Lab 7: Hash Table

Due: Wednesday, October 29 by 11:55 pm on Moodle

Exercise 1 – Resizable Hash Table

In this exercise, you will implement a class representing a resizable hash table.
NOTE: Use the provided `HashTable` module as a starting point.

The provided implementation of a hash table uses open addressing to resolve collisions. In the case of insertion, if two keys map to the same hash value, then the inserted item is placed in the next empty table slot. As is, however, it is only capable of storing **100** items. By the time **75** items have been hashed into the table, collisions may become more frequent than rare. Collisions take time to resolve, so it would be better if the hash table could grow to reduce their possibility.

Implement a `HashTable` class to resize once a certain threshold or *load factor* is reached. A good rule of thumb is that the threshold should be about **3/4** the size of the hash table. In this case, the threshold is reached when one quarter of the table contains **None** values and three quarters contain actual data. The table should at most double in size and never decrease in size.

In the **main** function of your module, test your implementation by:

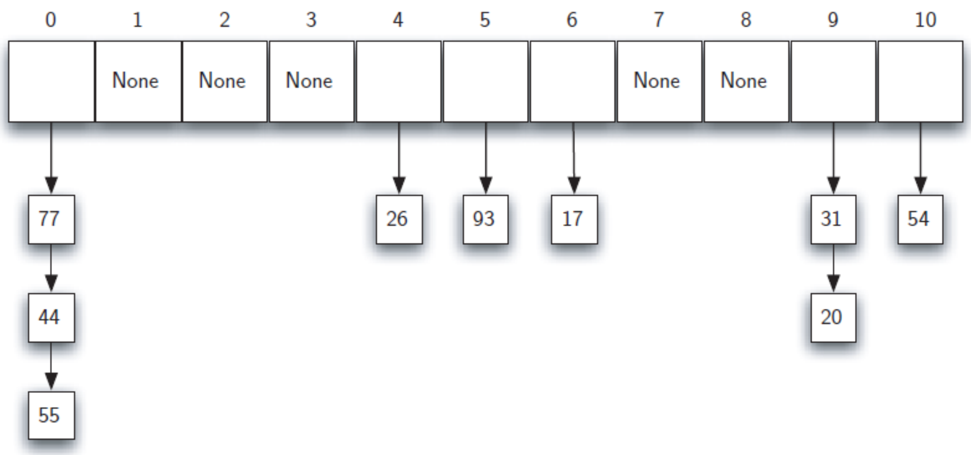
1. Insert items into the hash table until *just* before the threshold is reached.
2. Print the table's contents.
3. Add *just* one more item to the table.
4. Print the table's contents.

If the implementation has been successful, the hash table will have increased in size by some factor greater than one and less than two.

Exercise 2 – Chained Hash Table

In this exercise, you will implement a hash table that uses chaining for collision resolution. NOTE: Use the provided `HashTable` module as a starting point.

Implement a `HashTable` class that uses *chaining* instead of open addressing to resolve collisions. For every chain, use a list to store individual data items in a particular **slot**. Optionally, use any data structure that seems appropriate.



In the above image, a hash table is created with 11 slots. It appears that applying the *hash function* to the **data** 77, 44, and 55 all resulted in 0. Because of this, they were all placed in the same **slot**. Similarly, applying the hash function to data 31 and 20 resulted in 9, the slot in which they were placed.

Additionally, implement a __**delitem**__ function for the chained hash table. Its operation is a little similar to the __**getitem**__ and __**setitem**__ functions in the provided implementation of a hash table.

In the **main** function of your module, test your implementation by:

1. Inserting more than one value into every slot of the hash table.
2. Retrieving every value from every slot of the hash table.
3. Deleting every value from every slot of the hash table.

After each test, print the current state of the hash table to the console. It must be clear which slot items have been hashed to.

Final Deliverable

The **deliverable** will be a compressed zip file containing the following files:

- `resize_table.py` **# Implements a resizable hash table.**
- `chain_table.py` **# Implements a chained hash table.**