Hash tables

E. Rivas

Abril de 2014

Resumen

- Hash tables
  - Motivación
  - Idea
- Funciones hash
  - Cadenas
  - Enteros
- Implementación
- Resolución de colisiones
- Conclusión

Hash tables - motivación

Imaginemos que desamos mantener un arreglo asociativo: tener un conjunto de pares de claves y valores.

$$
\begin{array}{lcl}
\text{“Gabriel Guerrise”} & \mapsto & \text{4567-1234} \\
\text{“Fabio Pastrello”} & \mapsto & \text{4567-1235} \\
\text{“Alejandro Alaci”} & \mapsto & \text{4321-1241} \\
\text{“Quique Illid”} & \mapsto & \text{4653-2211} \\
 & \vdots & \\
\text{“Ricky Rua”} & \mapsto & \text{4655-3456}
\end{array}
$$

¿Cómo podemos almacenarlo?

Hash tables - motivación

Podríamos armar una lista:

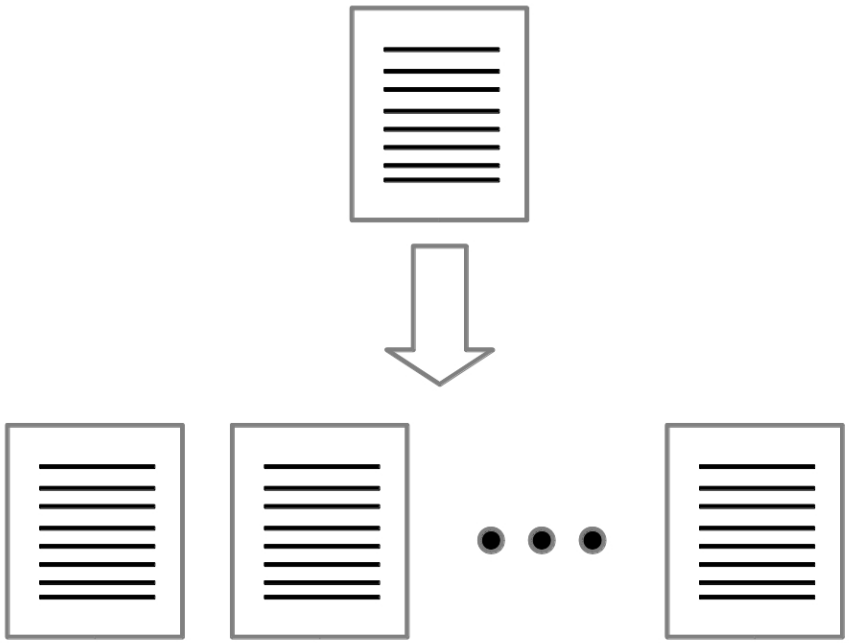
```
typedef struct _contacto {
  char key[50];
  char value[30];
} contacto;

contacto agenda[100];
```

Sin embargo esta puede resultar ser una representación poco conveniente.

Hash tables - motivación

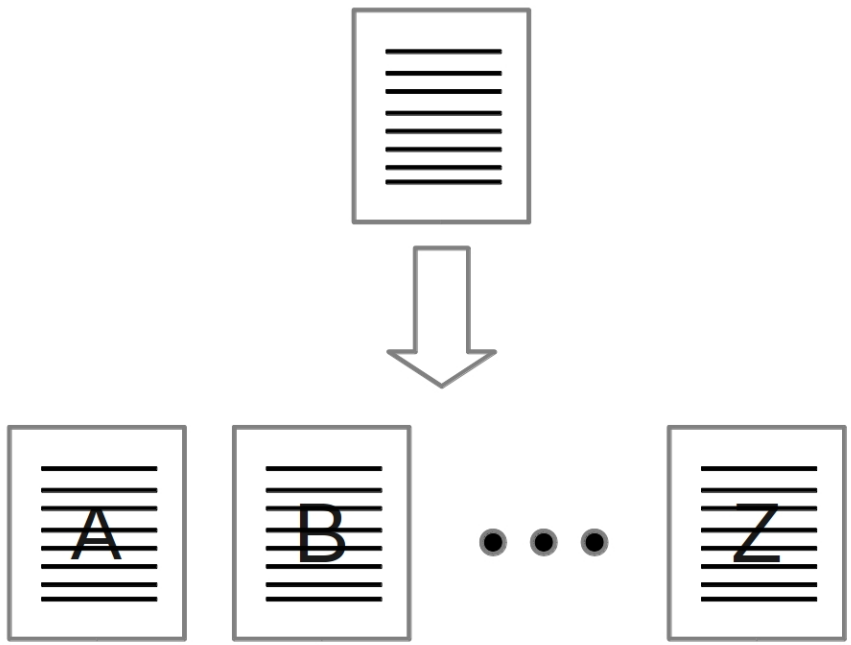
El problema de esta representación es que para recuperar un teléfono debemos recorrer varios items en la lista.

Necesitamos una idea... dividimos la agenda en partes.



Hash tables - idea

Por ej., podemos dividir la lista en 26 partes, una por cada letra del abecedario, y almacenamos en cada una solo los nombres que comienza con una letra dada.



Dado un nombre, solo tenemos que buscar en la lista de nombres que empiezan con esa letra.

Hash tables - idea

Sin embargo... todavía no es lo ideal. Sería mejor dividir el espacio de claves (nombres) en más *cubetas*, de manera que haya menos registros en cada cubeta, y sea más rápida la búsqueda.

Pregunta: ¿cuál sería el caso límite?
Poner tantas cubetas como nombres posibles.

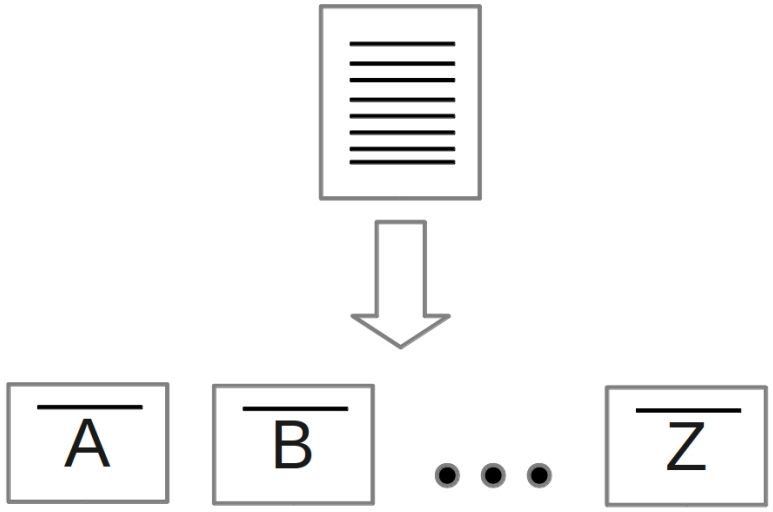
Pregunta: ¿cuál sería el problema?
Que ocuparíamos demasiado espacio.

Hash tables - idea general

Pero... ¿qué es una cubeta?

En una cubeta podríamos almacenar un solo elemento.

¿Que significaría esto? No se podría, en principio, almacenar más de un contacto que comience con la misma letra.



Hash tables - idea general

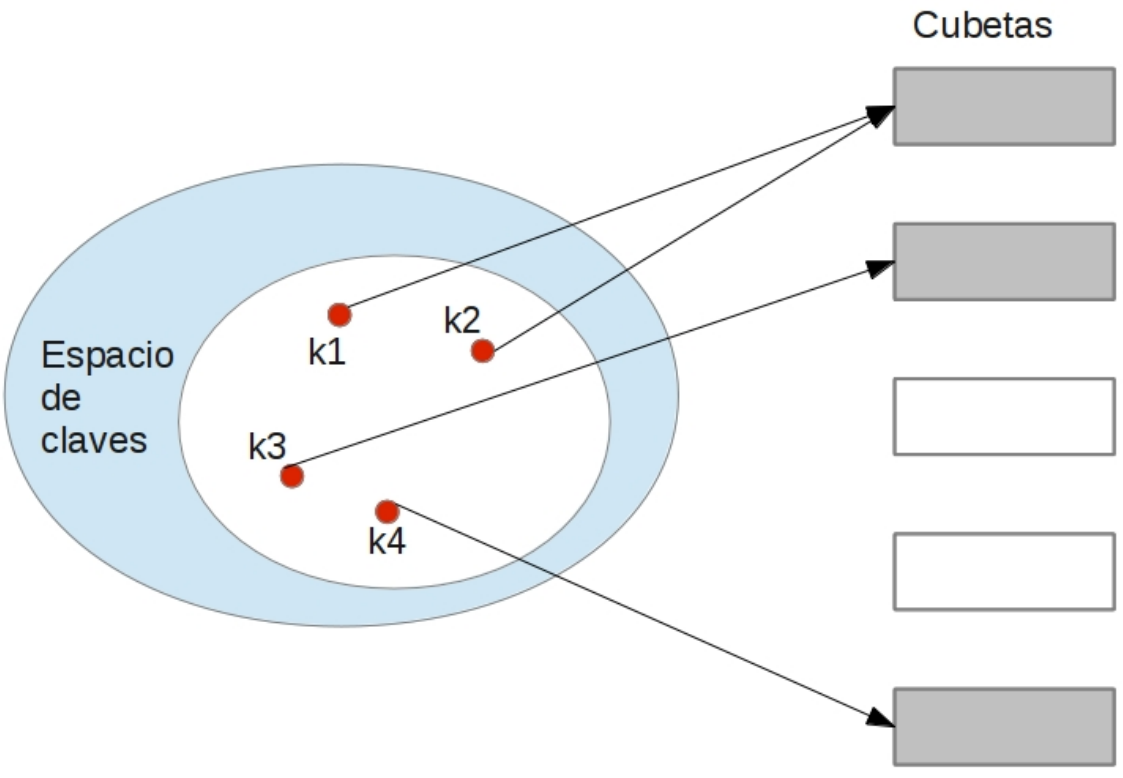
Supongamos que tenemos R cubetas donde almacenaremos los valores.

Dado un espacio de claves (K), utilizamos una función para separar los valores. En general, esta función tiene la forma:

$$\text{hash} : K \to [0, R-1]_{\mathbb{N}}$$

A un elemento con clave k, lo almacenamos en la cubeta hash(k). Cuando dos claves tienen van en la misma cubeta, entonces decimos que hay una *colisión*.

Hash tables - idea general



Funciones hash

Idealmente queremos conseguir funciones hash que:

- Sean fáciles de computar.
- Distribuyan las claves de manera uniforme entre todas las cubetas.

Queremos funciones hash para enteros, cadenas, etc.

Ejercicio: proponga una función hash para enteros para una tabla de 35 elementos.

Funciones hash - para cadenas

Si tenemos una función de hashing hashnat para naturales, entonces proponemos la función

$$\text{stringtonat}(c) = \sum_{i=0}^{L} c_i B^i$$

donde B es la base del conjunto de caracteres (128 o 256).

Luego podemos crear una función hash para strings como

$$\text{hashstring} = \text{hashnat} \circ \text{stringtonat}$$

Funciones hash - alternativa naturales

Una función hash alternativa para números naturales es la conocida como el "método de multiplicación".

Si tenemos R cubetas, entonces utilizamos la función:

$$\text{hashmult}(k) = \lfloor R(kA - \lfloor kA \rfloor) \rfloor$$

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

Implementación en C

En la implementación de un hash table llevaremos:

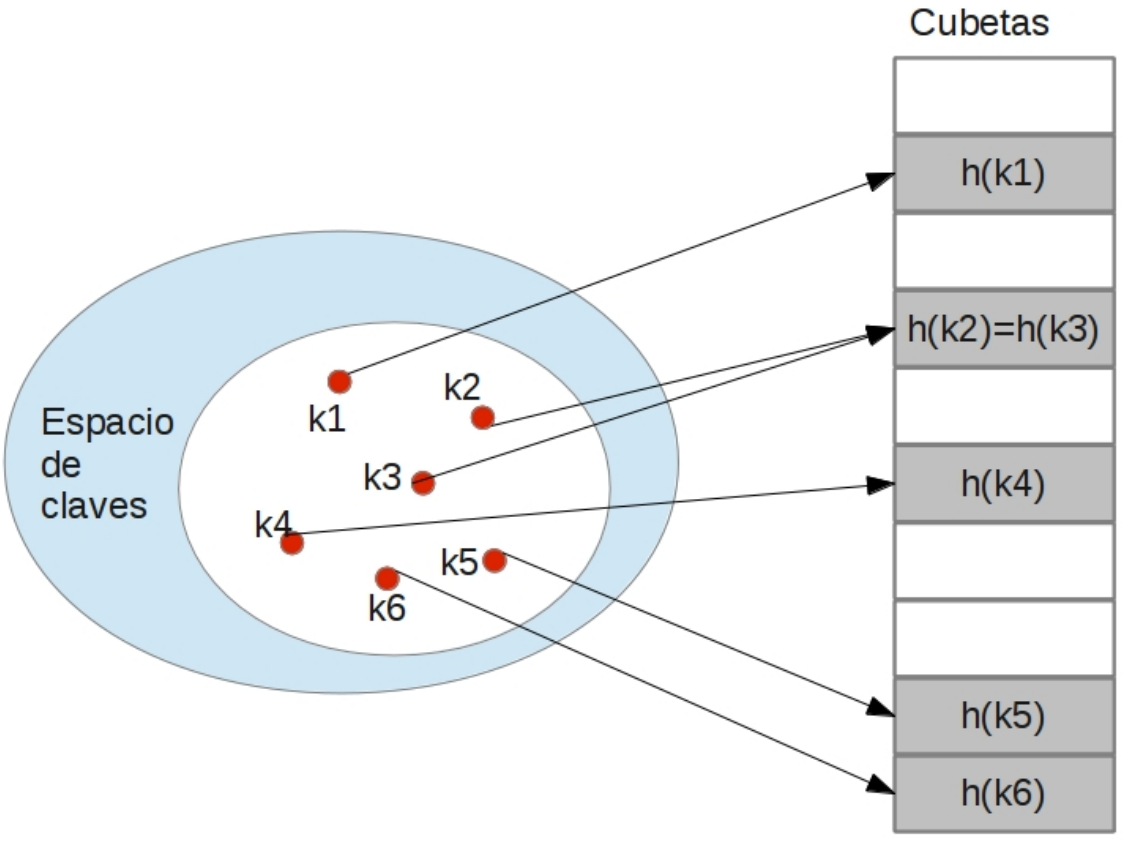
- Un puntero a las cubetas.
- La cantidad de cubetas reservadas.
- Un puntero a la función de hashing correspondiente.

Además, definimos un tipo de dato para cada una de las cubetas, compuestas de una clave y un dato asociado.

Implementación en C - Cubetas únicas

```
typedef struct _hashbucket {
  void *key;
  void *data;
} hashbucket;

typedef struct _hashtable {
  hashbucket *table;
  int nelems, size;
  int (*hash)(void *);
} hashtable;
```

Implementación en C - Cubetas únicas



Ejercicios

- Inicializar una hash table con una tabla de 20 cubetas, que opere con una función **int hashf(void *)**.
- Escribir una función **void hash_insert(hashtable *, void *key, void *data)** que agregue un nodo a una hash table dada.

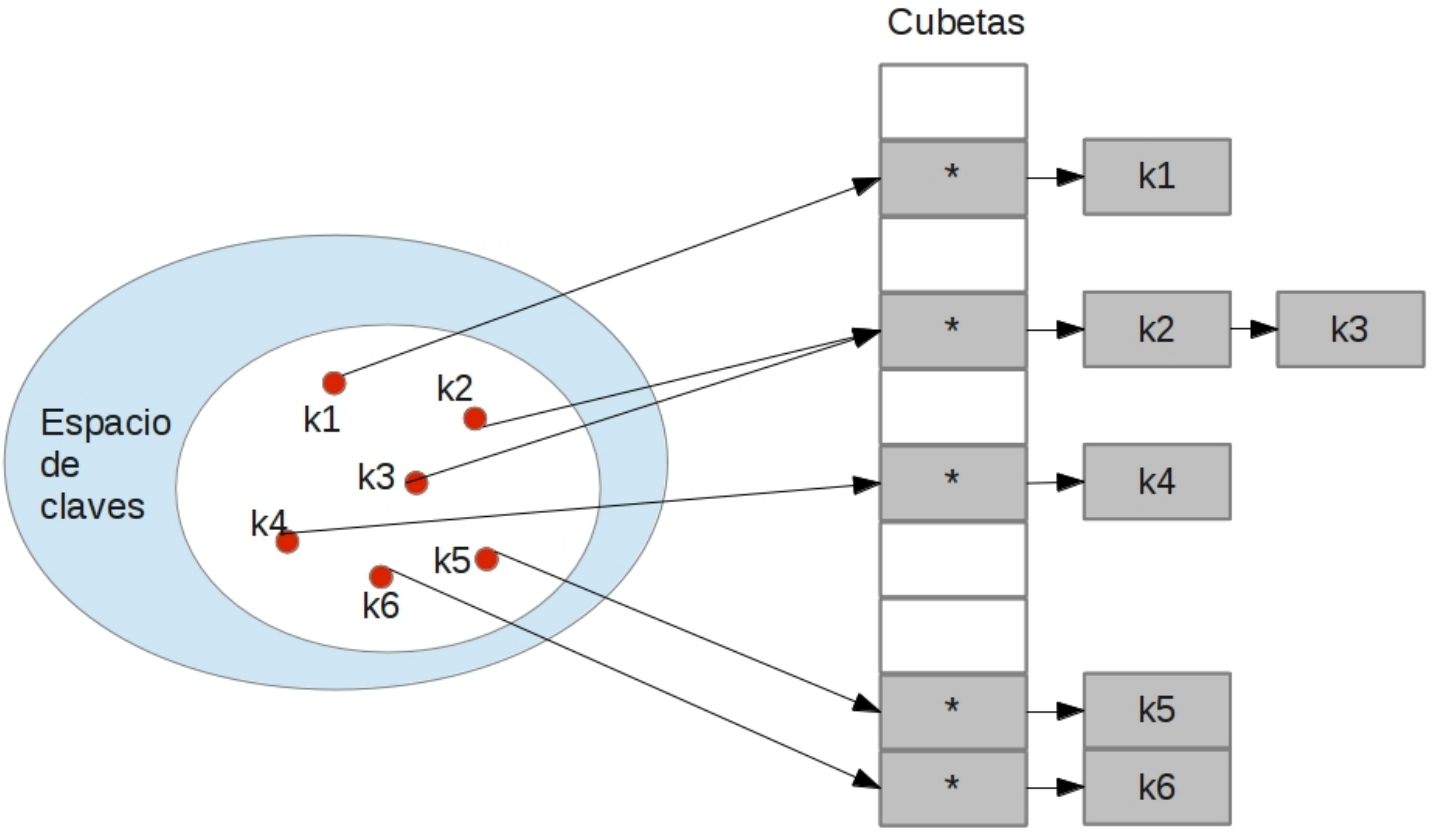
Resolución de colisiones - idea

En general, deberemos proponer una manera de resolver colisiones, en caso de que haya más de una clave con el mismo índice.

Algunas soluciones son:

- No resolverlas (máximo un elemento por índice).
- Encadenado (un puntero a una lista de elementos).
- Fusionado (se buscan slots vacios dentro de la tabla).

Resolución de colisiones - encadenado



Resolución de colisiones - encadenado en C

```
typedef struct _hashbucketnode {
  void *key;
  void *data;
  struct _hashbucketnode *next;
} hashbucketnode;

typedef hashbucketnode *hashbucket;

typedef struct _hashtable {
  hashbucket *table;
  int nelems, size;
  int (*hash)(void *);
} hashtable;
```

Conclusión

- Hemos visto lo que es una hash table: una manera de implementar un arreglo asociativo.
- Las hash tables son más rápidas que almacenar todos los datos en una lista.
- Debemos conseguir funciones hash adecuadas para cada espacio de claves.
- Debemos tener en cuenta cómo resolver las colisiones.