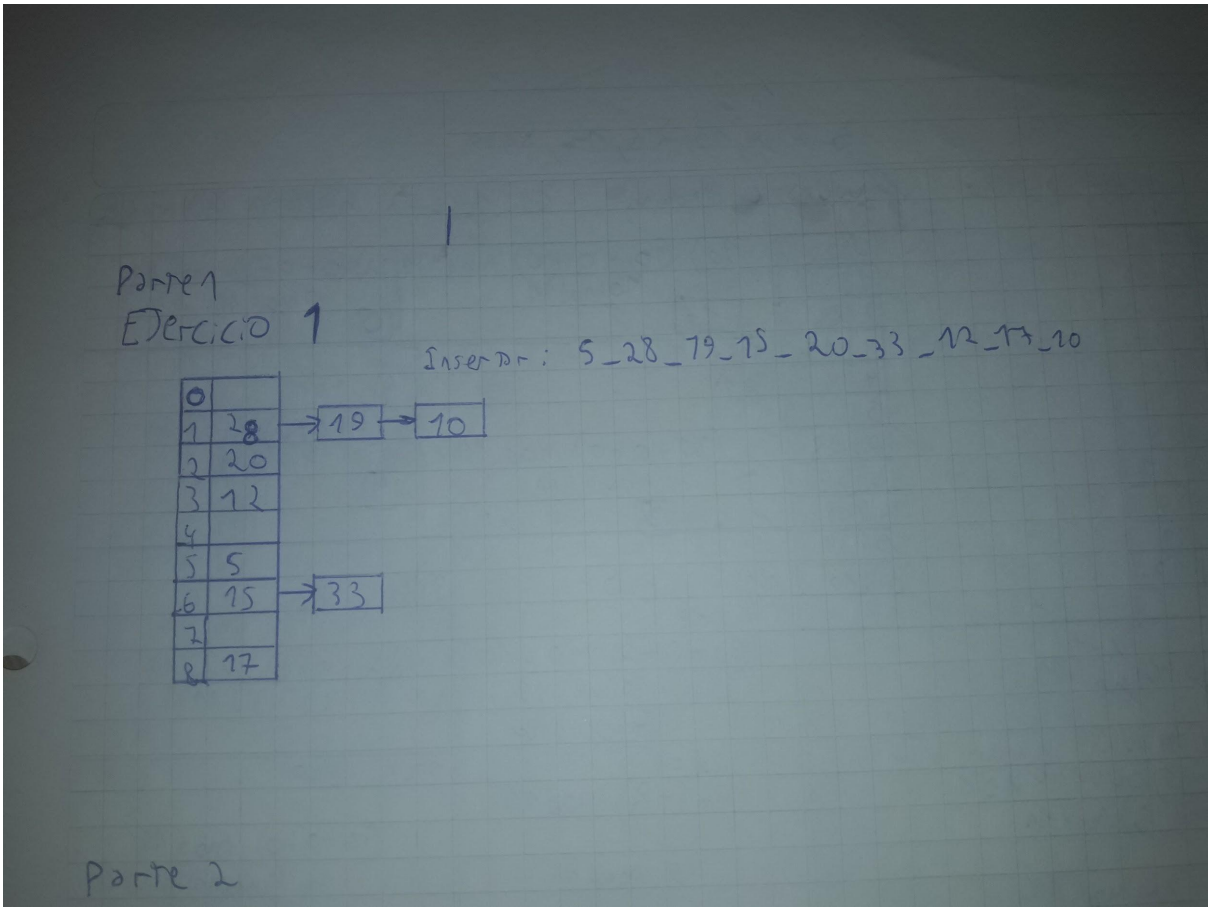
PARTE 1

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:

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



Ejercicio 2

A partir de una definición de diccionario como la siguiente:

```
dictionary = Array(m,0)
```

Crear un modulo 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 se quiere realizar la eliminación y el valor del key que se va a eliminar.

Salida: Devuelve **D**

```
def HashInt(D,key):
    modl=len(D)
    return key%len(D)
def HashAscii(D,key):
    n=0
    for i in range (len(key)):
        n=n+(ord(key[i])-48)
    modl=len(D)
    return n % modl

def insert_Dict(D,key,value,H):
    if search_Dict(D,key,H)!=None:
        return None
    newKey=H(D,key)
    if newKey==None:
        return None

    newNode=dictionaryNode()
    newNode.value=value
    newNode.key=key

    if D[newKey]==None:
        D[newKey]=dictionary()
```

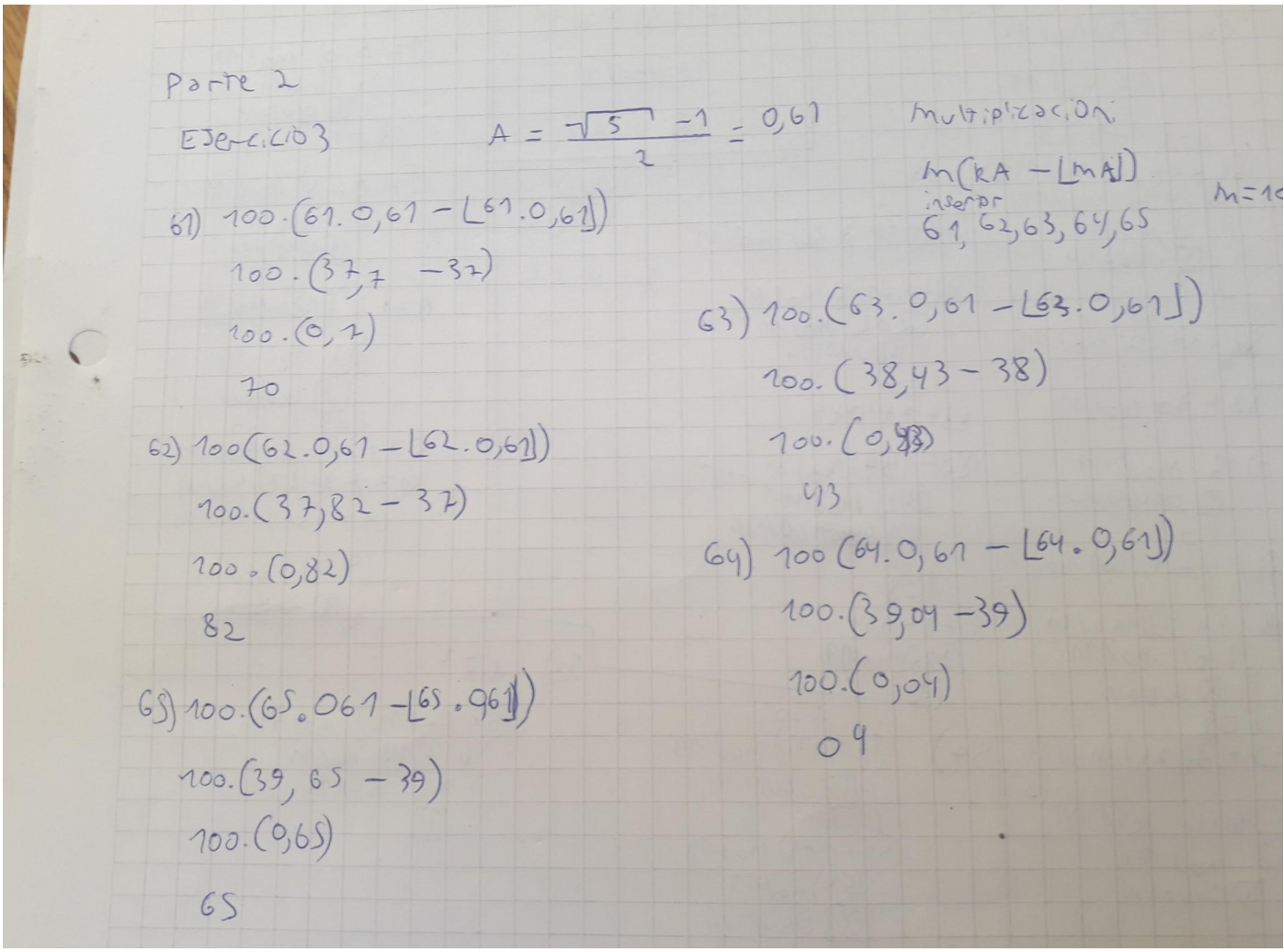
```
                D[newKey].head=newNode
        else:
                add_nodo(D[newKey],newNode)
                return D

def search_Dict(D,key,H):
        searchkey=H(D,key)
        if D[searchkey]==None:
                return None
        else:
                node=D[searchkey].head
                while node!=None:
                        if node.key==key:
                                return node.value
                        else:
                                node=node.nextNode
                return None



def delete_Dict(D,key,H):
        if search_Dict(D,key,H)==None:
                return
        newKey=H(D,key)
        node=D[newKey].head
        prevNode=None
        while node!=None:
                print(node.key)
                if node.key==key:
                        if node==D[newKey].head:
                                if node.nextNode==None:
                                        D[newKey]=None
                                else:
                                        D[newKey].head = node.nextNode
                        else:
                                if node.nextNode==None:
                                        prevNode.nextNode=None
                                else:
                                        prevNode.nextNode=node.nextNode
                        break
                else:
                        if prevNode==None:
                                prevNode=node
                        else:
                                prevNode=prevNode.nextNode
                        node=node.nextNode
```

return D

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.



Ejercicio 4

Implemente un algoritmo lo más eficiente posible que devuelva True o False a la siguiente proposición: dado dos strings $s_1...s_k$ y $p_1...p_k$, se quiere encontrar si los

caracteres de $p_1...p_k$ corresponden a una permutación de $s_1...s_k$. Justificar el coste en tiempo de la solución propuesta.

Ejemplo 1:
Entrada: S = ‘hola’ , P = ‘ahlo’
Salida: True, ya que P es una permutación de S

Ejemplo 2:
Entrada: S = ‘hola’ , P = ‘ahdo’
Salida: Falso, ya que P tiene al caracter ‘d’ que no se encuentra en S por lo que no es una permutación de S

```
def EsPermutacion(D,S1,S2):
	if len(S1)!=len(S2):
		return False
#	D=Array((len(S1)),dictionary())
	for i in range (len(S1)):
		ch=S1[i]
		insert_Dict(D,ch,ch)
	printDictionary(D)
	for i in range (len(S1)):
		print(S2[i])
		printDictionary(D)
		if search_Dict(D,S2[i])!=None:
			delete_Dict(D,S2[i])
		else:
			return False
	return True
```

La complejidad temporal es O(n^2)

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

```
def isUnique(D,L):
      cnode=L.head
      for i in range (length(L)):
            if insert_Dict(D,cnode.value_L,cnode.value_L)==None:
                  return False
```

```
            cnode=cnode.nextNode
        return True
```

El coste temporal es de O(n)

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.

```
def HashCodPost(D,key):
        return (ord(key[0]) -65)


def postalCode(D,key):
        key = key.upper()
        if len(key)!=8:
                return None
        for i in range (8):
                if i==0 or i>=5:

                        if ord(key[i])<65 or ord(key[i])>90:
                                return None
                else:
                        if ord(key[i])<48 or ord(key[i])>57:
                                return None

        if key[0]=="I" or key[0]=="O" or key[0]=="Ñ":
                return None
        insert_Dict(D,key,key,HashCodPost)
        return True
```

Ejercicio 7

Implemente un algoritmo para realizar la compresión básica de cadenas utilizando el recuento de caracteres repetidos. Por ejemplo, la cadena ‘*aabccccccaaa’* 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.

```
def LetterCount(D,S):
        cont=0
        contadorLetras=1
        insert_Dict(D,cont,(S[0]+str(contadorLetras)),HashInt)
        for i in range (1,len(S)):
                if S[i]!=S[i-1]:
                        cont=cont+1
                        contadorLetras=1
                        insert_Dict(D,cont,(S[i]+str(contadorLetras)),HashInt)
                else:
                        contadorLetras+=1
                        delete_Dict(D,cont,HashInt)
                        insert_Dict(D,cont,(S[i]+str(contadorLetras)),HashInt)
        returnString=""
        for i in range (len(D)):
                if D[i]!=None:
                        returnString=returnString+D[i].head.value
                else:
                        return returnString
        return returnString
```

El coste temporal es de O(n)

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 (abracadabra)

```
def isSubString(S1,S2):
        D=Array(len(S1),dictionary())
        cont=0
        for i in range (len (S1)):
                if S1[i]==S2[0]:
                        cont+=1
        for i in range (len(S1)):
```

```
            if S1[i]==S2[0]:
                if (len(S2)+i)<=len(S1):
                    stri=substr(S1,i,i+(len(S2)))
                    insert_Dict(D,i,stri,HashInt)
                else:
                    break
    printDictionary(D)
    for i in range (len(D)):
        if D[i]!=None:
            if S2==str(D[i].head.value):
                return i
```

La solución propuesta tiene una complejidad de O(L)

Ejercicio 9

Considerar los conjuntos de enteros $S = \{s1, \ldots, sn\}$ y $T = \{t1, \ldots, tm\}$. Implemente un algoritmo que utilice una tabla de hash para determinar si S ⊆ T (S subconjunto de T). ¿Cuál es la complejidad temporal de caso promedio del algoritmo propuesto?

```
def isSubGroup(AS,AT):
    if len(AS)>len(AT):
        return False
    D=Array(len(AT),dictionary())
    for i in range (len(AT)):
        insert_Dict(D,AT[i],AT[i],HashInt)
    printDictionary(D)
    for i in range (len(AS)):
        if search_Dict(D,AS[i],HashInt)==None:
            return False
    return True
```

La complejidad resulta en O(n^2)

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 10) 10, 22, 31, 4, 15, 28, 17, 88, 59

Linear probing | Cuadratic Probing $c1=1$ $c2=3$ $h'(k)=k$ $(h'(k)+c_1 i+c_2 i^2) \bmod m$

0	22
1	88
2	None
3	None
4	4
5	15
6	28
7	17
8	59
9	31
10	10

0	22
1	
2	88
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

10) $(10+0+0) \bmod 11$
$10 \bmod 11$
[10]

22) $(0+0+0) \bmod 11$
$0 \bmod 11$
[0]

31) $(31+0+0) \bmod 11$
$31 \bmod 11$
[9]

4) $i=0$
$(4+0+0) \bmod 11$
$4 \bmod 11$
[4]

15) $i=0$
$(15+0+0) \bmod 11$
$15 \bmod 11$
[4] ocupado
$i=1$
$(15+1+3) \bmod 11$
$19 \bmod 11$
[8]

28) $i=0$
$(28+0+0) \bmod 11$
$28 \bmod 11$
[6]

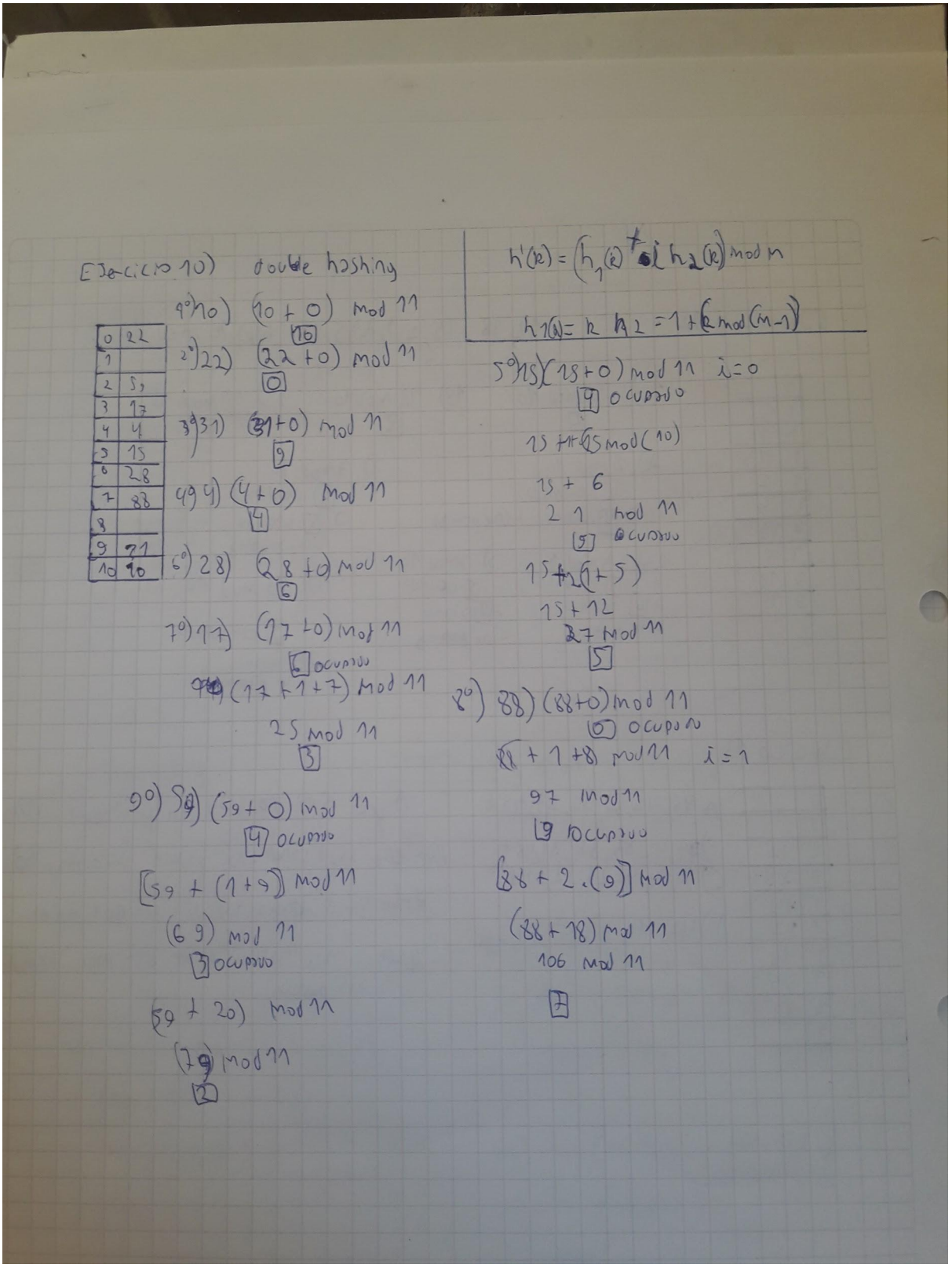
88) $(88+0+0) \bmod 11$ $i=0$
$88 \bmod 11$
[0] ocupado $i=1$
$(88+1+3) \bmod 11$
$92 \bmod 11$
[4] ocupado
$i=2$
$(88+2+12) \bmod 11$
$(90+12) \bmod 11$
$(102) \bmod 11$
[3] ocupado
$i=3$
$88+3+27 \bmod 11$
$88+30 \bmod 11$
$118 \bmod 11$
[8] ocupado
$i=4$
$(88+4+3.16) \bmod 11$
$(88+4+48) \bmod 11$
$(140) \bmod 11$
[8] ocupado
$i=5$
$(88+5+3.25) \bmod 11$
$(88+80) \bmod 11$
$168 \bmod 11$
[3] ocupado

$(88+6+3.36) \bmod 11$
$88+6+108) \bmod 11$
$(202) \bmod 11$
[4] ocupado
$i=7$
$88+7+147$
$(88+154) \bmod 11$
$(242) \bmod 11$
[0] ocupado
$i=8$
$88+8+3.64$
$(288) \bmod 11$
[2]

17) $i=0$
$(17+0+0) \bmod 11$
$17 \bmod 11$
[6] ocupado
$i=1$
$(17+1+3) \bmod 11$
$21 \bmod 11$
[9] ocupado
$i=2$
$(17+2+12) \bmod 11$
$31 \bmod 11$
[9] ocupado
$i=3$
$(17+3+3.9) \bmod 11$
$(17+3+27) \bmod 11$
$(17+30) \bmod 11$
$(47) \bmod 11$
[3] ocupado

59) $i=0$
$(59+0+0) \bmod 11$
[4] ocupado
$i=1$
$(59+1+3) \bmod 11$
$(63) \bmod 11$
[8] ocupado

59) $i=2$
$(59+2+12) \bmod 11$
$61+14 \bmod 11$
$73 \bmod 11$
[7]



0	22
1	
2	59
3	17
4	4
5	15
6	28
7	88
8	
9	31
10	10

Ejercicio 11

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?

La implementación es mejor con doblemente enlazada ya que es más fácil así conectar los nodos en la función delete() y dejarlos así en una complejidad más baja

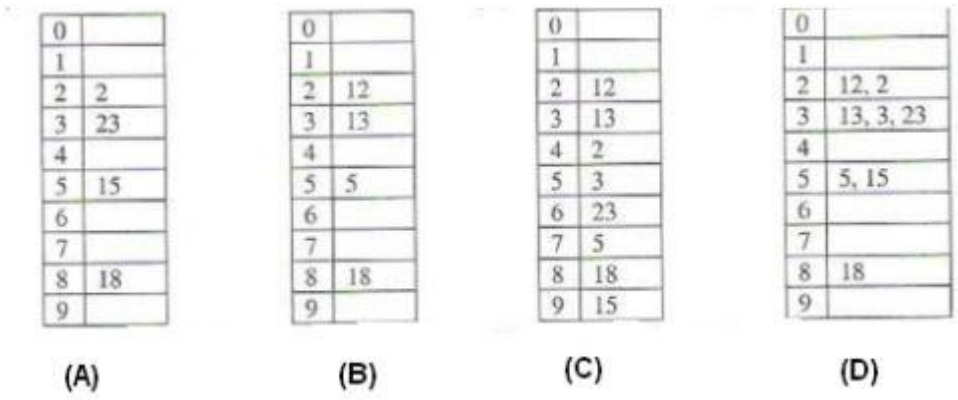
```
class doubleLinkedList:
    head=None
class doubleNode:
    value=None
    key=None
    nextNode=None
    prevNode=None
    redirect=None
    deleted=False

def dependentList(D):
    L=doubleLinkedList()
    for i in range (len(D)-1,-1,-1):
        node=doubleNode()
        node.redirect=LinkedList()
        D[i]=node
        addNode(L,node)
    print(length(L))
    return L


def addNode(L,node):
    node.nextNode=L.head
    if L.head!=None:
        L.head.prevNode=node
    L.head=node


def createList(N,L):
    for i in range (N):
        add_double(L,None,None)
```

```
def add_double(L,value,key):
        newNode=doubleNode()
        newNode.value=value
        newNode.key=key
        if L.head==None:
                L.head=newNode
        else:
                newNode.nextNode=L.head
                L.head.prevNode=newNone
                L.head=newNode

def dictInsertOpen(D,L,key,value,H):
        n=H(D,key)
        if D[n].key==None or D[n].deleted!=False:
                D[n].key=key
                D[n].value=value
                if n==0:
                        L.head=D[n].nextNode
                else:
                        D[n].prevNode.nextNode=D[n].nextNode
                        if n<len(D)-1:
                                D[n].nextNode.prevNode=D[n].prevNode
                        D[n].nextNode=L.head
                        D[n].prevNode=L.head.prevNode
                        if L.head.prevNode!=None:
                                L.head.prevNode.nextNode=D[n]
                        L.head.prevNode=D[n]
        else:
                L.head.value=value
                L.head.key=key
                add(D[n].redirect,L.head)
                if L.head.nextNode!=None:
                        L.head=L.head.nextNode

def dictDeleteOpen(D,L,key,H):
        n=H(D,key)
        ImprimirCola(D[n].redirect)
        if D[n].key==key:
                putNextHead(D[n],L)
                D[n].key=None
                D[n].value=None
                D[n].deleted=True
```

```
        else:
                nodoDel=delete(D[n].redirect,key)
                print("nodoeliminado",nodoDel.key)
                putNextHead(nodoDel,L)
                nodoDel.key=None
                nodoDel.value_L=None

def putNextHead(node,L):
        node.nextNode=L.head.nextNode
        node.prevNode=L.head
        if L.head.nextNode!=None:
                L.head.nextNode.prevNode=node
        L.head.nextNode=node
```

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 mod 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 mod 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

Ejercicio 12

12, 18, 13, 2, 3, 23, 5, 15 h(k): k mod 10

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

12) 12 mod 10 → 2

18) 18 mod 10 → 8

13) 13 mod 10 → 3

2) 2 mod 10 → 2 ocupado → 3 usando Line probing

3) 3 mod 10 → 3 ocupado → 5 usando Line probing

23) 23 mod 10 → 3 ocupado → 6 usando Line probing

5) 5 mod 10 → 5 ocupado → 7 usando Line probing

15) 15 mod 10 → 5 ocupado → 8 usando Line probing

Ejercicio 13

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

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 y C son posibles en primeros insertos porque analizando cada posible sucesión, 33 debe ser el último insertado porque es el que más "saltos" debe dar para llegar a esa posición

REPL.IT URL: https://repl.it/@PatricioHeduan/Dictionary