

# Algoritmos y estructuras de datos 2

TP Hash Table

## 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

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

Al insertar utilizando esta función de hash, tenemos la siguiente salida

```
[0][]
[1][(28, -1), (19, -1), (10, -1)]
[2][(20, -1)]
[3][(12, -1)]
[4][]
[5][(5, -1)]
[6][(15, -1), (33, -1)]
[7][]
[8][(17, -1)]
```

Donde se observa que las siguientes key tienen el mismo hash

$h(28) = h(19) = h(10) = 1$

$h(20) = 2$

$h(12) = 3$

$h(5) = 5$

$h(15) = h(33) = 6$

$h(17) = 8$

**Ejercicio 2:** Crear un módulo de nombre **dictionary.py** que **implemente** las siguientes especificaciones de las operaciones elementales para el **TAD diccionario**

La implementación la realice orientada a objetos, el método constructor contiene

```
# Initializes the table, in each slot an empty linked list is initialized
self.table = [[] for _ in range(slots)]
```

Que simplemente genera 'slots' listas enlazadas dentro de otra, simulando el array

El método insert

```
def insert(self, key, value):
    """Insertion. Appends the tuple (key, value) to the corresponding list"""
    hash_value = self.hash_function(key)
    self.table[hash_value].append((key, value))
```

El search

```
def search(self, key):
    """Given the key of the element, tries to find it's
    value inside the corresponding list"""
    hash_value = self.hash_function(key)

    linked_list = self.table[hash_value]

    for node in linked_list:
        node_key = node[0]
        if node_key == key:
            return node[1]
```

Y delete

```
def delete(self, key):
    """Removes the element from the list"""
    # This implementation of Dictionary allows repeated
    # keys, that's why only one of those is removed,
    # following FIFO
    hash_value = self.hash_function(key)
    linked_list = self.table[hash_value]

    # Tries to find the key inside the
    node_index = -1

    for i, node in enumerate(linked_list):
        node_key = node[0]
        if node_key == key:
            node_index = i
            break

    if node_index == -1:
        return

    del linked_list[node_index]
```

## 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.

```
Key=61 hash: 700
Key=62 hash: 318
Key=63 hash: 936
Key=64 hash: 554
Key=65 hash: 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.

```

32 def ejercicio4_is_permutation(str1, str2):
33
34     """
35     La complejidad de la implementación es de O(n) siendo n la cantidad de caracteres
36     esto se debe a que iteramos 2 veces sobre todos los caracteres, realizando operaciones
37     de insert, contains y delete, las cuales, gracias a trabajar con Dictionary son O(1),
38     por lo tanto, la complejidad del algoritmo queda determinada por la complejidad de los bucles
39     """
40     # Con permutación se sabe que la cantidad de caracteres en el original y
41     # el resultado debe ser la misma
42     if len(str1) != len(str2):
43         return False
44
45     # Creo la función de hash de caracteres, utilizando el código ascii
46     m = ord('z') - ord('a')
47     hash_function = lambda char: ord(char) % m
48
49     # Creo un diccionario que contiene todos los caracteres de str1
50     d = Dictionary(hash_function, m)
51
52     # Inserta todos los caracteres, sin importar las repeticiones, pues
53     # la implementación de la tabla permite repeticiones
54     for char in str1:
55         d.insert(char, 1)
56
57     is_permutation = True
58     for char in str2:
59
60         # Si se encuentra el caracter
61         if char in d:
62             # Lo borro, por las repeticiones
63             d.delete(char)
64         else:
65             is_permutation = False
66             break
67
68     return is_permutation

```

**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.

```
76 def ejercicio5_unique_elements(linked_list: list):
77
78     """
79     La complejidad del algoritmo es O(n).
80     Primero cargo la lista en Dictionary, operación O(n)
81     Luego itero por cada uno de los elementos de la lista O(n),
82     pero por cada key, itero por la estructura de datos del Diccionario O(c),
83     donde c es el factor de carga, el factor de carga es menor que n, luego
84     la complejidad resulta O(n)
85     """
86
87     # La idea es cargar la HashTable con todos los elementos de la lista,
88     # luego, acceder a cada uno de los elementos de la lista y verificar
89     # si la key es unica en cada una de las sub-listas
90
91     m = len(linked_list)
92     a = (5**.5 - 1) * 0.5
93     hash_func = lambda x: int(m * ((x * a) % 1))
94     hash_table = Dictionary(hash_func, m)
95
96     for key in linked_list:
97         hash_table.insert(key, None)
98
99     for key in linked_list:
100         sub_list = hash_table.table[key]
101         found = False
102         # Cuenta la cantidad de repeticiones de la key
103         for node in sub_list:
104             if node[0] == key:
105                 # Si ya se encontraba en la lista entonces estaba repetida
106                 if found > 1:
107                     return False
108                 found = True
109
110     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.

```
122 def ejercicio6():
123     m = 1009
124     code = "a1016ldb"
125
126     def hash_postal(code):
127         # Trato de que el código hash de lugares cercanos sea similar, es por
128         # eso que considero:
129         # - El primer caracter como la ciudad
130         # - Los dddd como la altura
131         # - Los ccc como la calle
132         # Entonces debo evaluar primero los c y luego los d, de esta manera
133         # garantizo que los códigos postales de lugares cercanos sean cercanos
134         # Entonces la ciudad y la calle deben tener mas peso que la altura
135
136         val = ord(code[0]) * 10_000 # Ciudad
137         val += ord(code[5]) * 1_000 # Calle
138         val += ord(code[6]) * 100   # Calle
139         val += ord(code[7]) * 10    # Calle
140         val += int(code[1:5])       # Altura
141
142         return val % m
143
144     print(hash_postal(code))
```

La implementación utiliza el método de la división, junto a una selección especial de potencias de los caracteres

**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.

```
153 def ejercicio7_compresion(patron):
154
155     compressed = ""
156
157     char = patron[0]
158     count = 0
159
160     for i in range(len(patron)):
161
162         # Si nos encontramos con el mismo char
163         if char == patron[i]:
164             # Se incrementa su contador
165             count += 1
166
167         # Cuando nos encontramos con otro
168         else:
169             # Agregamos los resultados comprimidos
170             compressed += char + str(count)
171
172             # Iniciamos el contador con el otro char
173             char = patron[i]
174             count = 1
175
176     # Agregamos el final
177     compressed += char + str(count)
178
179     # Retorno el patron o el comprimido, selecciono el mas corto
180     return patron if len(compressed) >= len(patron) else compressed
```

**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.

```
190 def ejercicio8(text, pattern):
191
192     # La complejidad de este algoritmo es de  $O(n)$ , siendo
193     #  $n$  la cantidad de caracteres de 'text', pues solo itero
194     # una vez por la cadena 'text'
195
196     matches = 0
197     start_index = -1
198     i = 0
199
200     while i < len(text):
201
202         # Cuando se obtiene una coincidencia
203         if text[i] == pattern[matches]:
204
205             # Almacena el primer índice
206             if matches == 0:
207                 start_index = i
208
209             matches += 1
210             # Esto significa que el patron fue encontrado
211             if matches == len(pattern):
212                 return start_index
213
214             # Todavía no termina, sigue evaluando
215             i += 1
216         else:
217
218             # Cuando el anterior caracter coincide, pero este no
219             if matches > 0:
220                 matches = 0
221                 start_index = -1
222                 # No se incrementa el contador, para que el caracter
223                 # actual entre en el bucle de comparación y no se saltee
224
225             # Caso contrario, se incrementa normalmente
226             else:
227                 i += 1
```



**Ejercicio 9:** Considerar los conjuntos de enteros  $S = \{s_1, \dots, s_n\}$  y  $T = \{t_1, \dots, t_m\}$ . 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?

```

235 def ejercicio9(subset_list, set_list):
236
237
238     # En la implementación asumo que no hay repetición de elementos, es una de las propiedades
239     # de los conjuntos. La complejidad del algoritmo que propongo es de  $O(n)$ , siendo n la
240     # cantidad de elementos del conjunto
241
242     # El 'subconjunto' no puede tener mas elementos
243     if len(subset_list) > len(set_list):
244         return False
245
246     # Creo la función de hash con el módulo
247     m = 109
248     hash_func = lambda x: x % m
249     set_dictionary = Dictionary(hash_func, m)
250
251     # Cargo el conjunto en el diccionario
252     for item in set_list:
253         set_dictionary.insert(item, -1)
254
255     # Itero por los elementos del 'subconjunto'
256     for subset_item in subset_list:
257
258         # Si lo contiene
259         if set_dictionary.contains(subset_item):
260             # Lo elimino
261             set_dictionary.delete(subset_item)
262
263         # Si no lo contiene, entonces no es subconjunto
264         else:
265             return False
266
267     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

```
[22, 88, 'deleted-none', 'deleted-none', 4, 15, 28, 17, 59, 31, 10]
```

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

```
[22, 'deleted-none', 88, 17, 4, 'deleted-none', 28, 59, 15, 31, 10]
```

3. Double hashing con  $h1(k) = k$  y  $h2(k) = 1 + (k \bmod (m - 1))$

```
[22, 'deleted-none', 59, 17, 4, 15, 28, 88, 'deleted-none', 31, 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  $h(k) = k \bmod 10$  y exploración lineal (linear probing). ¿Cuál es la tabla hash resultante? Justifique.

La tabla hash resultante es la C, pues es la única que cumple con lo siguiente

- 1) De entrada descarto la opción D, pues resuelve colisiones por encadenamiento
- 2) Se inserta 12 en el índice 2, pues  $12 \% 10 = 2$  (Descarto A)
- 3) Se inserta 13 en el índice 3, pues  $13 \% 10 = 3$
- 4) Se inserta 2 en el índice 4, pues  $2 \% 10 = 2$ , pero ya está ocupado, voy al siguiente índice 3, el cuál está ocupado, voy al siguiente índice 4, no ocupado y guardo el valor

**Ejercicio 13:** Una tabla hash de longitud 10 utiliza direccionamiento abierto con función hash  $h(k)=k \bmod 10$ , 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

Supongamos que es la D.

$42 \% 10 = 2$ , correcto

$46 \% 10 = 6$ , correcto

$33 \% 10 = 3$ , incorrecto, previamente ocupado por 23

Supongamos que es la C.

$46 \% 10 = 6$ , correcto

$34 \% 10 = 4$ , correcto

$42 \% 10 = 2$ , correcto

$23 \% 10 = 3$ , correcto

$52 \% 10 = 2$ , pero está ocupado, por lo tanto se desplaza linealmente hasta el índice 3, 4, y finalmente el 5 que no está ocupado, correcto

$33 \% 10 = 3$ , ocupado, se desplaza 4, 5, 6 hasta el 7 que no está ocupado.

Por lo tanto se ha comprobado que la respuesta correcta es la C