PARTE 1: Massacesi Juan Ignacio

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:

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

0	
1	28,19,10
2	20
3	12
4	
5	5
6	15,33
7	
8	17

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

```python
#Insert without Universal-function
def insert(D,key,value):
    if len(D) == 0:
        return None
    #Calculamos la nueva key para asignar un nuevo slot
    position = key % len(D)
    if D[position] == None:
        #Si el slot es Null, creamos una lista y agregamos la tupla(key,value)
        D[position] = []
        D[position].append((key,value))
    else:
        #Agregamos la tupla(key,value) a la lista en el slot dado por la hash-function
        D[position].append((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.

```python
#Search without Universal-function
def search(D,key):
    if len(D) == 0:
        return None
    #Calculo la nueva key usando la hash-function
    position = key % len(D)
    if D[position] == None:
        return None
    else:
        #Buscamos el value de la lista dentro del slot dado por la hash-function
        List = D[position]
        for i in range(len(List)):
            if List[i][0] == key:
                return List[i][1]
        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 nulo el **key** a eliminar.
Entrada: El diccionario sobre else quiere realizar la eliminación y el valor del key que se va a eliminar.
Salida: Devuelve **D**

```python
#Delete without Universal-function
def delete(D,key):
    if len(D) == 0:
        return None
    position = key % len(D)
    if D[position] != None:
        ListValue = D[position]
        for i in range(len(ListValue)):
            if ListValue[i][0] == key:
                ListValue.pop(i)
                break
        return D
    else:
        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.

h(k) = [m*(k*A MOD 1)]

h(61) = [1000*(61*($\sqrt{5}$ -1)/2 MOD 1)] = 700
h(62) = [1000*(62*($\sqrt{5}$ -1)/2 MOD 1)] = 318
h(63) = [1000*(63*($\sqrt{5}$ -1)/2 MOD 1)] = 936
h(64) = [1000*(64*($\sqrt{5}$ -1)/2 MOD 1)] = 554
h(65) = [1000*(65*($\sqrt{5}$ -1)/2 MOD 1)] = 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.

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 carácter ‘d’ que no se encuentra en S por lo que no es una permutación de S

```python
def PpermutacionS(D,s,p):
    if len(s) != len(p):
        return False
    elif s == p:
        return False
    else:
        #Insertamos cada letra de S en el hash
        for i in range(len(s)):
            insert(D,ord(s[i]),s[i])
        #Buscamos cada letra de P en el hash
        contLetrasDeP = 0
        for i in range(len(p)):
            value = search(D,ord(p[i]))
            delete(D,ord(p[i]))
            if value != None:
                contLetrasDeP += 1
        #Verificar si P es una permutación de S
        if contLetrasDeP == len(p):
            return True
        else:
            return False
```

El coste de la solución es de O(n), ya que se recorre un bucle n veces (tamaño de la cadena) para guardar en una variable la suma del ascii de cada letra de la palabra. Las operaciones de insert y search se ejecutan de manera constante O(1).

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

```python
def searchRepeated(D,List):
    if List == None:
        return True
    else:
        for i in range(len(List)):
            #Se busca si el elemento se ha ingresado al hash
            value = search(D,List[i])
            if value == None:
                #En caso de no encontrarse en el hash, se inserta
                insert(D,List[i],List[i])
            else:
                #En caso de encontrar el elemento se retorna False ya que el elemento a ingresar se encuentra en el hash
                return False
        return True
```

El coste del algoritmo es de O(n) ya que se recorre n veces un bucle for (n = longitud de la lista), a medida que recorremos la lista buscamos en el hash O(1) y si no se encuentra insertamos O(1).

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
def CodigoPostal(D,code):
    if code == "":
        return D
    #Calcular el código postal
    newCode = ""
    for i in code:
        if i == "c":
            #Calcular una letra aleatoria de la A-Z
            letter = str(random.choice(string.ascii_letters))
            newCode = newCode + str.upper(letter)
        elif i == "d":
            #Calcular el número aleatorio del 0-9
            number = random.randint(0,9)
            newCode = newCode + str(number)
    #Calcular la key del código postal
    #int(newCode[1:5]), newCode[inicio:final-1]
    newKey = (ord(newCode[0])*10^4) + int(newCode[1:5]) + (ord(newCode[5])*10^3) + (ord(newCode[6])*10^2) + (ord(newCode[7]
    #Insertar el código en el diccionario
    D = insert(D,newKey,newCode)
    return D
```

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.

```python
def CompressionString(string):
    if string == "":
        return string
    else:
        #Transformo las letras de la cadena a minúsculas
        string = string.lower()
        #Variables previas para calcular la cadena comprimida
        compressionString = ""
        countRep = 1
        firstLetter = string[0]
        #Calculo la cadena comprimida
        for i in range(1,len(string)):
            if string[i] == firstLetter:
                countRep += 1
            else:
                compressionString += firstLetter
                compressionString += str(countRep)
                firstLetter = string[i]
                countRep = 1
        compressionString += firstLetter
        compressionString += str(countRep)
        #Verifico que la cadena comprimida sea menor a la cadena original y la devuelvo
        #Caso contrario devuelvo la cadena original
        if len(compressionString) > len(string):
            return string
        else:
            return compressionString
```

El coste del algoritmo es O(n), recorremos un bucle n veces (n = longitud de la cadena a comprimir) para crear la cadena comprimida en base a la cadena a comprimir.

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 SearchOcurrencia(D,a1,p1):
    if a1 == "" or p1 == "":
        return
    else:
        p1 = p1.lower()
        a1 = a1.lower()
        keyP1 = 0
        pow = len(p1)
        for i in range(len(p1)):
            keyP1 += ord(p1[i]) * 10^(pow)
            pow -= 1
        #Insertar la cadena a1 en un slot del diccionario
        insert(D,keyP1,p1)
        #Calcular todas las cadenas de a1 de longitud igual a p1 e insertarlas en una lista
        flag = False
        i=0
        f = len(p1)
        List = []
        while flag == False:
            if len(a1[i:f]) == len(p1):
                List.append(a1[i:f])
                i += 1
                f += 1
            else:
                flag = True
        #Calcular el key de las cadenas de a1 de longitud igual a p1
        for i in range(len(List)):
            keyA1 = 0
            pow = len(List[i])
            for j in range(len(List[i])):
                print(pow)
                keyA1 += ord(List[i][j]) * 10^(pow)
                pow -= 1
            #Vefificar si se encontro la ocurrencia de a1 en p1 y retornar el indice donde comienza
            stringP1 = search(D,keyA1)
            if stringP1 == p1:
                return i
        return None
```

El coste del algoritmo es de O(m x n), donde m es la longitud de la lista formada por subcadenas de a1 de igual longitud a p1 y n es la cantidad de caracteres que tienen las palabras de la lista.

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

```python
def SsubconjuntoT(D,S,T):
    if S != [] and T == []:
        return False
    #El Vacio es subconjunto de todo conjunto
    if S == [] and T != []:
        return True
    #Un conjunto es subconjunto de si mismo
    elif S == T:
        return True
    else:
        #Insertar T en el hash
        for i in range(len(T)):
            insert(D,T[i],T[i])
        #Buscar S en el hash
        contSubconjunto = 0
        for i in range(len(S)):
            value = search(D,S[i])
            if value == S[i]:
                contSubconjunto += 1
        #Veficar que es un subconjunto de T
        if contSubconjunto == len(S):
            return True
        else:
            return False
```

El caso promedio sería de $\Theta(n + m)$, se recorre una vez S y T, quedaría determinado por la longitud de las listas S y T. En este caso las listas deberían de tener misma longitud o la lista S debería de ser más pequeña que la lista T.

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

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

10	10

2. Quadratic probing con *c1 = 1* y c2 = 3

llaves: 10; 22; 31; 4; 15; 28; 17; 88; 59

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

3. Double hashing con *h1(k) = k* y *h2(k) = 1 +(k mod (m - 1))*

llaves: 10; 22; 31; 4; 15; 28; 17; 88; 59

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

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 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 tabla de hash resultante es la (C), ya que cuando ingresemos es 2 al hash su posición estará ocupada por el 12 por ende se baja hasta la primer posición vacía en ese caso sería la número 4 (la 3 ocupada por el 13)y así pasaría con en el 3, 23, 5 y 15.

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 en primer lugar ingresamos el 46 en la posición 6 , segundo el 34 en la 4, tercero el 42 en la 2, cuarto el 23 en la 3, quinto el 52 en la 5 (según la función de hash iría en la 2 pero está ocupada por el 42 por lo tanto hay que bajar hasta la primera que esté vacía por eso la 5) y por último el 33 en la 7(según la función de hash iría en la 3 pero está ocupada por el 23 por lo tanto hay que bajar hasta la primera que esté vacía por eso la 7).

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.**~~