of the elements in the cluster. This scenario degrades the performance of the hash map.

To experiment, insert a number of keys into your OpenAddressingHashMap. Measure the size of the resulting clusters. Create a histogram showing the size of the clusters.