Programación 2

Tecnicatura Universitaria en Inteligencia Artificial

2022

Unidad 2

1. Introducción

Muchas aplicaciones requieren trabajar con **diccionarios**. Un diccionario es conjunto dinámico que admite las operaciones : **insertar** (`insert`), **buscar** (`search`) y **eliminar** (`delete`). Cada elemento del diccionario contiene una clave, la cual permite identificarlo.

Como ejemplo de un diccionario podemos nombrar la base de datos de un hospital, que permite recuperar los datos de un paciente a partir de su número de documento.

¿Cuál es la mejor forma de implementar un diccionario? Consideremos que las claves se encuentran en un universo U y analicemos.

2. Tabla de Direccionamiento Directo

Supongamos que una aplicación necesita un conjunto dinámico en el cual cada elemento posee una clave dentro de un universo $U = \{0, 1, \cdots, m-1\}$ donde m no es muy grande y las claves son todas diferentes. Para representar el conjunto dinámico utilizamos una tabla de direccionamiento directo (`DirectAccessTable`) $T[0, 1, \cdots, m-1]$ en la cual cada posición, o *slot*, corresponde a una clave en el universo U.

Una implementación en `Python` de esta tabla utilizando lista podría ser la siguiente:

```
class DirectAccessTable():
    def __init__(self, capacity):
        self.m = capacity
        self.T = [None] * self.m

    def insert(self, key, element):
        self.T[key] = element

    def search(self, key):
        return self.T[key]

    def delete(self, key):
        self.T[key] = None
```

La Figura 1 ilustra este enfoque. La posición k contiene un elemento del conjunto con clave k. Si el conjunto no contiene ningún elemento con clave k, entonces $T[k]$ está vacío, lo cual lo representamos indicando que contiene `None`.

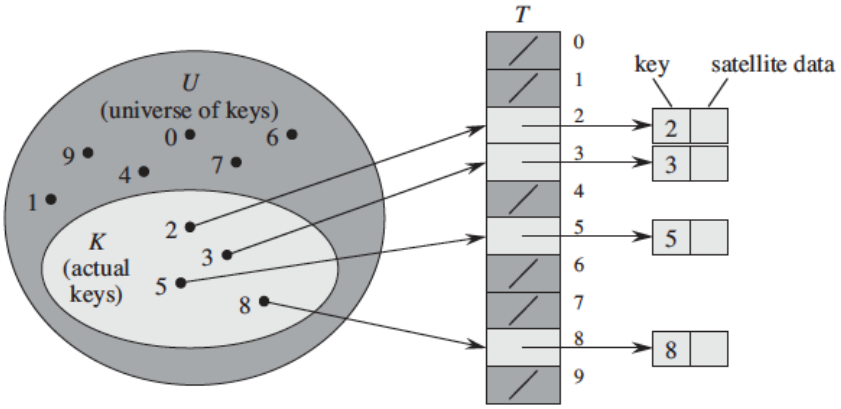

Figura 1: Tabla de Direccionamiento Directo

En este tipo de implementación, cada una de las operaciones de diccionario se ejecutan en tiempo constante. En cuanto al espacio de memoria ocupado es del orden del tamaño del universo U.

Respecto a las desventajas de este método, podemos decir que si el tamaño de U es grande puede ser poco práctico, o incluso imposible implementarlo, dada la memoria disponible en una computadora típica. Y además, si la cantidad de elementos del diccionario es mucho más pequeño que el tamaño de U, se desperdicia mucho espacio.

3. Tabla Hash (Hash Table)

Cuando el conjunto de claves almacenadas en un diccionario es mucho más pequeño que el universo U de todas las claves posibles, una **Tabla Hash** (`Hash Table`) requiere mucho menos espacio de almacenamiento que una Tabla de Direccionamiento Directo.

Este tipo de tabla reduce la cantidad de memoria necesaria a un orden similar al tamaño del diccionario. Se reduce así el requisito de almacenamiento mientras se mantiene el costo promedio de búsqueda de un elemento en un tiempo constante. Pero, ¿en qué consiste una Tabla Hash?

Para implementar una **Tabla Hash** debemos definir una tabla de tamaño m $T[0, 1, \cdots, m-1]$, donde el tamaño m es menor que el tamaño del universo U (o sea, $m < |U|$). También es necesario definir una función h, la cual denominamos **función hash** h, que calcula para cada clave k la posición o *slot* de la tabla que le corresponde. Se define así $h : U \to \{0, \cdots, m-1\}$. El elemento con clave k se almacena en $T[h(k)]$. La Figura 2 ilustra este enfoque.

Observando la Figura 2 podemos detectar un problema: dos claves, en este caso k_2 y k_5 mapean a la misma posición de la tabla, ya que $h(k_2) = h(k_5)$. A esta situación se la denomina **colisión**. Afortunadamente, contamos con técnicas efectivas para resolver colisiones.

Por supuesto, la solución ideal sería evitar las colisiones por completo. Podríamos intentar lograr este objetivo eligiendo una función hash h adecuada. Una idea sería lograr que h asigne posiciones "aleatorias", evitando así colisiones o al menos minimizando su número.

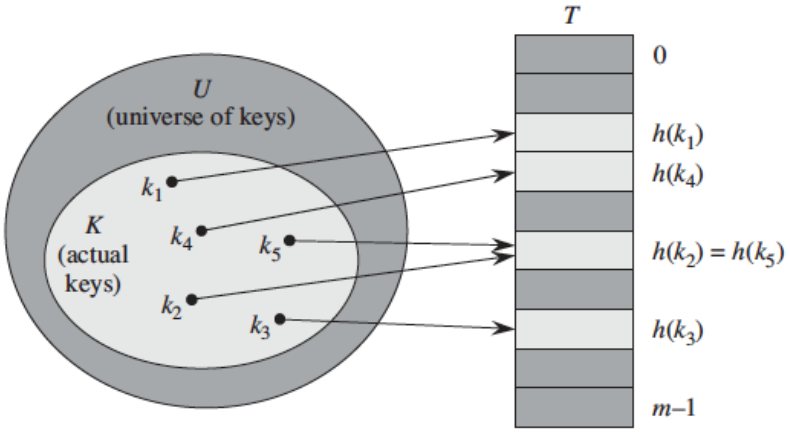

Figura 2: Uso de una función hash h para mapear las claves a las diferentes posiciones o *slots* de la tabla

Una primera implementación en `Python` de una tabla hash podemos plantearla como sigue:

```
class HashTable():
    def __init__(self, capacity, hashFunction):
        self.m = capacity
        self.h = hashFunction
        self.T = [None] * self.m

    def insert(self, key, element):
        pass

    def search(self, key):
        pass

    def delete(self, key):
        pass
```

3.1. Funciones Hash

3.1.1. ¿Qué hace que una función hash sea considerada buena?

Una buena función hash:

- Debe poder calcularse en tiempo constante.
- Satisfacer la hipótesis de hashing uniforme: es equiprobable que una clave dada tenga cualquier valor hash entre 0 y $m - 1$

Algunas funciones hash asumen que el universo de las claves es el conjunto $\mathbb{N}$ de números naturales. De esta forma, si las claves no son números naturales debemos encontrar la forma de interpretarlas como si lo fueran.

Por ejemplo, dada una cadena de caracteres c, podemos transformar la cadena usando la siguiente función:

$$stringtonat(c) = \Sigma_{i=0}^{length(c)-1} c_i \cdot B^i$$

donde B es la base del conjunto de caracteres.

3.1.2. Algunas funciones hash "conocidas"

Método del Resto

El método del resto propone la siguiente función hash:

$$h(k) = k \ mod \ m$$

Con respecto a este método:

- Debemos tener cuidado con el valor de m
- Funciona mal con valores $m = 2^p$, $h(k)$ estará dado por los primeros p bits de k, sin tener en cuenta los bits de orden superior.
- Funciona bien con m primos

Método de Multiplicación

Una función hash alternativa es la que propone el método de la multiplicación:

$$h(k) = \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$$

donde A es alguna constante en el rango $(0, 1)$.

Otras funciones hash

Existen otras funciones hash en la literatura. Investigue sobre otras funciones hash que podamos utilizar.

A continuación presentaremos algunas técnicas de resolución de colisiones.

3.2. Resolución de colisiones

3.2.1. Por encadenamiento

Cuando se resuelven las colisiones por encadenamiento, colocamos todos los elementos que mapean a una misma posición de la tabla hash en una lista enlazada, como muestra la Figura 3. La posición j referencia al primer elemento de la lista de todos los elementos almacenados para los cuáles la función h asigna el valor j; si no existen tales elementos, la posición j contiene **None**.

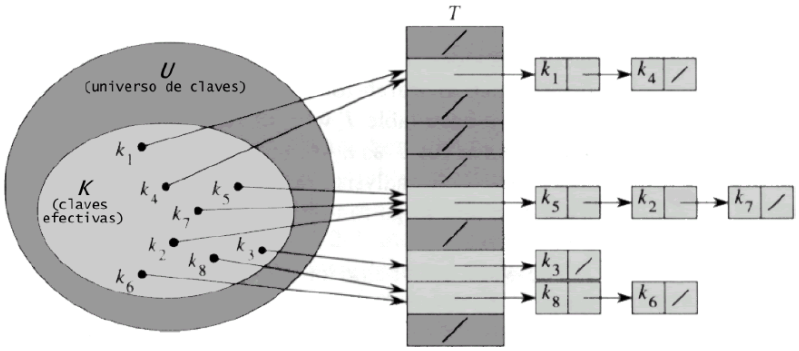

Figura 3: Tabla Hash - Resolución de colisiones por Encadenamiento

Podemos pensar los elementos de las listas enlazadas de tipo `Node2`

```
class Node2:
```

```
    def __init__(self, key=None, element=None, next=None):
        self.key = key
        self.element = element
        self.next = next
```

Y pensar en implementar una **Tabla Hash encadenada** (`HastTableChaining`) en `Python` de la siguiente manera:

```
class HastTableChaining(HashTable):

    def insert(self, key, element):
        node = Node2(key, element)
        index = self.h(key)
        node.next = self.T[index]
        self.T[index] = node

    def search(self, key):
        # search for an element with key 'key' in list T[h(key)]
        # complete this code


    def delete(self, key):
        # delete element with key key from list T[h(key)]
        # complete this code
```

Un `insert` se ejecuta siempre en tiempo constante, mientras que `search` y `delete` llevan un tiempo proporcional al tamaño de la lista en la posición correspondiente en el peor caso.

3.2.2. Por direccionamiento abierto

Ciertas aplicaciones, para ahorrar el espacio desperdiciado usado para referenciar el siguiente elemento en una lista enlazada, prefieren almacenar todos los elementos directamente en la propia tabla. Esto se puede lograr mediante el método llamado **direccionamiento abierto**.

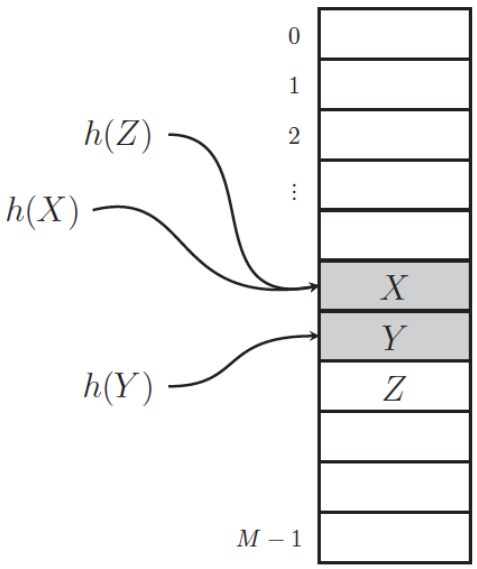

Figura 4: Tabla Hash - Resolución de colisiones por Direccionamiento Abierto

La función h devuelve un valor entre 0 y $m-1$ que se puede utilizar como un índice en la tabla hash T. Si calculamos $h(X)$ para una clave X y el lugar se encuentra vacío, el elemento de clave X se almacenará en T en el índice $h(X)$. En caso de que h asigne el mismo lugar a otro elemento con clave Z, o sea, ocurra que $h(Z) = h(X)$, estamos ante un caso de colisión. En el caso del direccionamiento directo, no se podrá almacenar en dicha posición. Lo que plantea el direccionamiento abierto es que el elemento con clave Z se ubique, por ejemplo, en la posición libre más cercana, saltando sobre un bloque de posiciones ocupadas. En el ejemplo de la Figura 4, el siguiente lugar también está ocupado, por otro elemento con clave Y, por lo cual debe almacenarse en el siguiente.

Si en la búsqueda de una posición vacía se alcanza la parte inferior de la tabla, se considera a la tabla como cíclica y se continúa la búsqueda desde la posición 0. Esto se implementa fácilmente calculando la siguiente posición de la tabla a la posición i de la siguiente manera: $(i+1)\ mod\ m$.

Esto funciona bien siempre que sólo se use una pequeña parte de la tabla. Cuando la tabla comience a llenarse, las celdas ocupadas tenderán a formar *clusters* (grupos). El problema es que cuanto mayor sea un cluster, mayor será la probabilidad de un nuevo elemento que caiga dentro de las posiciones ocupadas por el mismo, por lo cual será mayor la probabilidad de que dicho cluster se vuelva aún más grande: esto contradice la hipótesis la distribución uniforme de las claves.

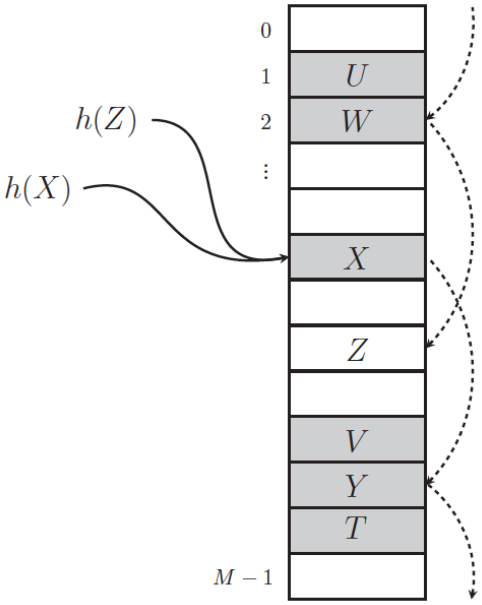

Figura 5: Tabla Hash - Resolución de colisiones por Direccionamiento Abierto con salto de tamaño k.

Por lo tanto, una mejor idea parecería ser la de buscar una celda vacía en saltos de tamaño k, para algún número fijo $k > 1$. Si k se elige de manera tal que el máximo común divisor entre k y m, $mcd(k, m)$, sea 1, entonces esta política de salto nos llevará de regreso al punto de partida sólo después de haber visitado todas las posiciones de la tabla. Este es un incentivo para elegir m como número primo, ya que cualquier tamaño de salto k logra el objetivo. En el ejemplo de la Figura 5, los valores usados son $m = 13$ y $k = 5$. Como $h(Z) = 5$ y dicha posición está ocupada por otro elemento de clave X, el índice actual se incrementa repetidamente en k. La secuencia de posiciones generada, llamada secuencia de sondeo, se representan a través de las flechas, y corresponden a las posicones 5, 10, 15 $mod\ m = 2$, y finalmente 7. Este es la primera ubicación vacía en el orden impuesto por la secuencia de sondeo, por lo que el elemento con clave Z será almacenado allí.

Lo cierto es que este último cambio es sólo cosmético: seguirá habiendo clusters como antes, solo podrían ser más difíciles de detectar.

Para evitar estos clusters, se puede reemplazar el salto de tamaño fijo k utilizado en el caso de colisión, por otro salto cuyo tamaño dependerá también de la clave a insertar. Esto sugiere usar **dos funciones hash independientes** $h_1(X)$ y $h_2(X)$, por eso el **método se llama doble hash**. En un primer intento se intentará almacenar el elemento con clave X en la dirección:

$$l \leftarrow h_1(X)$$

si dicha posición se encuentra ocupada, el tamaño de salto se define por:

$$k \leftarrow h_2(X)$$

de manera tal que $1 \leq k \leq m$.

La secuencia de sondeo para almacenar el elemento de clave X será:

$$(l + k)\ mod\ m,\ (l + 2k)\ mod\ m,\ (l + 3k)\ mod\ m,\ \ldots$$

hasta encontrar una posición vacía.

¿Cuál sería una buena opción para esta segunda función hash h_2?

Una sugerencia sería usar la siguiente función h_2:

$$h_2(X) = 1 + X\ mod\ m'$$

donde m' es el mayor primo menor que m.

4. Factor de Carga y Re-hashing

El factor de carga α es una estadística crítica de una tabla hash y se define de la siguiente manera:

$$\alpha = \frac{n}{m}$$

donde:

- n corresponde a la cantidad de elementos guardados en la tabla
- m corresponda al tamaño de la tabla.

La "performance" de la tabla hash se deteriora en relación con el factor de carga α. Por lo tanto, se realiza un re-hash de una tabla hash o se redimensiona si el factor de carga α se aproxima a 1. Una tabla también se redimensiona si el factor de carga cae por debajo de cierta cifra. Las cifras aceptables del factor de carga α se encuentran en entre 0,6 y 0,75.

Investigue cómo se realiza el re-hashing de una tabla.

Referencias

[1] Cormen T. H. et al, 2009. *Introduction to Algorithms*. 3era Edición. The MIT Press. Capítulo 11

Basic Concepts in Data Structures. Bar-Ilan University, Israel. Capítulo 9

[2] Cormen T. H. et al, *Introduction to Algorithms*, 3era Edición. The MIT Press, 2009. Capítulo 11.

[3] Shmuel Tomi Klein, *Basic Concepts in Data Structures*, Cambriadge University Press. Capítulo 9.