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. Nerdy Run: (Easy)

A group of game developers released a math game named ‘Nerdy Run’. In this game, there is a lengthy straight **path** that contains some numbers and a random number **k**. As you walk on the path, your goal is to find out if the path contains any duplicates within k distance. If you find a duplicate within k distance, you will type the number in a textbox; if not type None.
The constraint is that, the game control allows you to **move only once and move forward**. There is no backward movement controller in this game.

Constraint:

1. ***DO NOT USE*** Membership Operators in List, String for this task.
2. You can traverse the array ***ONLY ONCE and Always Move Forward*** (i.e. no nested loop).
3. You may use dictionary here.

Sample Input:
path = [6,7,8,9,5,9]
k = 3
Sample Output:
9
Explanation:
The number 9 has a duplicate at 2 distances away which is less than k(3). Therefore, the found duplicate within 3 distances is 9.

Sample Input:
path = [6,7,9,6,5,9]
k = 2
Sample Output:
None
Explanation:

No number repeated within distance 2

Sample Input:
path = [0.21,1.21,4.67,0.21,0.45,1.9]
k = 7
Sample Output:
0.21

2. Hashtable with Forward Chaining: (Medium)

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. Layered Hashtable: (Hard)

Complete the create_layered_hashtable() and search_element() methods in the given colab file. **Do not** change the given code; implement only the required methods. Printing the layered hashtable is already implemented in the class. You can take enough hints from the printing method. Do not initialize any other instance variable other than the given ones.

Let us learn a new type of hashing data structure for sorted linked lists. As you already know, searching in linked lists always requires a sequential search loop as there is no indexing (direct access) in this ADT (abstract data structure). So, to lessen the number of times a loop runs, we will use an array (of specific length) named **express_lane**. This array serves as the hashtable for sorted linked lists.

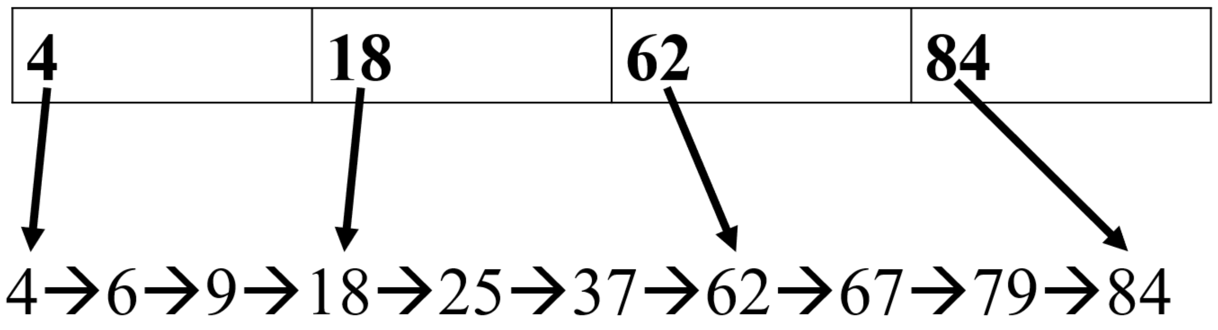
This express_lane contains some **specific** nodes of a sorted linked list. The nodes stored in express_lane[i] and express_lane[i+1] represent all the other nodes in the sorted linked list (normal lane).

For example,
Suppose, the size of an express_lane is 4.
Then, for the sorted linked list (normal lane):
4→6→9→18→25→37→62→67→79→84

The express_lane will be:

4	18	62	84

Here, each element in the express_lane is the node of the given linked list. Therefore, 4 in the express_lane indicates the node 4 in the given linked list (normal lane) and so on and so forth.



a. create_layered_hashtable(self, linked_list_head):

This method takes a sorted linked list(normal lane) and creates the express lane of a specific size.

Bucket: A bucket is the nodes represented by the nodes stored in express_lane[i] and express_lane[i+1].

In the mentioned example,
express_lane[0] contains node 4
express_lane[1] contains node 18
Therefore, this bucket represents all the nodes that are greater or equal to 4 and smaller than 18: 4→6→9→

express_lane[1] contains node 18
express_lane[2] contains node 62
Therefore, this bucket represents all the nodes that are greater or equal to 18 and smaller than 62: 18→25→37→

Which node to store in the express_lane array: First we identify the bucket size. For simplicity, to get the bucket size, we divide the number of nodes in the normal lane by the express_lane size.
In the mentioned example, for express lane size 4,

bucket size = (number of nodes in the normal lane/4)+1 = 10/4 +1 = 3

It indicates that every bucket has 3 nodes.
The express_lane contains the normal lane nodes whose positions are divisible by bucket size. In the mentioned example,
node 4 is at 0th index
Node 18 is at 3rd index
Node 62 is at 6th index
Node 84 is at 9th index
Thus the express_lane contains nodes:

4	18	62	84

b. search_element(self,k):

This method searches for k in the given sorted linked list and returns 'Found'/ 'Not Found'. (**Do not** implement sequential search, implement the hash based search.)

Suppose we need to search for an element in the sorted normal lane. Before, we needed to iterate through the whole linked list. In this layered hashtable, we need to find which bucket can possibly contain the element and iterate through only those nodes.
Since the normal lane is sorted, finding a bucket is easy. We need to find such an i in express_lane so that express_lane[i].element<=k<express_lane[i+1].element. Thus, we only need to iterate through a selective number of nodes in the normal lane.
In the mentioned example, to find 67, the probable bucket is the nodes from express_lane[2] to express_lane[3]. Thus, we only need to iterate 62→67→79.