

Estructuras de Datos

Direccionamiento abierto

Linear probing, tombstones, Swiss Tables y el dict de CPython

Gerson Garrido

Semana 9

UNIVERSIDAD NACIONAL DE INGENIERÍA

Hoy

1. Todo dentro del arreglo
2. Linear probing y agrupamiento primario
3. Borrar: tombstones y reconstruir
4. Encadenamiento frente a linear probing: lo que dicen el conteo y el reloj
5. Caso de estudio: Swiss Tables
6. Caso de estudio: el `dict` de CPython
7. Qué cambió y qué no
8. Laboratorio: corrector ortográfico

> **Lectura**
>
> Apuntes de la semana 9. Código en `code/week-09`.

SECCIÓN 01

Todo dentro del arreglo

01



La decisión que cambiamos

Encadenamiento (semana 7)

- Una lista por casilla.
- Cada clave: un nodo, una reserva, un puntero.
- α puede pasar de 1.

Direccionamiento abierto *(open addressing)*

- Una clave por casilla, todo en el arreglo.
- Sin nodos ni punteros.
- $\alpha \leq 1$, siempre.

Secuencia de probing

El orden en que se prueban las casillas para una clave: $s(k, 0), s(k, 1), s(k, 2), \ldots$ Insertar la recorre hasta una casilla libre. Buscar, hasta la clave o una casilla vacía.

Dos consecuencias inmediatas

- Una clave por casilla: $n \leq m$, así que α no pasa de 1.
- Toda búsqueda sin éxito termina en una casilla vacía.
- Por tanto, debe quedar al menos una casilla vacía. Si no, la búsqueda no termina.

Lo único que cambia

La misma función hash, las mismas m casillas, el mismo tipo abstracto. Las variantes solo se distinguen por su secuencia de probing.

SECCIÓN 02

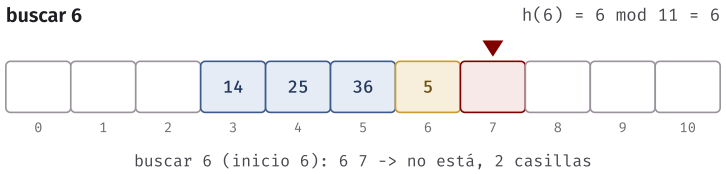
Linear probing

02

Linear probing

Definición

$s(k, i) = (h(k) + i) \bmod m$. Si la casilla inicial está ocupada, se prueba la siguiente, y al final se da la vuelta.



En la 6 hay otra clave, y en la 7 ya no hay nada: el 6 no está

El programa muestra cada paso

terminal

```
$ ./build/probe_trace 11 i14 i25 i36 i5 f36 f47 f6
insertar 14 (inicio 3): 3 <- aquí
insertar 25 (inicio 3): 3 4 <- aquí
insertar 36 (inicio 3): 3 4 5 <- aquí
insertar 5 (inicio 5): 5 6 <- aquí
buscar 36 (inicio 3): 3 4 5 -> está, 3 casillas
buscar 47 (inicio 3): 3 4 5 6 7 -> no está, 5 casillas
buscar 6 (inicio 6): 6 7 -> no está, 2 casillas
tabla: [0:_] [1:_] [2:_] [3:14] [4:25] [5:36] [6:5] [7:_] [8:_] [9:_] [10:_]
```

Cómo contamos

Casillas revisadas, incluida la última: la de la clave o la vacía donde termina.

Buscar

pseudocódigo

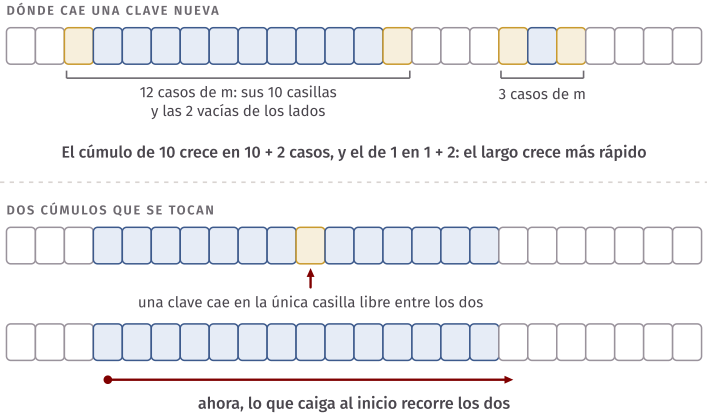
```
algoritmo buscar(tabla, m, k)
    i ← h(k) mod m
    mientras tabla[i] no esté vacía hacer
        si tabla[i] tiene la clave k entonces
            devolver i
        fin si
        i ← (i + 1) mod m
    fin mientras
    devolver -1
fin algoritmo
```

Invariante

Para toda clave en la casilla j, entre h(k) y j no hay ninguna casilla vacía.

Una casilla vacía en el camino es la prueba de que la clave no está. Toda operación debe conservar esta propiedad. Vaciar una casilla no la conserva.

Agrupamiento primario *(primary clustering)*



Por eso el costo no crece suave: se dispara cerca de $\alpha = 1$.

Las fórmulas clásicas (sin demostración)

Si las casillas iniciales son uniformes e independientes, el promedio de casillas revisadas es (Knuth, 1963):

Búsqueda con éxito: $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$

Búsqueda sin éxito: $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Hay que comprobarlas

Una fórmula sin demostración se comprueba contando. Eso es lo que hace `probing_counts`, sin reloj.

Comprobado contando, en 2^{20} casillas

terminal

```
$ ./build/probing_counts
Con éxito: casillas o nodos revisados por búsqueda, promedio
 alfa   linear fórmula   double  cadenas
 0.50     1.50     1.50     1.39     1.25
 0.90     5.64     5.50     2.56     1.45
 0.99    52.49    50.50     4.67     1.50

Sin éxito: casillas o nodos revisados por búsqueda, promedio
 alfa   linear fórmula   double  cadenas
 0.50     2.50     2.50     2.00     0.50
 0.70     6.05     6.06     3.33     0.70
 0.90    52.97    50.50     9.96     0.90
 0.99  4468.88  5000.50    99.87     0.99
```

La diferencia entre `linear` y `double` es el precio de los cúmulos. Mantengan α lejos de 1.

SECCIÓN 03

Borrar

03

Vaciar la casilla rompe el invariante

- Borramos el 25 vaciando su casilla, con 14, 25 y 36 en las casillas 3, 4 y 5.
- El 36 sigue en la casilla 5. Ya no se puede encontrar.
- El programa no se cae, y ASan no dice nada: ningún acceso a la memoria es inválido.
- Solo se pierden datos, y solo los del mismo camino que están después.

Con el programa

borrado incorrecto

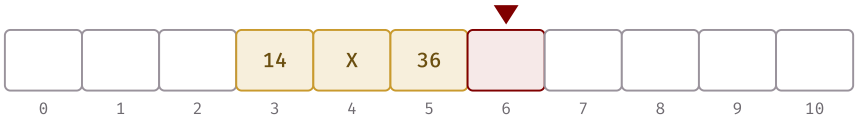
```
$ ./build/probe_trace 11 i14 i25 i36 e25 f14 f36
borrar 25 vaciando la casilla 4
buscar 14 (inicio 3): 3 -> está, 1 casillas
buscar 36 (inicio 3): 3 4 -> no está, 2 casillas
tabla: [0:_] [1:_] [2:_] [3:14] [4:_] [5:36] [6:_] [7:_] [8:_] [9:_] [10:_]
```

con tombstone

```
$ ./build/probe_trace 11 i14 i25 i36 t25 f14 f36
borrar 25 con tombstone en la casilla 4
buscar 14 (inicio 3): 3 -> está, 1 casillas
buscar 36 (inicio 3): 3 4 5 -> está, 3 casillas
tabla: [0:_] [1:_] [2:_] [3:14] [4:X] [5:36] [6:_] [7:_] [8:_] [9:_] [10:_]
```

Vaciar la casilla o dejar un tombstone

buscar 47




```
buscar 47 (inicio 3): 3 4 5 6 -> no está, 4 casillas
```

Una búsqueda sin éxito también pasa por el tombstone, y termina en la 6, que sí está vacía

Tombstones

Tombstone

Marca en una casilla que dice: aquí hubo una clave, ya no está, y el camino sigue.

Estado	Al buscar	Al insertar
Vacía	Termina: la clave no está	Se puede usar
Llena	Compara y, si no es, sigue	Compara y, si no es, sigue
Tombstone	Sigue, sin comparar	Se puede usar, si la clave no está más adelante

Error común: buscar hasta el primer tombstone

errors/stop_at_tombstone.cpp

```cpp
// Aquí está el error: un tombstone no dice nada sobre lo que viene después.
while (table[i] != kEmpty && table[i] != kTombstone) {
```

terminal

```
$ ./build/stop_at_tombstone
contains(8) = 1
contains(22) = 0  <- la clave sigue en la tabla
```

Por qué es peligroso: casi todas las pruebas lo dejan pasar. Solo falla si la clave buscada estaba después de la borrada, en el mismo camino.

Error común: insertar en el primer tombstone

La clave que insertan puede estar más adelante, pasando el tombstone.

terminal

```
$ ./build/duplicate_at_tombstone
copias de 15 en la tabla: 2
```

La regla correcta

1. Recorrer hasta una casilla vacía, recordando el primer tombstone.
2. Si sale la clave, reemplazar y terminar.
3. Si no, usar el tombstone recordado o, si no hubo, la vacía.

Una tabla con 100 claves y 1024 casillas

terminal

```
$ ./build/tombstone_counts
Se reúsan tombstones al insertar:
   cambios tombstones     casillas     rebuilds
      1000        601          3.2            0
      5000        921        185.9            0
     50000        924       1024.0            0
Se reúsan y se reconstruye al pasar 0.7:
   cambios tombstones     casillas     rebuilds
      1000        601          3.2            0
      5000        597          3.1            4
     50000        612          3.3           48
```

Idea clave: lo que decide el costo no es n/m, sino $(n + t)/m$. Para una búsqueda sin éxito, un tombstone pesa igual que una clave.

Limpiar y crecer son la misma operación

Reconstruir: reservar un arreglo, reinsertar solo las claves vivas, olvidar los tombstones.

- Si llegó al límite con muchas claves: duplicar.
- Si llegó con pocas claves y muchos tombstones: misma capacidad. Solo limpia.
- La frontera del laboratorio: las claves son como máximo la mitad del límite.

Cuesta O(1) amortizado

Limpiar cuesta $\Theta(m)$. Pero después hacen falta $\alpha_{\text{máx}} m/2$ inserciones para volver al límite: el mismo argumento de la semana 4.

Invariantes de la tabla del laboratorio

lab/probing_hash_map.h

```
1  // Invariantes:
2  // - Si capacity_ > 0, capacity_ es 8 multiplicado por una potencia de 2.
3  // - size_ cuenta las casillas llenas y tombstones_ los tombstones.
4  // - size_ + tombstones_ <= max_load_ * capacity_, con max_load_ < 1: siempre queda
       al menos una
5  //   casilla vacía, y por eso toda búsqueda termina.
6  // - No hay claves repetidas. Entre la casilla inicial de una clave y la casilla
       donde está, sin
7  //   contar esta última, no hay ninguna casilla vacía.
```

- `set_max_load_factor` rechaza $\alpha \geq 1$. Por defecto, 0.7.
- `new Slot[n]`: V necesita constructor por defecto.
- `probes_for` cuenta casillas, sin reloj.

SECCIÓN 04

Encadenamiento frente a linear probing

04

Dos argumentos opuestos

El conteo

α	linear	cadenas
0.50	2.50	0.50
0.70	6.05	0.70
0.90	52.97	0.90

Búsquedas sin éxito. Favorece al encadenamiento.

La memoria

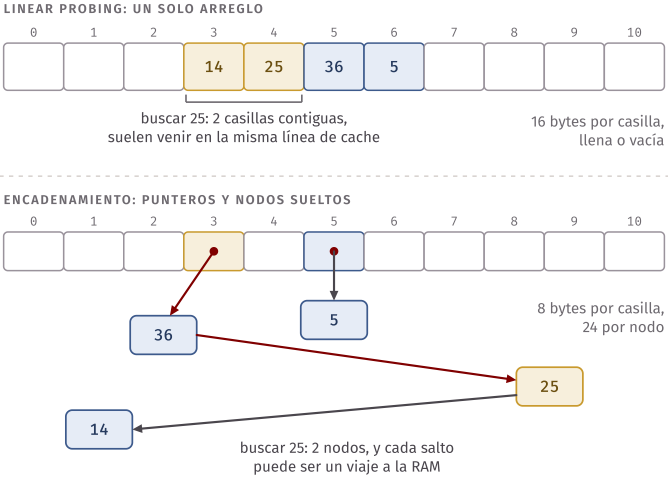
- La CPU trae bloques enteros: líneas de cache.
- La casilla de al lado suele salir gratis.
- Un nodo cualquiera puede costar un viaje a la RAM.

Favorecería al linear probing.

Hay que medir

Las mismas claves, las mismas casillas y el mismo α.

Las mismas claves en la memoria



Enteros: mediana de 5 rondas, 3 corridas

terminal

cas.	alfa	lin.hit	cad.hit	lin.miss	cad.miss	lin.B	cad.B
2^12	0.50	5.6	3.2	11.1	5.5	32.0	40.0
2^12	0.90	13.3	6.3	49.8	11.3	17.8	32.9
2^16	0.50	7.5	6.1	18.9	9.1	32.0	40.0
2^16	0.90	22.6	9.9	71.0	14.3	17.8	32.9
2^20	0.50	10.0	14.1	29.1	16.1	32.0	40.0
2^20	0.90	20.7	14.7	79.1	21.6	17.8	32.9
2^22	0.50	12.8	17.7	45.6	19.0	32.0	40.0
2^22	0.90	46.2	27.5	150.1	30.1	17.8	32.9

- Sin éxito: gana el encadenamiento en todas. Tablas pequeñas: también con éxito.
- Tablas grandes con $\alpha = 0.5$, con éxito: gana el linear probing, aunque revise más casillas.
- Tablas grandes con $\alpha = 0.9$: vuelve a ganar el encadenamiento. Memoria: menos el linear probing.

Qué podemos decir, y qué no

- **Medido:** ns, bytes por clave y casillas, y solo lo que se repitió en 3 corridas.
- **Probablemente:** el cambio entre 2^{16} y 2^{20} es la cache: una tabla pequeña cabe y una grande no.
- **No medido:** no tenemos `perf` aquí, y no contamos fallos de cache.
- **Sin explicar:** en tablas pequeñas el linear probing pierde por más de lo que dice el conteo.

Ojo con el eslogan

"El linear probing es más rápido por la cache" solo es cierto en una parte de esta tabla. Ninguno gana siempre.

SECCIÓN 05

Swiss Tables

05



Swiss Tables

- Equipo de Google, 2017. Abseil C++, 2018.
- Go 1.24 (2025) reescribió su `map` con este diseño.
- Sigue siendo direccionamiento abierto: un arreglo, probing, tombstones.

La observación

Con la tabla cargada, casi todas las comparaciones de claves son inútiles: son claves de otros. ¿Y si tuviéramos un resumen muy pequeño de cada una?

Un byte por casilla

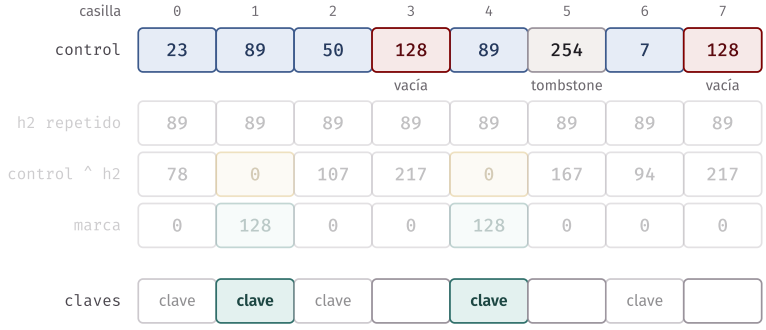
El hash de 64 bits se parte en dos: H1 (57 bits) elige dónde empezar y H2 (7 bits) va al byte de control.

Byte de control	Significado
`0xxxxxxx`	Llena, con H2 = `xxxxxxx`
`10000000`	Vacía
`11111110`	Borrada: un tombstone

Para qué sirven los 7 bits

Para evitar comparar claves. Si H2 no coincide, no es la clave. Si coincide, se compara. Dos claves distintas coinciden en H2 con probabilidad 1/128.

Un grupo de casillas y su palabra de control



Si ninguna es, match_empty marca la 3 y la 7. El grupo tiene vacías: la clave no está

En Go son 8 casillas y 8 bytes: una palabra de 64 bits.

El truco, en tres líneas

mini_swiss.h

```cpp
inline std::uint64_t match_h2(std::uint64_t ctrl, unsigned char h2) {
    const std::uint64_t v = ctrl ^ (kLowBits * h2);
    return (v - kLowBits) & ~v & kHighBits;
}
```

terminal

```
$ ./build/group_match
bytes por casilla: 23 89 50 128 89 254 7 128
buscar h2 =  89: casillas candidatas: 1 4
otra palabra, bytes por casilla: 2 3 128 128 128 128 128 128
buscar h2 =   2: casillas candidatas: 0 1
```

La casilla 1 de la otra palabra es un falso positivo. Nunca se pierde una coincidencia real.

Cuántas comparaciones de claves se ahorran

terminal

```
$ ./build/swiss_counts ../../data/palabras.txt
1048576 casillas. Promedios por búsqueda.
      --- con éxito: compara ---          --- sin éxito: compara ---
 alfa       linear        swiss   grupos       linear        swiss   grupos
 0.50         1.50         1.02     1.01         1.51       0.0333     1.06
 0.80         3.00         1.03     1.13        11.99       0.0989     1.92
 0.88         4.48         1.04     1.23        31.14       0.1694     3.01
```

Idea clave

La tabla no cambia lo que hace, sino dónde mira. Por eso puede trabajar con $\alpha = 7/8$, y gastar menos memoria.

Es una tabla de juguete: esto son conteos, no tiempos.

Un diseño, dos implementaciones

	Abseil (C++)	**Go 1.24**
Casillas por grupo	16 con SSE2 en x86; 8 en otras	8
Bytes de control	Arreglo aparte	Una palabra en cada grupo
Grupos	No necesariamente alineados	Bloques fijos
H2	7 bits (en el código revisado, los más altos)	Los 7 más bajos
Entre grupos	Quadratic probing	Quadratic probing
Carga máxima	7/8 en tablas grandes	7/8
Crecimiento	Toda la tabla	Por partes, de hasta 1024 entradas

Lo que dice el blog de Go

Hasta 60 % más rápido en microbenchmarks. Alrededor de 1.5 % en aplicaciones completas. Miren los dos números.

Un matiz de Go sobre borrar

- En Go, un grupo con una casilla vacía termina la búsqueda.
- Si el grupo de la clave borrada todavía tiene otra vacía, la casilla puede quedar vacía: no hace falta un tombstone.
- Solo se deja un tombstone si el grupo está lleno.

No contradice lo anterior

La unidad que termina la búsqueda es el grupo, no la casilla. En linear probing, una casilla es un grupo de 1.

SECCIÓN 06

El dict de CPython

06



Dos arreglos

- `dk_indices` es la tabla hash: índices de 1 a 8 bytes, `DKIX_EMPTY` o `DKIX_DUMMY` (el tombstone).
- `dk_entries`: entradas en el orden de inserción, para 2/3 de las casillas.
- Una casilla libre cuesta 1 byte, no 24.

Con un `dict` reducido

terminal

```
$ ./build/compact_dict
d[lima] = 0
d[cusco] = 1
d[puno] = 2
d[tacna] = 3
d[ica] = 4
  índices (8): 0 4 _ _ 1 2 3 _
  entradas: 0:lima=0 1:cusco=1 2:puno=2 3:tacna=3 4:ica=4
  recorrer: lima cusco puno tacna ica
```

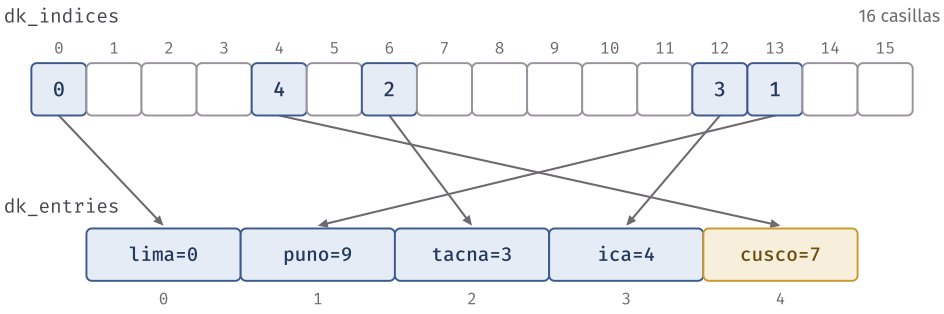
- Los índices están regados según el hash. Las entradas, en orden.
- Recorrer el diccionario es recorrer las entradas, sin mirar la tabla.
- Usa FNV-1a en vez del hash de Python: las casillas no son las reales.

Borrar, reemplazar y volver

terminal

```
$ ./build/compact_dict
del d[cusco]
  índices (8): 0 4 _ _ X 2 3 _
  entradas: 0:lima=0 1:(borrada) 2:puno=2 3:tacna=3 4:ica=4
  recorrer: lima puno tacna ica
d[puno] = 9 (reemplaza: no cambia de lugar)
  índices (8): 0 4 _ _ X 2 3 _
  entradas: 0:lima=0 1:(borrada) 2:puno=9 3:tacna=3 4:ica=4
d[cusco] = 7 (vuelve, y va al final)
  (reconstruye: 4 en uso, 8 -> 16 índices)
  índices (16): 0 _ _ _ 4 _ 2 _ _ _ _ _ 3 1 _ _
  entradas: 0:lima=0 1:puno=9 2:tacna=3 3:ica=4 4:cusco=7
  recorrer: lima puno tacna ica cusco
```

Los dos arreglos, paso a paso



d[cusco] = 7: reconstruye a 16 índices, sin tombstones ni huecos, y cusco queda al final

El orden es una consecuencia

- Python 3.6 adopta el diseño. Según su documentación, entre 20 % y 25 % menos memoria que 3.5. El orden es un detalle de implementación.
- Python 3.7: “Dictionary order is guaranteed to be insertion order”. Ya es parte del lenguaje.
- Reemplazar no cambia el orden. Una clave borrada que vuelve va al final.

El sentido de la causa

El diseño de dos arreglos da el orden, porque las entradas solo se agregan al final. El lenguaje lo garantiza después, y otra implementación de Python debe cumplirlo, aunque use otra estructura.

Python real, contra la predicción del código fuente

terminal

```
$ python3 dict_demo.py | tail -9
claves  sys.getsizeof  casillas  usables  predicción
     1            184         8         5         184
     6            272        16        10         272
    11            464        32        21         464
    22            832        64        42         832
    43           1584       128        85        1584
    86           3328       256       170        3328
   171           6576       512       341        6576
   342          13056      1024       682       13056
```

- Predicción: 64 + 32 + índices + entradas de 16 bytes. Con Python 3.14.10.
- Crece una clave después de 2/3 del tamaño: 5, 10, 21, 42...
- El tamaño nuevo sale de `used * 3`: con muchos borrados no crece. Es limpiar o crecer, con otra fórmula.

Una secuencia de probing que no es lineal

terminal

```
$ ./build/probe_order 8 0 9
casillas 8, hash 0: 0 1 6 7 4 5 2 3 0
$ ./build/probe_order 32 2654435769 12
casillas 32, hash 2654435769: 25 11 22 29 21 25 0 1 6 31 28 13
```

`perturb >>= 5;  i = (i*5 + perturb + 1) mod 2^k`

- `perturb` mete todos los bits del hash. Al final vale 0.
- Entonces $i \to 5i + 1$ visita todas las casillas.
- Por qué no lineal: `hash(i)` de un entero pequeño es i. Con claves consecutivas, la casilla siguiente es la de la próxima clave. “Linear probing is deadly in this case”, dice el comentario.
- Se pierde lo contiguo. CPython lo acepta con carga de hasta 2/3.

SECCIÓN 07

Qué cambió y qué no

07

Las ideas son las mismas

Idea	Esta clase	Swiss	`dict`
Hash y m casillas	Sí	Sí	Sí
Sin nodos	Sí	Sí	Sí (dos arreglos)
Termina en una vacía	Sí	Sí, en el grupo	Sí
Tombstones	Sí	Sí	Sí
α lejos de 1	Sí	7/8	2/3
Crecer reinsertando	Sí	Sí	Sí

Lo nuevo

No es matemática. Es la atención a la memoria: un byte de resumen por casilla, grupos que se comparan de una vez, índices de 1 byte para no pagar 24.

SECCIÓN 08

Laboratorio

08

Corrector ortográfico

A un cambio de distancia

Borrar una letra, reemplazarla o insertar una. Intercambiar dos letras vecinas no cuenta: son dos cambios.

- Letras, no bytes: `año` son 3 letras y 4 bytes. `lab/utf8.h` ya hace esta parte.
- Una palabra de L letras da $66L + 33$ candidatos. Con 8 letras, 561 preguntas.
- Casi todas responden que no: un corrector es un programa de búsquedas sin éxito.
- Cuatro diccionarios detrás de un `contains`: su tabla de la semana 7, la de hoy, `std::unordered_map` y un arreglo ordenado.

Lo que se ve al terminar

terminal

```
diccionario: 1045757 palabras
texto: 48004 palabras
desconocidas: 1378 apariciones de 708 palabras distintas
gatita (8): gaita gamita garita gotita
i (22): a di e ib id in ...
fué (114): fié fu fue fufé fui ful ...
rodillas (3): rodilla rodillos
```

No son errores del programa

Fué es una grafía antigua. A este diccionario le faltan *rodillas* y *gatita*. Las sugerencias salen en orden de bytes: á y ñ después de la z.

Los números: 3 corridas de `make bench`

terminal

diseño	máx	real	cargar	revisar	pregunta	MB
arreglo ordenado	-	-	106	99.2	119	33.5
unordered_map	1.00	-	410	67.9	66	60.5
encadenamiento	0.50	0.499	297	53.5	40	65.4
encadenamiento	1.00	0.997	341	60.9	56	57.4
linear probing	0.25	0.249	113	46.8	26	194.7
linear probing	0.50	0.499	109	62.2	67	98.7
linear probing	0.70	0.499	132	62.2	66	98.7

- Cargar: el linear probing tarda menos de la mitad. El encadenamiento reserva un nodo por palabra.
- Preguntar, con el mismo α: gana el encadenamiento. Casi todo son búsquedas sin éxito.
- Memoria: con cadenas, cada casilla ocupa 48 bytes, llena o vacía: aquí el probing gasta más.
- El máximo no es el real: con este tamaño, 0.5 y 0.7 dan la misma tabla.

El laboratorio

Sesión 1

- `ProbingHashMap<V>`: lo demás ya está hecho.
- Las mismas pruebas de la semana 7.
- Su prueba: insertar y borrar muchas veces, contando casillas.

Sesión 2

- Copiar su `hash_map.h`.
- Candidatos, sugerencias, informe y los cuatro diccionarios.
- `make bench` y `report.md`, con sus números.

terminal

```
make test-table
make test-spell
make test-lab
make bench
```

Ninguna prueba

mide tiempo: todas cuentan.

Para la próxima clase

- Leer los apuntes de la semana 10: árboles binarios de búsqueda.
- Laboratorio: corrector ortográfico.
- Resuelvan a mano las preguntas de repaso de la guía y compárenlas con `probe_trace`.

Pregunta

Si su corrector solo hiciera búsquedas con éxito, ¿cambiaría su elección de diseño? Respondan con la tabla de `probing_counts` y con una medición.