Vitamins

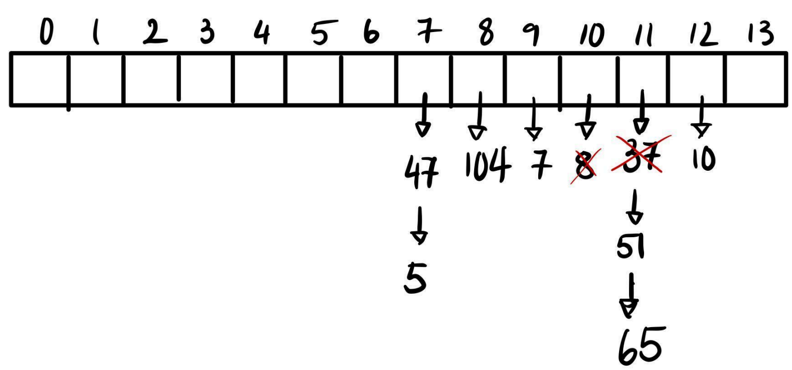
1. Given an array of size **N = 7**, perform the following operation using **separate chaining**. Draw the abstract representation of the array on a piece of paper. Trace all the changes. Use **Multiply-Add-Divide (MAD) for compression method**. The MAD method maps an integer i as follows:

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

p is a prime number
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. (10 minutes)

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

__________. 2000/25 = 80

3. Analyze the worst-case and average runtime of the following function, and give it an appropriate name(5 minutes):

#same character sets, is_permutation, etc

```
def is_anagram(s1, s2): # s1 (m), s2 (n)
    fMap = ChainingHashTableMap( ) #fMap = frequency Map

    for char in s1:
        if char not in fMap:  O(1) average
            fMap[char] = 0

        fMap[char] += 1       O(1) average

    for char in s2:
        if char not in fMap:  O(1) average
            return False

        fMap[char] -= 1       O(1) average

    for key in fMap:
        if fMap[key] != 0:    O(1) average
            return False

    return True
```

n = len(s1)
m = len(s2)

Average runtime: O(n + m)

What are the outputs of the following?

```
print(mystery("cheaters", "teachers")) True

print(mystery("engineering", "gnireenigne")) True

print(mystery("Python", "nohtyp")) False
```

Coding (See CS1134 Lab13 Solutions.py)

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

Download the **ChainingHashTableMap.py** & **UnsortedArrayMap.py** file under Resources/Lectures on NYU Classes

1.
   a. Implement the function (10 minutes):

```
def most_frequent(lst):
```

   This function is given lst, a list of numbers, and returns the number that appears most frequently. You can assume that the most frequent number in lst is unique. Give an implementation that optimizes the **average-case** runtime.

   ex) 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.

   b. Using your implementation from above, write the following function (10 minutes):

```
def first_unique(lst):
```

   This function is given lst, a list of numbers, and returns the first number that is not repeated in the list. Give an implementation that optimizes the **average-case** runtime.

   ex) lst = [5,9,2,9,0,5,9,7] the call first_unique(lst) should return 2, since it is the first non duplicate number, the others being 0 and 7.

   c. What is the worst-case extra space complexity of your two functions? Suppose that instead of a list of integers, you are given a string of lower-case and upper-case letters (symbols not included). Does the extra space complexity change? O(1) space if strings only

2. In homework 2, you had to find two values in a list of integers that sum up to a target value. You will do the same here, except this time the list is **not sorted**.

   Given a list of integers and a target value, write a function that returns a tuple of two integer values representing the indices summing up to the target. If no pair is possible, return (None, None).

   Give an implementation that optimizes the **average-case** run-time of O(n). (30 minutes)

```
def two_sum(lst, target):
```

   ex)

   lst = [-2, 11, 15, 21, 20, 7], target = 22, the function should return (2,5)______

   lst = [-2, 11, 15, 21, 20, 17], target = 22, the function should return (None, None)___

   Hint: You should use the ChainingHashTableMap function to achieve the run-time.

3. In this question, we will look at the `PlayList` ADT. This ADT is used to maintain a sequential collection of songs. Each song could be played individually, or all songs could be played sequentially (in the order they were inserted in). (35 minutes)

   Define this ADT, which supports the following behavior:
   - `pl = PlayList( )` - creates an empty PlayList object.
   - `pl.add_song(new_song_name)` - adds the song new_song_name to the end of the songs sequence
   - `pl.add_song_after(song_name, new_song_name)` - adds the song new_song_name to the songs sequence, right after song_name; or raise KeyError exception if song_name not in the play list
   - `pl.play_song(song_name)` - plays the song song_name; or raise KeyError exception if song_name not in the play list
   - `pl.play_list( )` - plays all the songs in the PlayList by their sequential order.

   To simulate playing a song, have the methods `print("Playing " + song_name)`.

   <u>**Implementation Requirements**</u>:
   1. `add_song`, `add_song_after`, `play_song` should run in $\Theta(1)$ **average time**.
   2. `play_list` should run in $\Theta(n)$ average time, n being the number of songs in the play list.
   3. You may not use Python's built-in dict. Use the `ChainingHashTableMap` instead, which has the same behavior. If needed, you may use ONE additional data structure implemented in class to help you:

      ```
      ArrayStack
      ArrayQueue
      DoublyLinkedList
      BinarySearchTree
      ```

Test Code for the PlayList ADT:

```
#Feel free to listen to these itunes top hits while you code :)

p1 = PlayList( )
p1.add_song("Jana Gana Mana")
p1.add_song("Kimi Ga Yo")
p1.add_song("The Star-Spangled Banner")
p1.add_song("March of the Volunteers")
p1.add_song_after("The Star-Spangled Banner", "La Marcha Real")
p1.add_song_after("Kimi Ga Yo", "Aegukga")
p1.add_song("Arise, O Compatriots")
p1.add_song("Chant de Ralliement")
p1.add_song_after("Chant de Ralliement", "Himno Nacional Mexicano")
p1.add_song_after("Jana Gana Mana", "God Save The Queen")

p1.play_song("The Star-Spangled Banner")
p1.play_song("Jana Gana Mana")

p1.play_list( )
```

Output:

```
"Playing The Star-Spangled Banner"
"Playing Jana Gana Mana"
"Playing Jana Gana Mana"
"Playing God Save The Queen"
"Playing Kimi Ga Yo"
"Playing Aegukga"
"Playing The Star-Spangled Banner"
"Playing La Marcha Real"
"Playing March of the Volunteers"
"Playing Arise, O Compatriots"
"Playing Chant de Ralliement"
"Playing Himno Nacional Mexicano"
```

4. For this question, you will implement another data type, `Set`, which is similar to a `Map`. A set is similar to a map in that it has the following properties:

   - **all keys in the collection are unique**
   - **all keys in the collection are unordered**

   Instead of having a (key, value) pair, **sets only have keys**. Therefore, it is ideal to use a set if you only care about unique keys instead of setting the value of each key to None.

   For familiarity, Python has a built-in set (`set`) and map (`dict`).

   Using dict literals `{ }`, we can create a dictionary with the following:

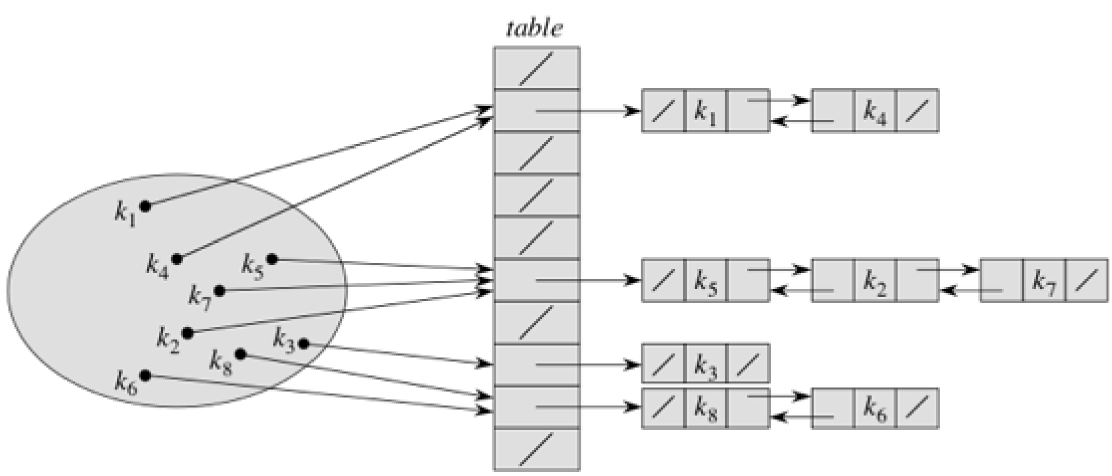
```
dict1 = {1 : "apple", 2 : "banana", 3: "orange"}
print(type(dict1)) #<class 'dict'>
```

   However, if we use the same literals with only keys, we end up creating a set:

```
set1 = {1, 2, 3}
print(type(set1)) #<class 'set'>
```

Note: Since we only have unique keys, we do not need the item class. Instead of using the UnsortedArrayMap file, we will just use a **DoublyLinkedList** as our secondary collections.

Here is a representation of the `ChainingHashTableSet`:



Each key is mapped to a slot (index) in the hash table using a hash function, and is then placed into the doubly linked list bucket in the respective slot.

Download the **ChainingHashTableMap.py** file under Resources/Labs on NYU Classes

The `ChainingHashTableSet` will use the same hash function that was implemented in lectures.

Your task is to modify the existing methods in the ChainingHashTableMap file for a ChainingHashTableSet and additionally define the add, remove methods. (35 minutes)

```
class ChainingHashTableSet:

    def __init__(self, N=64):
    #modify this to support the set ADT

    def rehash(self, new_size):
    #modify this to support the set ADT

    def __iter__(self):
    #modify this to support the set ADT

    def __contains__(self, key):
    #modify this to support the set ADT


    def add(self, key):
    ''' Adds a key to the set. If the key already exists, do
    nothing. You may want to refer to the __setitem__
    implementation of the ChainingHashTableMap. '''


    def remove(self, key):
    ''' Removes a key from the set. If the key doesn't exist, raise
    a KeyError. You may want to refer to the __delitem__
    implementation of the ChainingHashTableMap  '''



def print_hash_table(hset):
#modify this to support the set ADT
```

OPTIONAL **SETS WILL NOT APPEAR ON THE FINAL EXAM!**

Sets are commonly used with logic operations: **AND, OR**. You will learn more about sets when you take your Discrete Mathematics course; here is a head start with some extra credit for you!

The **AND** operation, also called the **intersection of 2 sets** is a set containing keys that exist in **BOTH** sets. We use the ∩ or & to denote the operator.

ex)

```
set1 = {1, 2, 3, "apple", "banana"}
set2 = {1, 3, "orange"}

set1 & set2 returns {1, 3}
set1.intersection(set2) returns {1, 3}


#commutative property: a & b == b & a
set2 & set1 returns {1, 3}
```

The **OR** operation, also called **the union of 2 sets** is a set containing keys that exist in **EITHER** sets. We use the ∪ or | to denote the operator.

ex)

```
set1 = {1, 2, 3, "apple", "banana"}
set2 = {1, 3, "orange"}

set1 | set2 returns {1, 2, 3, "apple", "banana", "orange"}
set1.union(set2) returns {1, 2, 3, "apple", "banana", "orange"}


#commutative property: a | b == b | a
set2 | set1 returns {1, 2, 3, "apple", "banana", "orange"}
```

Finally, there is the **difference of 2 sets**, which is a set containing keys that exist in **ONE SET BUT NOT THE OTHER**. We use the - to denote the operator.____

ex)

```
set1 = {1, 2, 3, "apple", "banana"}
set2 = {1, 3, "orange"}

#notice that unlike with OR and AND, the operation is not commutative!

set1 - set2 returns {2, "apple", "banana"}
set2 - set1 returns {"orange"}

set1.difference(set2) returns {2, "apple", "banana"}
set2.difference(set1) returns {"orange"}
```

Add the following methods to your `ChainingHashTableSet` class:

```
def intersection(self, other):
#returns a new set containing the intersection of the two sets
#self is set1, and other is set2

def __and__(self, other):
#same as intersection, but allows you to do set1 & set2


def union(self, other):
#returns a new set containing the union of the two sets
#self is set1, and other is set2

def __or__(self, other):
#same as union, but allows you to do set1 | set2


def difference(self, other):
#returns a new set containing the difference of the two sets
#self is set1, and other is set2


def __sub__(self, other):
#same as difference, but allows you to do set1 - set2
```

Hint: the operators implementations __and__, __or__, and __sub__ should only be driver methods. You should use the methods implemented in part 4.