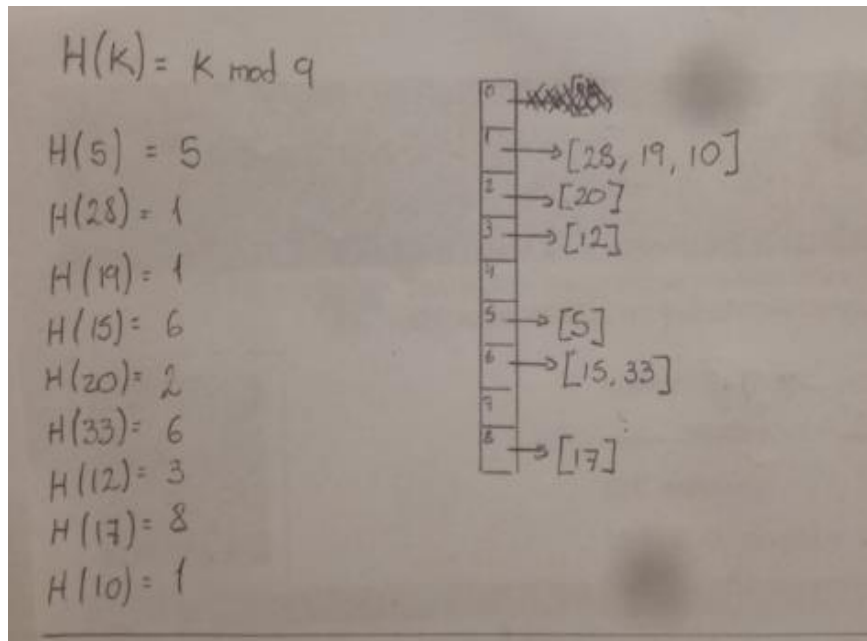


PARTE 1

Ejercicio 1

Ejemplificar que pasa cuando insertamos las llaves 5, 28, 19, 15, 20, 33, 12, 17, 10 en un **HashTable** con la colisión resulta por el método de chaining. Permita que la tabla tenga 9 slots y la función de hash:

$$H(k) = k \bmod 9 \quad (1)$$



Ejercicio 2

A partir de una definición de diccionario como la siguiente:

dictionary = Array(m,0)

Crear un módulo de nombre **dictionary.py** que **implemente** las siguientes especificaciones de las operaciones elementales para el **TAD diccionario**.

Nota: puede **dictionary** puede ser redefinido para lidiar con las colisiones por encadenamiento

insert(D, key, value)

Descripción: Inserta un key en una posición determinada por la función de hash (1) en el diccionario (dictionary). Resolver colisiones por encadenamiento. En caso de keys duplicados se anexan a la lista.

Entrada: el diccionario sobre el cual se quiere realizar la inserción y el valor del key a insertar

Salida: Devuelve D

search(D, key)

Descripción: Busca un key en el diccionario

Entrada: El diccionario sobre el cual se quiere realizar la búsqueda

(dictionary) y el valor del key a buscar.

Salida: Devuelve el value de la key. Devuelve **None** si el key no se encuentra.

delete(D,key)

Descripción: Elimina un key en la posición determinada por la función de hash (1) del diccionario (dictionary)

Poscondición: Se debe marcar como nulo el **key** a eliminar.

Entrada: El diccionario sobre el se quiere realizar la eliminación y el valor del key que se va a eliminar.

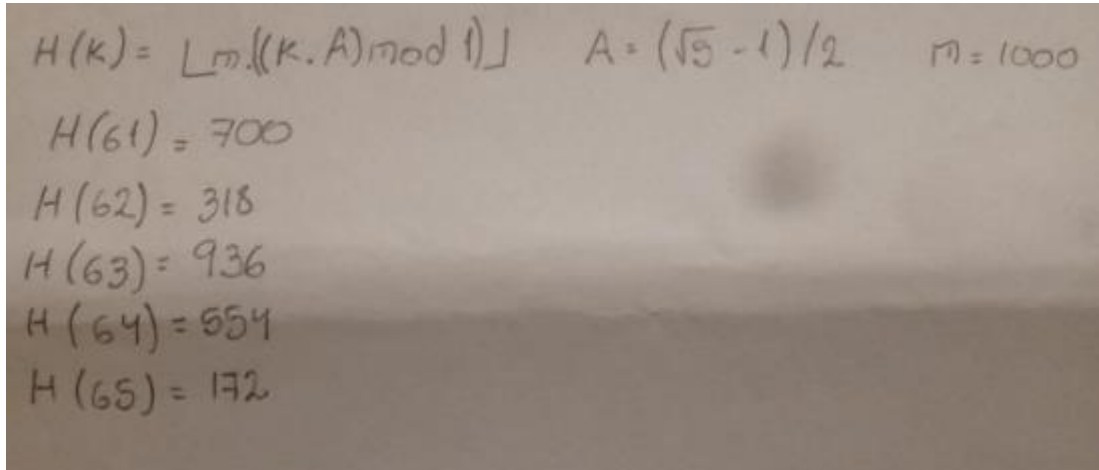
Salida: Devuelve D

```
1 class DictionaryNode:
2     def __init__(self, key = None, value = None):
3         self.key = key
4         self.value = value
5
6
7 class Dictionary:
8     def __init__(self, hash_function = None, longitud = 13):
9         self.D = [None] * longitud
10
11         if hash_function != None:
12             self.hash_function = hash_function
13         else:
14             self.hash_function = lambda k: k % longitud
15
16
17 # def Ordered      # Estaria bueno implementar una manera de insertar los elementos en la tabla y que queden ordenados
18 def insert(self, D, k, val):
19
20     new_k = self.hash_function(k)
```


PARTE 2

Ejercicio 3

Considerar una tabla hash de tamaño $m = 1000$ y una función de hash correspondiente al método de la multiplicación donde $A = (\sqrt{5}-1)/2$. Calcular las ubicaciones para las claves 61, 62, 63, 64 y 65.



Handwritten calculations for a hash function:

$$H(k) = \lfloor m \cdot ((k \cdot A) \bmod 1) \rfloor \quad A = (\sqrt{5} - 1) / 2 \quad m = 1000$$
$$H(61) = 700$$
$$H(62) = 318$$
$$H(63) = 936$$
$$H(64) = 554$$
$$H(65) = 172$$

Ejercicio 4

Implemente un algoritmo lo más eficiente posible que devuelva **True** o **False** a la siguiente proposición: dado dos strings $s_1 \dots s_k$ y $p_1 \dots p_k$, se quiere encontrar si los caracteres de $p_1 \dots p_k$ corresponden a una permutación de $s_1 \dots s_k$. Justificar el coste en tiempo de la solución propuesta.

Ejemplo 1:

Entrada: $S = \text{'hola'}$, $P = \text{'ahlo'}$

Salida: True, ya que P es una permutación de S

Ejemplo 2:

Entrada: $S = \text{'hola'}$, $P = \text{'ahdo'}$

Salida: Falso, ya que P tiene al carácter 'd' que no se encuentra en S por lo que no es una permutación de S

```
1 import dictionary as d
2 import math
```

```
20 def isPermutation_ver2(elem1, elem2):
21
22     """
23     El orden de complejidad es de O(n), las operaciones de busqueda e insercion son O(1) pero depende de la longitud del elemento
24     insertado por lo que sera O(n)
25     """
26
27     flag = True
28
29     if len(elem1) != len(elem2):
30         flag = False
31
32     m = 27 # caracteres desde la a - z
33     hash_function = lambda k : ord(k) % m
34
35     dicc = d.Dictionary(hash_function, m)
36     for i in elem1:
37         dicc.insert(dicc.D, i, str(i))
38     for i in elem2:
39         pos = dicc.search(dicc.D, i)
40
41         if pos == None:
42             flag = False
43
44     return flag
```

Ejercicio 5

Implemente un algoritmo que devuelva True si la lista que recibe de entrada tiene todos sus elementos únicos, y Falso en caso contrario. Justificar el coste en tiempo de la solución propuesta.

Ejemplo 1:

Entrada: L = [1,5,12,1,2]

Salida: Falso, L no tiene todos sus elementos únicos, el 1 se repite en la 1ra y 4ta posición

```
57 def isConj_v2(elem):
58
59     """
60     Complejidad de O(n) ya que el las operaciones de insercion busqueda y delete son O(1)
61     """
62
63     m = len(elem)
64
65     A = (math.sqrt(5)-1)/2
66
67     hash_function = lambda k : int(m*((k*A)% 1))
68
69     dicc = d.Dictionary(hash_function, m)
70
71     for key in elem:
72         dicc.insert(dicc.D, key, str(key))
73
74     count = 0
75
76     for key in elem:
77         if dicc.search(dicc.D, key) != None:
78             dicc.delete(dicc.D, key)
79             count += 1
80
81     if count == m:
82         return True
83     else:
84         return False
```

Ejercicio 6

Los nuevos códigos postales argentinos tienen la forma cddddccc, donde c indica un carácter (A - Z) y d indica un dígito 0, . . . , 9. Por ejemplo, C1024CWN es el código postal que representa a la calle XXXX a la altura 1024 en la Ciudad de Mendoza. Encontrar e implementar una función de hash apropiada para los códigos postales argentinos.

```
87     # Ejercicio 6
88
89     def codigo_postal(dict, codigo):
90
91         """
92         La complejidad sera de O(1) ya que la longitud de los codigos esta definida, las operaciones son aritmeticas (O(1)) y las operaciones
93         de insercion y busqueda son O(1)
94         """
95
96         print(f"el codigo postal se insertara en la posicion {Codigo_postal_hash(codigo)}")
97         dict.insert(dict.D, codigo, str(codigo))
98         return dict.D
99
100    def Codigo_postal_hash(cp):
101
102        ## Como la catidad de combinaciones es muy grande (Ronda los 40 billones) elejimos un n° primo muy grande
103        m = 100003
104        parte_num = int(cp[1:5])
105        parte_string = cp[6:] + cp[5:]
106        parte_string_ascii = sum(ord(c)*10**k for c,k in zip(parte_string, [4,3,2,1]))
107        val = parte_num + parte_string_ascii
108        hash_function = val % m
109
110        return hash_function
111
```

Ejercicio 7

Implemente un algoritmo para realizar la compresión básica de cadenas utilizando el recuento de caracteres repetidos. Por ejemplo, la cadena 'aabcccccaaa' se convertiría en 'a2blc5a3'. Si la cadena "comprimida" no se vuelve más pequeña que la cadena original, su método debería devolver la cadena original. Puedes asumir que la cadena sólo tiene letras mayúsculas y minúsculas (a - z, A - Z). Justificar el coste en tiempo de la solución propuesta.

```
115     # Ejercicio 7
116
117     """
118     Complejidad de O(n) ya que itero una lista de longitud n, no encuentre aplicacion efectiva en diccionarios
119     """
120    def compress(elem):
121
122        if isConj(elem):
123            return elem
124        else:
125            lista = ""
126            cont = 0
127
128            for i in range(len(elem)):
129                if i == 0:
130                    lista += elem[i]
131                    cont += 1
132                    continue
133                if elem[i] == lista[-1]:
134                    cont += 1
135                else:
136                    lista += str(cont) + elem[i]
137                    cont = 1
138            lista += str(cont)
139
140            return lista
141
```

Ejercicio 8

Se requiere encontrar la primera ocurrencia de un string $p_1...p_k$ en uno más largo $a_1...a_L$. Implementar esta estrategia de la forma más eficiente posible con un costo computacional menor a $O(K*L)$ (solución por fuerza bruta). Justificar el coste en tiempo de la solución propuesta.

Ejemplo 1:

Entrada: S = 'abracadabra' , P = 'cada'

Salida: 4, índice de la primera ocurrencia de P dentro de S (abra**cad**abra)

```
184 def isIn_v2(text, sub):
185
186     """
187     La complejidad sera de O(m+n) ya que depende de la longitud del texto insertado en la tabla (m) y la longitud del subtexto (n)
188     que van a ser iterados, en uno para la insercion en tabla, y el otro en la busqueda
189     """
190
191
192     aux = len(sub)
193     dict = d.Dictionary(hash_text, 97)
194
195     for i in range(len(text)):
196         if i+aux <= len(text):
197             new_char = text[i:i+aux]
198             dict.insert(dict.D, new_char, new_char)
199             found = dict.search(dict.D, sub)
200
201     return found != None
202
203
204     return False
206 def hash_text(text):
207
208     char_ascii = 0
209
210     for i in range(len(text)):
211         char_ascii += ord(text[i])*10**i
212
213     hash_function = char_ascii % 97
214
215     return hash_function
216
```

Ejercicio 9

Considerar los conjuntos de enteros $S = \{s_1, \dots, s_n\}$ y $T = \{t_1, \dots, t_m\}$. Implemente un algoritmo que utilice una tabla de hash para determinar si $S \subseteq T$ (S subconjunto de T). ¿Cuál es la complejidad temporal del caso promedio del algoritmo propuesto?

```
217 def isSubConj(C, S):
218
219     """
220     La complejidad sera de O(n), siendo n la longitud de la cadena mas larga
221     """
222
223     dict = d.Dictionary(None, 37) # elijo un numero primo
224
225     for i in C:
226         dict.insert(dict.D, i, str(i))
227
228     for i in S:
229
230         aux = dict.search(dict.D, i)
231
232         if aux != str(i):
233             return False
234
235     return True
236
```

Parte 3

Ejercicio 10

Considerar la inserción de las siguientes llaves: 10; 22; 31; 4; 15; 28; 17; 88; 59 en una tabla hash de longitud $m = 11$ utilizando direccionamiento abierto con una función de hash $h'(k) = k$. Mostrar el resultado de insertar estas llaves utilizando:

1. Linear probing
2. Quadratic probing con $c1 = 1$ y $c2 = 3$
3. Double hashing con $h1(k) = k$ y $h2(k) = 1 + (k \bmod (m - 1))$

Ejercicio 11 (opcional)

Implementar las operaciones de `insert()` y `delete()` dentro de una tabla hash vinculando todos los nodos libres en una lista. Se asume que un slot de la tabla puede almacenar un indicador (flag), un valor, junto a una o dos referencias (punteros). Todas las operaciones de diccionario y manejo de la lista enlazada deben ejecutarse en $O(1)$. La lista debe estar doblemente enlazada o con una simplemente enlazada alcanza?

Ejercicio 12

Las llaves 12, 18, 13, 2, 3, 23, 5 y 15 se insertan en una tabla hash inicialmente vacía de longitud 10 utilizando direccionamiento abierto con función hash $h(k) = k \bmod 10$ y exploración lineal (linear probing). ¿Cuál es la tabla hash resultante? Justifique.

0	
1	
2	2
3	23
4	
5	15
6	
7	
8	18
9	

(A)

0	
1	
2	12
3	13
4	
5	5
6	
7	
8	18
9	

(B)

0	
1	
2	12
3	13
4	2
5	3
6	23
7	5
8	18
9	15

(C)

0	
1	
2	12, 2
3	13, 3, 23
4	
5	5, 15
6	
7	
8	18
9	

(D)

Ejercicio 13

Una tabla hash de longitud 10 utiliza direccionamiento abierto con función hash $h(k)=k \bmod 10$, y exploración lineal (linear probing). Después de insertar 6 valores en una tabla hash vacía, la tabla es como se muestra a continuación.

0	
1	
2	42
3	23
4	34
5	52
6	46
7	33
8	
9	

¿Cuál de las siguientes opciones da un posible orden en el que las llaves podrían haber sido insertadas en la tabla? Justifique

- (A) 46, 42, 34, 52, 23, 33
- (B) 34, 42, 23, 52, 33, 46
- (C) 46, 34, 42, 23, 52, 33
- (D) 42, 46, 33, 23, 34, 52

A tener en cuenta:

1. Usen lápiz y papel primero
2. ~~No se puede utilizar otra Biblioteca mas allá de algo1.py y las bibliotecas desarrolladas durante Algoritmos y Estructuras de Datos I.~~

