

1. Hasling

El motivo del uso del backtracking es la obtención de la búsqueda en una eficiencia constante O(1).

Consiste en asignar a cada elección una única posición dentro de una tabla y a cada posición de la tabla un único elemento.

Se usarán funciones hash. Por tanto nuestros objetivos son encontrar funciones hash que generen el menor número de colisiones y proporcionar soluciones para las mismas.

2. Tables Hash

Es un **contenedor** que permite el **almacenamiento y recuperación eficientes** a partir de **claves**.

Resumen de la tabla.

$h(\text{clue}) = \text{valor}$ La idealness que la sea inyectiva, pero esto es
muy difícil de lograr.

Tabla hash		
$h(k)$	k	Posición dentro del fichero
0	239	i
1	500	n
	.	
	.	
	.	
n	733	1

		Fichero
	k	Contenido
1	733	bla bla bla bla bla
	.	
	.	
	.	
	.	

Si, en particular, h
e **biyectividad** **funciones**
perfecto

3. Functions Hash

$$h: A \longrightarrow \mathcal{C}$$

- S conjunto de claves
 - C conjunto de índices sobre la tabla hash (regIN)

Coudicoues

- Potencia de calorías
 - $b(x) = c$ \rightarrow En caso contrario no es una función const.
 - Ser random \rightarrow es pseudoraleatoria
 - Nº de colisiones mínima

La más común es $h(x) = k \cdot \text{west}(x) + k \cdot \text{espresso}$. k = tamaño de la tabla. Se deben tener las siguientes consideraciones:

- i) $k > w^2$ claves si es menor \rightarrow habrá colisiones
- ii) k primo \rightarrow obligatorio pero si eficaz.
- iii) Equilibrio uso de espacio y evasión de colisiones

\rightarrow depende del tipo de hashing

4. Colisiones

Situación: $h_i(x) = h_j(y)$

Solución:

1. Depende de la estructura de datos elegida
2. Depende si es hash abierto o hash cerrado.

Hashing abierto

Consiste en usar una ED auxiliar por cada índice de la tabla, normalmente una lista.

El tamaño de la tabla se fija a priori y suele ser un vector de punteros a la estructura usada.

En realidad, el tamaño de la tabla no es fijo; sin embargo, el espacio requerido por los punteros y las listas provoca un desgaste de memoria y se espera la eficiencia, que depende de la estructura auxiliar. \rightarrow No usar colas, pilas, poniendo estructuras como pilas o colas provocaría una degradación en la eficiencia.

En este tipo de hashing se usa el factor de carga, definido por el factor de carga de la estructura auxiliar, se busca que sea próximo a 1.

Rendimiento: w^2 medios de intentos para insertar.

Hashing cerrado

Consiste en usar un vector para albergar la tabla hash.

Debido a las colisiones, se usa el rehashing para su resolución asignándole otro valor hash a la clave hasta encontrar un libre. $\rightarrow h_1(x)$

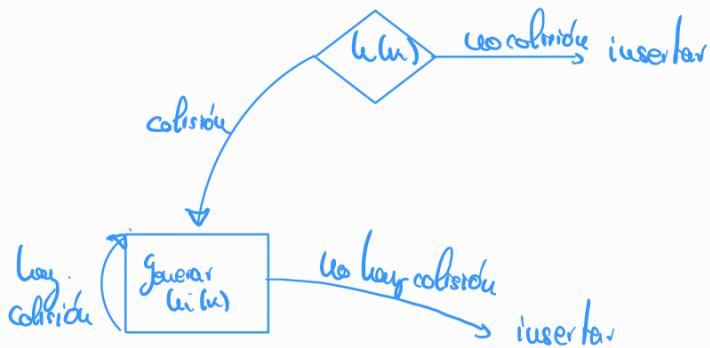
- Búsqueda siguiendo la misma secuencia de inserción.
- La casilla borrada debe marcarse como tal, pero a tener en cuenta que al insertar con respecto a estar libre estuviera así para la búsqueda, pues significaría que está ocupada implicando que se siga buscando.

borrada = ocupada (el borrar) borrada = libre (la inserción)

- En ocasiones, cuando se ocupe toda la tabla se hace necesario aumentar el tamaño de la misma y volver a rehashing. Normalmente consiste en cambiar la función hash. No tiene por qué ser siempre por el mismo.

Rehashing lineal

$$h_i(u) = [h(u) + (i-1)] \% m \quad i \geq 2 \text{ con } h(u) = u \% m$$

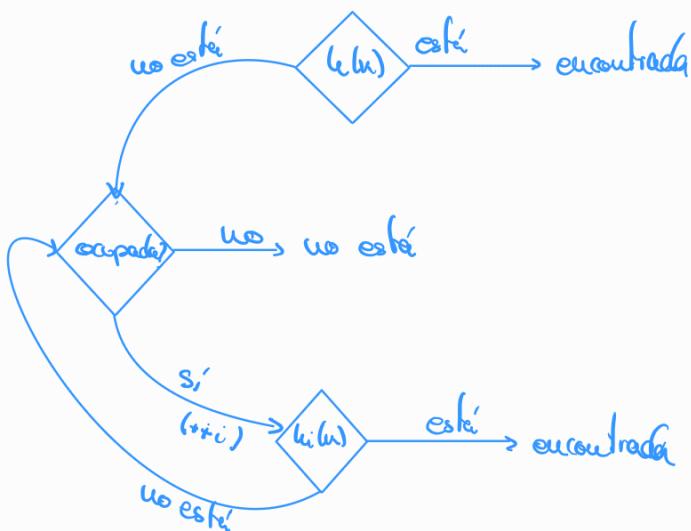


Esta solución tiende a crear agrupaciones primarias, es decir, una sucesión de casillas ocupadas en una tabla más contiguas. Estos elementos causan degradación de eficiencia en las inserciones y búsquedas.

Solución:

- Estructuras que gestionen el manejo del fin de las agrupaciones primarias
- Métodos de distribución alternativa de bloques.

Para buscar:



Scandeo aleatorio

$$h_i(u) = (h_0(u) + (i-1) \cdot c) \% m \quad i \in \mathbb{N}, i \geq 2 \quad \text{c primos relativos con m}$$
$$h_i(u) = [h_0(u) + c] \% m \quad i \in \mathbb{N}, i \geq 2 \quad \text{Es decir } P(c, m) \neq 0$$
$$\hookrightarrow h_0(u) = h(u)$$

Este estructura hash genera agrupaciones secundarias de orden c

Rehashing doble (es el más preventivo y usado).

$$h_i(u) = (h_{i-1}(u) + h_0(u)) \% m \quad i \in \mathbb{N}, i \geq 2$$

$$h_0(u) = 1 + (u \% (m-1))$$

$$h_1(u) = h(u)$$

Condiciones $h_0(u)$:

- $h_0(u) \neq 0$
- $h_0(u)$ no constante

Es buena cuando m y $(m-1)$ son primos relativos