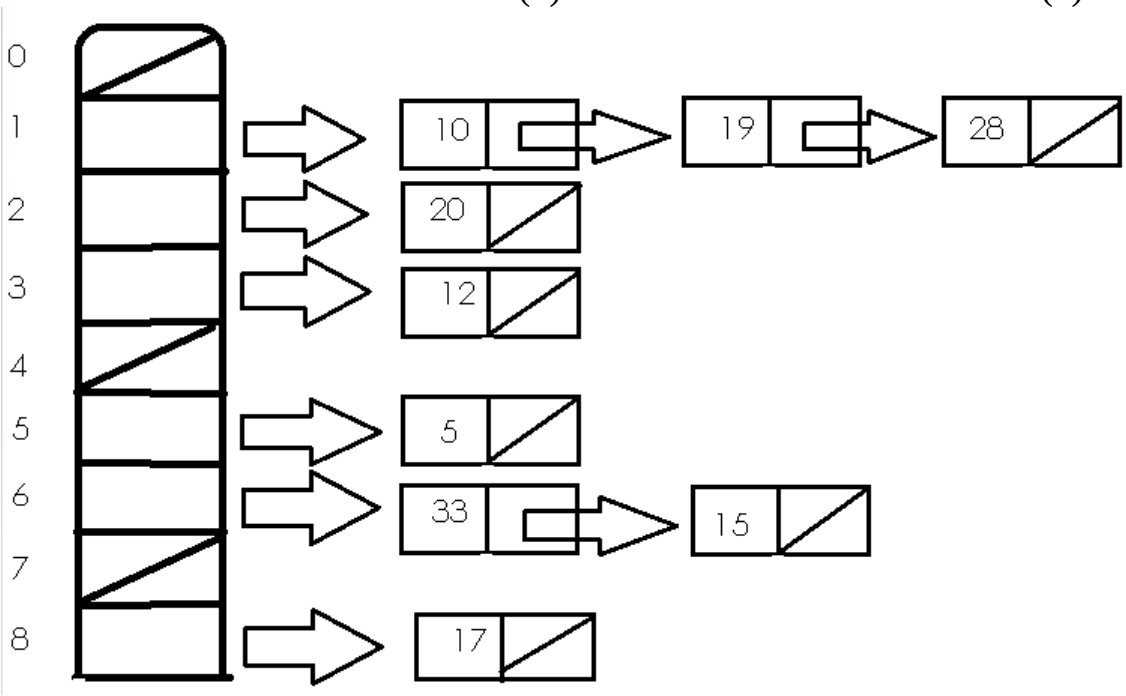
Hash Tables

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 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

```
11  #Insert del hash
12  def hashinsert(D,key,value):
13      node = dictionarynode()
14      node.value = value
15      node.key = key
16      hashindex = hashfunction(key,len(D))
17      if D[hashindex] == None:
18          list = []
19          list.append(node)
20          D[hashindex] = list
21      else:
22          D[hashindex].append(node)
23      return key
```

```
7   #Función hash de la división
8   def hashfunction(k,m):
9       return (k%m)
```

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.

```
25  #Search del hash
26  def hashsearch(D,key):
27      hashindex = hashfunction(key,len(D))
28      if D[hashindex] == None: return
29      for i in range(0,len(D[hashindex])):
30          if key == D[hashindex][i].key:
31              return key
32      return
```

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**

```
34  #Delete del hash
35  def hashdelete(D,key):
36      hashindex = hashfunction(key,len(D))
37      if D[hashindex] == None: return
38      for i in range(0,len(D[hashindex])):
39          if key == D[hashindex][i].key:
40              D[hashindex].pop(i)
41              if len(D[hashindex]) == 0:
42                  D[hashindex] = None
43              return D
44      return
```

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.

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 $s_1...s_k$ 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.

```python
#Ejercicio 4
def ispermutation(s,p):
    #Si la longitud de las dos cadenas son disti
    if len(s) != len(p) or s == p: return False
    hashtable = []
    hashtable = generartabla(hashtable)
    asciiS = 0
    asciiP = 0
    for i in range(0,len(s)):
        asciiS = asciiS + ord(s[i])
        asciiP = asciiP + ord(p[i])
    #Con el codigo ascii como key inserto una ca
    hashinsert(hashtable,asciiS,s)
    key = hashsearch(hashtable,asciiP)
    if key != None:
        return True
    return False
```

Coste del tiempo: Siendo n la longitud de las dos palabras (ya que si es una permutación deben tener la misma longitud), el coste del algoritmo va a ser de O(n), ya que recorremos las dos palabras para generar dos llaves.
Si las llaves son iguales entonces estamos hablando de una permutación.

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.

```python
#Ejercicio 5
def uniquelist(list):
    hashtable = []
    hashtable = generartabla(hashtable)
    hashinsert(hashtable,list[0],list[0])
    for i in range(1,len(list)):
        if hashsearch(hashtable,list[i]) != None:
            return False
        hashinsert(hashtable,list[i],list[i])
    return True
```

Coste del tiempo: El tiempo computacional es O(n) siendo n la longitud de la lista ya que en el peor caso la lista es única, por lo tanto recorremos toda la lista.

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.

```python
#Ejercicio 6
def codigopostal(codigo,hashtable):
    key = 0
    sizecodigo = len(codigo)
    for i in range(0,sizecodigo):
        key = key + (ord(codigo[i])) * pow(26,sizecodigo-1-i)
    hashinsert(hashtable,key,codigo)
    return hashtable
```

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.

```python
#Ejercicio 7
def compressed(s):
    #Hago toda la cadena en minúscula y inicializo una cadena
    s = s.lower()
    compressedstring = ""
    character = s[0]
    j = 1
    for i in range(1,len(s)):
        if s[i] == character:
            j += 1
        else:
            compressedstring = compressedstring + character
            compressedstring = compressedstring + str(j)
            j = 1
            character = s[i]
    compressedstring = compressedstring + character
    compressedstring = compressedstring + str(j)
    #Si la longitud de la cadena comprimida es mayor o igual
    if len(compressedstring) >= len(s):
        return s
    return compressedstring
```

Coste del tiempo: El coste computacional del algoritmo es o(N) siendo n la longitud de la palabra ya que la recorremos para ir buscando las veces que se repite una letra.

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.

```python
#Ejercicio 8
def ocurrencia(p,a):
    sizep = len(p)
    sizea = len(a)

    hashtable = []
    hashtable = generartabla(hashtable)
    keyp = 0
    for i in range(0,sizep):
        keyp = keyp + (ord(p[i]) * pow(256,sizep-1-i))
    hashinsert(hashtable,keyp,p)

    hashtexto = 0
    for i in range(0,sizep):
        hashtexto = hashtexto + (ord(a[i]) * pow(256,sizep-1-i))

    for i in range(0,sizea - sizep + 1):
        if hashsearch(hashtable,hashtexto) != None:
            if a[i:i+sizep] == p:
                return i
        if i < sizea - sizep:
            hashtexto = (hashtexto - ord(a[i]) * pow(256,sizep-1)) * 256 + ord(a[i+sizep])
    return -1
```

Coste computacional: El coste computacional del algoritmo es O(L) siendo l la longitud del patrón en el mejor o caso promedio ya que primero recorremos todo el patrón para luego insertarlo en la tabla, eso se hace en o(l). Luego buscamos en la palabra una subcadena con longitud l para buscarlo en el hash y verificar si esa es la cadena.
En el peor caso que no se encuentre la palabra sería de o(K*L) ya que tendremos que recorrer todo k siendo k la longitud de la palabra.

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 ⊆ T (S subconjunto de T). ¿Cuál es la complejidad temporal del caso promedio del algoritmo propuesto?

```python
#Ejercicio 9
def subconjunto(t,s):
    #T conjunto,s subconjunto
    hashtable = []
    hashtable = generartabla(hashtable)
    for i in range(0,len(t)):
        hashinsert(hashtable,t[i],t[i])
    for i in range(0,len(s)):
        if hashsearch(hashtable,s[i]) == None:
            return False
    return True
```

Coste computacional: El coste computacional del algoritmo va a ser de o(t+s) siendo t la longitud del conjunto y s del subconjunto. En el caso que los dos juntos tengan la misma longitud seria de o(t).

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))*

Linear Probing:

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

Quadratic probing:

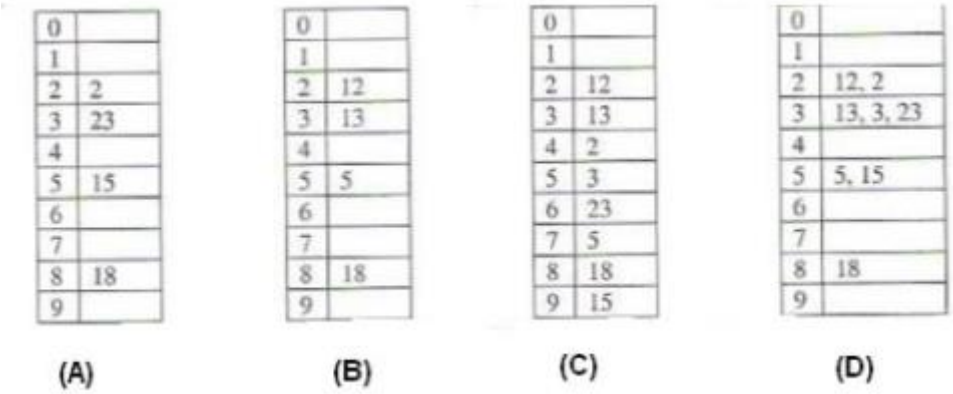
0	22
1	
2	88
3	17
4	4
5	
6	28
7	59
8	15
9	31
10	10

Double hashing:

0	22
1	
2	59
3	17
4	4
5	15
6	28
7	88
8	
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)

La table resultante es la C ya que las dos primera tablas no están insertadas todas las key y en la D la colisión se resuelve a través de encadenamiento.

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

La respuesta correcta es la C, ya que las tablas quedan así.

A)

		42	52	34	23	46	33		

B)

		42	23	34	52	33	46		

C)

		42	23	34	52	46	33		

D)

		42	33	23	34	46	52		