Escrito por **Adrián Quiroga Linares**.

El problema de búsqueda es uno de los problemas más importantes y repetidos. Se quiere buscar un elemento entre un conjunto. En función de si el conjunto está ordenado o no, se puede realizar una **búsqueda lineal** (*recorre todos los elementos del primero al último buscando el elemento buscado, por lo que el orden de complejidad es de* $O(n)$), una búsqueda binaria (*solo posible para arrays ordenados, reduciendo el orden de complejidad a* $O(\log_2(n))$), o una **búsqueda por clave** (*con la que se accede a cualquier elemento por medio de su clave, teniendo una complejidad* $O(1)$).

Siguiendo este último método, se han implementado las **tablas hash**, que proporcionan un tiempo de búsqueda constante, independiente del número de elementos y aunque estos no estén ordenados, al acceder a cada uno a través de su clave. Además, la inserción y el borrado son también constantes. Esto se consigue obteniendo la posición de cada elemento por medio de una fórmula matemática llamada **función hash**. Ejecutar dicha fórmula para obtener la posición se hace de manera constante, independientemente del tamaño de la tabla, y una vez se sabe la posición, se puede acceder al elemento también de forma constante.

Estas son claras **ventajas** sobre el resto de estructuras de datos para almacenar conjuntos de datos. Sin embargo, como **desventajas** tenemos:

- Los datos se almacenan en un array (estructura estática) por lo que debe fijarse un tamaño máximo desde el principio, que no se podrá sobrepasar. Si el tamaño es demasiado grande, se desperdiciará mucho espacio.
- No hay forma de recorrer los elementos directamente, ya que no tienen porque encontrarse en posiciones consecutivas, pudiendo haber varias posiciones vacías de array entre cada par de elementos.
- Tampoco se pueden almacenar los elementos ordenados, ya que su posición viene dada por el resultado devuelto por la fórmula matemática.
- Se puede dar el caso en el que la función hash asigne a un elemento una posición ya ocupada en el array. A esto se le conoce como **colisión**, y es necesario plantear una solución para redirigir el elemento a otra posición, y tener forma localizarlo después.

Las claves (*elemento identificativo de los datos que queremos ordenar*) pueden ser números o cadenas de caracteres.

8.1 Funciones Hash

Existen diversas funciones hash con sus ventajas e inconvenientes, que se muestran a continuación.

8.1.1 Función hash por módulo

Es muy simple de entender y de implementar: $h(K) = K \mod N$, siendo h la función hash, K la clave del dato, mod el operando módulo (resto de la división), y N el tamaño de la tabla (tamaño del array en el que se insertan los datos).

Sin embargo, a pesar de su sencillez, tiene el gran inconveniente de que ciertos valores

de N pueden producir muchas colisiones para muchas claves (aquellos que tengan una factorización con muchos factores). Una forma de solucionarlo es seleccionar para el valor de N un número primo. Solo sirve para claves formadas por **números enteros**.

8.1.2 Función hash cuadrado

Se eleva la clave al cuadrado y del resultado se seleccionan los dígitos centrales: $h(K) =$ dígitos centrales(K^2). Para determinar cuántos dígitos centrales se cogen, se puede utilizar una función matemática en función del tamaño de la tabla, por ejemplo: dígitos centrales $= \log(N)$.

En este caso el número de colisiones no va a depender de los factores del valor del tamaño de la tabla tan directamente, sino que toma un carácter un poco más aleatorio, aunque por lo general se reducen para tamaños de tabla grandes. Solo sirve para **claves formadas por números enteros**.

8.1.3 Función hash por plegamiento

Se divide la clave en partes con el mismo número de dígitos (*salvo la última, que puede tener menos si la división no es exacta*) y se realiza una operación de suma o multiplicación sobre las partes, y sobre la solución, se queda con las cifras menos significativas. El número de cifras para realizar las divisiones en partes y el número de cifras menos significativas que se escogen de la solución dependen también del tamaño de la tabla. Solo sirve para claves formadas por **números enteros**.

8.1.4 Función hash por el método de la división

Para claves formadas por cadenas de caracteres, una opción es sumar los valores ASCII de cada uno de los caracteres que forman la cadena y al valor entero obtenido calcularle la función hash por módulo.

Como en el caso de la función hash por módulo, uno de los **inconvenientes** puede ser el valor del tamaño de la tabla escogido. Además, también presenta problemas con valores de tamaño muy grandes si las cadenas tienes todas pocos caracteres, ya que no se llenará gran parte de la tabla, desperdiciando espacio mientras se pueden estar provocando colisiones en un número reducido de posiciones.

8.1.5 Función hash por suma ponderada

Otro método para calcular la posición cuando las claves están **formadas por cadenas de caracteres** es obtener los valores ASCII de cada carácter, multiplicarlo por un número en función de su posición en la cadena y luego sumar los valores.

Para evitar obtener valores no válidos, hay que realizar el módulo del valor del tamaño de la tabla para cada valor de cada carácter. Como el código ASCII recoge 256 caracteres, usar una numeración en base 256 puede ser una buena opción. En ese caso, el último carácter de la cadena tendrá simplemente su valor ASCII, el siguiente tendrá su valor ASCII más 256, el posterior tendrá su valor ASCII más $256 \cdot 2$, y así sucesivamente, sumando después todos esos valores.

8.2 Resolución de colisiones

Como se explicó antes, cuando una clave es asignada a una posición ya ocupada se produce una **colisión**, que es necesario resolver para poder asignar una posición válida a dicha clave y poder localizarla después. Una forma de **evitar las colisiones** primeramente es seleccionar una **buena función hash**. Sin embargo, siempre hay colisiones que son inevitables. Para resolver estas existen dos alternativas: **recolocación y encadenamiento**.

8.2.1 Recolocación

Si la posición asignada al elemento está ya ocupada, se busca otra posición. Hay 3 tipos de recolocaciones:

- **Recolocación simple**: Si la posición asignada está ocupada, se prueba en la siguiente; si también está ocupada, en la siguiente, y así sucesivamente hasta encontrar una posición vacía. Para buscar elementos, se sigue el mismo proceso: si no está en la posición correspondiente, se recorren las siguientes hasta encontrarlo o, si no está, hasta llegar a un espacio vacío o recorrer toda la tabla. Esto supone distinguir entre posiciones vacías porque se ha borrado un elemento y aquellas que están vacías porque aún no se insertó ningún elemento.

  Desventajas:

  - El coste de la inserción, búsqueda y borrado aumenta si hay que recorrer muchas posiciones.
  - Pueden formarse grandes bloques de posiciones ocupadas consecutivas, lo cual incrementa el problema anterior.

- **Recolocación por salto**: Similar a la recolocación simple, pero en lugar de avanzar una posición cada vez, se avanza de a **en** a**, donde** a **es un número entre 2 y** $N-1$ (*con* N *siendo el tamaño de la tabla*). Un **problema** es que, en función del valor de a, puede no encontrarse una posición vacía a pesar de que sí las haya. Esto se soluciona seleccionando un valor de a que sea primo con N.

  Se conoce como factor de carga a la relación entre el número de datos almacenados y el tamaño de la tabla. Si este valor se mantiene **por debajo de 1/2**, se logra una media de colisiones constante.

- **Recolocación cuadrática**: Cada vez que ocurre una colisión, se busca en la posición resultante de sumar el número de colisión al cuadrado a la posición original. Esto evita formar bloques de posiciones consecutivas llenas. Sin embargo, es posible no encontrar una posición libre, aunque solo ocurre cuando la tabla está casi llena. Por ello, es necesario mantener el **factor de carga por debajo de 1/2**, lo cual se logra mediante la **redispersión**, que se comentará más adelante.

8.2.2 Encadenamiento

Cada posición del vector de datos contiene una lista enlazada, lo que permite insertar un nuevo elemento al final de la lista sin producir ninguna colisión, incluso si ya hay

elementos en esa posición. Así, **la inserción se realiza siempre en un tiempo constante**.

Sin embargo, la **búsqueda y eliminación** requieren calcular la función hash y recorrer la lista hasta encontrar el elemento, lo cual supone un **coste lineal** en el peor de los casos. Además de la reducción general en el tiempo de inserción, otra ventaja es que no existe un número máximo de elementos, ya que las listas pueden crecer según sea necesario.

Si todos los elementos se concentran en unas pocas posiciones del array, se tendrán listas largas en esas posiciones, mientras otras permanecerán vacías, reduciendo la eficiencia. Para evitar este problema, se debe mantener el **factor de carga por debajo de 3/4 mediante la redispersión**.

8.3 Redispersión

Cuando en una tabla hash, ya sea con recolocación o encadenamiento, se supera el **factor de carga máximo establecido**, es necesario reducirlo para no afectar la eficiencia. Para ello, se realiza la operación de **redispersión**, que **aumenta el tamaño de la tabla**, generalmente duplicándolo, y recoloca los elementos en sus nuevas posiciones. Aunque es una **operación costosa**, al realizarse pocas veces y mantener la eficiencia de la tabla hash, se considera **aceptable**.

8.4 Aplicaciones

Algunas de las aplicaciones más típicas de las tablas hash son:

- Tablas de símbolos (variables, por ejemplo) de intérpretes y compiladores.
- Gestión de un conjunto de personas, identificados por su DNI o número de la seguridad social.