

Hashing

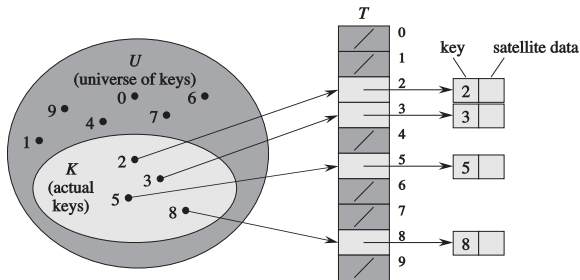
Algoritmos y Estructuras de Datos II

1^{er} cuatrimestre de 2019

Motivación

Queremos implementar un diccionario con sus operaciones de búsqueda, inserción y remoción muy eficientes.

- **Primera idea:** para un diccionario $\text{dicc}(U, V)$ tenemos un arreglo A de largo $|U|$ (tamaño del universo U) donde A_i guarda NULL o el significado del elemento $i \in U$ ¹.



¹Si no son números, podemos usar una *codificación* $U \leftrightarrow \{0, \dots, |U| - 1\}$.

Motivación

Ventajas de nuestra idea:

- ▶ Consultar es $O(1)$.
- ▶ Insertar es $O(1)$.
- ▶ Borrar es $O(1)$.
- ▶ Sencilla de implementar.

Problemas de nuestra idea:

- ▶ Inicializar es $O(|U|)$.
- ▶ Necesitamos $O(|U|)$ memoria, incluso para 'guardar' un solo elemento.

El universo U podría ser enorme dependiendo del contexto. Por ejemplo en un `dicc(int,int)`, o `dicc(string,int)`.

Motivación

Ventajas de nuestra idea:

- ▶ Consultar es $O(1)$.
- ▶ Insertar es $O(1)$.
- ▶ Borrar es $O(1)$.
- ▶ Sencilla de implementar.

Problemas de nuestra idea:

- ▶ Inicializar es $O(|U|)$.
- ▶ Necesitamos $O(|U|)$ memoria, incluso para 'guardar' un solo elemento.

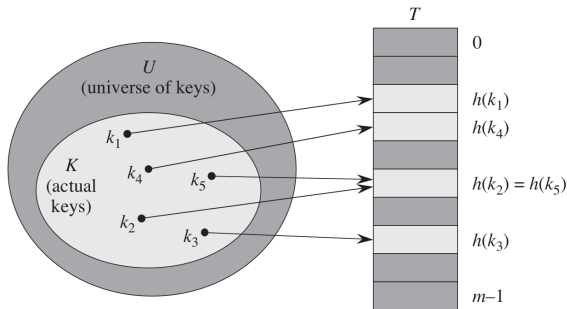
El universo U podría ser enorme dependiendo del contexto. Por ejemplo en un `dicc(int,int)`, o `dicc(string,int)`.

- **Segunda idea:** tabla de hash.

Tablas de hash

Una tabla de hash está basada en los siguientes elementos:

- ▶ Un arreglo T de tamaño m para guardar los datos. Usualmente m es proporcional al número de claves almacenadas.
- ▶ Una función de hash $h : U \rightarrow \{0, \dots, m - 1\}$
- ▶ Un procedimiento para tratar los casos de colisiones.



Colisiones

En una función de hash $h : A \rightarrow B$, una colisión se da cuando dos elementos $x, y \in A$ tienen igual imagen en la función de hash: $h(x) = h(y)$.

En nuestro diccionario sobre tabla de hash, ¿pueden ocurrir colisiones en $h : U \rightarrow \{0, \dots, m - 1\}$?

- ▶ Si nuestra tabla tiene el tamaño del universo de las claves y nuestra función de hash es ‘buena’, podríamos no tener colisiones (*perfect hashing*).
- ▶ Pero esa era nuestra primera idea. En general la tabla no es tan grande como $|U|$.

```
int hash()
```

¿Como codificamos un numero?

int hash()

```
int hashfunction(int n)
    return n % m;
```



```
int hash()
```

¿Y un string?

int hash()

```
int hashfunction(string s)
    return (int(s[0]) % m );
```

```
int hash()
```

¿Que pasa si ingreso strings con el mismo prefijo: 1,11,111?

int hash()

¡Horror!



int hash(): Ejemplo2

```
int hashfunction(Char *s){  
    int i;  
    for( i=0; *s; s++ ) i = 131*i + *s;  
    return( i % m );  
}
```

Notar que usamos un número primo para la codificación, para reducir la probabilidad de colisión

Colisiones

Con nuestra función $h : U \rightarrow \{0, \dots, |T| - 1\}$ podríamos guardar $|T|$ elementos sin colisiones. Si tenemos que guardar más, agrandamos la tabla y cambiamos nuestra función h .

Entonces, ¿es realmente necesario lidiar con colisiones?

Colisiones

Con nuestra función $h : U \rightarrow \{0, \dots, |T| - 1\}$ podríamos guardar $|T|$ elementos sin colisiones. Si tenemos que guardar más, agrandamos la tabla y cambiamos nuestra función h .

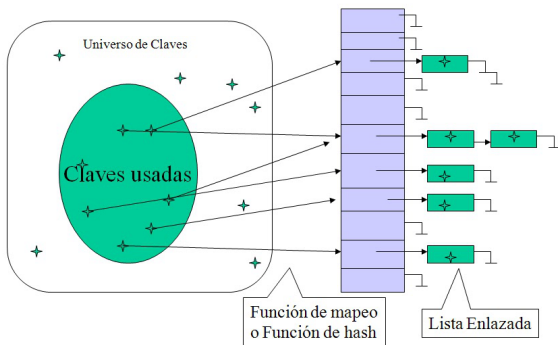
Entonces, ¿es realmente necesario lidiar con colisiones?

En general sí, es necesario:

- ▶ No sabemos por adelantado qué m claves se van a insertar. Peor aún, normalmente ni conocemos la distribución.
- ▶ Dos soluciones usuales para lidiar con colisiones:
 - ▶ Hashing abierto.
 - ▶ Hashing cerrado.

Hashing abierto (*chaining*)

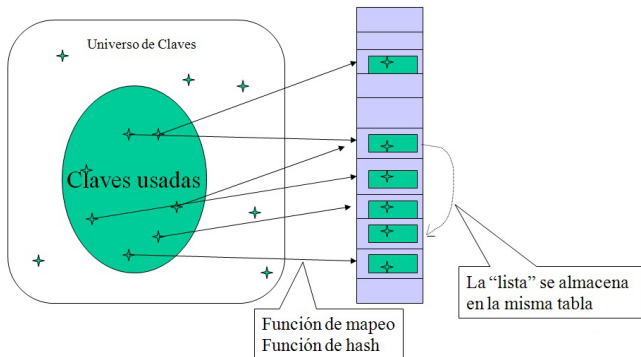
En cada posición de nuestra tabla tenemos una lista enlazada. Los elementos que tienen igual imagen en h se guardan en la misma lista.



► ¿Complejidades de las operaciones?

Hashing cerrado (*probing*)

Dada una clave $k \in U$, no trabajamos con un único hash $h(k)$, sino una secuencia $[h_1(k), \dots, h_i(k), \dots, h_m(k)]^2$, que vamos probando hasta encontrar un casillero libre.



► ¿Complejidades de las operaciones?

²Probe sequence.

Hashing cerrado (*probing*)

Para obtener una secuencia de probing hay 3 técnicas usuales:

- ▶ *Linear probing*:

$$h(k, i) = (h_i(k) + c.i) \mod m$$

- ▶ *Quadratic probing*:

$$h(k, i) = (h_i(k) + c_1.i + c_2.i^2) \mod m$$

- ▶ *Double hashing*:

$$h(k, i) = (h_{i1}(k) + i.h_{i2}(k)) \mod m$$

- ▶ Donde $[c_j]$ sera el tamaño del salto j

La secuencia para una clave k será $[h(k, 0), \dots, h(k, m - 1)]$.

Factor de carga

Sobre el tamaño de nuestra tabla...

- ▶ ¿Qué pasa si es muy chico? (Caso límite: $|T| = 1$).
- ▶ ¿Qué pasa si es muy grande? (Caso límite: $|T| = |U|$).

Definimos el factor de carga como: $\alpha = \frac{n}{|T|}$

Donde n es el número de elementos guardados y $|T|$ el tamaño de la tabla.

- ▶ Cuando el factor de carga es grande, agrandamos la tabla.
- ▶ Cuando el factor de carga es chico, achicamos la tabla.

¿Cuáles serían buenas cotas para α ?

Otros usos de funciones de hash

Las funciones de hash sirven para muchas cosas...

- ▶ Diccionarios sobre hash.

Otros usos de funciones de hash

Las funciones de hash sirven para muchas cosas...

- ▶ Diccionarios sobre hash.

- ▶ Criptografía.

Otros usos de funciones de hash

Las funciones de hash sirven para muchas cosas...

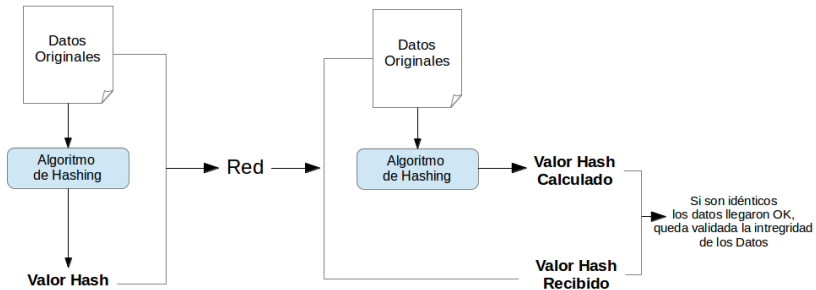
- ▶ Diccionarios sobre hash.
- ▶ Criptografía.
- ▶ Detección de errores, integridad de datos.

Otros usos de funciones de hash

Las funciones de hash sirven para muchas cosas...

- ▶ Diccionarios sobre hash.
- ▶ Criptografía.
- ▶ Detección de errores, integridad de datos.
- ▶ Comparación de datos.

Otros usos de funciones de hash



Taller de hoy

- ▶ Implementar un diccionario de claves string sobre una estructura que utilice *hashing abierto (chaining)*.
- ▶ Cuando el *factor de carga* supere cierto valor, debemos redimensionar el vector.
- ▶ Debemos implementar: definido?, definir, significado, borrar... *redimensionar* y *hash*.

FAQ

Pregunta 1

- ▶ ¿Qué es redimensionar? ¿cuándo y cómo lo invocamos?

Pregunta 1

- ▶ ¿Qué es redimensionar? ¿cuándo y cómo lo invocamos?

Respuesta

- ▶ Redimensionar duplica la longitud del vector y vuelve a insertar todos los valores.
- ▶ Se invoca cuando *definimos* una nueva clave se corrobora que no se supere el umbral predefinido.

FAQ II

Pregunta 2

- ▶ ¿Cómo utilizamos la función de hash?

Pregunta 2

- ▶ ¿Cómo utilizamos la función de hash?

Respuesta

- ▶ Cada vez que queremos determinar la posición de una clave en el vector:

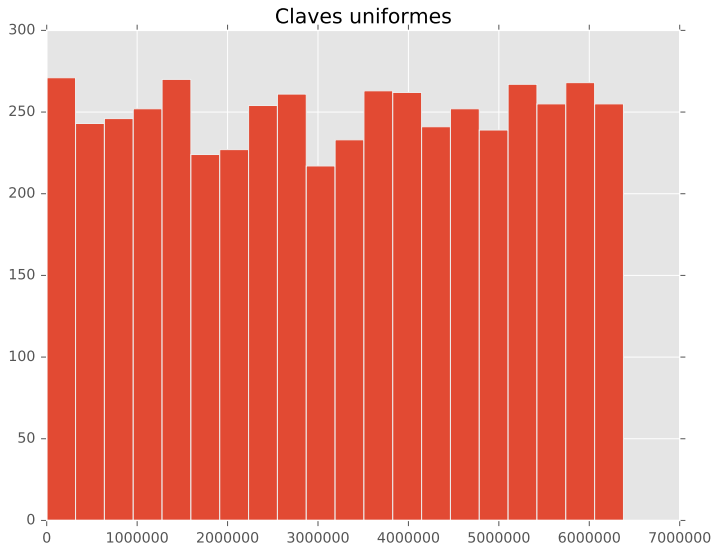
$$\text{posición} = \text{fn_hash}(\text{clave}) \bmod |T|$$

- ▶ Cormen et al. *Introduction to Algorithms*. Sección 11.3.
- ▶ Knuth. *The Art of Computer Programming*. Sección 6.4.

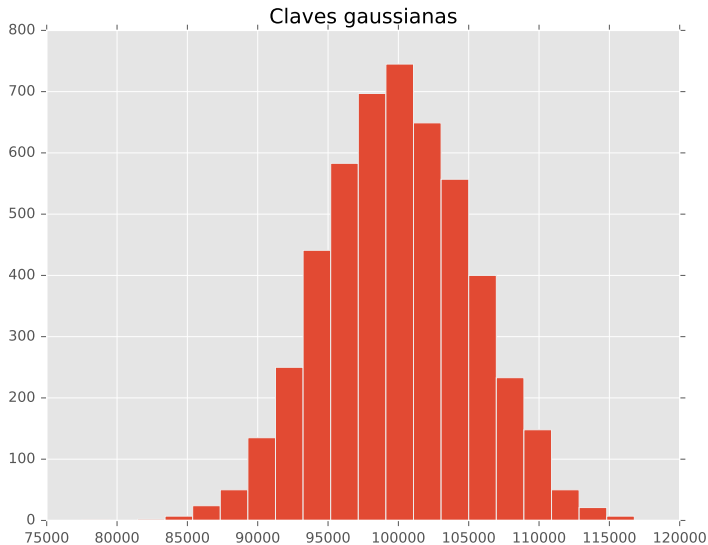
Más

- ▶ El testing también evaluará la performance de la estructura.
- ▶ Inserciones masivas con distintas distribuciones.

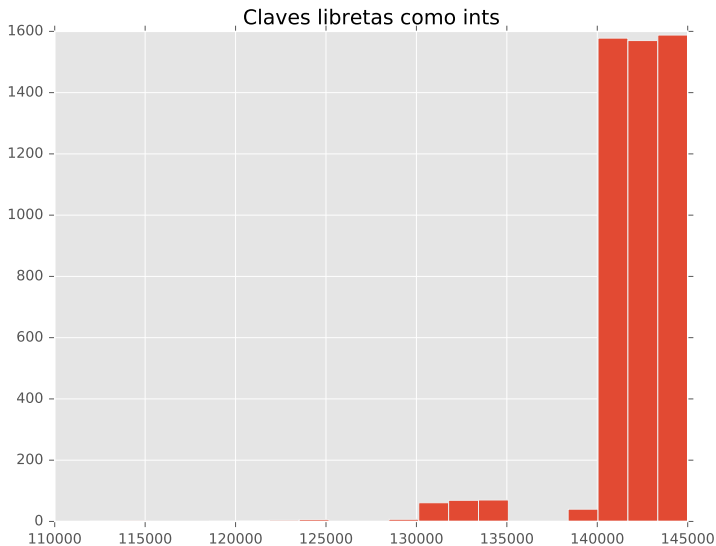
Testing (X: cant. buckets, Y: cant. claves)



Testing (X: cant. buckets, Y: cant. claves)



Testing (X: cant. buckets, Y: cant. claves)



¿Preguntas?