Hashing and Collision Resolve [CO3]

Instructions for students:

- Complete the following methods on Hashing.
- You may use any language to complete the tasks.
- All your methods must be written in one single .java or .py or .pynb file. DO NOT CREATE separate files for each task.
- If you are using JAVA, you must include the main method as well which should test your other methods and print the outputs according to the tasks.
- If you are using PYTHON, then follow the coding templates shared in this folder.

NOTE:

- **YOU CANNOT USE ANY BUILT-IN FUNCTION EXCEPT** len **IN PYTHON. [negative indexing, append is prohibited]**
- **YOU HAVE TO MENTION SIZE OF ARRAY WHILE INITIALIZATION**

1. Hashtable with Forward Chaining:

You have to create a hashTable from a vehicle_info array. The array contains vehicles as tuples and where each tuples indicates → (brand name, vehicle_type, rent, no_of_passengers)

Vehicle_info = [('Toyota', 'Private Car', 500, 4), ('Jeep', 'SUV', 950, 6), ('Lamborghini', 'SUV', 6900, 6), ('Hyundai', 'Bike', 100, 1), ('BMW', 'Private Car', 1000, 8), ('Honda', 'Bike', 150, 1), ('Ferrari', 'Private Car', 2500, 4), ('BMW', 'Minivan', 5800, 7)]

The print_vehicle_hashtable() methods have been done for you. Write the __hash_function() and insert_vehicle() methods.

def __hash_function(self, brand): 4 marks

The hash function takes the brand name as string calculates its hash key and returns the key. The hash key can be calculated by summing the ascii values of each character of the brand name and modding it by the length of the vehicle_info array. i.e. sum of ascii values%length of vehicle_info array.

[HAVE TO USE RECURSION]
[You may use helper function]

For example,
Brand = 'Toyota'
Ascii Values → T: 84, o:111, y:121, t:116, a:97
Sum of ascii values= 84+111+121+111+116+97= 640
sum of ascii values%length of vehicle_info array = 640%8 = 0 (hash key)

-

insert_vehicle(self, vehicle): 6 marks

You need to insert each vehicle from the **Vehicle_info** array into the hashtable on the basis of its hash function. If you want to insert a vehicle to an index of the hashTable where another vehicle info already exists, you have to check the brands of the vehicles. If the brand matches, then chain the current vehicle as a linked list in the hashed index. If it does not, find the earliest empty slot using the find_empty_slot() function and place the vehicle in that empty spot.
For example:

__hash_function('Jeep') will return 4.
0 : (Brand: Toyota, Type: Private Car, Rent: 500, No. of Passengers: 4)---->None
1: None
2: None
3: None
4: (Brand: Jeep, Type: SUV, Rent: 950, No. of Passengers: 6)---->None
5: None
6: None
7: None
8: None

__hash_function('Lamborghini') will return 4. Since index 4 has Jeep and the brands do not match, it goes to the earliest empty index 1.
0 : (Brand: Toyota, Type: Private Car, Rent: 500, No. of Passengers: 4)---->None
1: (Brand: Lamborghini, Type: SUV, Rent: 6900, No. of Passengers: 6)
2: None
3: None
4: (Brand: Jeep, Type: SUV, Rent: 950, No. of Passengers: 6)---->None
5: None
6: None
7: None
8: None

Again, for the 'BMW' vehicles, the chain is created since the brand is the same.

2. Searching in hashtable:

Complete the __hash_function() and search_hashtable() methods in the given colab file . **Do not** change the given code; implement only the required methods. Creating and Inserting into a hash table using forward chaining is already done in the class. Do not initialize any other instance variable other than the given ones.

a. search_hashtable(self,s) → this instance method takes a string s and searches for the string in the instance variable **ht**. If the string s is found, this method returns 'Found', else returns 'Not Found'. (**Do not** implement sequential search, implement the hash based search.)

b. __hash_function(self,s) → this instance method takes a key-value pair (string, int), calculates the hashed index on key and returns the index. This hash function takes consecutive two letters of the key string, concatenates their ascii values into an integer and sums all the concatenated integers. Then it finds out the modulus of the summation (**think for yourself with which number should we mod the summation**) as the hashed index. For instance, for a string 'Mortis', the consecutive two letters are Mo, rt, is. The concatenated integer for
Mo is 77111 (Ascii of M is 77, o is 111);
rt is 114116 (Ascii of r is 114, t is 116);
'is' is 105115 (Ascii of i is 105, s is 115).
The summation is = 77111+114116+105115
Mod the summation with _____ (**fill in the gap**) and return the answer as the hashed index.

(As for an odd length string, add the letter 'N' at the end of it. Thus, 'Morti' becomes 'MortiN' and the consecutive two letters are Mo, rt, iN)

3. Hashtable with Forward Chaining 2.0

Write the insert() function and hash_function() of the HashTable class which uses an array to store a key-value pair, where the key is a string representing a fruit, and the value is an int representing its respective price.

hash_function(key)

Takes a key which is a string, and calculates its length. If length is even, the sum of the ascii values of the even characters is calculated, otherwise, the sum of the ascii values of odd characters is calculated.
Finally, it returns the sum modded by length of the array

If we call __hash_function(“apple”):
As len(“apple”) is odd, characters in odd indexes (p, l) are taken:

sum = ord(p) + ord(l) = 112 + 108 = 220

If the Hashtable object is initialized with length 5, then:

sum % length of Hashtable array = 220 % 5 = 0
So, the function returns 0

insert(key, value)

Creates a node that contains a tuple which stores the key-value pair as (key, value). Finds index by passing key to hash_function(). If there is no collision, the node is placed in the index.

If there is a collision, forward chaining is applied. Here, the chain should be arranged in **descending order**, i.e, if the node being inserted has the highest value (price), it will be the head of the chain. Otherwise, you should iterate the chain and insert it in the appropriate position.

4. Deletion from hashtable

You are given the hash function, **h(key) = (key + 3) % 6** for a hash-table of length 6. In this hashing, forward chaining is used for resolving conflict and a 6-length array of singly linked lists is used as the hash-table. In the singly linked list, each node has a next pointer, an Integer key and a string value, for example: (4 (key) , “Rafi” (value)). The hash-table stores this key-value pair.

Implement a function **remove(hashTable, key)** that takes a key and a hash-table as parameters. The function removes the key-value pair from the aforementioned hashtable if such a key-value pair (whose key matches the key passed as argument) exists in the hashtable.

Consider,Node class and hashTable are given to you. You just need to complete the remove(hashTable, key) function.

```
class Node:
  def __init__(self, key, value, next=None):
      self.key, self.value, self.next = key, value, next
```

Sample Input and Output:
Some Key-value pairs:
(34, “Abid”) , (4, “Rafi”), (6, “Karim”), (3, “Chitra”), (22, “Nilu”)

HashTable is given in the following:
0: (3, “Chitra”)
1: (22, “Nilu”) → (4, “Rafi”) → (34, “Abid”)
2: None
3: (6, “Karim”)
4: None
5: None

remove(hashTable, key=4) returns the changed hashTable where (4, “Rafi”) is removed.

New HashTable Output:
0: (3, “Chitra”)
1: (22, “Nilu”) → (34, “Abid”)
2: None
3: (6, “Karim”)
4: None
5: None

remove(hashTable, key=9) returns the same given hashTable since 9 doesn’t exist in the table.