Lab 2: Modular Hashing Lab

This is an INDIVIDUAL assignment. Due date is as indicated on BeachBoard. Follow ALL instructions otherwise you will lose points. In this lab, you will be implementing two functions from a class called `hash_table`. This will require the use of hashing functions and linear probing.

Background:

Searching for a certain value in a large database or list can take a long time especially if that value is stored in a space that is the last place that you look. See example below:

Index	0	1	2	3	4	5	6	…	20483958	20483959	20483960	20483961
Value	#	#	#	#	#	#	#	…	49842	428	32532	35246

If you tried to find 32532 in the array using traditional brute force, it would take a long time because 32532 is at the end of the array. You would have to check around 20483960 memory addresses before finding it.

A hash function is a function that generates a key for a value. This key allows us to

1. Strategically place your data
2. Easily find your data when you need it

There are many types of hashing, but we will focus on modular hashing.

$$h(k) = k \bmod m$$

- *h(k): mapping*
- *k: key*
- *m: number of addresses*

Another example:

293587
85023840
54234914
39482905
9850293
209524
98520398
623407

Use the numbers on the left as our example. There are 8 numbers in our array and we want to arrange them in a way that allows us to find them as quickly as possible. Because there are 8 spots in the array, we are going to use mod 8 to define these locations for us. For example: `293587: 293587 mod 8 = 3`. So, we'll put this number in index 3 `85023840:85023840 mod 8 =0`. We'll put this number in index 0 Continue this pattern….

Finally we'll have an array that looks like below. This is called a hash table:

0	85023840
1	39482905
2	54234914
3	293587
4	209524
5	9850293
6	98520398
7	623407

Let's say I want to find or have access to a certain number such as `98520398`.

If I wanted to use a brute force algorithm to find it in the array above, it would take me 7 tries if I started from the beginning of the array and 2 tries if I started at the end of the array.

However, if I want to find it in my hash table, I would plug it into the hash function that I used to create the hash table `x mod 8` and I would get a remainder of 7. In 1 try, I can find the value I was looking for.

In real life, we can have collisions. To deal with collisions, there are multiple strategies that programmers use, but we will cover linear probing ONLY. We won't talk about chaining, quadratic probing, or clustering in this class. You can learn that in cecs328
Collision: When a hash function maps two different keys to the same address
Linear probing: A technique used to overcome collision where the key is re-hashed by placing the value in the next available slot in the table.

Example:

0	
1	
2	10 / 66
3	
4	
5	
6	
7	

If you try to insert `10`, you would put it in address `2` because `10 mod 8 = 2`
If you try to insert `66`, that would also give you address `2` because `66 mod 8 = 2`
This is what we call a collision.

How to circumvent the issue (linear probing)

- Re-hash the value by placing it in the next available slot in the table

0	
1	
2	10
3	66
4	
5	
6	
7	

If you try to insert `83`, you would put it in address `4` because `83 mod 8 = 3`, but slots `3` is already taken. The next available position would be `4`
If you try to insert `42`, you would put it in address `5` because `42 mod 8 = 2`, but slots `2, 3, 4` are already taken. The next available position would be `5`

0	
1	
2	10
3	66
4	83
5	42
6	
7	

In the case that you reach the end of the hash table during linear probing, you want to loop back around and continue checking from index `0`.

Instructions:

1. Take a close look at the `hashing.py` file. There are some functions that have already been implemented. In addition, there are two empty functions: `linear_probe(value, start_index)` and `hash(value)`. Read through both of their descriptions carefully. Remember, you will lose points if you do not follow the instructions. We are using a grading script.

`Linear_probe(value, start_index)`	
input	• value– value to be inserted • start_index– where linear probing starts
output	• returns the index of the hash_table that the value should be inserted after linear probing
restrictions	• Although you can implement this function with just one input, DO NOT alter the function heading
assumptions	• value will always be an integer • your table will always be big enough

`to_hash(value)`	
input	• value– value to be inserted
output	• Do not return anything. Just insert value into the proper position in self.table. Utilize linear_probe and insert in this function
restrictions	•
assumptions	• value will always be an integer • your table will always be big enough

2. Please note that a few functions are already completed: `insert(value, index)` `get_table()` and `__str__()`. You are required to use the `insert()` function in your `hash()` function. However, the other two functions `get_table()` and `__str__()` are provided for debugging purposes (if you're having issues with your code) and do not have to be used in any of your functions. Do NOT alter the already implemented functions in ANY significant way.

3. Your job is to implement both `linear_probe()` and `hash()` so that it passes any test case. There are two sample test cases provided for you, but these are not the only cases that we will test. We will be testing other test cases in the same way the test cases are presented.
   Note: The grading script will test `linear_probe()` and `hash()` separately! Although you can implement `hash()` without `linear_probe()`, not implementing `linear_probe()` will result in lost points.
   Another note: For `linear_probe()`, you do not have to use both inputs. If you think that you can implement this function with just one of the inputs, then you can do that. Two inputs were provided for testing purposes.

4. After completing these functions, comment out the test cases (or delete them) or else the grading script will pick it up and mark your program as incorrect. At the minimum, the test cases must be commented/deleted. It is up to you if you want to remove the `checker()` function.

5. Convert your `hashing.py` file to a `.txt` file. Submit your `hashing.py` file and your `.txt` file on BeachBoard. Do NOT submit it in compressed folder.

6. Do not email us your code asking us to verify it. We will answer general questions, but we will not debug your code over email.

Grading rubric

Points	Requirement
15	Implemented `linear_probe()` correctly
15	Implemented `hash()` correctly
5	Passes 2 original test cases (also commented out or deleted the test cases)

*** Note: If your program has an error, you will automatically get a 0 on the lab! Double check to make sure that you do not have any errors!!!
** If we find that you have significantly altered `insert()` or `get_table()`, then you will get an automatic 0! Regardless of your other functions being correct or not.