Universidad Nacional de Cuyo
Facultad de Ingeniería
Licenciatura en Ciencias de la Computación

Trabajo Práctico Nº 4

Algoritmos y Estructuras de Datos

Hash Tables

2024

Gonzalo Padilla Lumelli

Mayo 2024

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:

$$H(k) = k \text{ mód } 9 \tag{1}$$

Solución

Para cada llave se obtienen las siguientes posiciones en la tabla a partir de la función hash:

1. $H(5) = 5$
2. $H(28) = 1$
3. $H(19) = 1$ (Colisión)
4. $H(15) = 6$
5. $H(20) = 2$
6. $H(33) = 6$ (Colisión)
7. $H(12) = 3$
8. $H(17) = 8$
9. $H(10) = 1$ (Colisión)

Como utilizamos el método de chaining, cuando se produce una colisión, encadenamos la key a la última key en esa posición, utilizando una lista enlazada. La tabla resultante quedaría:

i	Keys
0	
1	$28 \rightarrow 19 \rightarrow 10$
2	20
3	12
4	
5	5
6	$15 \rightarrow 33$
7	
8	17

Ejercicio 2

A partir de una definición de diccionario como la siguiente:

dictionary = Array(m,0) # una sugerencia de implementación, se puede usar una lista de python.

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

Nota: 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 None 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.

Solución

```python
def insert(d, key, value):
    tableIndex = hash_function(key, len(d))  # Calculamos índice
    lista = d[tableIndex]           # Obtenemos la lista en ese índice

    # Si no existe una lista, creamos una conteniendo al par (key, value)
    if not lista:
        d[tableIndex] = [(key, value)]
        return d

    # Si existe, vemos si ya está la key en ella
    # En tal caso, actualizamos el valor asociado
    for i, (k, v) in enumerate(lista):
        if k == key:
            lista[i] = (key, value)
            return d

    # Si no existe la key en la lista, agregamos el par (key, value)
    lista.append((key, value))
    return d
```

```python
def search(d, key):
    tableIndex = hash_function(key, len(d))  # Calculamos índice en la tabla
    lista = d[tableIndex]           # Obtenemos la lista en ese índice

    if not lista:                   # Si no existe, no está el elemento
        return None

    # Si existe una lista, buscamos la key en ella
    for i, (k, v) in enumerate(lista):
        if k == key:                # Si existe en la lista,
            return v                # retornamos el valor asociado

    return None                     # Si no existe en la lista, retornamos None
```

```python
def delete(d, key):
    tableIndex = hash_function(key, len(d))  # Calculamos índice en la tabla
    lista = d[tableIndex]                    # Obtenemos la lista en ese índice

    if not lista:                            # Si no existe, no está el elemento
        return d

    # Si existe una lista, buscamos la key en ella
    for i, (k, v) in enumerate(lista):
        if k == key:                         # Si existe en la lista,
            lista.pop(i)                     # la sacamos de la lista
            if len(lista) == 0:              # Si quedó vacía, la cambiamos por None
                d[tableIndex] = None
            break

    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.

Solución

Se obtienen las siguientes ubicaciones:

- $H(61) = \lfloor m\ (61\ (\sqrt{5} - 1)/2 \bmod 1) \rfloor = 700$
- $H(62) = \lfloor m\ (62\ (\sqrt{5} - 1)/2 \bmod 1) \rfloor = 318$
- $H(63) = \lfloor m\ (63\ (\sqrt{5} - 1)/2 \bmod 1) \rfloor = 936$
- $H(64) = \lfloor m\ (64\ (\sqrt{5} - 1)/2 \bmod 1) \rfloor = 554$
- $H(65) = \lfloor m\ (65\ (\sqrt{5} - 1)/2 \bmod 1) \rfloor = 172$

Ejercicio 4

Implemente un algoritmo lo más eficiente posible que devuelva True o False a la siguiente proposición: dado dos strings $s_1 \dots s_k$ y $p_1 \dots p_k$, se quiere encontrar si los caracteres de $p_1 \dots p_k$ corresponden a una permutación de $s_1 \dots s_k$. Justificar el costo en tiempo de la solución propuesta.

Solución

```python
def isPermutation(stringS, stringP):
    if not stringS or not stringP:
        return False

    # Si son de distinto largo, no es permutación
    if len(stringS) != len(stringP):
        return False

    stringS = stringS.lower()
    stringP = stringP.lower()

    # Contamos los caracteres de cada palabra
    charsCountsS = dict()
    charsCountsP = dict()
    for i in range(len(stringS)):
        if stringS[i] in charsCountsS:
            charsCountsS[stringS[i]] += 1
        else:
            charsCountsS[stringS[i]] = 1
        if stringP[i] in charsCountsP:
            charsCountsP[stringP[i]] += 1
        else:
            charsCountsP[stringP[i]] = 1

    # Verificamos que coincidan
    for c in stringS:
        if c not in charsCountsP or charsCountsP[c] != charsCountsS[c]:
            return False

    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 costo en tiempo de la solución propuesta.

Solución

```python
def noRepeatedElements(lista):
    if lista is None:
        return False

    dictionary = dict()         # Guardamos cada elemento en un diccionario

    for item in lista:
        if item in dictionary:  # Si ya hay un elemento igual en el diccionario
            return False        # La lista tiene elementos repetidos
        dictionary[item] = item # Si no, lo añadimos al diccionario

    return True                 # Retornamos True si no hubo repeticiones
```

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.

Solución

```python
def postalCodeHash(code):
    code = code.upper()

    # Cantidad de valores posibles por posición del código postal
    ranges = [26, 10, 10, 10, 10, 26, 26, 26]
    currentBase = 1 # Base con la cual multiplicar a un caracter
    codeHash = 0    # Hash que se está calculando

    # Recorremos el código postal de derecha a izquierda
    # Multiplicamos al valor (mapeado de 0 a 25 para las letras)
    # por la base actual y lo sumamos al hash. Luego actualizamos la base.
    for i in range(len(code) - 1, -1, -1):
        if code[i].isalpha():
            codeHash += (ord(code[i]) - 65 ) * currentBase
        else:
            codeHash += int(code[i]) * currentBase
        currentBase *= ranges[i]

    return codeHash
```

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 costo en tiempo de la solución propuesta.

Solución

```python
def compress_string(string):
    if string is None:
        return None

    if not string:
        return ""

    currentChar = string[0]
    charCount = 1
    newStringArray = [''] * len(string)
    arrayIndex = 0

    for i in range(1, len(string)):
        if string[i] == currentChar:
            charCount += 1
        else:
            newStringArray[arrayIndex] = currentChar
            newStringArray[arrayIndex + 1] = str(charCount)

            arrayIndex += 2
            charCount = 1
            currentChar = string[i]

            if len(string) - arrayIndex < 3:
                return string

    newStringArray[arrayIndex] = currentChar
    newStringArray[arrayIndex + 1] = str(charCount)

    return ''.join(newStringArray[:arrayIndex + 2])
```

Ejercicio 8

Se requiere encontrar la primera ocurrencia de un string $p_1 \dots p_k$ en uno más largo $a_1 \dots a_L$. Implementar esta estrategia de la forma más eficiente posible con un costo computacional menor a $O(K \cdot L)$ (solución por fuerza bruta). Justificar el coste en tiempo de la solución propuesta.

Solución

```python
def find_index(s, p):
    if not s or not p or len(s) > len(p):
        return None

    s = s.lower()
    p = p.lower()

    lettersInS = dict()

    uniqueLetters = 0
    for letter in s:
        if letter not in lettersInS:
            lettersInS[letter] = uniqueLetters
            uniqueLetters += 1

    radix = uniqueLetters
    largestBase = radix ** (len(s) - 1)
    hashS = 0
    hash = 0
    correctLettersInHash = 0

    for letter in s:
        hashS *= radix
        hashS += lettersInS[letter]

    for i in range(len(s)):
        hash *= radix
        letter = p[i]
        if letter in lettersInS:
            correctLettersInHash += 1
            hash += lettersInS[letter]

    if correctLettersInHash == len(s) and hashS == hash:
        return 0

    j = 0
    for i in range(len(s), len(p)):
        lastLetter = p[j]
        nextLetter = p[i]

        if lastLetter in lettersInS:
            correctLettersInHash += -1
            hash += - lettersInS[lastLetter] * largestBase

        hash *= radix

        if nextLetter in lettersInS:
            correctLettersInHash += +1
            hash += lettersInS[nextLetter]

        if correctLettersInHash == len(s) and hashS == hash:
            return j + 1

        j += 1

    return None
```

Ejercicio 9

Considerar los conjuntos de enteros $S = s_1, \ldots, s_n$ y $T = t_1, \ldots, t_m$. 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?

Solución

```python
def is_subset(s, p):
    pDictionary = dict()
    for element in p:
        pDictionary[element] = 1

    for element in s:
        if not pDictionary[element]:
            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 $c_1 = 1$ y $c_2 = 3$.
3. Double hashing con $h_1(k) = k$ y $h_2(k) = 1 + (k \text{ mód } (m-1))$.

Solución

i	Linear	Quadratic	Double Hashing
0	22	22	22
1	88		
2		88	59
3		17	17
4	4	4	4
5	15		15
6	28	28	28
7	17	59	88
8	59	15	
9	31	31	31
10	10	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 $h(k) = k$ mód 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)

Solución

Para cada llave se obtienen las siguientes posiciones en la tabla a partir de la función hash:

1. $h(12, 0) = 2$
2. $h(18, 0) = 8$
3. $h(13, 0) = 3$
4. $h(2, 0) = 2$ (Colisión)
5. $h(2, 1) = 3$ (Colisión)
6. $h(2, 2) = 4$
7. $h(3, 0) = 3$ (Colisión)
8. $h(3, 1) = 4$ (Colisión)
9. $h(3, 2) = 5$
10. $h(23, 0) = 3$ (Colisión)
11. $h(23, 1) = 4$ (Colisión)
12. $h(23, 2) = 5$ (Colisión)
13. $h(23, 3) = 6$
14. $h(5, 0) = 5$ (Colisión)
15. $h(5, 1) = 6$ (Colisión)
16. $h(5, 2) = 7$
17. $h(15, 0) = 5$ (Colisión)
18. $h(15, 1) = 6$ (Colisión)
19. $h(15, 2) = 7$ (Colisión)
20. $h(15, 3) = 8$ (Colisión)
21. $h(15, 4) = 9$

Lo cual coincide con la tabla (C).

Ejercicio 13

Una tabla hash de longitud 10 utiliza direccionamiento abierto con función hash $h(k) = k$ mód 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

Solución

(A) No es un orden posible, ya que el 52, con $h(52) = 2$, debería ocupar la posición 3 (hasta ahora libre), al estar ocupada la posición 2 por el 42 insertado anteriormente.

(B) No es un orden posible, ya que el 33, con $h(33) = 3$, debería ocupar la posición 6 (hasta ahora libre), al estar ocupada la posición 3 por el 23, y las siguientes por 34 y 52.

(C) Es un orden posible, ya que las únicas colisiones son producidas por el 52, con $h(52) = 2$, que termina ocupando la posición 5 al estar ocupadas desde las posiciones desde la 2 hasta la 4; y el 33, con $h(33) = 3$, que termina ocupando la posición 7 al estar ocupadas las posiciones desde la 3 hasta la 6.

(D) No es un orden posible, ya que el 33, con $h(33) = 3$, termina ocupando la posición 3, ya que no está ocupada, en lugar de la 7.