7. Tablas de Dispersión

Las **tablas de dispersión** o **tablas hash** son aquellas que almacenan elementos de un conjunto mediante el uso de una **función hash**, que dado un elemento devuelve su posición exacta en la tabla que se almacena.

El tiempo de búsqueda de dicho elemento es de coste $O(1)$, ya que tenemos acceso directo a dichos elementos de un conjunto, esto es así ya que el tiempo que se tarda en calcular la función hash es constante.

Una idea propuesta es que cada elemento que se almacena se compone de una clave, estas claves son números entero y dependen del tamaño del vector M, donde el rango de las mismas es $[0, M-1]$, por tanto, si queremos acceder al elemento 'i', será de la manera $h(i) = i$.

También podemos encontrar problemas con dicha idea, ya que pueden que no estén acotadas (demasiado grandes para el tamaño del vector), la claves no son de tipo entero..

7.1. Funciones Hash

Las **funciones hash** transforman una clave de un cierto tipo (número, cadena, fecha, etc) en una **dirección** o **índice** de la propia tabla hash, $h : C \rightarrow [0, M-1]$, donde C es el conjunto de posibles claves.

Debido a esto encontramos las **colisiones** entre claves que se produce cuando a dos claves diferentes le corresponden como resultado el mismo valor de función hash. Si esto ocurre decimos que ambas son **claves sinónimas**.

Por tanto, una **función hash perfecta** es aquella que no produce ninguna colisión entre todas las claves posibles $C \leq M$ (el cardinal de C es menor al tamaño del vector).

Pero esto es algo teórico, ya que en la práctica el número de claves posibles es mayor al tamaño del vector $C > M$, por tanto, es inevitable que se produzcan colisiones entre claves.

Propiedades que encontramos en las buenas funciones hash:

- Eficiencia temporal → Rapida de calcular.
- Eficiencia espacial → Función sobreyectiva(todas las posciones de la tabla corresponden a una clave).
- Distribución uniforme → Reduce la posibilidad de colisiones entre claves.

Esto no es algo sencillo y presenta algunas dificultades.

Una función hash la podemos construir en dos partes:

1. Definir una función que hash **H(x)** que transformen las claves a números enteros.
2. Trasladar cualquier número a una dirección obtenida mediante el calculo de su módulo respecto al tamaño del vector → $h(x) = H(x)\ mod\ M$.

7.2. Resolución de colisiones

Como hemos comentado anteriormente, en la práctica las colisiones son algo *inevitable*, para hacerles frente podemos definir diferentes tipos de hashings (hashing cerrado, abierto o encadenamiento mezclado).

Tenemos que diferenciar que la **sinonímia** $\rightarrow$ **colisión**, pero la **colisión** $\nrightarrow$ **sinonímia**.

7.3. Hashing Cerrado

Cuando se produce una colisión, se busca una nueva posción libre en la tabla. Aquí encontramos una **secuencia de exploración** para poder localizar dicha posición libre de la tabla. Esto se realiza mediante una serie de **M funciones hash** $\{h_0, h_1, h_2, ..., h_{M-1}\}$, donde h_0 es la función hash original.

Exploración lineal

Tenemos que $h_i = (h_0 + i * \alpha)\ mod\ M$, donde la distancia entre celdas $\alpha \in \mathbb{N}$ y es primo relativo con M.

Este tipo de hash produce **agrupamientos primarios** $\rightarrow$ posiciones ocupadas $\alpha = 1$, esto aumenta el tiempo de posteriores colisiones debido a que se puede producir un gran bloque de claves, lo que denominamos **degradación muy rápida de la eficiencia de las operaciones**.

Exploración cuadrática

Ahora tenemos $h_i = (h_0 + i^2)\ mod\ M$, ahora la distancia entre celdas aumenta exponencialmente.

Ahora encontramos que se producen **agrupamientos secundarios** de claves sinónimas debido a que la secuencia de exploración desde h_0(+1, +4, +9, +16,..), además estos agrupamientos secundarios se entrelanzan pero no se unen para formar otros mayores, es decir, son menos perjudiciales para la eficiencia temporal que los agrupamientos primarios.

Exploración aleatoria

Este es otro tipo de hahing cerrado, donde $h_i = (h_0 + x_i)\ mod\ M$, donde x_i se obtiene mediante una generador de números aleatorios (GNA).

El GNA tiene que generar dichos números cuya distribución debe de ser uniforme, es decir, que tengan la misma probabilidad para permitir recorrer todas las posciones.

Este tipo de hashing cerrado evita los agrupamientos de claves sinónimas si usamos una semilla diferente para cada clave, de modo que tendremos secuencias de exploración diferentes.

Hashing doble

En este tipo de hashing encontramos una segunda función hash $h`$, quedando la función hash como $h_i = (h_0 + i * h`) \ mod \ M$.

Este método es parecido a la *exploración lineal*, pero con una diferencia: la distancia entre celdas exploradas es variable, debido a que ahora depende de incrementos de $h`$ posiciones.

Como ahora cada clave tiene una secuencia de exploración diferente, se **evita la colisión de claves sinónimas**.

Esta nueva función hash secundaria tiene que cumplir ciertas condiciones:

- $h`(x) \neq 0$, para cualquier clave 'x', de lo contrario no sirve para resolver colisiones, ya que si no todas las funciones de exploración coincidirían con la función hash original h_0.
- $h` \neq h_0$. Si ambas son iguales, la secuencia de exploración sería la misma, por ende, habría colisiones entre claves.
- $h`(x)$ y M deben de ser primos relativos para garantizar que se recorre toda la tabla.

Es importante saber distinguir entre casillas libres y ocupadas en la tabla, así como las borradas.

Operaciones

- **Búsqueda:** Seguir con la secuencia de exploración hasta:
  - Encontramos la cable buscada (búsqueda con éxito).
  - Encontramos una casilla *libre* (búsqueda sin éxito).
  - Se ha recorrido todas las casillas (búsqueda sin éxito).
- **Inserción:** Seguir con la secuencia de exploración hasta:
  - Encontramos una casilla que está *libre* y realizar la inserción.
  - Encontramos la clave a insertar (ya existe en la tabla).
  - Se ha recorrido toda la tabla (concluimos con que está llena).
- **Eliminación:** Seguir con la secuencia de exploración hasta:
  - Encontramos la clave a eliminar y realizamos la eliminación (marcar la casilla como *borrada*).
  - Encontrar una casilla libre o se ha recorrido toda la tabla (clave no existe).

La eficiencia de las operaciones se degrada rápidamente a medida de que se va llenando de claves la tabla.

7.4. Hashing abierto

Una **tabla hash abierta** (también denominadas con *direccionamiento cerrado* o *encadenamiento separado*), cada posición de la misma es un cubículo con una estructura que permite alojar varios elementos cuyas claves son **sinónimas**.

La estructura más sencilla es '**tabla de M lista enlazada**', cuya longitud media es n/M, donde n es el número de claves insertadas. Esto determina el tiempo medio de búsqueda y la función hash distrubuirá uniformemente las claves para que no se acumulen en pocos cubículos.

Si seleccionamos una M demasiado grande, las listas serán bastante cortas y, en consecuencia, **no supone una mejora importante de la eficiencia de la búsqueda mantener las listas ordenadas** o **sustituirlas por árboles de búsqueda**.

7.5. Encadenamiento mezclado

Es una *combinación del hashin cerrado y abierto*, donde las claves se insertan en casillas libres de la tabla (hashing cerrado), pero aquellas que la función hash lleva a casillas ocupadas se enlazan formando listas dentro de la tabla (hashin abierto).

Estas listas pueden contener una mezcla de **claves sinónimas** y **no sinónimas**, a diferencia del hashin abierto.

La búsqueda con esta combinación **es más rápida** que las búsquedas que se realizan en el *hashing cerrado*, puesto a que solo hay que buscar el fragmento de la lista que comienza en la casilla $h(k)$, para localizar la clave k.

Por el contrario, la búsqueda en el *encadenamiento mezclado* no son tan rápidas como en el *hashing abierto*, ya que estas listas contienen tanto claves no sinónimas como sinónimas, por ende, son más grandes.

7.6. Eficiencia

- **Mejor caso**: No hay colisiones, hay una clave en cada casilla de la tabla, por tanto, las búsquedas son de orden $O(1)$.
- **Peor caso**: Todas la claves de la tabla tienen colisiones, debido a esto las búsquedas son de orden $O(n)$.
- **Caso promedio**: Si asumimos que la distribución de claves es uniforme, la búsqueda depende del factor de carga ($\alpha = n/M$) y la distribución de las propias claves.

7.7. Comparación entre métodos

Si el factor de carga se mantiene por **debajo del umbral**, el número de comparaciones medio estará acotado por una constante y esto hace que la **búsqueda** tenga un coste promedio de $O(1)$.

El *hashing cerrado* requiere prever el número máximo de elementos a almacenar (degradación muy rápida de la eficiencia y rendimiento), mientras que *hashing abierto* requiere un mayor tamaño de la tabla (aunque el uso de memoria es menor, ya que no usamos punteros).

La exploración lineal es el método **menos eficiente**, con una curva exponencial (*véase en la Figura 7.1: Gráfica comparativa entre diferentes tipos de hashing*). Aunque esta aplicación puede ser útil en algunos casos, ya que su implementación no es complicada.

La exploración cuadrática tiene una eficiencia equiparable a las exploración aleatoria o *hashing doble*, esto confirma que los **agrupamientos secundarios** no afectan tanto en la eficiencia como los **agrupamientos primarios**.

Como vemos el rendimiento del *Encadenamiento mezclado* se aproxima al del *hashing abierto*, el cual es el **más eficiente** peor acosta de consumir más memoria debido al uso de punteros de las listas enlazadas.

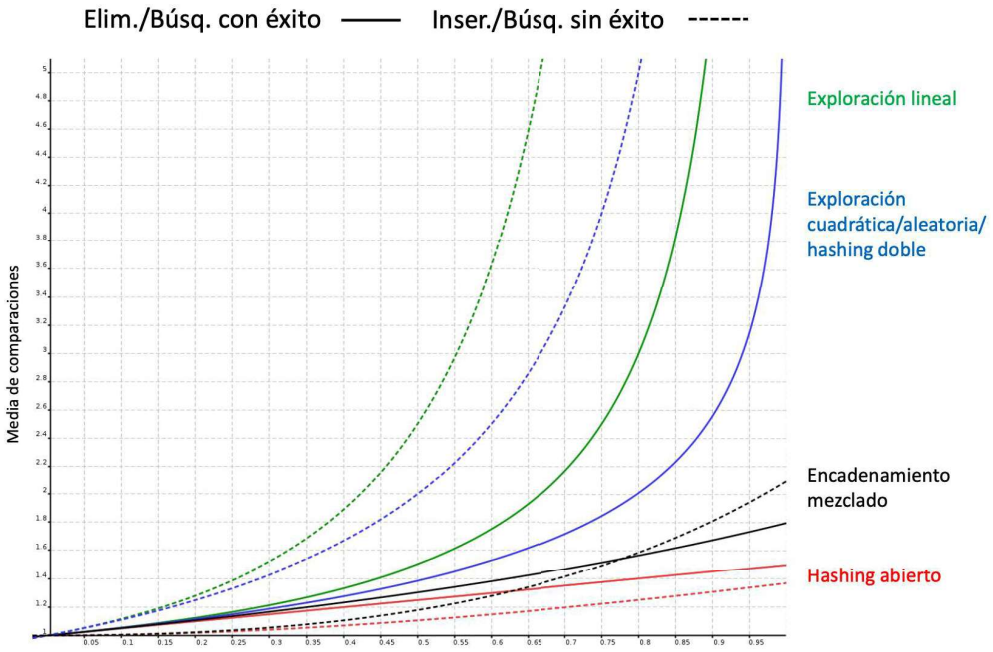

Figura 7.1: Gráfica comparativa entre diferentes tipos de hashing

7.8. Redimensionamiento y Rehashing

Cuando una tabla tiene un factor de carga alto, es decir, está muy llena, se realiza un **redimensionamiento** de la misma (se crea una tabla más grande y se reinsercionan de nuevo los elementos de la anterior) o un **rehashing** (se reinsercionan los elementos en la misma tabla para eliminar las casillas que están marcadas como borradas).

Esto hace que se restaure la eficiencia de la tabla.

7.9. Comparación con árboles de búsqueda

Sabemos que las tablas hash ofrecen un tiempo de búsqueda del orden $O(1)$ si controlamos el facto de carga, pero un coste de $O(n)$ para el peor caso.

Como vimos en el tema de árboles de búsqueda, estos ofrecen un tiempo de búsqueda del orden $O(log\ n)$, con la ventaja de que estos árboles son dinámicos, es decir, no tenemos que prever el número máximo de elementos a almacenar.

Otra ventaja es que los **árboles de búsqueda** ofrecen un conjunto de operaciones más amplios, donde destaca el *acceso a los elementos en orden*, gracias a esto podemos acceder los mínimos y máximos valores respecto a otro o buscar el minimo valor o máximo del árbol.

Como conclusión tenemos que si queremos realizar búsquedas rápidas la elección correcta es el uso de **tablas de dispersión o hash** cuando se elige una buena función hash y controlamos adecuadamente el factor de carga. Sin embargo, si necesitamos tener los elementos ordenados, haremos uso de un **árbol de búsqueda**.