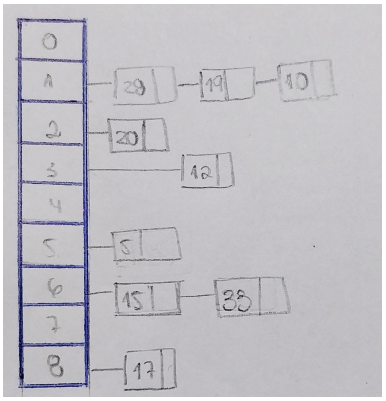
PARTE 1

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



Ejercicio 2

A partir de una definición de diccionario como la siguiente:

`dictionary = Array(m,0) # una sugerencia de implementacion,se puede usar una lista de python`

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

```
def insert(D,key,value):
    pos = key % 9
    m = 9
    if len(D) == 0: #esto lo va a hacer solo una vez
        for i in range(0,m):
            D.append(None)
    if D[pos] == None:
        D[pos] = LinkedList()
    print(key)
    add(D[pos],key,value)
    return 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.

```
def search(D,key):
    pos = key % 9
    if D[pos] != None:
      node = searchNode(D[pos],key)
      if node != None:
        return node.value #devuelve el value
    return None
```

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 None el **key** a eliminar.

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

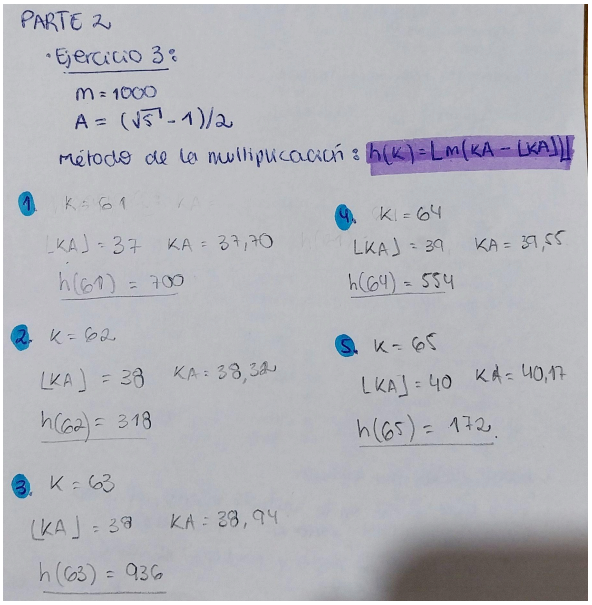
Salida: Devuelve **D**

```
def delete(D,key):
    pos = key % 9
    if D[pos] != None:
      deleteNode(D[pos],key)
      return D
    return None
```

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 costo en tiempo de la solución propuesta.

```
def esPermutacion(S,P):
  D = []
  if len(S) == len(P):
    m = numPrimo(len(S))

    for i in range(0,len(S)): #hago una hashT con los caracteres de S
      insert1(D,ord(S[i]),S[i],m)

    i = 0
    while i < len(P): #busco en la hashT los caracteres de P
      if search1(D,ord(P[i]),m) == None:
        print("Falso, ya que",P, "tiene al carácter",P[i], "que no se encuentra en",S,"por
lo que no es una permutación de", S)
        return False
      delete1(D,ord(P[i]),m)
      i += 1
    print("True, ya que", P, "es una permutacion de", S)
    return True
  else:
    print("Falso, ya que las palabras no tienen el mismo tamaño")
    return False
```

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 costo 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 elementosUnicos(L):
  m = numPrimo(len(L))
  D = inicializaDic([],m)
  flag = True
  node = None
  for i in range(0,len(L)): #defino una hashT con lo elementos de L
    node = search1(D,L[i],m) #busco el elemento antes de insertarlo
    if node != None: #si ya se encuentra en la hashT, devuelve false
      print("Falso,", L[i],"se repite en la", node.value,"y en la",
i+1,"posicion" )
      flag = False
    else: #sino lo ingresa
      insert1(D,L[i],i+1,m)
  return flag
```

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 hashPostal(codigo): #C1024CWN
  val = 0
  D = inicializaDic([],1009)
  for i in range(1,5):
    val += int(codigo[i])

  key = ord(codigo[0])*10^3 + ord(codigo[5])*10^2 + ord(codigo[6])*10+
ord(codigo[7]) + val
#sumo el valor entero y los codigos ascii de las letras con peso(para que las
permutaciones de los caracteres no me devuelvan el mismo valor) segun su
posicion esto me va a devolver un valor unico para cada codigo postal

  insert2(D,key,codigo,1009,(sqrt(5)-1)/2)
```

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 costo en tiempo de la solución propuesta.

```
def compresionCadenas(cad):
  m = numPrimo(len(cad)) #calculo el num primo
  D = inicializaDic([],m) #inicializa las pos de la hashT en none
  string = ""
  for i in range(0,len(cad)):
    letra = cad[i]
    node = search1(D,ord(letra),m) #busco el node que tiene key=letra en la hashT
    if node == None:
      insert1(D,ord(letra),1,m) #si no esta lo inserto
      node = search1(D,ord(letra),m) #lo guardo en una var
    else:
      node.value += 1 #si esta aumento el value
      #el campo value me va a indicar la cantidad de rep de la letra

    if i < len(cad)-1: #no estamos en el ultimo caracter
      if cad[i+1] != letra: #si la letra que sigue es diferente
        string += letra + str(node.value) #concateno la letra con el value
        node.value = 0 #reinicio el value porque la letra se puede repetir mas adelant
    else: #estamos en el ultimo caracter
      if i == len(cad)-1: #estoy en el ultimo char de la cadena
        string += letra + str(node.value) #concateno la letra con el value
        node.value = 0

  if len(string) < len(cad):
    return string
  else:
    return cad
```

costo computacional: O(n^2/m) ya que tengo un bucle for que recorre toda la cadena O(n) para insertar (O(1)) los valores en la hash table e ir aumentando el campo value y dentro de ese bucle uso una función search (O(n/m)) para buscar los elemento en la hash table

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 primeraOcurrencia(S,P):

  for i in range(0,len(S)): #recorro la cadena S
    if S[i] == P[0]: #si el primer caracter de P es igual a un caracter de S
      if S[i:i+len(P)] == P: #si la subcadeba de S que sigue a partir de i de long = 
len(P) es igual a P
        print(i)
        return True
      else:
        return False
```

costo computacional: O(n) ya que en el peor caso tengo que recorrer la toda cadena más larga

Ejercicio 9

Considerar los conjuntos de enteros $S = \{s1, . . . , sn\}$ y $T = \{t1, . . . , tm\}$. 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?

```
def esSubconjunto(S,T): #como son conjuntos los elementos no se repiten
  if len(S) > len(T):
    return False
  else:
    m = numPrimo(len(T))
    D = inicializaDic([],m)
    for i in range(0,len(T)): #hago una hashT con el conjunto mas grande
      insert1(D,T[i],T[i],m)

    for i in range(0,len(S)): #busco los elementos del conjunto las pequeño en la 
hashT
      if search1(D,S[i],m) == None:
        return False #si hay un elemento que no esta en la hashT no es subconjunto
    return True
```

costo computacional: O(n^2/m) ya que tengo un bucle for (O(n)) que recorre uno de los conjuntos y adentro una función search en una hashT de complejidad O(n/m)

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 mod (m - 1))*

1.

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

2.

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

3.

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

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)

R: la tabla resultante es la C

- Primero se inserta el 12 en la posición 2, descartamos la tabla A y como es direccionamiento abierto descartamos la tabla D
- Se inserta el 18 en la posición 8
- Se inserta el 13 en la posición 3
- Se intenta insertar el 2 en la posición 2 pero como ya está ocupado pasa a la posición desocupada que seria la 4
- Asi sucesivamente con los valores restantes

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

R: la opción C

1. inserta 46 en la pos 6
2. inserta 34 en la pos 4
3. inserta 42 en la pos 2
4. inserta 23 en la pos 3
5. intenta insertar 52 en la pos 2 pero como está ocupada busca la siguiente posición desocupada, la cual es la pos 5
6. intenta insertar 33 en las pos 3 por lo cual busca en las siguientes posiciones hasta llegar a la 7 que desocupada y lo inserta