Programación II

Tecnicatura Universitaria en Inteligencia Artificial

2023

Diccionarios

1. Introducción

Muchas aplicaciones requieren trabajar con tres operaciones básicas: insertar, buscar, y eliminar. Un ejemplo muy simple de esto puede ser una agenda telefónica en la que el usuario encuentra números de teléfono (que no recuerda porque son muy largos) a partir del nombre de pila de la persona a quien quiere llamar. En estos casos el usuario puede añadir nuevos contactos ingresando el nombre con el que lo identificará y luego el número de teléfono a guardar. De un modo similar también puede borrar contactos existentes.

Otro ejemplo en el que las operaciones de inserción, búsqueda y eliminación son fundamentales podría ser el sistema de caché de un navegador. A través de este caché, el navegador guarda las respuestas que recibió de determinados pedidos (por ejemplo, *HTTP requests*) a distintos servidores. De esta forma, puede reutilizar las respuestas guardadas en pedidos subsiguientes. Esta reutilización tiene ventajas tanto para el usuario que está usando el navegador, ya que la respuesta será más rápida (¡está guardada en la misma máquina física!), como para el servidor, quien no tendrá que procesar un nuevo pedido. En estos casos, podemos pensar que el caché del navegador guardará la URL de los recursos junto con su contenido (por ejemplo, una determinada imagen). Así, la próxima vez que el navegador detecte que un sitio web necesita un recurso en la misma URL, la buscará en su caché, y mostrará la imagen sin necesidad de pedírsela nuevamente al servidor. Pasado cierto tiempo, el navegador borrará la entrada en el caché asociada a la URL (usualmente este proceso se denomina *invalidación de caché*), e irá agregando nuevas a medida que el usuario navegue en la web.

Además de las tres operaciones básicas ya mencionadas, estas aplicaciones tienen en común otra cuestión: la necesidad de utilizar *claves* y *valores*; es que justamente, en estos casos, la operación de búsqueda trae inherentemente estos conceptos: se busca algo por su clave (nombre de pila, URL), y se obtiene su valor asociado (el número de teléfono, la imagen).

Estas aplicaciones son ejemplos de lo que en ciencias de la computación se conoce como *diccionarios*.

El **TAD Diccionario** representa un conjunto dinámico donde cada elemento del diccionario contiene una clave, la cual permite identificarlo, y un valor asociado a dicha clave, que admite siguientes operaciones:

- `__init__` inicializa un nuevo diccionario vacío.
- `insert(k, v)` para agregar un elemento con clave k y valor v al diccionario.

- `delete(k)` Dado una clave k, si la misma se encuentra en el diccionario, borra la clave y su valor asociado.
- `search(k)` dada una clave k, decide si se encuentra en el diccionario. Si se encuentra, devuelve el valor asociado a dicha clave.

Además, incluiremos otra restricción: queremos que los tres métodos se ejecuten tan rápido como sea posible.

En este apunte estudiaremos distintas formas de implementar diccionarios.

2. Tabla de Direccionamiento Directo

Supongamos que una empresa de turismo nacional decide hacer descuentos en ciertos destinos dependiendo del mes. Por ejemplo, en enero los descuentos son en paquetes para viajar a Bariloche, en febrero en paquetes para viajar a Ushuaia, en marzo en paquetes para viajar a Córdoba, etc. Quien esté a cargo del sistema informático de la empresa podría decidir guardar esta información en un diccionario, en donde las claves son números naturales representando el mes correspondiente y, los valores, la información asociada al descuento de dicho mes. Así, en este caso, el universo de claves se define con el conjunto $U = [0, 1, ..., 11]$, en donde los números del 0 al 11 representan a los meses de Enero a Diciembre en su orden habitual.

Cuando se trabaja con aplicaciones que necesitan un conjunto dinámico en el cual cada elemento posee una clave dentro de un universo $U = \{0, 1, \ldots, m-1\}$ donde m no es muy grande y las claves son todas diferentes, para representar al diccionario podemos utilizar una tabla de direccionamiento directo (`DirectAccessTable`) $T[0, 1, \ldots, m-1]$ en la cual cada posición, o slot, corresponde a una clave en el universo U. [1] Esta técnica funciona muy bien cuando el universo U de las claves es razonablemente pequeño.

Una implementación en `Python` de esta tabla utilizando listas 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
```

Con dicha clase podríamos representar el sistema de descuentos de la empresa de turismo mediante una tabla de direccionamiento directo de 12 elementos [2], como se muestra a continuación.

```
discounts = DirectAccessTable(12)
discounts.insert(0, "Bariloche")
discounts.insert(1, "Ushuaia")
discounts.insert(2, "Cordoba")
# insertar los descuentos para el resto de los meses
```

[1] Notar que podemos pensar un arreglo como una tabla de direccionamiento directo donde donde las claves son simplemente los índices del arreglo.

[2] Como siempre, comenzamos a numerar los meses desde 0, es decir, el 0 representa el mes de Enero, el 1 es el mes de Febrero y así sucesivamente

La Figura 1 ilustra el enfoque de las tablas de direccionamiento directo de modo general. Cada clave corresponde a un índice en la tabla (arreglo) T. El conjunto U representa todas las claves posibles, mientras que el conjunto K muestra solo aquellas claves que están en uso. En la tabla, las posiciones en desuso están marcadas con un gris mas oscuro. 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`.

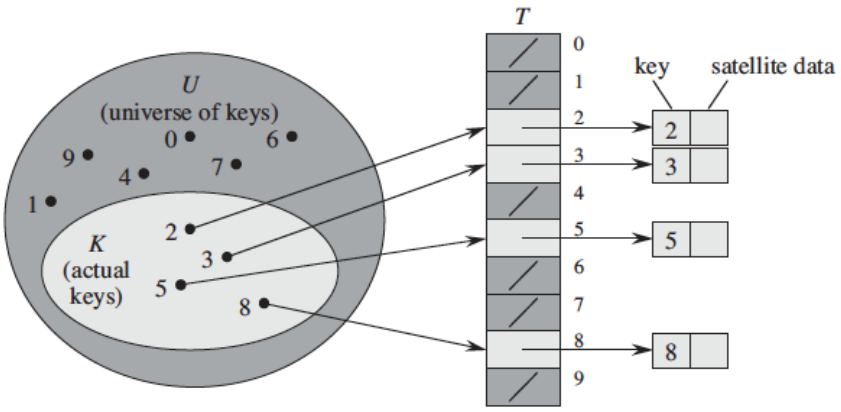

Figura 1: Tabla de Direccionamiento Directo.

En este tipo de implementación, cada una de las operaciones de diccionario se ejecutan en tiempo constante, dado que las operaciones se reducen a operaciones constantes en un arreglo de tamaño fijo. En cuanto al espacio de memoria ocupado es del orden del tamaño del universo U.

Ahora bien, ¿qué pasaría si en lugar de trabajar con un sistema de descuentos por mes, nos fuéramos a otro campo y, por ejemplo, quisiéramos indexar los artículos de Wikipedia? Si consideráramos el título de un artículo la clave mediante la cual se recupera el contenido del mismo, ¿cuál sería el universo de las claves? ¿Qué porcentaje de ese universo correspondería realmente a títulos válidos de artículos existentes?

Podemos notar que, cuando el tamaño del universo U de claves es grande, implementar un diccionario mediante tablas de direccionamiento directo puede ser poco práctico (o incluso imposible), dada la memoria disponible en una computadora típica. Además, si la cantidad de elementos del diccionario es mucho más pequeño que el tamaño de U, se desperdicia mucho espacio.

Ejercicios

1. Suponga que un diccionario S es representado mediante una tabla de acceso directo T de longitud m. Describa un procedimiento para encontrar el máximo elemento del diccionario y estime su complejidad.

2. El TAD Conjunto de Claves se define igual que un diccionario, pero las claves no poseen valores asociados. Explique como implementaría este TAD utilizando una Tabla de Direccionamiento Directo.

3. Tabla Hash (Hash Table)

Como se vio en la sección anterior, cuando el conjunto K de claves almacenadas en un diccionario es mucho más pequeño que el universo U de todas las claves posibles, las tablas de direccionamiento directo desperdician mucho espacio. Este es el caso si queremos, por ejemplo, indexar los artículos de Wikipedia sobre computación (en español) mediante su título. Si consideramos que un título está definido por valores alfanuméricos de hasta n caracteres de longitud, el conjunto K contendría los títulos de los artículos de computación existentes, mientras que el universo U de claves contendría todas las combinaciones posibles. Como se explicó anteriormente, con el direccionamiento directo, se reserva espacio para cada valor posible de

clave, y un elemento con clave k, se aloja en el slot k. Suponiendo que existen cinco mil artículos de Wikipedia sobre computación que queremos indexar, sería más razonable, por ejemplo, asignar claves únicamente en el rango $[0, 5000]$. Para esto necesitaríamos una función capaz de transformar un valor cualquiera en un número natural en dicho rango. De esta forma, reduciríamos la cantidad de memoria necesaria a un orden similar a la cantidad de claves que realmente tendrá el diccionario. La idea sigue siendo la misma: aprovechar que en un arreglo podemos acceder eficientemente por posición, con la diferencia de que ahora la clave no es la posición directamente, si no que a partir de la clave vamos a calcular la posición que le corresponda. Esta es la idea detrás una *tabla hash*.

Para implementar una tabla hash debemos contar con dos puntos fundamentales:

- Por un lado, debemos definir una tabla de tamaño m, $T[0, 1, \cdots, m-1]$, donde m es la máxima cantidad de claves que soportará. Se espera que m sea menor que el tamaño del universo de claves U (o sea, $m < |U|$).
- Por otro lado, también es necesario definir una función $h : U \to \{0, \cdots, m-1\}$, la cual denominamos *función hash*. h calcula para cada clave k la posición o *slot* de la tabla que le corresponde.

En una tabla hash, el elemento con clave k se almacena en $T[h(k)]$. La Figura 2 ilustra este enfoque.

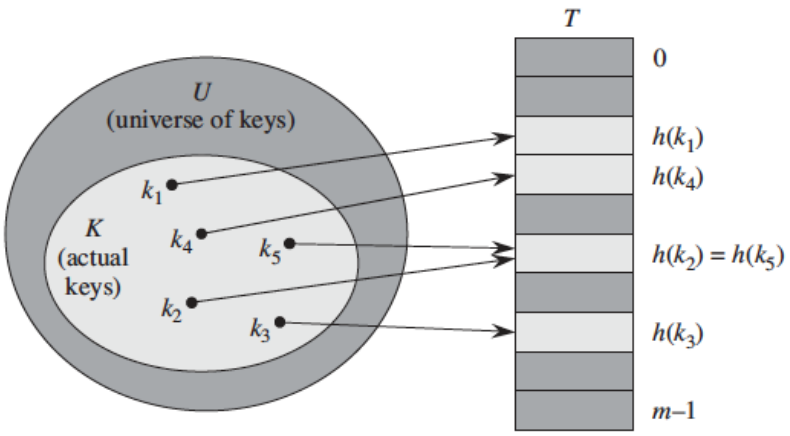

Figura 2: Uso de una función hash h para mapear las claves a las diferentes posiciones o *slots* de la tabla

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.

A continuación se presenta una primera versión de la clase `HashTable` en `Python`.

```
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 cumplir dos requisitos fundamentales:

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

Es importante notar que algunas funciones hash conocidas 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].

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 $(0, 1)$.

A continuación presentaremos algunas técnicas de resolución de colisiones.

[3] Una **base** del conjunto de caracteres es un método que permite codificar cada carácter de un lenguaje natural (el "alfabeto" de ese lenguaje) en un número. Algunas bases conocidas son la norma ASCII y UTF-8, por citar algunos ejemplos.

3.2. Resolución de colisiones 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 estructura de datos adicional, por ejemplo 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 cuales la función h asigna el valor j; si no existen tales elementos, la posición j contiene `None`.

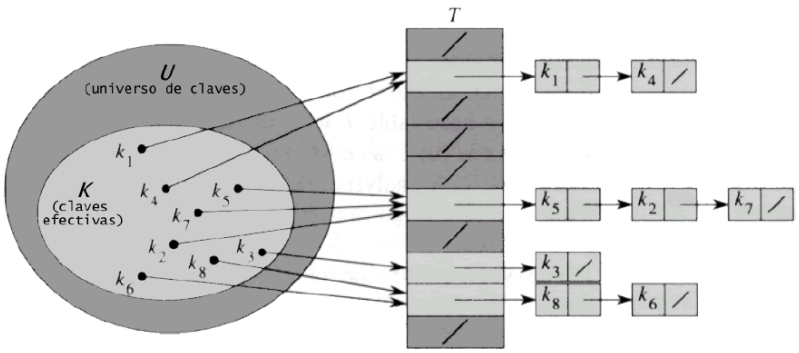

Figura 3: Tabla Hash - Resolución de colisiones por Encadenamiento

Supongamos que para definir una lista enlazada contamos con la clase `Node2` que se presenta a continuación.

```
class Node2:
    def __init__(self, key=None, element=None, next=None):
        self.key = key
        self.element = element
        self.next = next
```

Luego, podemos pensar en implementar una *tabla hash encadenada* (`HashTableChaining`) en `Python` de la siguiente manera:

```
class HashTableChaining(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.

3.2.1. Longitud de la lista más larga

Por "longitud" de la lista más larga nos referimos a la máxima cantidad de elementos en una posición de la tabla.

La performance de la tabla se deteriora a medida que la longitud aumenta. Usualmente, esto significa que si se supera cierto valor fijado de ante-mano, se considera que la tabla se ha deteriorada demasiado y se re-dimensiona la tabla, re-hasheando todos los elementos.

3.3. Resolución de colisiones 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*.

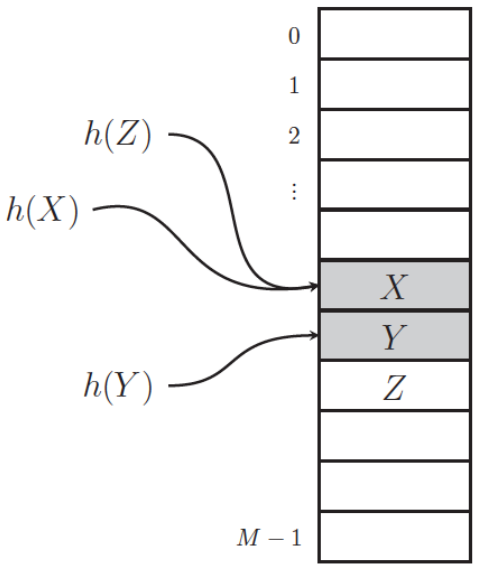

Figura 4: Tabla Hash - Resolución de colisiones por Direccionamiento Abierto

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 de la distribución uniforme de las claves.

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 $k = 5$.

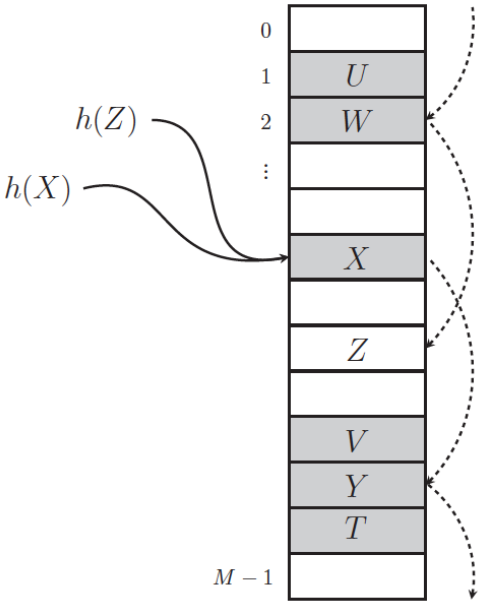

Figura 5: Tabla Hash - Resolución de colisiones por Direccionamiento Abierto con salto de tamaño k.

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.

En la Figura 6 se muestra un ejemplo simplificado del método de direccionamiento abierto con doble hash.

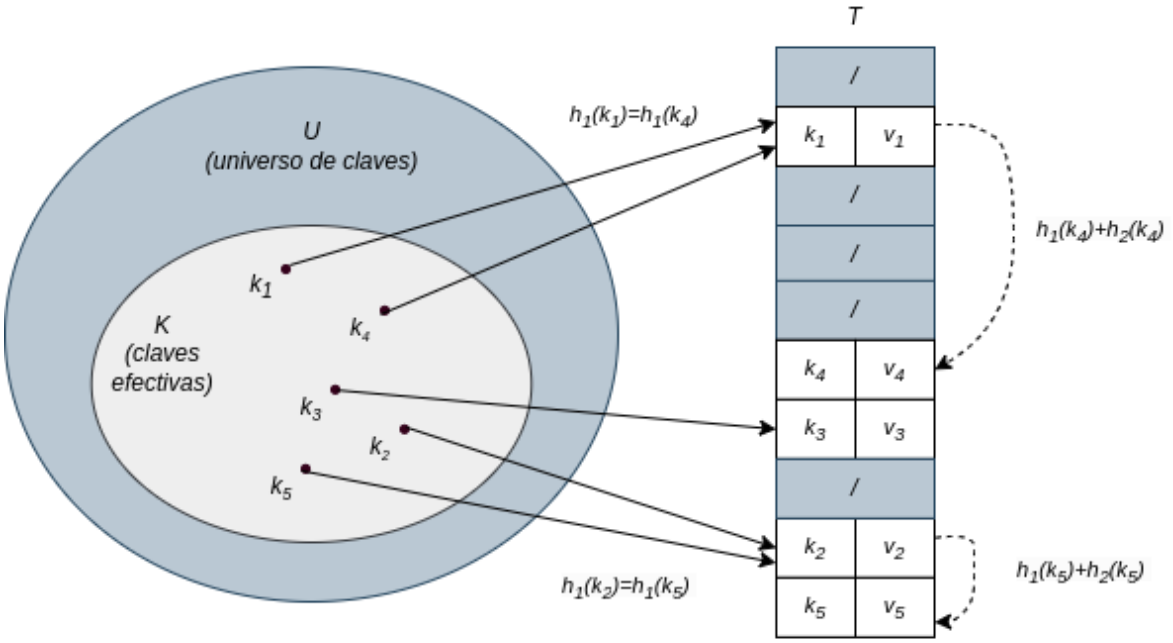

Figura 6: Tabla Hash - Resolución de colisiones por Direccionamiento Abierto con doble hash.

¿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.

3.3.1. Cluster más largo y Factor de Carga

Cuando se resuelven las colisiones por direccionamiento abierto, la performance de la tabla se degrada a medida que se agrandan los clusters de datos. Podríamos tomar como medida para monitorear la performance el cluster más grande en la tabla, pero dicho valor puede ser díficil de calcular. En su lugar se monitorea una métrica relacionada: el **factor de carga**.

El factor de carga α 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.

Es decir, es la proporción de ocupación de la tabla.

Si se resuelven las colisiones por direccionamiento abierto, 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 - para ahorrar memoria. Las cifras aceptables del factor de carga α se encuentran en entre 0,6 y 0,75.

4. Conclusiones

La técnica de hashing es extremadamente efectiva y práctica, las operaciones que definen al TAD Diccionario se ejecutan en tiempo constante en promedio[4].

Referencias

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

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

[4] Esto quiere decir que, si bien hay casos donde el tiempo que tarda la operación no es constante, la estructura de datos asegura que este peor caso no puede recurrir siempre.