

Estructuras de Datos

Tablas hash

Funciones hash, encadenamiento y lo que puede salir mal

Gerson Garrido

Semana 7

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. El diccionario, y lo que cuesta con lo que ya sabemos
2. De una clave grande a un índice pequeño
3. Colisiones: son inevitables
4. Función hash: sumar falla, FNV-1a funciona
5. Encadenamiento y factor de carga
6. Crecer: duplicar y reinsertar todo
7. El peor caso, provocado a propósito
8. `std::unordered_map` y `std::hash`
9. ¿Tabla hash o arreglo ordenado?

> **Lectura**
>
> Apuntes de la semana 7. Código en `code/week-07`.

SECCIÓN 01

El diccionario

01

Guardar, buscar y borrar por clave

Diccionario *(dictionary, map)*

Tipo abstracto de datos que guarda pares (clave, valor), con claves que no se repiten.

- `set(k, v)`: inserta o sobrescribe
- `get(k)`: el valor, o avisa que no está
- `contains(k)`
- `remove(k)`
- `size()`

Ya usaron uno: la tabla de variables `Env` del intérprete de Introducción a la Programación.

`x = 3` guarda con la clave `"x"`. `x + 1` la busca.

La usaron por su encabezado. Hoy la construyen.

Con lo que ya sabemos

Representación	`get`	`set` clave nueva	`remove`
Arreglo sin orden	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Lista enlazada	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Arreglo ordenado	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Tabla hash, esperado	$O(1)$	$O(1)$	$O(1)$

¿Por qué `set` es $\Theta(n)$ aun al final de un arreglo?

Un diccionario no repite claves: antes de insertar, hay que buscar.

SECCIÓN 02

De la clave al índice

02

Cuando la clave es el índice

Variables de una sola letra: un arreglo de 26 casillas.



- Guardar y buscar: $\Theta(1)$, sin comparar ninguna clave. Es una **tabla de acceso directo**.
- Claves de 10 bytes: $256^{10} \approx 1.2 \times 10^{24}$ valores posibles. No hay memoria.
- Y un diccionario del español tiene cerca de un millón de palabras.

La salida

Separar las claves **posibles** de las que **de verdad hay**. Un arreglo proporcional a n, y una función que traduzca.

Función hash y tabla hash

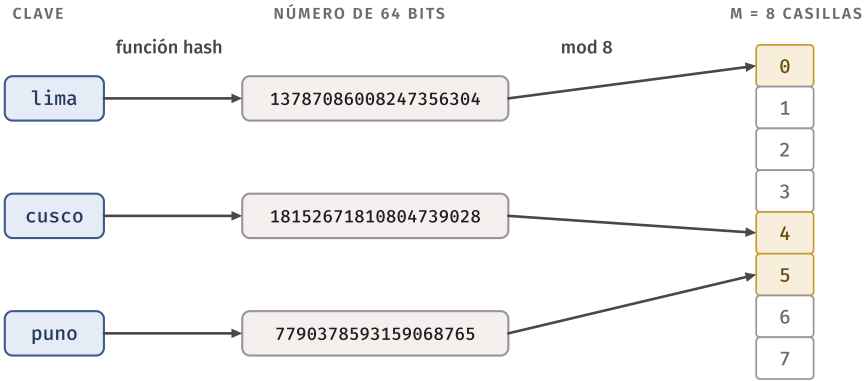
Función hash ***(hash function)***

Recibe una clave y devuelve un número. Una **tabla hash** es un arreglo de m casillas en el que la clave k se guarda en la casilla `h(k) mod m`.

Idea clave

La búsqueda binaria *descarta* comparando. La tabla hash *calcula* dónde debería estar la clave.

De la clave a la casilla



SECCIÓN 03

Colisiones

03

Colisiones: dos claves, una casilla

Colisión ***(collision)***

Dos claves distintas caen en la misma casilla.

- **Principio del palomar:** si hay más claves posibles que casillas, alguna casilla se repite.
- Ni los 64 bits lo evitan: hay más cadenas que números de 64 bits.
- Pero ¿y si guardamos muchas menos claves que casillas?

La primera colisión llega muy pronto

terminal

```
$ build/first_collision ../../data/palabras.txt
m           mediana  raiz(m)  repeticiones
1000        33       32       51
100000      381      316      51
1000000     1082     1000     51
10000000    3910     3162     51
```

- Un millón de casillas: colisión tras unas mil palabras, con la tabla al 0.1 %.
- Paradoja del cumpleaños: crecen las **parejas**, $k(k-1)/2$. Cada una choca con probabilidad $1/m$.
- Cuando k es del orden de $\sqrt{m}$, ya es de esperar una.

Decisión de diseño: no evitarlas, sino lograr que sean pocas y que cuesten poco.

SECCIÓN 04

Una buena función hash

04

Qué pedirle a una función hash

1. **Determinista.** Misma clave, mismo número.
2. **Rápida.** Se calcula en cada operación: $\Theta(L)$ con L bytes.
3. **Que reparta bien.** Como si cada clave eligiera su casilla al azar.

La regla que no se puede romper

Si dos claves son iguales, su hash debe ser igual. Lo contrario no se pide: eso es una colisión, algo normal.

`h(k) = 0` es determinista, rapidísima y **correcta**. Solo que la tabla es una lista. Reparte mal: lenta. No cumple la regla: pierde datos.

La primera idea: sumar los bytes

hash_functions.h

```cpp
inline std::uint64_t sum_hash(const std::string& key) {
    std::uint64_t hash = 0;
    for (char c : key) {
        hash += static_cast<unsigned char>(c);
    }
    return hash;
}
```

Lo bueno

Es determinista y rápida. ¿Cómo reparte?

La suma no ve el orden

terminal

```
$ build/hash_demo
palabra  suma  suma%8  fnv1a             fnv1a%8
amor     431   7       8c778a83af1454b2  2
roma     431   7       a35c141ff0caad88  0
mora     431   7       0d1beba2c400c74c  4
ramo     431   7       6daffc1fd1e8c544  4
ad       197   5       089c3e07b5458e38  0
bc       197   5       08a63507b54dd372  2
```

No ve el orden: todo anagrama colisiona. Tampoco hace falta ser anagrama.

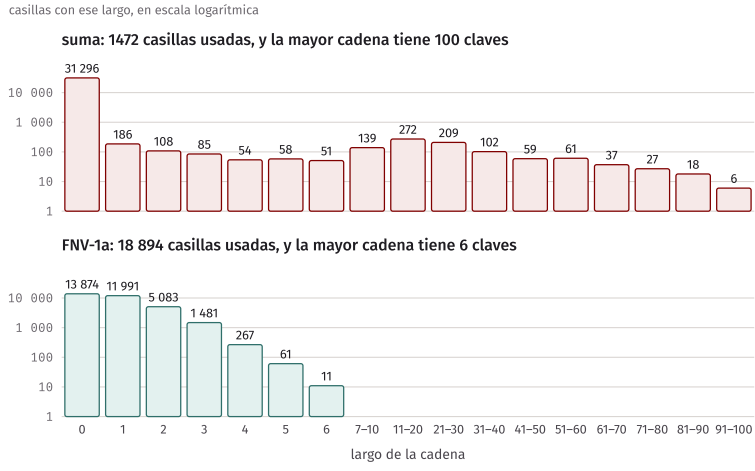
El defecto grave: da números pequeños

terminal

```
$ build/hash_compare ../../data/terminos.tsv
28039 palabras; la suma da 1472 valores distintos, de 97 a 2212
hash           m   alfa    usadas  la_mayor comp_acierto
suma        1024  27.38      1021       100        21.7
fnv1a       1024  27.38      1024        49        14.7
suma       32768   0.86      1472       100        20.7
fnv1a      32768   0.86     18894         6         1.4
suma     1048576   0.03      1472       100        20.7
fnv1a    1048576   0.03     27642         3         1.0
```

- Un millón de casillas: la suma usa 1472, y sigue con una cadena de 100.
- Agrandar no ayuda: todas las sumas caben en las primeras 2212 casillas.
- Y con pocas casillas, FNV-1a tampoco se salva: 14.7 comparaciones. La tabla tiene que ser proporcional a n.

El largo de las cadenas, con m = 32 768



FNV-1a

hash_functions.h

```cpp
inline std::uint64_t fnv1a(const std::string& key) {
    std::uint64_t hash = 14695981039346656037ULL;
    for (char c : key) {
        hash ^= static_cast<unsigned char>(c);
        hash *= 1099511628211ULL;
    }
    return hash;
}
```

- Para cada byte: XOR, y multiplicar por un primo. Constantes oficiales de sus autores.
- El overflow de un entero **sin signo** está bien definido: da la vuelta módulo 2^{64}. Con signo, sería comportamiento indefinido.
- Se prueba con vectores oficiales: `"a"` da `af63dc4c8601ec8c`, `"foobar"` da `85944171f73967e8`.
- Sirve para una tabla, no para seguridad: no es criptográfica.

FNV-1a, byte por byte

terminal

```
$ build/fnv_trace
clave "ab"
  inicio         cbf29ce484222325
  xor  61        cbf29ce484222344
  por el primo   af63dc4c8601ec8c
  xor  62        af63dc4c8601ecee
  por el primo   089c4407b545986a
clave "ba"
  inicio         cbf29ce484222325
  xor  62        cbf29ce484222347
  por el primo   af63df4c8601f1a5
  xor  61        af63df4c8601f1c4
  por el primo   08a63307b54dd00c
```

- El XOR cambia solo los bits bajos.
- La multiplicación reparte el cambio hacia los altos.
- Cada byte se mezcla con todo lo anterior: el orden importa.
- Lo que le faltaba a la suma: ve el orden y usa los 64 bits.

Bytes, tildes y `char`

`std::string` guarda bytes: `ñandú` son 5 letras y 7 bytes. Para hashear está bien, pero `char` puede tener signo o no, según la plataforma.

Error común

Olvidar `unsigned char`. Una tilde es un byte mayor que 127: con signo, vale negativo.

terminal

```
$ g++ -std=c++20 -funsigned-char errors/signed_char_hash.cpp -o build/sc1 && build/sc1
bad_sum("lima") = 419
bad_sum("ñandú") = 1060
$ g++ -std=c++20 -fsigned-char errors/signed_char_hash.cpp -o build/sc2 && build/sc2
bad_sum("lima") = 419
bad_sum("ñandú") = 36
```

Las pruebas del laboratorio usan `-fsigned-char`, para que se note en cualquier máquina.

SECCIÓN 05

Encadenamiento

05

Encadenamiento: una lista por casilla

Encadenamiento ***(chaining)***

Cada casilla guarda una lista enlazada, su **cadena**, con las claves que caen ahí.

La representación

chain_table.h

```
1  struct Node {
2      std::string key;
3      V value;
4      Node* next;
5  };
6
7  Node** slots_;
8  std::size_t capacity_;
9  std::size_t size_ = 0;
10 HashFunction hash_;
```

- `new Node*[capacity]()`: los `()` dejan todos los punteros nulos.
- Con un puntero a función se prueba la misma tabla con las dos funciones.

Invariantes

Cada nodo de la cadena i cumple `hash_(key)% capacity_ == i`.
Una clave aparece en un solo nodo.
`size_` es el total de nodos.

Buscar

chain_table.h

```
1  Node* find_node(const std::string& key) const {
2      for (Node* node = slots_[slot_of(key)]; node != nullptr; node = node->next) {
3          if (node->key == key) {
4              return node;
5          }
6      }
7      return nullptr;
8  }
```

Idea clave: el hash solo dice **dónde mirar**. Por eso el nodo guarda la clave completa, y por eso se compara.

Guardar: dos caminos

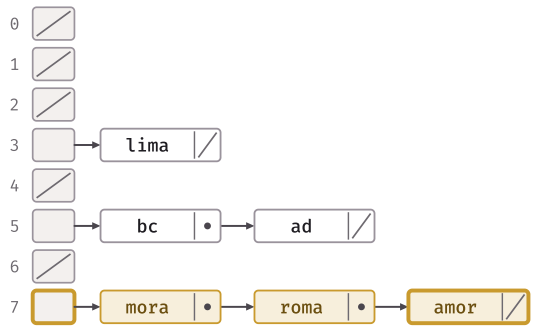
chain_table.h

```
1  void set(const std::string& key, const V& value) {
2      Node* node = find_node(key);
3      if (node != nullptr) {
4          node->value = value;
5          return;
6      }
7      std::size_t slot = slot_of(key);
8      slots_[slot] = new Node{key, value, slots_[slot]};
9      size_++;
10 }
```

- Si ya está: cambia el valor, y `size_` no se mueve.
- Si no: el mismo `push_front` de la semana 5.

Llenar una tabla y buscar en ella

```
get(ramo)   suma 431   431 mod 8 = 7                    comparaciones: 3
```



ramo tiene la misma suma, y no está: 3 comparaciones. Un fallo mira toda la cadena

Con la suma de bytes.

Tres búsquedas en la misma cadena

La casilla 7 tiene `mora`, `roma` y `amor`, en ese orden.

- `mora`: está primero. **1 comparación.**
- `amor`: está al final. **3 comparaciones.** Es un acierto.
- `ramo`: su suma es 431, y no está. **3 comparaciones.** Un fallo siempre mira la cadena completa.
- Una casilla vacía responde “no está” con 0 comparaciones.

Borrar: sacar un nodo de una lista simple

pseudocódigo

```
algoritmo borrar(tabla, clave)
    i ← h(clave) mod m
    anterior ← NULO
    nodo ← tabla[i]
    mientras nodo ≠ NULO hacer
        si nodo.clave = clave entonces
            si anterior = NULO entonces
                tabla[i] ← nodo.siguiente
            sino
                anterior.siguiente ← nodo.siguiente
            fin si
            liberar nodo
            devolver verdadero
        fin si
        anterior ← nodo
        nodo ← nodo.siguiente
    fin mientras
    devolver falso
fin algoritmo
```

- Dos casos: si no es el primero, cambia el `next` del anterior. Si es el primero, cambia la **casilla**.
- Una casilla es un puntero, igual que un `next`: un puntero a puntero unifica los dos casos.

Error común

Olvidar el primer nodo: la casilla queda apuntando a memoria liberada.

Borrar, paso a paso



Se libera mora. Queda amor, y la casilla apunta a ese nodo

SECCIÓN 06

Factor de carga

06

El factor de carga

Factor de carga *(load factor)*

α = n/m: el largo promedio de las cadenas, contando las vacías.

Suposición: cada clave cae en cualquier casilla con igual probabilidad. Es un modelo: la probabilidad formal viene en otro curso.

- **Fallo:** la cadena tiene en promedio α claves, y se miran todas: α.
- **Acierto:** ella misma, más las que tiene delante, en promedio la mitad: cerca de 1 + α/2.

Idea clave

O(1 + α) esperado: 1 por la casilla, α por la cadena. Con α acotado, O(1).

Contando comparaciones de verdad

terminal

```
$ build/load_experiment ../../data/palabras.txt
     m       n   alfa   acierto  1+alfa/2     fallo      alfa
   100     300   3.00      2.43      2.50      3.07      3.00
  1024     512   0.50      1.26      1.25      0.46      0.50
  1024    1024   1.00      1.49      1.50      1.06      1.00
  1024    2048   2.00      2.03      2.00      1.99      2.00
  1024    5120   5.00      3.52      3.50      5.02      5.00
  1024  102400 100.00     51.00     51.00    100.00    100.00
```

- Son conteos: no dependen de la máquina.
- **100 casillas, 300 claves:** α = 3. Fallo: unas 3 comparaciones. Acierto: unas 2.5.
- Con α chico, fallar cuesta *menos* que acertar: muchas veces cae en una casilla vacía.
- Repartir al azar no es repartir parejo: cerca de $e^{-\alpha}$ de las casillas quedan vacías.

Dos promedios distintos

Costo esperado

Promedio sobre cómo caen las claves, **suponiendo** que la función reparte bien.

Una función mala o un adversario lo rompen.

Costo amortizado

Promedio sobre una secuencia de operaciones, **sin suposiciones** ni azar.

El de la semana 4. Vuelve a salir en un momento.

SECCIÓN 07

Crecer

07

Crecer: otra vez duplicar

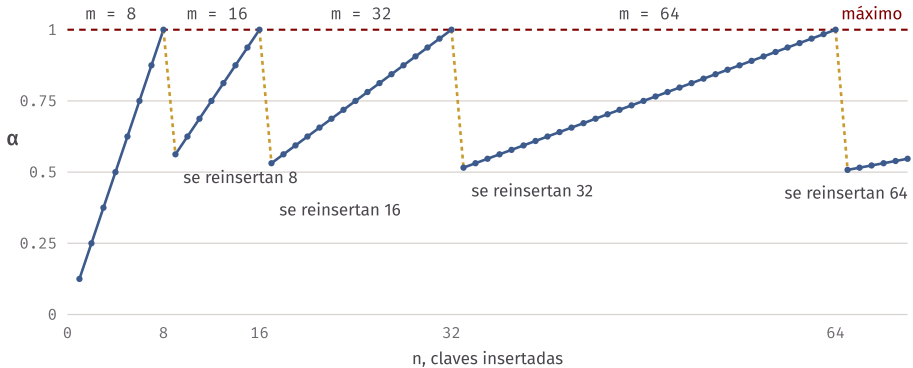
Si m queda fijo y n crece, α crece y el costo también. Regla: si una clave **nueva** superaría el máximo, primero se duplica.

n	m	α	qué pasó
8	8	1.0	Justo en el máximo: no crece
9	16	0.5625	9/8 superaría 1: se reinsertan 8 y entra la nueva
16	16	1.0	Otra vez en el máximo
17	32	0.53125	Se reinsertan 16

Error común

Crecer al sobrescribir. Una clave que ya está no aumenta n: primero se busca, y se crece solo si es nueva.

α mientras se inserta



Superadas las primeras casillas, α se mantiene entre 0.5 y 1

¿Basta con copiar las casillas?

terminal

```
$ build/naive_copy ../../data/terminos.tsv
lima: hash % 8 = 0, hash % 16 = 0
cusco: hash % 8 = 4, hash % 16 = 4
tacna: hash % 8 = 6, hash % 16 = 6
puno: hash % 8 = 5, hash % 16 = 13
arequipa: hash % 8 = 1, hash % 16 = 1
con 1024 casillas: 28039 de 28039 encontradas
copiando a 2048:   13892 de 28039 encontradas
reinsertando:      28039 de 28039 encontradas
```

- Copiar pierde la mitad. Siguen en la memoria, y `get` las busca en otro lugar.
- ¿Por qué la mitad? Con m potencia de 2, al duplicar se mira un bit más. Si es 1, el índice aumenta en el m antiguo: 5 + 8 = 13.

Idea clave: cada clave está donde `h(k) mod m` lo decía, y m acaba de cambiar.

Reinsertar sin copiar

pseudocódigo

```
algoritmo crecer(tabla)
    nueva ← arreglo de 2 × m punteros NULO
    para i desde 0 hasta m - 1 hacer
        nodo ← tabla[i]
        mientras nodo ≠ NULO hacer
            siguiente ← nodo.siguiente
            j ← h(nodo.clave) mod (2 × m)
            nodo.siguiente ← nueva[j]
            nueva[j] ← nodo
            nodo ← siguiente
        fin mientras
    fin para
    liberar tabla
    tabla ← nueva
    m ← 2 × m
fin algoritmo
```

- Los nodos ya existen: solo se vuelven a **enlazar**.
- Se guarda `siguiente` antes de tocar el nodo.
- Ningún nodo se mueve: un puntero de `get` sigue válido al crecer. Muy distinto de `std::vector`.

De una cadena salen, a lo sumo, dos



Se libera el arreglo antiguo. De la cadena 0 salieron dos: la 0 y la 4 = 0 + 4

Crecer nunca mezcla cadenas distintas, y el orden dentro de una cadena no importa.

El costo: otra vez amortizado

terminal

```
$ build/growth_count
         n   casillas reinsertadas por_insercion
      1000       1024          1016         1.02
     10000      16384         16376         1.64
   1000000    1048576       1048568         1.05
   1048577    2097152       2097144         2.00
```

- Una inserción que crece cuesta $\Theta(n)$. El total: $8 + 16 + \ldots + m/2 = m - 8$.
- Si creció hasta m, había más de $m/2$ claves: $m < 2n$. Menos de 2 reinserciones por clave.
- El 2.00 es el peor momento, redondeado: en realidad 1.99999.

`set` es O(1) **amortizado** (una sola puede costar $\Theta(n)$) y **esperado** (la cadena depende de la función).

SECCIÓN 08

El peor caso

08

El peor caso sigue siendo $\Theta(n)$

- Todas las claves en una casilla: la tabla es una lista.
- Buscar cuesta $\Theta(n)$. Construirla, $\Theta(n^2)$.
- Por casualidad, con FNV-1a: muy improbable.
- A propósito: fácil. La función es pública y no tiene secreto.

La receta

Probar `k0`, `k1`, `k2`, … y quedarse con las claves cuyo hash sea múltiplo de 8192. Caen en la casilla 0 de toda tabla con $m \leq 8192$ potencia de 2: **crecer no sirve de nada**.

Claves elegidas

terminal

```
$ build/flood
4000 claves; la ultima elegida es k32475944
normales   m=4096  la mayor cadena     5, comparaciones por busqueda      1.4
elegidas   m=4096  la mayor cadena  4000, comparaciones por busqueda   2000.5
elegidas   m=8192  la mayor cadena  4000, comparaciones por busqueda   2000.5
unordered_map con reserve(4000): bucket_count 4027
normales   la casilla de keys[0] tiene     1,      0.23 ms
elegidas   la casilla de keys[0] tiene  4000,     19.66 ms
```

- Conteos exactos: de 1.4 a 2000.5 comparaciones por búsqueda.
- Tiempos con `-O2`, mediana de 5; dependen de la máquina. En cuatro corridas, entre 84 y 92 veces más.
- Las claves se eligieron para ese `bucket_count`, que el programa conocía por usar `reserve`.

No es teoría

- Crosby y Wallach, *Denial of Service via Algorithmic Complexity Attacks*.
- Aviso oCERT-2011-003: varios lenguajes a la vez, con especial exposición en servidores web.
- Un servidor guarda los parámetros de una petición en una tabla. El atacante manda miles de nombres elegidos.

La defensa: mezclar en el hash un secreto elegido al arrancar. Python lo hace con las cadenas:

terminal

```
$ python3 -c "print(hash('lima'))"
-5712963166023579098
$ python3 -c "print(hash('lima'))"
3007673079197261O
```

Idea clave: O(1) esperado no es O(1). Un adversario puede romper la suposición a propósito.

SECCIÓN 09

En la biblioteca

09

`std::unordered_map`

Diccionario	`std::unordered_map`
`set(k, v)`	`m[k] = v`, o `m.insert_or_assign(k, v)`
`get(k)`	`m.find(k)`: un iterador, o `end()`
`contains(k)`	`m.contains(k)`, desde C++20
`remove(k)`	`m.erase(k)`: cuántos borró, 0 o 1
`size()`	`m.size()`

Cada elemento es un `std::pair<const Key, Value>`, con `first` y `second`.

Error común

Preguntar con `m[k]`: si la clave no está, **la inserta**. Para preguntar, `contains` o `find`.

El estándar, y esta biblioteca

El estándar pide

- Constante en promedio, lineal en el peor caso.
- `max_load_factor()` empieza en 1.0.
- Reorganizar invalida iteradores, no punteros ni referencias a elementos.
- Orden de recorrido sin especificar.
- `std::hash`: claves iguales, hash igual, **en una misma ejecución**.

libstdc++, en este entorno

- `_Hash_bytes`, con semilla **fija**: sin secreto.
- Todos los nodos en una sola lista simple.
- Tamaños primos: 13, 29, 59, 127…

Lo de la derecha puede cambiar con otra biblioteca o versión. No dependan de ello.

Lo que muestra un programa

terminal

```
$ build/library_demo
al inicio: bucket_count 1, max_load_factor 1
crece en n = 1 14 30 60 128 258 542 1110
bucket_count: 1 13 29 59 127 257 541 1109 2357
con n = 2000: load_factor 0.848536
orden: ica tacna puno cusco piura lima
tras rehash(100), bucket_count 103: lima piura cusco puno tacna ica
contains("ica") 1, contains("nazca") 0, size 6
cities["nazca"] 0, size 7
```

- La misma regla: con 13 casillas cabe la clave 13, y la 14 hace crecer.
- **Por qué las claves no salen ordenadas:** la posición la decide el hash, y un buen hash no tiene relación con el orden.
- El orden puede cambiar sin tocar ninguna clave. No lo usen, ni en una prueba.

Error común: modificar una clave en la tabla

```
terminal

$ build/mutated_key
invariantes: 1
size: 1
get("lima"): no está
get("cusco"): no está
invariantes: 0
```

Cambiar `lima` por `cusco` a través de un puntero.

- El dato sigue ahí, `size` lo cuenta, y nadie lo encuentra: está en una casilla que no le corresponde.
- Ningún sanitizer se queja: la memoria está bien, y la tabla, mal. Solo el invariante lo ve.
- Por eso en la biblioteca la clave es `const`, y `entry.first = "cusco"` no compila.
- Para cambiar una clave: borrar, e insertar con la nueva.

SECCIÓN 10

¿Tabla o arreglo ordenado?

10

El mismo diccionario, dos estructuras

terminal

```
$ build/lookup_bench ../../data/palabras.txt
1045757 palabras, 200000 consultas
                        ns/consulta    comparaciones         MB
arreglo ordenado                510             20.0       35.2
std::unordered_map              165              1.4       71.8
empiezan con cas: 1537, de cas a casó
la tabla debe revisar 1045757 claves para hallar las mismas 1537
```

- `-O2`, mediana de 5; los tiempos dependen de la máquina. En cuatro corridas, la tabla fue entre 2.4 y 3.1 veces más rápida.
- Mucho menos que 14 a 1: paga el hash, y probablemente los saltos por la memoria. No lo medimos.
- El arreglo ocupa cerca de la mitad.

Cuándo prefieren un arreglo ordenado

Arreglo ordenado

- Rangos, prefijos, mínimo, máximo, siguiente.
- Datos que casi no cambian.
- Importa la memoria.
- Peor caso garantizado: $\Theta(\log n)$, sin suposiciones.

Tabla hash

- Clave exacta.
- Los datos cambian mucho.
- El orden no importa.

Idea clave

Una tabla hash solo responde “¿qué valor tiene exactamente esta clave?”. Una buena función destruye toda relación entre claves parecidas.

“Orden” significa orden de bytes

terminal

```
antes de la primera b: aúpo; ultima antes de la primera con a acentuada: zúñiga; primera: ábaco
```

- `std::string` compara bytes. En UTF-8, `á` empieza con 0xC3, mayor que cualquier letra sin tilde.
- `ábaco` va después de todas las palabras con `z`. El prefijo `a` no la encuentra.
- `casó` sí entra en el prefijo `cas`: su tilde viene después del prefijo.
- Para el orden del idioma, hay que comparar con otra función.

SECCIÓN 11

Laboratorio

11

Una sola sesión: 2 horas

La otra sesión de la semana es para la **Práctica 2**. Por eso hay más código dado.

1. **`HashMap<V>`** (60 min). Dados: `Node`, regla de cinco, `check_invariants`. Ustedes: `fnv1a`, `find_node`, `set`, `get`, `contains`, `remove`, `grow`.
2. **Al azar contra `std::unordered_map`, con ASan** (15 min). Ya están escritas: entender y corregir.
3. **Tabla de variables** (15 min). El evaluador está dado: faltan cuatro métodos de `VarTable`.
4. **Largo de las cadenas** (25 min). `sum_hash`, `chain_lengths`, predicción y `lab/report.md`.

Dos cuidados

`grow` no crea nodos: las pruebas cuentan reservas y copias. Tras un movimiento, `capacity()` es 0: cuidado con `% 0`.

Para correrlo

terminal

```
make test-map
make test-calc
make test-hash
make test-lab
make report
```

- Cada función sin hacer detiene el programa y dice cuál falta.
- Con ASan y UBSan: una fuga hace fallar la prueba.
- En la semana 9 copiarán su `hash_map.h`: déjenlo limpio.

Para la próxima clase

- Laboratorio: la tabla de variables, ahora como tabla hash.
- Práctica 2 en la otra sesión: arreglos dinámicos, colas, listas enlazadas, deques y ordenamiento.
- La semana 8 es el examen parcial. Después, en la semana 9, otra manera de resolver las colisiones.

Pregunta

Sin nodos ni punteros, ¿dónde podría ir una clave que encuentra su casilla ocupada? ¿Y qué pasa con una búsqueda si después se borra alguna clave en el camino?