Franco Yudica 13922

Algoritmos y estructuras de datos 2

TP Hash Table

Franco Yudica 13922

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

```python
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

```python
def delete(self, key):
    """Removes the element from the list"""
    # This implementation of Dictiorary 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
```

Franco Yudica 13922

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
def ejercicio4_is_permutation(str1, str2):

    """
    La complejidad de la implementación es de O(n) siendo n la cantidad de caracteres
    esto se debe a que iteramos 2 veces sobre todos los caracteres, realizando operaciones
    de insert, contains y delete, las cuales, gracias a trabajar con Dictionary son O(1),
    por lo tanto, la complejidad del algoritmo queda determinada por la complejidad de los bucles
    """
    # Con permutación se sabe que la cantidad de caracteres en el original y
    # el resutado debe ser la misma
    if len(str1) != len(str2):
        return False

    # Creo la función de hash de caracteres, utilizando el código ascii
    m = ord('z') - ord('a')
    hash_function = lambda char: ord(char) % m

    # Creo un diccionario que contiene todos los caracteres de str1
    d = Dictionary(hash_function, m)

    # Inserta todos los caracteres, sin importar las repeticiones, pues
    # la implementación de la tabla permite repeticiones
    for char in str1:
        d.insert(char, 1)

    is_permutation = True
    for char in str2:

        # Si se encuentra el caracter
        if char in d:
            # Lo borro, por las repeticiones
            d.delete(char)
        else:
            is_permutation = False
            break

    return is_permutation
```

Franco Yudica 13922

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
def ejercicio5_unique_elements(linked_list: list):

    """
    La complejidad del algoritmo es O(n).
    Primero cargo la lista en Dictiorary, operación O(n)
    Luego itero por cada uno de los elementos de la lista O(n),
    pero por cada key, itero por la estructura de datos del Diccionario O(c),
    donde c es el factor de carga, el factor de carga es menor que n, luego
    la complejidad resulta O(n)
    """

    # La idea es cargar la HashTable con todos los elementos de la lista,
    # luego, acceder a cada uno de los elementos de la lista y verificar
    # si la key es unica en cada una de las sub-listas

    m = len(linked_list)
    a = (5**.5 - 1) * 0.5
    hash_func = lambda x: int(m * ((x * a) % 1))
    hash_table = Dictionary(hash_func, m)

    for key in linked_list:
        hash_table.insert(key, None)

    for key in linked_list:
        sub_list = hash_table.table[key]
        found = False
        # Cuenta la cantidad de repeticiones de la key
        for node in sub_list:
            if node[0] == key:
                # Si ya se encontraba en la lista entonces estaba repetida
                if found > 1:
                    return False
                found = True

    return True
```

Franco Yudica 13922

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 ejercicio6():
    m = 1009
    code = "a1016idb"

    def hash_postal(code):
        # Trato de que el código hash de lugares cercanos sea similar, es por
        # eso que considero:
        # - El primer caracter como la ciudad
        # - Los dddd como la altura
        # - Los ccc como la calle
        # Entonces debo evaluar primero los c y luego los d, de esta manera
        # garantizo que los códigos postales de lugares cercanos sean cercanos
        # Entonces la cuidad y la calle deben tener mas peso que la altura

        val = ord(code[0]) * 10_000     # Ciudad
        val += ord(code[5]) * 1_000     # Calle
        val += ord(code[6]) * 100       # Calle
        val += ord(code[7]) * 10        # Calle
        val += int(code[1:5])           # Altura

        return val % m

    print(hash_postal(code))
```

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

Franco Yudica 13922

Ejercicio 7: Implemente un algoritmo para realizar la compresión básica de cadenas utilizando el recuento de caracteres repetidos. Por ejemplo, la cadena ‘*aabccccaaa’* 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 ejercicio7_compresion(patron):

    compressed = ""

    char = patron[0]
    count = 0

    for i in range(len(patron)):

        # Si nos encontramos con el mismo char
        if char == patron[i]:
            # Se incrementa su contador
            count += 1

        # Cuando nos encontramos con otro
        else:
            # Agregamos los resultados comprimidos
            compressed += char + str(count)

            # Iniciamos el contador con el otro char
            char = patron[i]
            count = 1

    # Agregamos el final
    compressed += char + str(count)

    # Retorno el patron o el comprimido, selecciono el mas corto
    return patron if len(compressed) >= len(patron) else compressed
```

Franco Yudica 13922

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
def ejercicio8(text, pattern):

    # La complejidad de este algoritmo es de O(n), siendo
    # n la cantidad de caracteres de 'text', pues solo itero
    # una vez por la cadena 'text'

    matches = 0
    start_index = -1
    i = 0

    while i < len(text):

        # Cuando se obtiene una coincidencia
        if text[i] == pattern[matches]:

            # Almacena el primer índice
            if matches == 0:
                start_index = i

            matches += 1
            # Esto significa que el patron fue encontrado
            if matches == len(pattern):
                return start_index

            # Todavia no termina, siguie evaluando
            i += 1
        else:

            # Cuando el anterior caracter coincide, pero este no
            if matches > 0:
                matches = 0
                start_index = -1
                # No se incrementa el contador, para que el caracter
                # actual entre en el bucle de comparación y no se saltee

            # Caso contrario, se incrementa normalmemte
            else:
                i += 1
```

Franco Yudica 13922

Alternativamente, hice la siguiente implementación. El código resulta más fácil de leer, y la idea es mucho más clara, tiene una complejidad $O(n)$, pues se cargan casi n patrones en el diccionario. El problema es la complejidad espacial, pues hay que crear una estructura de datos extra para poder buscar el patrón, mientras que en la otra implementación no hace falta, pues solo itero por las cadenas originales.
Para la función hash, solo considero el código ascii de los primeros 3 caracteres del patrón, nótese que el patrón puede tener una longitud menor a 3, y es por esa misma razón que considero el mínimo de 3 y la longitud de la cadena

```python
def ejercicio8_hashtable(text, pattern):
    # Complejidad O(n)
    # La idea de este método, es ir agregando
    # las sub-cadenas del texto de la misma longitud
    # que las de pattern, de manera secuencial.

    # Hash function toma los primeros tres caracteres del patron p
    # (en caso de que tenga esa cantidad de caracteres, por eso uso min)
    # nótese que también se multiplican por potencias de 10
    m = 17
    hash_function = lambda p: sum([ord(p[i]) * 10 ** i for i in range(min(3, len(p)))]) % m

    dictionary = Dictionary(hash_function, m)
    pattern_length = len(pattern)

    for i in range(0, len(text) - pattern_length + 1):

        # Creo el patron de la misma longitud que pattern
        sub_pattern = text[i : i + pattern_length]

        # Lo inserto en el diccionario con su respectivo índice
        dictionary.insert(sub_pattern, i)

    # Se se encuentra, retorno el índice, de otra manera None
    return dictionary[pattern] if pattern in dictionary else None
```

Franco Yudica 13922

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?

```
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_item) > 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
```

Franco Yudica 13922

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 mod (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 mod 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 12 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

Franco Yudica 13922

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

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