Hashing

You do not need to write the Node class, Hashtable class etc. unless specified in the problem.

Just design the function(s) specified in the problem.

No need to write driver code.

**This document is a compilation of previous semester's exam (midterm/final) problems and lab quiz problems. Try to practice relevant problems from online judges as well.*

1.

Suppose you are given a hash function called check(). In this hashing, forward chaining is used for resolving conflict, and a 3-length array of singly linked lists is used as the hash table. In the singly linked list, each node has a next pointer, a string key and a string value, for example: ("7B12C" (key), "CSE220" (value)). The hash-table stores this key-value pair. The check function is given below:

For your reference ASCII value of A is 65.

```
Hash Function
def check(key):
    v = 0
    for i in key:
        if i.isnumeric():
            v+=int(i)
        else:
            v+=ord(i)
    return v%3
```

I. What is the hashed-index of ("4G14", "MNS") key-value pair in the above hashTable?

II. 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 and return the table. If the key does not exist, the function returns the same hash table.

Consider, Node class, hash_function 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


Python Notation:
def remove(hashTable, key):
        # To Do
```

Sample Input and Output

Input Table	Returned Table
0: ("13D", "ZMD") → ("10A", "ABW") 1: ("31B", "NZF") 2: ("2A4", "RAK") → ("C7B", "FAF") → ("1A2", "MNY") Function call: remove(hashTable, key="C7B")	0: ("13D", "ZMD") → ("10A", "ABW") 1: ("31B", "NZF") 2: ("2A4", "RAK") → ("1A2", "MNY")

2.

You have an interesting hash function for a hash table. The hash-function takes a singly linked list where node values are characters and the characters are put in the list using some unknown permutation. The hash function takes a string key as input and the head of the linked list. Then the hash for the key is the mod by 10 of the sum of the indexes of the characters of the list in the string. Implement this hash function. If any node value is not present in the string, then consider its index -1. If any character of the string occurs multiple times, then consider the index value of the last occurrence. **You can not use any built-in function except range. Consider the Node class has elem and next variable. No need to write the Node class. You can not use any data structures other than the given data structures.**

Python Notation:

```
def hash(list, st):
        # To Do
```

Sample Input	Sample Output	Explanation
hash(list, st) list = D → A → T → A st = "SADA"	7	Index of D is 2 and A is 3. T is not present in the string hence -1. (2 + 3 -1 + 3) % 10 = 7

3.

You have an array S containing names. You need to convert these names into integers using the **conFun()** function, which assigns a numerical value to each letter of the alphabet (A=1, B=2, ..., Z=26). The hash value for a name is the sum of the values of its letters modulo 10.

a. Generate an array EL of the same length as the given array S. For each element e in S, convert it into an integer using the **conFun()** function and store the result in the corresponding index of EL (One is done for you).

ABC	EBC	JKL	MLK	BAC
6				

b. What kind of advantage will we be able to take from this change?
c. To resolve any collision, which methods can we use?

4.

2. Write a function that **inserts** an **integer** element **elem** into a hashtable **ht** using **forward chaining** and the hash function **hash = (key*2)%size.**

5.

Tasks from Lab 4 - Lab 4 - Secondary Data Structures.docx

6.

In this task, you are asked to implement a **`HashTable`** class that stores key-value pairs, where the key is a **string (representing an employee ID)** and the value is a **string (representing the department name)**. The class should include a **hash_function** that computes the hash index based on the sum of ASCII values of each character in the key, then Multiply each ASCII value by its position index (starting from 1) and finally sum these weighted values and take the modulus with the size of the hash table. The **`insert()`** method should insert a new key-value pair or update the value if the key already exists, using **forward chaining** to handle collisions. **If the key already exists, its value should be updated. [You are not allowed to use any built-in functions except len(). Assume the display method is already implemented]**

Sample Input:	Sample Output:	Explanation:

`ht = HashTable(4)`
 	
`ht.insert("E123", "HR")`
`ht.insert("BA", "Finance")`
`ht.insert("XY", "Engineering")`
`ht.insert("YX", "Marketing")`
 	
`print("\nHash table after insertions with collisions:")`
`ht.display()`
 	
`ht.insert("E123", "Admin")`
`print("\nHash table after update:")`
`ht.display()`
	Hash table after insertions with collisions:
	Index 0: (BA: Finance) -> None
	Index 1: (E123: HR) -> (YX: Marketing) -> None
	Index 2: (XY: Engineering) -> None
	Index 3: None
	
	Hash table after update:
	Index 0: (BA: Finance) -> None
	Index 1: (E123: Admin) -> (YX: Marketing) -> None
	Index 2: (XY: Engineering) -> None
	Index 3: None
		For E123:
		
		ASCII values: 'E' = 69, '1' = 49, '2' = 50, '3' = 51.
		Weighted sum: (1 × 69) + (2 × 49) + (3 × 50) + (4 × 51) = 69 + 98 + 150 + 204 = 521.
		Index: 521 % 4 = 1.
		
		For BA:
		
		ASCII values: 'B' = 66,'A' = 65.
		Weighted sum: (1x66)+(2 × 65) = 296.
		Index: 296 % 4 = 0.

7.

You are tasked with implementing a **hash table** that stores student records. Each record contains **a student name (a string) and a student ID (an integer)**. You will implement a hash table that uses **forward chaining** (a linked list) to handle collisions. Implement the following methods:

Hash Function: You are given a string representing the student name. You need to calculate a hash index for this string using the following rules: Take the sum of the ASCII values of the characters in the string. If the sum is odd, return the sum modulo the size of the hash table. If the sum is even, return the sum divided by 2, modulo the size of the hash table.

Insert: Implement the insert() method that takes a student name (string) and student ID (integer) and stores them in the hash table. If a student in the same index already exists in the hash table (i.e., there is a collision), use forward chaining (linked list) to store multiple student records at the same hash index. **Note**: If the key-value pair already exists, you will print an error message saying "Student already exists" and discard the key-value pair.

Search: Implement the search() method that takes a student name (string) and returns the corresponding student ID (integer). If not found, then return None. **[No built-in function except len(). Assume the display method and Node class are already implemented]**

Sample Input:	**Sample Output:**	**Explanation:**

`ht = HashTable(5)`
`ht.insert("Alice", 123456) ht.insert("Bob", 135927)`
`ht.insert("Charlie", 348247)`
`ht.insert("David", 124382)`
`print("\nHash table after insertions and rehashing:")`
`ht.display()`
 	
`ht.insert("Alice", 123456)`
`ht.search("David")`
	`Hash table after insertions:`
	`Index 0: ("Bob", 135927)`
	`Index 1: None`
	`Index 2: None`
	`Index 3:("Charlie", 348247)`
	`Index 4: ("Alice", 123456)->("David", 124382)`
	
	`Student already exists`
	
	`ID of David is 124382`
		`For 'Alice', hash function calculation, sum of ASCII values = 65 + 108 + 105 + 99 + 101 = 478. Since 478 is even, the hash index = 478 / 2 % 5 = 4.`
		`For 'Bob', sum of ASCII values = 66 + 111 + 98 = 275. Since 275 is odd, the hash index = 275 % 5 = 0.`
		`When we try to insert ("Alice", 123456) again, since the (key, value) pair already exists, an error message is printed.`
		`David is found at index 4 and his ID returned.`

8.

In this task, you are asked to implement a **`HashTable`** class that stores key-value pairs, where the key is a **string (representing an employee ID)** and the value is a **string (representing the department name)**. The class should include a **hash_function** that computes the hash index based on the sum of ASCII values of each character in the key, sum each ASCII value by its position index (starting from 1) and finally sum these weighted values and take the modulus with the size of the hash table. The **`insert()`** method should insert a new key-value pair or update the value if the key already exists, using **forward chaining** to handle collisions. **If the key already exists, its value should be updated. [You are not allowed to use any built-in functions except len(). Assume the display method is already implemented]**

Sample Input:	Sample Output:	Explanation:

`ht = HashTable(4)`
 	
`ht.insert("E123", "HR")`
`ht.insert("BA", "Finance")`
`ht.insert("XY", "Engineering")`
`ht.insert("YX", "Marketing")`
 	
`print("\nHash table after insertions with collisions:")`
`ht.display()`
 	
`ht.insert("E123", "Admin")`
`print("\nHash table after update:")`
`ht.display()`
	`Hash table after insertions with collisions:`
	`Index 0: (XY: Engineering) -> (YX: Marketing) -> None`
	`Index 1: (E123: HR) -> None`
	`Index 2: (BA: Finance) -> None`
	`Index 3: None`
	
	`Hash table after update:`
	`Index 0: (XY: Engineering) -> (YX: Marketing) -> None`
	`Index 1: (E123: Admin) -> None`
	`Index 2: (BA: Finance) -> None`
	`Index 3: None`
		`For E123:`
		
		`ASCII values: 'E' = 69, '1' = 49, '2' = 50, '3' = 51.`
		`Weighted sum: (1 + 69) + (2 + 49) + (3 + 50) + (4 + 51) = 70 + 51 + 53 + 55 = 229.`
		`Index: 229 % 4 = 1.`
		
		`For BA:`
		
		`ASCII values: 'B' = 66,'A' = 65.`
		`Weighted sum: (1+66)+(2 + 65) = 134.`
		`Index: 134 % 4 = 2.`

9.

You are tasked with implementing a **hash table** that stores student records. Each record contains **a student name (a string) and a student ID (an integer)**. You will implement a hash table that uses **forward chaining** (a linked list) to handle collisions. Implement the following methods:

Hash Function: You are given a string representing the student name. You need to calculate a hash index for this string using the following rules: Take the sum of the ASCII values of the characters in the string. If the sum is odd, return the sum modulo the size of the hash table. If the sum is even, return the sum divided by 2, modulo the size of the hash table.

Search: Implement the search() method that takes a student name (string) and returns the corresponding student ID (integer). If the student is found in the hash table, return the student ID. If the student is not found, return None.

Delete: Implement the delete() method that takes a student name (string) and removes the corresponding record from the hash table. If the student is not found, return None.

[No built-in function except len(). Assume the display method and Node class are already implemented]

Sample Input:	Sample Output:	Explanation:

`ht = HashTable(5)`
`// assume the insert method is called multiple times and some entries are already in the hashtable`
 	
`print('Hash table after insertions:\n')`
`ht.display()`
 	
`ht.search("David")`
 	
`ht.delete("Alice", 123456)`
 	
`print('Hash table after deletions:\n')`
`ht.display()`
	`Hash table after insertions:`
	`Index 0: ("Bob", 135927)`
	`Index 1: None`
	`Index 2: None`
	`Index 3:("Charlie", 348247)`
	`Index 4: ("Alice", 123456)->("David", 124382)`
	
	`ID of David is 124382`
	
	`Hash table after deletions:`
	`Index 0: ("Bob", 135927)`
	`Index 1: None`
	`Index 2: None`
	`Index 3:("Charlie", 348247)`
	`Index 4: ("David", 124382)`
		`For 'Alice', hash function calculation, A → 65, l → 108, i → 105, c → 99, e → 101. The sum of ASCII values = 65 + 108 + 105 + 99 + 101 = 478. Since 478 is even, the hash index = 478 / 2 % 5 = 4.`
		`For 'Bob', B → 66, o → 111, b → 98. The sum of ASCII values = 66 + 111 + 98 = 275. Since 275 is odd, the hash index = 275 % 5 = 0.`
		`David is found at index 4 and his ID returned.`

10.

In this task, you are asked to implement a **`HashTable`** class that stores key-value pairs, where the key is a **string (representing a package ID)** and the value is a **string (representing the package status)**. The class should include a **hash_function** that computes the hash index based on the sum of ASCII values of the first three characters of the key, adding 'X' if the key is shorter than three characters. The **`insert()`** method should insert a new key-value pair or update the value if the key already exists, using **forward chaining** to handle collisions. **If the key already exists, its value should be updated.**

[You are not allowed to use any built-in functions except len(). Assume the display method is already implemented]

Sample Input:	Sample Output:	Explanation:

`ht = HashTable(10)`
 	
`ht.insert("PKG123", "In Transit")`
`ht.insert("AB", "Delivered")`
`ht.insert("PKG456", "Returned")`
 	
`print("\nHash table after insetons :")`
`ht.display()`
 	
`ht.insert("PKG123", "Delivered")  # Updating PKG123 status`
 	
`print("\nHash table after updates:")`
`ht.display()`
	Hash table after insertions:
	Index 6: PKG123 (In Transit)
	PKG456 (Returned)
	
	Index 9: AB (Delivered)
	
	Hash table after updates:
	Index 6: PKG123 (Delivered)
	PKG456 (Returned)
	
	Index 9: AB (Delivered)
		For PKG123,Hash function calculation, P' = 80,'K' = 75,'G' = 71.Total sum = 80 + 75 + 71 = 226. So, index=226%10=6
		For AB 'A' = 65, 'B' = 66. Since the key is less than 3 characters, the ASCII value of 'X' (88) is added to the sum.Total sum = 65 + 66 + 88 = 219
		So, index = 219 % 10 = 9
		When we try to insert PKG123 again, since the key already exists its value will be updated from In Transit to Delivered.

11.

You are asked to implement a HashTable class that stores key-value pairs, where the key is a string (representing a student ID) and the value is an integer (representing the student's grade).

The class should include a hash_function that calculates the hash index by summing the ASCII values of the **first two characters** of the key. If the key is only one character, the ASCII value of 'Y' (89) should be added as the second character.

The insert() method should add a new key-value pair or update the value if the key already exists. Use **forward chaining** to handle collisions.

You are **not allowed** to use any built-in functions except len(). Assume the display() method is already implemented to show the hash table.

Python

Sample Input	Sample Output	Explanation

`# Sample Input`
`ht = HashTable(7)`
`ht.insert("S1", 85)`
`ht.insert("A", 90)`
`ht.insert("S2", 78)`
`print("\nHash table after insertions:")`
`ht.display()`
 	
`ht.insert("S1", 88)  # Updating S1 grade`
`print("\nHash table after update:")`
`ht.display()`
	`Hash table after insertions:`
	`Index 0: A (90) -> S2 (78) -> None`
	`Index 1: None`
	`Index 2: None`
	`Index 3: None`
	`Index 4: None`
	`Index 5: None`
	`Index 6: S1 (85) -> None`
	
	`Hash table after update:`
	`Index 0: A (90) -> S2 (78) -> None`
	`Index 1: None`
	`Index 2: None`
	`Index 3: None`
	`Index 4: None`
	`Index 5: None`
	`Index 6: S1 (88) -> None`
		For S1:
		'S' = 83, '1' = 49.
		Total = 83 + 49 = 132.
		Index = 132 % 7 = 6.
		For A:
		'A' = 65, 'Y' = 89.
		Total = 65 + 89 = 154.
		Index = 154 % 7 = 0.
		For S2:
		'S' = 83, '2' = 50.
		Total = 83 + 50 = 133.
		Index = 133 % 7 = 0.

Sample Input	Sample Output	Explanation

`// Sample Input`
`HashTable ht = new HashTable(7);`
`ht.insert("S1", 85);`
`ht.insert("A", 90);`
`ht.insert("S2", 78);`
`System.out.println("\nHash table after insertions:");`
`ht.display();`
 	
`ht.insert("S1", 88);//Updating S1 grade`
`System.out.println("\nHash table after update:");`
`ht.display();`
	`Hash table after insertions:`
	`Index 0: A (90) -> S2 (78) -> null`
	`Index 1: null`
	`Index 2: null`
	`Index 3: null`
	`Index 4: null`
	`Index 5: null`
	`Index 6: S1 (85) -> null`
	
	`Hash table after update:`
	`Index 0: A (90) -> S2 (78) -> null`
	`Index 1: null`
	`Index 2: null`
	`Index 3: null`
	`Index 4: null`
	`Index 5: null`
	`Index 6: S1 (88) -> None`
		For S1:
		'S' = 83, '1' = 49.
		Total = 83 + 49 = 132.
		Index = 132 % 7 = 6.
		For A:
		'A' = 65, 'Y' = 89.
		Total = 65 + 89 = 154.
		Index = 154 % 7 = 0.
		For S2:
		'S' = 83, '2' = 50.
		Total = 83 + 50 = 133.
		Index = 133 % 7 = 0.

12.

Question 1 [15 Points]

You are given an array of positive integers containing n elements. Your task is to find the first repeating element in the array using a hash-based approach. A repeating element is one that appears more than once in the array, and among all repeating elements, you must return the one that repeats at the **smallest index difference**. If no element repeats, return -1.

Use a hashmap to efficiently solve the problem by storing and checking the elements as you traverse the array.

Sample Input	Sample Output
6 10 5 3 4 3 5	3 Explanation: Both 3 and 5 repeat, but 3 repeats with a smaller index difference (indices 2 and 4). The difference of indexes for 3 is 2 (4-2 = 2) and 5 is 4 (5-1 = 4)
5 1 2 3 4 5	-1 Explanation: No elements repeat, so the output is -1.
6 1 1 3 4 3 3	1 Explanation: Both 1 and 3 repeat, but 1 repeats with a smaller index difference indices (0 and 1) 3 appears at index no 2 and first repeat occurs at index no 4 . Since we are taking the first repeating element, we will not consider the very last 3 at index no 5. So the difference of indexes for 1 is 1 (1-0 = 1) and 3 is 2 (4-2 = 2)

13.

In this task, you are asked to implement a **HashTable** class that stores key-value pairs, where the key is **a string (representing a product ID)** and the value **is a float (representing the product's price)**. The class should include a **hash_function** that computes the hash index based on the sum of the ASCII values of the first three characters of the key. If the key is shorter than three characters, it should add the ASCII value of '0' (48) to make the key length three.The **insert()** method should insert a new key-value pair into the hash table. If a collision occurs, the method will use forward chaining (linked lists) to store multiple entries at the same index. If the key already exists, it should update the value by adding the new price to the previous price.

You are not allowed to use any built-in functions except len(). Assume the display method is already implemented.

Sample Input:	Sample Output:	Explanation:

`ht = HashTable(10)`
 	
`ht.insert("P123", 19.99)`
`ht.insert("AB", 15.50)`
`ht.insert("P456", 25.75)`
 	
`print("\nHash table after insertions:")`
`ht.display()`
 	
`ht.insert("P123", 21.99) # Updating price for P123 by adding the new price`
 	
`print("\nHash table after update:")`
`ht.display()`
	`Hash table after insertions:`
	`Index 5: P456 (25.75)`
	
	`Index 9: P123 (19.99)`
	`A45 (15.50)`
	
	`Hash table after updates:`
	`Index 5: P456 (25.75)`
	`Index 9: P123 (41.98) # 19.99 + 21.99`
	`Index 9: A45 (15.50)`
		`For P123,Hash function calculation, 'P' = 80, '1' = 49, '2' = 50.Total sum = 80 + 49 + 50 = 179. So, index=179%10=9`
		`For AB 'A' = 65, 'B' = 66. Since the key is less than 3 characters, the ASCII value of '0' (48) is added to the sum.Total sum = 65 + 66 + 48 = 179`
		`So, index = 179 % 10 = 9`
		`When we try to insert P123 again, since the key already exists its value will be updated by adding 19.99 with 21.99.`

14.

You are asked to implement a HashTable class that stores key-value pairs, where the key is a string (representing a student ID) and the value is an integer (representing the student's grade).

The class should include a hash_function that calculates the hash index by summing the ASCII values of the **first two characters** of the key. If the key is only one character, the ASCII value of 'Z' (90) should be added as the second character.

The insert() method should add a new key-value pair or update the value if the key already exists. Use **forward chaining** to handle collisions.

You are **not allowed** to use any built-in functions except len(). Assume the display() method is already implemented to show the hash table.

Python

Sample Input	Sample Output	Explanation

`# Sample Input`
`ht = HashTable(7)`
`ht.insert("S1", 85)`
`ht.insert("A", 90)`
`ht.insert("S2", 78)`
`print("\nHash table after insertions:")`
`ht.display()`
 	
`ht.insert("S1", 88)  # Updating S1 grade`
`print("\nHash table after update:")`
`ht.display()`
	`Hash table after insertions:`
	`Index 0: S2 (78) -> None`
	`Index 1: A (90) -> None`
	`Index 2: None`
	`Index 3: None`
	`Index 4: None`
	`Index 5: None`
	`Index 6: S1 (85) -> None`
	
	`Hash table after update:`
	`Index 0: S2 (78) -> None`
	`Index 1: A (90) -> None`
	`Index 2: None`
	`Index 3: None`
	`Index 4: None`
	`Index 5: None`
	`Index 6: S1 (88) -> None`
		For S1:
		'S' = 83, '1' = 49.
		Total = 83 + 49 = 132.
		Index = 132 % 7 = 6.
		For A:
		'A' = 65, 'Z' = 90.
		Total = 65 + 90 = 155.
		Index = 155 % 7 = 1.
		For S2:
		'S' = 83, '2' = 50.
		Total = 83 + 50 = 133.
		Index = 133 % 7 = 0.

Sample Input	Sample Output	Explanation

`// Sample Input`
`HashTable ht = new HashTable(7);`
`ht.insert("S1", 85);`
`ht.insert("A", 90);`
`ht.insert("S2", 78);`
`System.out.println("\nHash table after insertions:");`
`ht.display();`
 	
`ht.insert("S1", 88);// Updating S1 grade`
`System.out.println("\nHash table after update:");`
`ht.display();`
	`Hash table after insertions:`
	`Index 0: S2 (78) -> null`
	`Index 1: A (90) -> null`
	`Index 2: null`
	`Index 3: null`
	`Index 4: null`
	`Index 5: null`
	`Index 6: S1 (85) -> null`
	
	`Hash table after update:`
	`Index 0: S2 (78) -> null`
	`Index 1: A (90) -> null`
	`Index 2: null`
	`Index 3: null`
	`Index 4: null`
	`Index 5: null`
	`Index 6: S1 (88) -> null`
		For S1:
		'S' = 83, '1' = 49.
		Total = 83 + 49 = 132.
		Index = 132 % 7 = 6.
		For A:
		'A' = 65, 'Z' = 90.
		Total = 65 + 90 = 155.
		Index = 155 % 7 = 1.
		For S2:
		'S' = 83, '2' = 50.
		Total = 83 + 50 = 133.
		Index = 133 % 7 = 0.

15.

You are given an array of positive integers containing n elements. Your task is to find the first repeating element in the array using a hash-based approach. A repeating element is one that appears more than once in the array, and among all repeating elements, you must return the one that repeats at the **largest index difference**. If no element repeats, return -1.

Use a hashmap to efficiently solve the problem by storing and checking the elements as you traverse the array.

Sample Input	Sample Output
6 10 5 3 4 3 5	5 Explanation: Both 3 and 5 repeat, but 5 repeats with a larger index difference (indices 1 and 5). The difference of indexes for 3 is 2 (4-2 = 2) and 5 is 4 (5-1 = 4)
5 1 2 3 4 5	-1 Explanation: No elements repeat, so the output is -1.
6 1 1 3 4 3 3	3 Explanation: Both 1 and 3 repeat, but 3 repeats with a higher index difference indices (2 and 4) 3 appears at index no 2 and first repeat occurs at index no 4 . Since we are taking the first repeating element, we will not consider the very last 3 at index no 5. So the difference of indexes for 1 is 1 (1-0 = 1) and 3 is 2 (4-2 = 2)