UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

Algoritmos y estructuras de datos II
Hash Tables

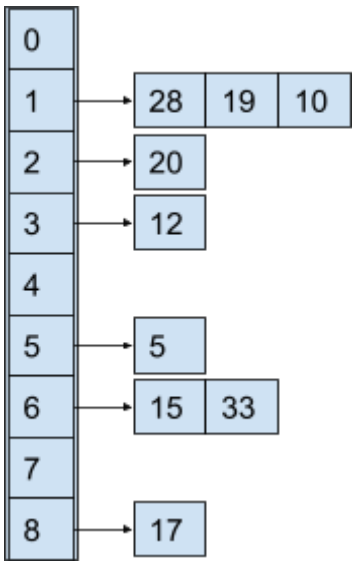
PARTE 1

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:

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

```
12  def exercise1():
13      print('Exercise 1')
14      hash_function1 = lambda k : k % 9
15      hash_table = dictionary.Dictionary(9,hash_function1)
16      keys_set = [5,28,19,15,20,33,12,17,10]
17      for key in keys_set:
18          hash_table.insert(key,'')
19      print(hash_table.table)
20      print('')
```



UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

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

`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.

`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**

```python
class Dictionary:
    def __init__(self,slots,hash_function):
        # slots equivale a 'm' en la teoria
        self.slots = slots
        self.hash_function = hash_function
        self.table = [[] for _ in range(slots)]

    def insert(self,key,value):
        table_slot = self.hash_function(key)
        self.table[table_slot].append((key,value))
        return self.table

    def search(self,key):
        table_slot = self.hash_function(key)
        for tuple in self.table[table_slot]:
            if tuple[0] == key:
                return tuple[1]
        return

    def delete(self,key):
        table_slot = self.hash_function(key)
        for tuple in self.table[table_slot]:
            if tuple[0] == key:
                self.table[table_slot].remove(tuple)

```

UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

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.

```python
def exercise3():
    print('Exercise 3')
    A = (math.sqrt(5)-1) / 2
    hash_function3 = lambda k: math.floor(1000*(k*A % 1))
    keys_set3 = [61,62,63,64,65]

    for key in keys_set3:
        print('h(%d):' %key,hash_function3(key))
    print('')
```

```
Exercise 3
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.

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 is_permutation(string1,string2):
    if len(string1) != len(string2) or string1 == string2:
        return False

    str_length = len(string1)
    A = (math.sqrt(5)-1) / 2
    hash_function = lambda k: math.floor(str_length*(k*A % 1))
    hash_table = dictionary.Dictionary(str_length,hash_function)

    for char in string1:
        hash_table.insert(ord(char),char)

    for char in string2:
        if hash_table.search(ord(char)) == None:
            return False

    return True
```

UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

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

```
95  def has_unique_elements(list):
96      list_length = len(list)
97
98      A = (math.sqrt(5)-1) / 2
99      hash_function = lambda k: math.floor(list_length*(k*A % 1))
100     hash_table = dictionary.Dictionary(list_length,hash_function)
101
102     for i in range(len(list)):
103         position = 'position: %d' %(i+1)
104         hash_table.insert(list[i],position)
105         if len(hash_table.table[i]) > 1:
106             return 'Falso, la lista no tiene todos sus elementos unicos',hash_table.table[i]
107     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.

```
118 def postal_code_hash_function(code,m):
119     # Codigo postal argentino
120     province_id = code[0]
121     territorial_subdivision_id = code[1:5]
122     city_block_side = code[5:]
123
124     # Construyo el key numerico
125     code_key = ord(province_id) * 10**4
126     code_key += int(territorial_subdivision_id)
127
128     exp = 2
129     for i in range(len(city_block_side)):
130         code_key += (ord(city_block_side[i]) - ord('A')) * 10**exp
131         exp -= 1
132
133     return code_key % m
```

UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737

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 compress_string(s):
    if not s:
        return s  # Si la cadena está vacía, se devuelve tal cual

    compressed = []
    count = 1  # Contador para llevar el recuento de caracteres repetidos
    prev_char = s[0]  # Variable para almacenar el caracter previo inicialmente con el primer caracter de la cadena

    for i in range(1, len(s)):
        if s[i] == prev_char:
            count += 1
        else:
            compressed.append(prev_char + str(count))
            prev_char = s[i]
            count = 1

    # Se agrega el último caracter y su contador a la lista comprimida después de finalizar el bucle
    compressed.append(prev_char + str(count))

    # Se crea la cadena comprimida uniendo los elementos de la lista comprimida
    compressed_str = ''.join(compressed)

    # Se compara la longitud de la cadena original y la cadena comprimida y se devuelve la cadena original si es más corta
    if len(compressed_str) >= len(s):
        return s
    else:
        return compressed_str
```

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
# Implementacion del algoritmo de Knuth-Morris-Prat para string matching

def kmp_matcher(pattern, text):
    m = len(pattern)
    n = len(text)
    prefix = prefix_table(pattern, m)
    # Searching
    i = 0
    j = 0
    while i < n:
        if text[i] == pattern[j]:
            if j == m - 1:
                return i - m + 1
            j += 1
            i += 1
        else:
            if j > 0:
                j = prefix[j - 1]
            else:
                i += 1
    return -1
```

```python
def prefix_table(pattern, pattern_lenght):
    prefix = [0] * pattern_lenght
    j = 0
    for i in range(1, pattern_lenght):
        if pattern[i] == pattern[j]:
            prefix[i] = j + 1
            j += 1
        else:
            j = 0
    return prefix

# Complejidad temporal = O(m+n)
```

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

```python
# S y T son sets, es decir no tienen elementos repetidos.

def is_subset(S,T):
    if len(S) > len(T):
        return False

    m = len(T)

    hash_function = lambda k: k % m
    hash_tableT = dictionary.Dictionary(m,hash_function)

    for num in T:
        hash_tableT.insert(num,num)

    print(hash_tableT.table)
    for s in S:
        if hash_tableT.search(s) != s:
            return False
    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 mod (m - 1))*

```python
# Ejercicio 10

keys_list = [10,22,31,4,15,28,17,88,59]

def hashing_linear_probing(keys_list,m):
    hash_function = lambda k,i: (k + i) % m
    hash_table = Dictionary(m,hash_function)

    for key in keys_list:
        hash_table.insert(key)
    print('Linear probing hashing:',hash_table.table)

hashing_linear_probing(keys_list,11)
# Resultado = [22, 88, None, None, 4, 15, 28, 17, 59, 31, 10]

def hashing_cuadratic_probing(keys_list,m):
    c1,c2 = 1,3
    hash_function = lambda k,i: (k + (c1*i) + (c2*(i**2))) % m
    hash_table = Dictionary(m,hash_function)

    for key in keys_list:
        hash_table.insert(key)
    print('Cuadratic probing hashing:',hash_table.table)

hashing_cuadratic_probing(keys_list,11)
# Resultado = [22, None, 88, 17, 4, None, 28, 59, 15, 31, 10]
```

```python
def double_hashing(keys_list,m):
    hk1 = lambda k: k
    hk2 = lambda k: 1 + (k % (m-1))
    hash_function = lambda k,i: (hk1(k) + i*hk2(k)) % m
    hash_table = Dictionary(m,hash_function)

    for key in keys_list:
        hash_table.insert(key)
    print('Double hashing:',hash_table.table)

double_hashing(keys_list,11)
# Resultado = [22, None, 59, 17, 4, 15, 28, 88, 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.

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)

```python
def ejercicio12():
    keys_list = [12,18,13,2,3,23,5,15]
    hashing_linear_probing(keys_list,10)

ejercicio12()
```

```
Linear probing hashing: [None, None, 12, 13, 2, 3, 23, 5, 18, 15]
```

La tabla resultante es la C

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

```python
def ejercicio13():
    keys_list = [46,34,42,23,52,33]
    hashing_linear_probing(keys_list,10)
```

```
Linear probing hashing: [None, None, 42, 23, 34, 52, 46, 33, None, None]
```

La respuesta correcta es la C

UNCUYO - Licenciatura en ciencias de la computación
Augusto Robles 11737