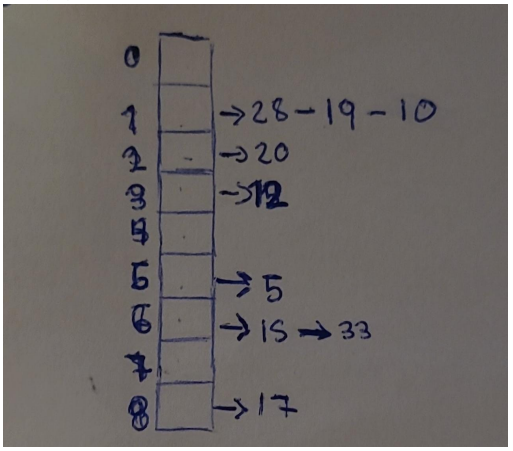
MAURO JOSÉ SORBELLO DIAZ LEGAJO: 14006

# 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

```python
def insert(D, key, value):
    ind = hash_mod(key, len(D))
    if D[ind] == None:
        D[ind] = []
        D[ind].append((key, value))
    else:
        D[ind].append((key, value))
```

**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 searchD(D, key):
    ind = hash_mod(key, len(D))
    if D[ind] == None:
        return None
    else:
        L = D[ind]
        i = 0
        while L[i] != None:
            tupla = L[i]
            if tupla[0] == key:
                return [i, tupla[1]]
            else:
                i += 1
    return None


def search(D, key):
    L = searchD(D, key)
    if L != None:
        return L[1]
    else:
        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 el se quiere realizar la eliminación y el valor del key que se va a eliminar.
**Salida:** Devuelve **D**

```
def delete(D, key):
    L = searchD(D, key)
    if L != None:
        ind = int(hash_mod(key, len(D)))
        i = L[0]
        D[ind][i] = None
        return D
    else:
        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.

```
A = (math.sqrt(5) - 1) / 2

m = 1000

print(hash_mult(61, m, A))  # 700
print(hash_mult(62, m, A))  # 318
print(hash_mult(63, m, A))  # 936
print(hash_mult(64, m, A))  # 554
print(hash_mult(65, m, A))  # 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
# Puedo guardar cada letra tomando como key su valor ascii,
# una vez guardada en mi hash table, busco la otra palabra por letra en mi hash table.
# Si todas coinciden es una permutacion, sino no lo es.

# El costo va a ser el largo de la primer palabra, O(n) dado a que tengo que recorrer
# la palabra, mejorando el costo O(n^2) de si lo hicieramos con bucles anidados

# No encuentro una forma en la que sea O(1), ya que si o si debemos recorrer la palabra.

# Como el codigo ascii es unico para cada letra, no se repetiran las keys


def permutacion(D, str1, str2):
    if len(str1) != len(str2):
        return False
    for i in range(0, len(str1)):
        key = int(ord(str1[i]))
        insert(D, key, str1[i])
    j = 0
    while j < (len(str2) - 1):
        L = search(D, ord(str2[j]))
        if L == None:
            return False
        j += 1
    return True
```

## 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
# Es O(n^2/m) en el peor de los casos porque el search es O(n/m), insert O(1).
# En el caso promedio es O(n) porque no recorre toda la lista y
# si esta bien distribuida es tener el primer for loop nomas.


def unico(D, L):
    for i in range(0, len(L)):
        key = L[i]
        check = search(D, key)
        if check != None:
            return False
        insert(D, key, 0)
    return True
```

## 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
# La calle se va a repetir, y nosotros no queremos que se forme una lista dentro de nuestro hash
# Toma el codigo ascii dde los primeros 4 y multiplica el primero por 1000, el segundo por 100,
# el tercero por 10 y el cuarto por 1, se suman todos los numeros.
# Despues toma de los segundos 4 y lo multiplica por 1000, el segundo por 100,
# el tercero por 10, y el ultimo por 1,se suman todos los numeros.
# De esa forma aunque la suma de los codigo ascii se repita.
# Luego se sumaran a las sumas de los numeros multiplicados y seran unicos
# La multiplicacion por distintos factores de 10,
# hacen que si hay una permutacion de esos numeros, no se repita el total.
# Asi se generara la key que sera unica, despues por metodo de la division hacemos mod m

def hash_cod_post(str, m):
    key = 0
    j = 1000
    for i in range(0, 8):
        num = ord(str[i]) * j
        key += num
        if i / 8 != 0.375:
            j = j / 10
        else:
            j = 1000
    return key % m
```

## 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 comp_str(str1):
    L = []
    L.append([str1[0], 1])
    cont = 0
    for i in range(1, len(str1)):
        if str1[i] == str1[i-1]:
            L[cont][1] += 1
        else:
            cont += 1
            L.append([str1[i], 1])
    str2 = ""
    for j in range (0, len(L)):
        if L[j] != None:
            if L[j][1] != 1:
                str2 = str2 + L[j][0] + str(L[j][1])
            else:
                str2 = str2 + L[j][0]
    return str2

#COSTE O(n) porque los loops no son anidados
```

## 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)

```python
#HASH TABLE
#Orden O(K), porque recorro la primer palabra y la inserto. Despues busco la otra palabra.
#Como el search es O(k/m), cuando recorra la otra palabra para buscar será O(1) dado a que nuestra funcion
#ordena correctamente
#Despues el insert sera tambien O(k/m)
#El problema es que depende de que tan bien se implemento la funcion de hash y el m elegido,
#para que el searchD y el insertMod conserven la complejidad O(n)
#En el caso de que nuestra m sea del mismo tamaño que la cantidad de letras en el abecedario,
#No habra colisiones debido a que cada numero % m sera diferente
```

```python
def insertMod(D, key, value, k):
    ind = hash_mod(key, len(D))
    if D[ind] == None:
        D[ind] = []
        D[ind].append([key, value, [k]])
    else:
        L = searchD(D, key)
        if L != None:
            ind = int(hash_mod(key, len(D)))
            i = L[0]
            D[ind][i][2].append(k)
            return D
        D[ind].append([key, value, [k]])

def ocurrenciaHash(D, st1, st2):
    if (len(st1)) > len(st2):
        strbig = st1
        strshort = st2
    else:
        strbig = st2
        strshort = st1
    for i in range(0, len(strbig)):
        insertMod(D, ord(strbig[i]), [strbig[i]], i+1)
    searchFirst = searchD(D, ord(strshort[0]))
    for j in range(1, len(strshort)):
        if searchD(D, ord(strshort[j])) == None:
            return False
    if searchFirst != None:
        ind = int(hash_mod(ord(strbig[0]), len(D)))
        pos =  searchFirst[0]
        indices = D[ind][pos][2]
    if len(indices) == 1:
        return indices[0]
    else:
        for w in range(0, len(indices)):
            indice = indices[w]
            if strbig[indice+1] == strshort[1]:
                return indice
    return indice
```

```python
#Otra forma es sin utilizar tabla Hash.
#En este caso el orden es O(n) siendo n la longitud de la palabra mas larga.
#Esto es debido a que en el peor caso se recorrera toda la palabra larga, sin haber coincidencia

def ocurrencia(st1, st2):
    if (len(st1)) > len(st2):
        strbig = st1
        strshort = st2
    else:
        strbig = st2
        strshort = st1
    j = 0
    coincidencia = ""
    indices = []
    for i in range(0, len(strbig)):
        if strbig[i] == strshort[j]:
            coincidencia = coincidencia + strbig[i]
            indices.append(i)
        if strbig[i] != strshort[j]:
            if len(coincidencia) != len(strshort):
                indices = []
                coincidencia = ""
                j = 0
        else:
            if j + 1 < len(strshort):
                j += 1
            else:
                return indices[0]
```

## 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
#En el caso promedio la complejidad es O(n), con n la longitud del conjunto mas largo.
#Porque va a depender del search, que depende de que este bien implementada la funcion hash


def subconjunto(D, T1, T2):
    if len(T1) > len(T2):
        subLarge = T1
        subShort = T2
    else:
        subLarge = T2
        subShort = T1
    for i in range(0, len(subLarge)):
        key = int(subLarge[i])
        insert(D, key, subLarge[i])
    j = 0
    while j < (len(subShort) - 1):
        L = search(D, subShort[j])
        if L == None:
            return False
        j += 1
    return True
```

# 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 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 | |

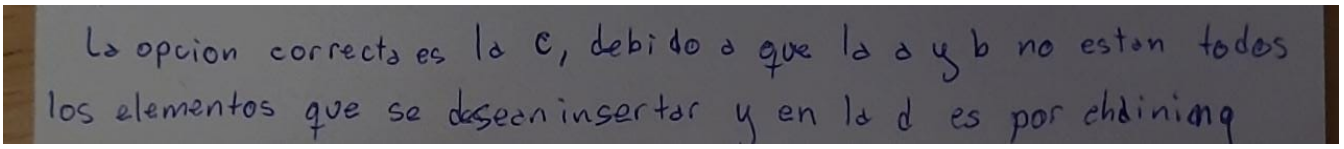
(B)

| | |
|---|---|
| 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 | |

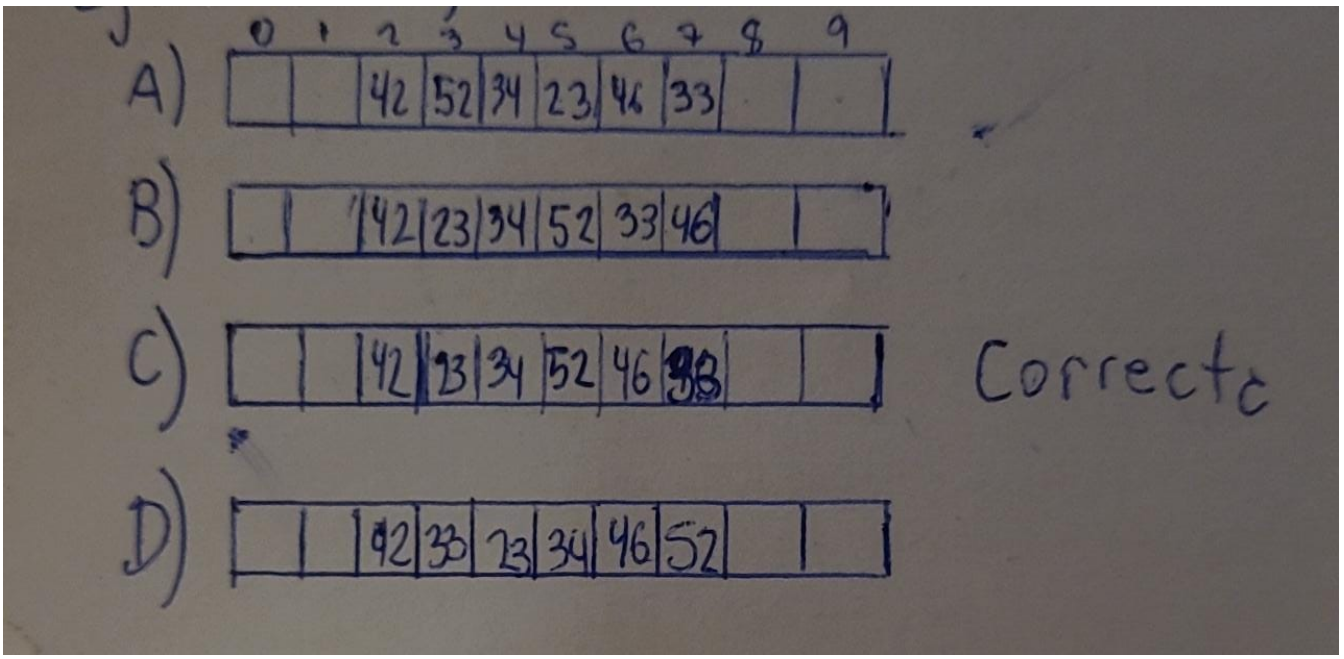
¿Cuál de las siguientes opciones da un posible orden en el que las llaves podrían haber sido insertadas en la tabla? Justifique

(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




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.**~~