Algorithms for Data Science
Lab 6
HashTable ADT

Below you will find a hashtable ADT class called MyHashtable. This class implements the standard hashtable functions: insert, member, and delete. When a MyHashtable is created, one specifies the size of the hashtable, that is, the size of the underlying list used to implement the table. This specific hashtable implementation assumes that only strings will be inserted into the table. When working with hashtables, one needs a hash function defined on the elements inserted. For this specific example, we use the ASCII value of the first character of the string. For example, "amy" → 'a' → 97. Note that in the code below, we created a hashtable that was only of size 10. Thus, we needed to mod (%) the hashcode (97) by the table size (10) in order to always produce an index that is in the correct range for our table. For example, 97%10 → 7 so "amy" would stored at location 7. The last thing to notice about this hashtable implementation is that it uses chaining. That is, at every location there is a list, and when there are hash conflicts, multiple items are added to the index by placing them in the list. Please explore this code and see what it does. Note that printing the hashtable itself is useful for debugging because it allows you to see what is actually stored in the table at any given moment.

For this lab, it is your job to create another hashtable implementation. It should have all the same functions—insert, member, and delete—along with a constructor that takes the size. In fact, you should use the exact same testing code to test it. The difference is that your implementation should use open addressing with linear probing.

Open addressing means that each location in the table holds only one value. So do not make empty lists at each location when creating the table. Instead store none at each location to start. Additionally, you will need a second table that holds the status of each index. I called my second table Status. The status of each index will either be "empty" (which they all should be set to initially), "filled," or "deleted."

When inserting (or deleting or finding a member) you will need to find the location to start looking in the table. Do this exactly the same way as in the chaining version (i.e., the ASCII value of first letter of the string inserted). However, if you try to insert into a spot that is already taken, you will then use linear probing to find the next available spot to insert the element. Basically, you will keep searching until you find a spot that isn't "filled." Don't forget to wrap back around to the start of the table with a mod when probing. Once you find a spot that isn't "filled," assign the element to it and mark its status as "filled." Note that for this lab, you may assume the hashtable will never fill up and thus you will always eventually find a "filled" location.

Finding a member and deleting are similar. You first jump to the hashed location. If the element there isn't the one you are looking for, then you have to start probing down the table to try to find it. Obviously, you would stop looking if you find it. However, you should also stop looking when you find an "empty" spot because this indicates that the element doesn't exist.
For the member function, you just return true or false. For the delete function, if you find it, then you can set the table value to none and status to "deleted." You don't need to do anything if you try to delete an element that doesn't exist.

Test your code using the same test case provided below. Again, it is useful for debugging to print out the hashtable between steps if things don't seem correct.

```
# HashTable ADT with chaining implementation
# This hashtable accepts only strings and hashes based on their
#  ASCII value of the first char
# The constructor takes in the size of the table
class MyHashtable(object):
    def init  (self, size): # Creates an empty hashtable
        self.size = size
        # Create the list (of size) of empty lists (chaining)
        self.table = []
        for i in range(self.size):
            self.table.append([])
    def str  (self): # for print
        return str(self.table)
    def insert(self, elem): # Adds an element into the hashtable
        hash = ord(elem[0]) % self.size
        self.table[hash].append(elem)
    def member(self, elem): # Returns if element exists in hashtable
        hash = ord(elem[0]) % self.size
        return elem in self.table[hash]
    def delete(self, elem): # Removes an element from the hashtable
        hash = ord(elem[0]) % self.size
        self.table[hash].remove(elem)

# Testing code
s = MyHashtable(10)
s.insert("amy") #97
s.insert("chase") #99
s.insert("chris") #99
print(s.member("amy"))
print(s.member("chris"))
print(s.member("alyssa"))
s.delete("chase")
print(s.member("chris"))
# You can use print(s) at any time to see the contents
#   of the table for debugging
#print(s)
```