# Tablas de hash 1

Ariel Bendersky[1,2], Nicolás D'ippolito[1,2]

[1]ICC, UBA-CONICET
[2]Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Algoritmos y Estructuras de Datos II
Segundo cuatrimestre de 2019

# (2) El desafío de hoy

- ¿Se puede buscar en $O(1)$?

# (2) El desafío de hoy

- ¿Se puede buscar en $O(1)$?
- Con más precisión: ¿puede un diccionario obtener() en $O(1)$?

# (3) ¿Y si las claves son los números del 1 al 1000?

- Para algunos tipos particulares de diccionarios es fácil: si $\alpha$ es nat en el rango $1 \ldots n$.

# (3) ¿Y si las claves son los números del 1 al 1000?

- Para algunos tipos particulares de diccionarios es fácil: si $\alpha$ es nat en el rango $1 \ldots n$.
- En realidad, también puedo si el rango es $k \ldots n + k$.

# (3) ¿Y si las claves son los números del 1 al 1000?

- Para algunos tipos particulares de diccionarios es fácil: si $\alpha$ es nat en el rango $1 \ldots n$.
- En realidad, también puedo si el rango es $k \ldots n + k$.
- ¿Y si el rango es muy grande pero ralo?

# (3) ¿Y si las claves son los números del 1 al 1000?

- Para algunos tipos particulares de diccionarios es fácil: si $\alpha$ es nat en el rango $1 \ldots n$.
- En realidad, también puedo si el rango es $k \ldots n + k$.
- ¿Y si el rango es muy grande pero ralo?
- Dejemos de lado las opciones que desperdician memoria.

# (4) Relajemos

- Si relajamos algunos de los requerimientos podemos empezar a obtener resultados interesantes.

# (4) Relajemos

- Si relajamos algunos de los requerimientos podemos empezar a obtener resultados interesantes.
- Vamos a conformarnos con tener $O(1)$ en el caso esperado sabiendo que podría haber casos peores.

# (4) Relajemos

- Si relajamos algunos de los requerimientos podemos empezar a obtener resultados interesantes.
- Vamos a conformarnos con tener $O(1)$ en el caso esperado sabiendo que podría haber casos peores.
- Supongamos que queremos almacenar $n$ elementos, cuyos claves son valores numéricos en $1 \ldots N$ y sabemos que $N \gg n$.

# (4) Relajemos

- Si relajamos algunos de los requerimientos podemos empezar a obtener resultados interesantes.
- Vamos a conformarnos con tener $O(1)$ en el caso esperado sabiendo que podría haber casos peores.
- Supongamos que queremos almacenar $n$ elementos, cuyos claves son valores numéricos en $1 \ldots N$ y sabemos que $N \gg n$.
- Por ejemplo, 100 DNIs, y $\beta$ son todos los datos de la persona.

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...
- ...vamos a tener un arreglo $A[1 \ldots n]$ de conj($\langle \alpha, \beta \rangle$).

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...
- ...vamos a tener un arreglo $A[1 \ldots n]$ de conj($\langle \alpha, \beta \rangle$).
- Y además una función: $h$: DNI $\rightarrow [0 \ldots n-1]$.

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...
- ...vamos a tener un arreglo $A[1 \ldots n]$ de conj($\langle \alpha, \beta \rangle$).
- Y además una función: $h$: DNI $\to [0 \ldots n-1]$.
- Por ejemplo: $h(d) = d \bmod 100$.

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...
- ...vamos a tener un arreglo $A[1 \ldots n]$ de conj($\langle \alpha, \beta \rangle$).
- Y además una función: $h$: DNI $\to [0 \ldots n-1]$.
- Por ejemplo: $h(d) = d \bmod 100$.
- Forma de uso: los datos del DNI $d$ están en $A[h(d)]$.

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...
- ...vamos a tener un arreglo $A[1 \ldots n]$ de conj($\langle \alpha, \beta \rangle$).
- Y además una función: $h$: DNI $\to [0 \ldots n-1]$.
- Por ejemplo: $h(d) = d \bmod 100$.
- Forma de uso: los datos del DNI $d$ están en $A[h(d)]$.
- Eso sí: tal vez no sean los únicos que están ahí.

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...
- ...vamos a tener un arreglo $A[1 \ldots n]$ de conj($\langle \alpha, \beta \rangle$).
- Y además una función: $h$: DNI $\to [0 \ldots n-1]$.
- Por ejemplo: $h(d) = d \bmod 100$.
- Forma de uso: los datos del DNI $d$ están en $A[h(d)]$.
- Eso sí: tal vez no sean los únicos que están ahí.
- Ésa es la idea básica detrás de una estructura de datos llamada *tabla de hash*.

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...
- ...vamos a tener un arreglo $A[1 \ldots n]$ de conj($\langle \alpha, \beta \rangle$).
- Y además una función: $h$: DNI $\to [0 \ldots n-1]$.
- Por ejemplo: $h(d) = d \bmod 100$.
- Forma de uso: los datos del DNI $d$ están en $A[h(d)]$.
- Eso sí: tal vez no sean los únicos que están ahí.
- Ésa es la idea básica detrás de una estructura de datos llamada *tabla de hash*.
- $h$ es la función de hash (o función hash, o función de hashing).

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...
- ...vamos a tener un arreglo $A[1 \ldots n]$ de conj($\langle \alpha, \beta \rangle$).
- Y además una función: $h$: DNI $\to [0 \ldots n-1]$.
- Por ejemplo: $h(d) = d \bmod 100$.
- Forma de uso: los datos del DNI $d$ están en $A[h(d)]$.
- Eso sí: tal vez no sean los únicos que están ahí.
- Ésa es la idea básica detrás de una estructura de datos llamada *tabla de hash*.
- $h$ es la función de hash (o función hash, o función de hashing).
- ¿Y si $\alpha$ no es numérico?

# (5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de $\beta$...
- ...vamos a tener un arreglo $A[1 \ldots n]$ de conj($\langle \alpha, \beta \rangle$).
- Y además una función: $h$: DNI $\to [0 \ldots n-1]$.
- Por ejemplo: $h(d) = d \bmod 100$.
- Forma de uso: los datos del DNI $d$ están en $A[h(d)]$.
- Eso sí: tal vez no sean los únicos que están ahí.
- Ésa es la idea básica detrás de una estructura de datos llamada *tabla de hash*.
- $h$ es la función de hash (o función hash, o función de hashing).
- ¿Y si $\alpha$ no es numérico?
- Alcanza con encontrar una función total de $\alpha \to [0 \ldots n-1]$.

# (6) Formalicemos

- Una tabla de hash es una tupla $\langle A, h \rangle$, donde $A$ es un arreglo de $k$ posiciones y $h$ la función de hash $\alpha \rightarrow [0 \ldots k-1]$.

# (6) Formalicemos

- Una tabla de hash es una tupla $\langle A, h \rangle$, donde $A$ es un arreglo de $k$ posiciones y $h$ la función de hash $\alpha \rightarrow [0 \ldots k-1]$.
- Tarea: pensar en invariante y función de abstracción.

# (6) Formalicemos

- Una tabla de hash es una tupla $\langle A, h \rangle$, donde $A$ es un arreglo de $k$ posiciones y $h$ la función de hash $\alpha \to [0 \ldots k-1]$.
- Tarea: pensar en invariante y función de abstracción.
- (Cada posición del arreglo se suele llamar *bucket*.)

# (6) Formalicemos

- Una tabla de hash es una tupla $\langle A, h \rangle$, donde $A$ es un arreglo de $k$ posiciones y $h$ la función de hash $\alpha \to [0 \dots k-1]$.
- Tarea: pensar en invariante y función de abstracción.
- (Cada posición del arreglo se suele llamar *bucket*.)
- Si $h$ distribuye "bien" a los $\alpha$ tendremos un caso "esperado" de $O(1)$.

# (6) Formalicemos

- Una tabla de hash es una tupla $\langle A, h \rangle$, donde $A$ es un arreglo de $k$ posiciones y $h$ la función de hash $\alpha \to [0 \dots k-1]$.
- Tarea: pensar en invariante y función de abstracción.
- (Cada posición del arreglo se suele llamar *bucket*.)
- Si $h$ distribuye "bien" a los $\alpha$ tendremos un caso "esperado" de $O(1)$.
- El peor caso será bastante mayor. Cuánto?

# (6) Formalicemos

- Una tabla de hash es una tupla $\langle A, h \rangle$, donde $A$ es un arreglo de $k$ posiciones y $h$ la función de hash $\alpha \to [0 \ldots k-1]$.
- Tarea: pensar en invariante y función de abstracción.
- (Cada posición del arreglo se suele llamar *bucket*.)
- Si $h$ distribuye "bien" a los $\alpha$ tendremos un caso "esperado" de $O(1)$.
- El peor caso será bastante mayor. Cuánto?
- Depende de cómo resolvamos las colisiones.

# (7) Hashing perfecto vs colisiones

- Se dice que una función de hash es perfecta si:
  $\forall c_1, c_2 \in \alpha, c_1 \neq c_2 \Rightarrow h(c_1) \neq h(c_2)$

# (7) Hashing perfecto vs colisiones

- Se dice que una función de hash es perfecta si:
  $\forall c_1, c_2 \in \alpha, c_1 \neq c_2 \Rightarrow h(c_1) \neq h(c_2)$
- Para eso necesitamos que $k \geq |\alpha|$, lo que muy rara vez sucede.

# (7) Hashing perfecto vs colisiones

- Se dice que una función de hash es perfecta si:
  $\forall c_1, c_2 \in \alpha, c_1 \neq c_2 \Rightarrow h(c_1) \neq h(c_2)$
- Para eso necesitamos que $k \geq |\alpha|$, lo que muy rara vez sucede.
- De hecho, suele suceder que $|\alpha| \gg k$.

# (7) Hashing perfecto vs colisiones

- Se dice que una función de hash es perfecta si:
  $\forall c_1, c_2 \in \alpha, c_1 \neq c_2 \Rightarrow h(c_1) \neq h(c_2)$
- Para eso necesitamos que $k \geq |\alpha|$, lo que muy rara vez sucede.
- De hecho, suele suceder que $|\alpha| \gg k$.
- Por ende, es muy probable (más de lo que uno se imagina) que $h(c_1) = h(c_2)$. Es decir, que $c_1$ y $c_2$ colisionen.

# (7) Hashing perfecto vs colisiones

- Se dice que una función de hash es perfecta si:
  $\forall c_1, c_2 \in \alpha, c_1 \neq c_2 \Rightarrow h(c_1) \neq h(c_2)$
- Para eso necesitamos que $k \geq |\alpha|$, lo que muy rara vez sucede.
- De hecho, suele suceder que $|\alpha| \gg k$.
- Por ende, es muy probable (más de lo que uno se imagina) que $h(c_1) = h(c_2)$. Es decir, que $c_1$ y $c_2$ colisionen.
- Ejercicio: proponer una función hash perfecta para el caso en que las claves sean strings de largo 3 en el alfabeto $\{a, b, c\}$.

# (8) Nadie es perfecto

- Si *perfecto* es mucho pedir, ¿qué pasa con “bueno”?

# (8) Nadie es perfecto

- Si *perfecto* es mucho pedir, ¿qué pasa con "bueno"?
- Supongamos que conocemos la distribución de frecuencias de $\alpha$ y llamamos $P(c)$ a la probabilidad de la clave $c$.

# (8) Nadie es perfecto

- Si *perfecto* es mucho pedir, ¿qué pasa con "bueno"?
- Supongamos que conocemos la distribución de frecuencias de $\alpha$ y llamamos $P(c)$ a la probabilidad de la clave $c$.
- Nos gustaría que

$$\forall i \in [0 \dots k-1], (\sum_{c|h(c)=i} P(c)) \sim 1/k$$

# (8) Nadie es perfecto

- Si *perfecto* es mucho pedir, ¿qué pasa con "bueno"?
- Supongamos que conocemos la distribución de frecuencias de $\alpha$ y llamamos $P(c)$ a la probabilidad de la clave $c$.
- Nos gustaría que

$$\forall i \in [0 \dots k-1], (\sum_{c|h(c)=i} P(c)) \sim 1/k$$

- Es decir, que para cada posición del arreglo, la probabilidad de que algún $\alpha$ vaya a parar ahí, sea aproximadamente uniforme.

# (8) Nadie es perfecto

- Si *perfecto* es mucho pedir, ¿qué pasa con "bueno"?
- Supongamos que conocemos la distribución de frecuencias de $\alpha$ y llamamos $P(c)$ a la probabilidad de la clave $c$.
- Nos gustaría que

$$\forall i \in [0 \dots k-1], (\sum_{c|h(c)=i} P(c)) \sim 1/k$$

- Es decir, que para cada posición del arreglo, la probabilidad de que algún $\alpha$ vaya a parar ahí, sea aproximadamente uniforme.
- Eso se llama uniformidad simple, y es muy difícil de lograr.

# (8) Nadie es perfecto

- Si *perfecto* es mucho pedir, ¿qué pasa con "bueno"?
- Supongamos que conocemos la distribución de frecuencias de $\alpha$ y llamamos $P(c)$ a la probabilidad de la clave $c$.
- Nos gustaría que

$$\forall i \in [0 \dots k-1], (\sum_{c|h(c)=i} P(c)) \sim 1/k$$

- Es decir, que para cada posición del arreglo, la probabilidad de que algún $\alpha$ vaya a parar ahí, sea aproximadamente uniforme.
- Eso se llama uniformidad simple, y es muy difícil de lograr.
- En gran parte, porque $P$ suele ser desconocida.

# (9) Nadie es perfecto (cont.)

- A modo de ejemplo, pensemos en $A[0 \ldots 5]$ y $h(c) = c \bmod 5$.

# (9) Nadie es perfecto (cont.)

- A modo de ejemplo, pensemos en $A[0 \ldots 5]$ y $h(c) = c \bmod 5$.
- Dos conjuntos de datos muy similares $\{1, 7, 10, 14\}$ vs $\{1, 6, 11, 16\}$.

# (9) Nadie es perfecto (cont.)

- A modo de ejemplo, pensemos en $A[0 \ldots 5]$ y $h(c) = c \bmod 5$.
- Dos conjuntos de datos muy similares $\{1, 7, 10, 14\}$ vs $\{1, 6, 11, 16\}$.
- En el primer caso, ocupan las posiciones

# (9) Nadie es perfecto (cont.)

- A modo de ejemplo, pensemos en $A[0 \ldots 5]$ y $h(c) = c \bmod 5$.
- Dos conjuntos de datos muy similares $\{1, 7, 10, 14\}$ vs $\{1, 6, 11, 16\}$.
- En el primer caso, ocupan las posiciones

# (9) Nadie es perfecto (cont.)

- A modo de ejemplo, pensemos en $A[0 \ldots 5]$ y $h(c) = c \bmod 5$.
- Dos conjuntos de datos muy similares $\{1, 7, 10, 14\}$ vs $\{1, 6, 11, 16\}$.
- En el primer caso, ocupan las posiciones 1, 2, 0 y 4.
- En el segundo caso,

# (9) Nadie es perfecto (cont.)

- A modo de ejemplo, pensemos en $A[0 \ldots 5]$ y $h(c) = c \bmod 5$.
- Dos conjuntos de datos muy similares $\{1, 7, 10, 14\}$ vs $\{1, 6, 11, 16\}$.
- En el primer caso, ocupan las posiciones 1, 2, 0 y 4.
- En el segundo caso,

# (9) Nadie es perfecto (cont.)

- A modo de ejemplo, pensemos en $A[0 \ldots 5]$ y $h(c) = c \bmod 5$.
- Dos conjuntos de datos muy similares $\{1, 7, 10, 14\}$ vs $\{1, 6, 11, 16\}$.
- En el primer caso, ocupan las posiciones 1, 2, 0 y 4.
- En el segundo caso, todos quedan en la 1.
- Como conocer las distribuciones es muy difícil, lo que se busca en la práctica es tener independencia de la distribución de los datos.

# (9) Nadie es perfecto (cont.)

- A modo de ejemplo, pensemos en $A[0 \ldots 5]$ y $h(c) = c \bmod 5$.
- Dos conjuntos de datos muy similares $\{1, 7, 10, 14\}$ vs $\{1, 6, 11, 16\}$.
- En el primer caso, ocupan las posiciones 1, 2, 0 y 4.
- En el segundo caso, todos quedan en la 1.
- Como conocer las distribuciones es muy difícil, lo que se busca en la práctica es tener independencia de la distribución de los datos.
- Y además, saber que las colisiones son prácticamente inevitables y habrá que lidiar con ellas sí o sí.

# (9) Nadie es perfecto (cont.)

- A modo de ejemplo, pensemos en $A[0 \ldots 5]$ y $h(c) = c \bmod 5$.
- Dos conjuntos de datos muy similares $\{1, 7, 10, 14\}$ vs $\{1, 6, 11, 16\}$.
- En el primer caso, ocupan las posiciones 1, 2, 0 y 4.
- En el segundo caso, todos quedan en la 1.
- Como conocer las distribuciones es muy difícil, lo que se busca en la práctica es tener independencia de la distribución de los datos.
- Y además, saber que las colisiones son prácticamente inevitables y habrá que lidiar con ellas sí o sí.
- ¿Son inevitables?

# (10) Paradoja del cumpleaños

- Si tenemos una tabla con $k = 365$ y $h$ distribuye uniformemente entre las celdas, ¿cuál es la probabilidad de tener colisiones?

# (10) Paradoja del cumpleaños

- Si tenemos una tabla con $k = 365$ y $h$ distribuye uniformemente entre las celdas, ¿cuál es la probabilidad de tener colisiones?
- (No es una paradoja en sentido estricto: es una contradicción entre la intuición y el resultado matemático.)

# (10) Paradoja del cumpleaños

- Si tenemos una tabla con $k = 365$ y $h$ distribuye uniformemente entre las celdas, ¿cuál es la probabilidad de tener colisiones?
- (No es una paradoja en sentido estricto: es una contradicción entre la intuición y el resultado matemático.)
- Si elegimos 23 personas al azar, la probabilidad de que dos de ellas cumplan años el mismo día es $> \frac{1}{2}$ (aprox. 50,7 %).

# (10) Paradoja del cumpleaños

- Si tenemos una tabla con $k = 365$ y $h$ distribuye uniformemente entre las celdas, ¿cuál es la probabilidad de tener colisiones?
- (No es una paradoja en sentido estricto: es una contradicción entre la intuición y el resultado matemático.)
- Si elegimos 23 personas al azar, la probabilidad de que dos de ellas cumplan años el mismo día es $> \frac{1}{2}$ (aprox. 50,7 %).
- Es decir, aún suponiendo distribución uniforme de los nacimientos, hay alta probabilidad de festejo conjunto.

# (10) Paradoja del cumpleaños

- Si tenemos una tabla con $k = 365$ y $h$ distribuye uniformemente entre las celdas, ¿cuál es la probabilidad de tener colisiones?
- (No es una paradoja en sentido estricto: es una contradicción entre la intuición y el resultado matemático.)
- Si elegimos 23 personas al azar, la probabilidad de que dos de ellas cumplan años el mismo día es $> \frac{1}{2}$ (aprox. 50,7 %).
- Es decir, aún suponiendo distribución uniforme de los nacimientos, hay alta probabilidad de festejo conjunto.
- Tarea: pensar en la demo (pista: calcular la inversa, es decir, la probabilidad de que no haya dos personas que colisionen).

# (10) Paradoja del cumpleaños

- Si tenemos una tabla con $k = 365$ y $h$ distribuye uniformemente entre las celdas, ¿cuál es la probabilidad de tener colisiones?
- (No es una paradoja en sentido estricto: es una contradicción entre la intuición y el resultado matemático.)
- Si elegimos 23 personas al azar, la probabilidad de que dos de ellas cumplan años el mismo día es $> \frac{1}{2}$ (aprox. 50,7 %).
- Es decir, aún suponiendo distribución uniforme de los nacimientos, hay alta probabilidad de festejo conjunto.
- Tarea: pensar en la demo (pista: calcular la inversa, es decir, la probabilidad de que no haya dos personas que colisionen).
- Corolario: si tenemos una tabla con $k = 365$ y $h$ distribuye uniformemente entre ellas, aún así tenemos probabilidad $> \frac{1}{2}$ de que luego de 23 inserciones se produzca una colisión.

# (11) Resolviendo las colisiones

- Dado que las colisiones son inevitables, una buena tabla de hash tiene que poder lidiar con ellas.

# (11) Resolviendo las colisiones

- Dado que las colisiones son inevitables, una buena tabla de hash tiene que poder lidiar con ellas.
- Dos familias de métodos:

# (11) Resolviendo las colisiones

- Dado que las colisiones son inevitables, una buena tabla de hash tiene que poder lidiar con ellas.
- Dos familias de métodos:
  - Hashing (o direccionamiento) abierto: los elementos se guardan en la tabla.

# (11) Resolviendo las colisiones

- Dado que las colisiones son inevitables, una buena tabla de hash tiene que poder lidiar con ellas.
- Dos familias de métodos:
  - Hashing (o direccionamiento) abierto: los elementos se guardan en la tabla.
  - Hashing (o direccionamiento) cerrado: en cada $A[i]$ hay algún tipo de contenedor que almacena todos los elementos que son hasheados a $i$.

# (11) Resolviendo las colisiones

- Dado que las colisiones son inevitables, una buena tabla de hash tiene que poder lidiar con ellas.
- Dos familias de métodos:
  - Hashing (o direccionamiento) abierto: los elementos se guardan en la tabla.
  - Hashing (o direccionamiento) cerrado: en cada $A[i]$ hay algún tipo de contenedor que almacena todos los elementos que son hasheados a $i$.
- Ojo: la bibliografía a veces alterna los nombres y las definiciones. Lo que importa, como siempre, es la idea más que el nombre.

# (12) Hashing cerrado por concatenación

- Tenemos varias alternativas para el contenedor a usar en un hahshing cerrado.

# (12) Hashing cerrado por concatenación

- Tenemos varias alternativas para el contenedor a usar en un hahshing cerrado.
- Por ejemplo, podrían ser AVLs...

# (12) Hashing cerrado por concatenación

- Tenemos varias alternativas para el contenedor a usar en un hahshing cerrado.
- Por ejemplo, podrían ser AVLs...
- Pero perderíamos el ansiado $O(1)$.

# (12) Hashing cerrado por concatenación

- Tenemos varias alternativas para el contenedor a usar en un hahshing cerrado.
- Por ejemplo, podrían ser AVLs...
- Pero perderíamos el ansiado $O(1)$.
- Empecemos por las listas y analicemos sus complejidades:

# (12) Hashing cerrado por concatenación

- Tenemos varias alternativas para el contenedor a usar en un hahshing cerrado.
- Por ejemplo, podrían ser AVLs...
- Pero perderíamos el ansiado $O(1)$.
- Empecemos por las listas y analicemos sus complejidades:
  - Inserción: podemos tener una lista que permita agregar el elemento en $O(1)$.

# (12) Hashing cerrado por concatenación

- Tenemos varias alternativas para el contenedor a usar en un hahshing cerrado.
- Por ejemplo, podrían ser AVLs...
- Pero perderíamos el ansiado $O(1)$.
- Empecemos por las listas y analicemos sus complejidades:
  - Inserción: podemos tener una lista que permita agregar el elemento en $O(1)$.
  - Búsqueda y borrado: lineal en la cantidad de elementos del bucket.

# (12) Hashing cerrado por concatenación

- Tenemos varias alternativas para el contenedor a usar en un hahshing cerrado.
- Por ejemplo, podrían ser AVLs...
- Pero perderíamos el ansiado $O(1)$.
- Empecemos por las listas y analicemos sus complejidades:
  - Inserción: podemos tener una lista que permita agregar el elemento en $O(1)$.
  - Búsqueda y borrado: lineal en la cantidad de elementos del bucket.
- ¿Cuánto pueden medir esas listas?

# (13) Factor de carga

- Definimos el factor de carga $fc = n/k$.

# (13) Factor de carga

- Definimos el factor de carga $fc = n/k$.
- Es decir, la relación entre la cant. de elementos presentes y el tamaño de la tabla.

# (13) Factor de carga

- Definimos el factor de carga $fc = n/k$.
- Es decir, la relación entre la cant. de elementos presentes y el tamaño de la tabla.
- Teorema: suponiendo que $h$ es simplemente uniforme y se usa hashing cerrado por concatenación, en promedio

# (13) Factor de carga

- Definimos el factor de carga $fc = n/k$.
- Es decir, la relación entre la cant. de elementos presentes y el tamaño de la tabla.
- Teorema: suponiendo que $h$ es simplemente uniforme y se usa hashing cerrado por concatenación, en promedio
  - una búsqueda fallida requiere $\Theta(1 + fc)$, y

# (13) Factor de carga

- Definimos el factor de carga $fc = n/k$.
- Es decir, la relación entre la cant. de elementos presentes y el tamaño de la tabla.
- Teorema: suponiendo que $h$ es simplemente uniforme y se usa hashing cerrado por concatenación, en promedio
  - una búsqueda fallida requiere $\Theta(1 + fc)$, y
  - una búsqueda exitosa requiere $\Theta(1 + fc/2)$.

# (13) Factor de carga

- Definimos el factor de carga $fc = n/k$.
- Es decir, la relación entre la cant. de elementos presentes y el tamaño de la tabla.
- Teorema: suponiendo que $h$ es simplemente uniforme y se usa hashing cerrado por concatenación, en promedio
  - una búsqueda fallida requiere $\Theta(1 + fc)$, y
  - una búsqueda exitosa requiere $\Theta(1 + fc/2)$.
- Corolario: si $n \leq k$ o incluso si $n \sim k$, tenemos $O(1)$.

# (13) Factor de carga

- Definimos el factor de carga $fc = n/k$.
- Es decir, la relación entre la cant. de elementos presentes y el tamaño de la tabla.
- Teorema: suponiendo que $h$ es simplemente uniforme y se usa hashing cerrado por concatenación, en promedio
  - una búsqueda fallida requiere $\Theta(1 + fc)$, y
  - una búsqueda exitosa requiere $\Theta(1 + fc/2)$.
- Corolario: si $n \leq k$ o incluso si $n \sim k$, tenemos $O(1)$.
- Por ende, es muy importante dimensionar bien la tabla.

# (14) Hashing abierto

- Veamos cómo funciona el hashing abierto.

# (14) Hashing abierto

- Veamos cómo funciona el hashing abierto.
- La idea es que ante una colisión vamos a ubicar al elemento en otra celda de la tabla que esté libre.

# (14) Hashing abierto

- Veamos cómo funciona el hashing abierto.
- La idea es que ante una colisión vamos a ubicar al elemento en otra celda de la tabla que esté libre.
- Los métodos varían según cómo hacen para encontrar esa otra posición.

# (14) Hashing abierto

- Veamos cómo funciona el hashing abierto.
- La idea es que ante una colisión vamos a ubicar al elemento en otra celda de la tabla que esté libre.
- Los métodos varían según cómo hacen para encontrar esa otra posición.
- La función de hash incorpora como segundo parámetro al número de intento:

# (14) Hashing abierto

- Veamos cómo funciona el hashing abierto.
- La idea es que ante una colisión vamos a ubicar al elemento en otra celda de la tabla que esté libre.
- Los métodos varían según cómo hacen para encontrar esa otra posición.
- La función de hash incorpora como segundo parámetro al número de intento:
- El $i$-ésimo intento para $c$ se corresponde con $h(c, i)$.

# (14) Hashing abierto

- Veamos cómo funciona el hashing abierto.
- La idea es que ante una colisión vamos a ubicar al elemento en otra celda de la tabla que esté libre.
- Los métodos varían según cómo hacen para encontrar esa otra posición.
- La función de hash incorpora como segundo parámetro al número de intento:
- El $i$-ésimo intento para $c$ se corresponde con $h(c, i)$.
- En hashing abierto el borrado suele ser problemático.

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:

  1 $i = 0$

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$
     - $A[h(c, i)] \neq \text{null} \wedge$

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$
     - $A[h(c, i)] \neq$ null $\wedge$
     - $A[h(c, i)]$.clave $\neq c$

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c,i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c,i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$
     - $A[h(c,i)] \neq$ null $\wedge$
     - $A[h(c,i)]$.clave $\neq c$
  3. si ($i < |A|$) $\wedge$ $A[h(c,i)] \neq$ null return $A[h(c,i)]$.valor

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c,i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c,i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$
     - $A[h(c,i)] \neq$ null $\wedge$
     - $A[h(c,i)]$.clave $\neq c$
  3. si ($i < |A|$) $\wedge$ $A[h(c,i)] \neq$ null return $A[h(c,i)]$.valor
  4. sino return no está

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$
     - $A[h(c, i)] \neq$ null $\wedge$
     - $A[h(c, i)]$.clave $\neq c$
  3. si ($i < |A|$) $\wedge$ $A[h(c, i)] \neq$ null return $A[h(c, i)]$.valor
  4. sino return no está
- Borrado:

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c,i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c,i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$
     - $A[h(c,i)] \neq$ null $\wedge$
     - $A[h(c,i)]$.clave $\neq c$
  3. si ($i < |A|$) $\wedge$ $A[h(c,i)] \neq$ null return $A[h(c,i)]$.valor
  4. sino return no está
- Borrado:
  - ¿Marcamos como null?

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$
     - $A[h(c, i)] \neq$ null $\wedge$
     - $A[h(c, i)]$.clave $\neq c$
  3. si ($i < |A|$) $\wedge$ $A[h(c, i)] \neq$ null return $A[h(c, i)]$.valor
  4. sino return no está
- Borrado:
  - ¿Marcamos como null?

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c,i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c,i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$
     - $A[h(c,i)] \neq$ null $\wedge$
     - $A[h(c,i)]$.clave $\neq c$
  3. si ($i < |A|$) $\wedge$ $A[h(c,i)] \neq$ null return $A[h(c,i)]$.valor
  4. sino return no está
- Borrado:
  - ¿Marcamos como null? Ojo con la búsqueda.
  - Es mejor marcarlos como borrados.

# (15) Algoritmos de hashing abierto

- Inserción de clave $c$ en $A$:
  1. $i = 0$
  2. mientras ($i < |A| \wedge A[h(c, i)]$ esté ocupada) incrementar $i$
  3. si ($i < |A|$), el elemento va en $A[h(c, i)]$
  4. si no, overflow!
- Búsqueda de clave $c$ en $A$:
  1. $i = 0$
  2. incrementar $i$ mientras:
     - $i < |A| \wedge$
     - $A[h(c, i)] \neq$ null $\wedge$
     - $A[h(c, i)]$.clave $\neq c$
  3. si ($i < |A|$) $\wedge$ $A[h(c, i)] \neq$ null return $A[h(c, i)]$.valor
  4. sino return no está
- Borrado:
  - ¿Marcamos como null? Ojo con la búsqueda.
  - Es mejor marcarlos como borrados.
  - Aunque empeora la performance de la búsqueda para el caso exitoso. ¿Por qué?

# (16) Repaso y continuación

- La próxima veremos cómo construir las $h(c, i)$,

# (16) Repaso y continuación

- La próxima veremos cómo construir las $h(c, i)$,
- los problemas que se pueden presentar.

# (16) Repaso y continuación

- La próxima veremos cómo construir las $h(c, i)$,
- los problemas que se pueden presentar.
- y cómo solucionarlos.

# Tablas de hash 2

Ariel Bendersky[1,2], Nicolás D'ippolito[1,2]

[1]ICC, UBA-CONICET
[2]Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Algoritmos y Estructuras de Datos II
Segundo cuatrimestre de 2019

# (2) Repasemos tablas de hash

- Queríamos buscar en algo así como $O(1)$.

# (2) Repasemos tablas de hash

- Queríamos buscar en algo así como $O(1)$.
- El problema eran las colisiones.

# (2) Repasemos tablas de hash

- Queríamos buscar en algo así como $O(1)$.
- El problema eran las colisiones.
- Vimos que podíamos manejarlas con hashing cerrado o abierto.

# (2) Repasemos tablas de hash

- Queríamos buscar en algo así como $O(1)$.
- El problema eran las colisiones.
- Vimos que podíamos manejarlas con hashing cerrado o abierto.
- En hashing abierto teníamos que definir $h(c, i)$ como la función que nos indicaba en qué posición de $A$ debíamos intentar ubicar la clave $c$ en el intento $i$.

# (2) Repasemos tablas de hash

- Queríamos buscar en algo así como $O(1)$.
- El problema eran las colisiones.
- Vimos que podíamos manejarlas con hashing cerrado o abierto.
- En hashing abierto teníamos que definir $h(c, i)$ como la función que nos indicaba en qué posición de $A$ debíamos intentar ubicar la clave $c$ en el intento $i$.
- Hoy vamos a ver distintas formas de definir esa función.

# (3) Barrido (y limpieza?)

- ¿Tendría sentido una función $h(c, i)$ que deje posiciones de la tabla sin explorar?

# (3) Barrido (y limpieza?)

- ¿Tendría sentido una función $h(c, i)$ que deje posiciones de la tabla sin explorar?

# (3) Barrido (y limpieza?)

- ¿Tendría sentido una función $h(c, i)$ que deje posiciones de la tabla sin explorar? No.
- Por eso se habla de *barrido*:

# (3) Barrido (y limpieza?)

- ¿Tendría sentido una función $h(c, i)$ que deje posiciones de la tabla sin explorar? No.
- Por eso se habla de *barrido*:
  - Lineal.

# (3) Barrido (y limpieza?)

- ¿Tendría sentido una función $h(c, i)$ que deje posiciones de la tabla sin explorar? No.
- Por eso se habla de *barrido*:
  - Lineal.
  - Cuadrático.

# (3) Barrido (y limpieza?)

- ¿Tendría sentido una función $h(c, i)$ que deje posiciones de la tabla sin explorar? No.
- Por eso se habla de *barrido*:
  - Lineal.
  - Cuadrático.
  - Hashing doble (o rehashing).

# (3) Barrido (y limpieza?)

- ¿Tendría sentido una función $h(c, i)$ que deje posiciones de la tabla sin explorar? No.
- Por eso se habla de *barrido*:
  - Lineal.
  - Cuadrático.
  - Hashing doble (o rehashing).
- Veamos...

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)]$,

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)]$,

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)]$, $A[h'(c) + 1]$, $\ldots$,

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)]$, $A[h'(c) + 1]$, $\ldots$, $A[|A|]$,

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)]$, $A[h'(c) + 1]$, ..., $A[|A|]$, $A[0]$, ...
- Ejemplo:

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)], A[h'(c) + 1], \ldots, A[|A|], A[0], \ldots$
- Ejemplo:
  - $h(c, i) = (h'(c) + i) \bmod 101$, $h'(c) = c \bmod 101$

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)]$, $A[h'(c) + 1]$, ..., $A[|A|]$, $A[0]$, ...
- Ejemplo:
  - $h(c, i) = (h'(c) + i) \bmod 101$, $h'(c) = c \bmod 101$
  - Insertemos 2, 103, 104, 105, etc., en ese orden.

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)]$, $A[h'(c) + 1]$, ..., $A[|A|]$, $A[0]$, ...
- Ejemplo:
  - $h(c, i) = (h'(c) + i) \bmod 101$, $h'(c) = c \bmod 101$
  - Insertemos 2, 103, 104, 105, etc., en ese orden.
- Cada inserción colisiona varias veces antes de poderse realizar.

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)]$, $A[h'(c) + 1]$, ..., $A[|A|]$, $A[0]$, ...
- Ejemplo:
  - $h(c, i) = (h'(c) + i) \bmod 101$, $h'(c) = c \bmod 101$
  - Insertemos 2, 103, 104, 105, etc., en ese orden.
- Cada inserción colisiona varias veces antes de poderse realizar.
- Eso se llama aglomeración primaria, y si bien es un caso extremo, puede darse.

# (4) Barrido lineal

- La idea es ir recorriendo todas las posiciones en secuencia:
- $h(c, i) = (h'(c) + i) \bmod |A|$, donde $h'(c)$ es otra función de hashing.
- Veamos la secuencia que se genera:
  $A[h'(c)]$, $A[h'(c) + 1]$, ..., $A[|A|]$, $A[0]$, ...
- Ejemplo:
  - $h(c, i) = (h'(c) + i) \bmod 101$, $h'(c) = c \bmod 101$
  - Insertemos 2, 103, 104, 105, etc., en ese orden.
- Cada inserción colisiona varias veces antes de poderse realizar.
- Eso se llama aglomeración primaria, y si bien es un caso extremo, puede darse.
- Cuando hay aglomeración primaria, si dos secuencias colisionan en algún momento, siguen colisionando.

# (5) Barrido cuadrático

- $h(c, i) = (h'(c) + a\ i + b\ i^2) \bmod |A|$, donde $h'(c)$ es un función de hash y $a$ y $b$ son constantes.

# (5) Barrido cuadrático

- $h(c, i) = (h'(c) + a\ i + b\ i^2) \bmod |A|$, donde $h'(c)$ es un función de hash y $a$ y $b$ son constantes.
- Es una forma de evitar la aglomeración primaria, ya que el polinomio varía dependiendo del número de intento.

# (5) Barrido cuadrático

- $h(c, i) = (h'(c) + a\ i + b\ i^2)$ mod $|A|$, donde $h'(c)$ es un función de hash y $a$ y $b$ son constantes.
- Es una forma de evitar la aglomeración primaria, ya que el polinomio varía dependiendo del número de intento.
- Sin embargo, puede producirse aglomeración secundaria: si hay colisión en el primer intento, sigue habiendo colisiones ($h'(c_1) = h'(c_2) \Rightarrow h(c_1, i) = h(c_2, i)$).

# (6) Hashing doble o rehashing

- Idea: que el barrido también dependa de la clave.

# (6) Hashing doble o rehashing

- Idea: que el barrido también dependa de la clave.
- $h(c, i) = (h_1(c) + i\ h_2(c)) \bmod |A|$, $h_1(c)$ y $h_2(c)$ son dos funciones de hashing.

# (6) Hashing doble o rehashing

- Idea: que el barrido también dependa de la clave.
- $h(c, i) = (h_1(c) + i\ h_2(c)) \bmod |A|$, $h_1(c)$ y $h_2(c)$ son dos funciones de hashing.
- ¿Qué pasa con la aglomeración primaria y la secundaria?

# (6) Hashing doble o rehashing

- Idea: que el barrido también dependa de la clave.
- $h(c, i) = (h_1(c) + i\ h_2(c)) \bmod |A|$, $h_1(c)$ y $h_2(c)$ son dos funciones de hashing.
- ¿Qué pasa con la aglomeración primaria y la secundaria?
- El hashing doble es muy poco probable que tenga aglomeración primaria.

# (6) Hashing doble o rehashing

- Idea: que el barrido también dependa de la clave.
- $h(c, i) = (h_1(c) + i\ h_2(c)) \bmod |A|$, $h_1(c)$ y $h_2(c)$ son dos funciones de hashing.
- ¿Qué pasa con la aglomeración primaria y la secundaria?
- El hashing doble es muy poco probable que tenga aglomeración primaria.
- Y reduce los fenómenos de aglomeración secundaria.

# (7) No todo es natural...

- Si las claves no son naturales, ¿cómo construimos buenas funciones de hash?

# (7) No todo es natural...

- Si las claves no son naturales, ¿cómo construimos buenas funciones de hash?
- La idea general es asociar a cada clave un número entero, en una forma que dependerá del contexto.

# (7) No todo es natural...

- Si las claves no son naturales, ¿cómo construimos buenas funciones de hash?
- La idea general es asociar a cada clave un número entero, en una forma que dependerá del contexto.
- Ejemplo: código ASCII de cada caracter, multiplicado por cierta base elevada a la posición.

# (7) No todo es natural...

- Si las claves no son naturales, ¿cómo construimos buenas funciones de hash?
- La idea general es asociar a cada clave un número entero, en una forma que dependerá del contexto.
- Ejemplo: código ASCII de cada caracter, multiplicado por cierta base elevada a la posición.
- Ie, $\text{ascii}(\text{prim}(s)) \ \ 2^0 + \text{ascii}(\text{prim}(\text{resto}(s))) \ \ 2^1 + \ldots$

# (8) Más allá de mod

- Una vez que nuestras claves son numéricas tenemos que ver cómo terminar de definir la función.

# (8) Más allá de mod

- Una vez que nuestras claves son numéricas tenemos que ver cómo terminar de definir la función.
- Hay varias alternativas que varían en complejidad y en comportamiento con respecto a la aglomeración.

# (9) Basadas en división

- $h(c) = c \bmod |A|$

# (9) Basadas en división

- $h(c) = c \bmod |A|$
- Tiene baja complejidad pero...

# (9) Basadas en división

- $h(c) = c \bmod |A|$
- Tiene baja complejidad pero...
- Si $|A| = 10^p$ para algún $p$ todas las claves cuyos últimos digitos coincidan colisionarán...

# (9) Basadas en división

- $h(c) = c \bmod |A|$
- Tiene baja complejidad pero...
- Si $|A| = 10^p$ para algún $p$ todas las claves cuyos últimos digitos coincidan colisionarán...
- Y si $|A| = 2^p$ para algún $p$ lo mismo sucederá con los $p$ bits menos significativos.

# (9) Basadas en división

- $h(c) = c \bmod |A|$
- Tiene baja complejidad pero...
- Si $|A| = 10^p$ para algún $p$ todas las claves cuyos últimos digitos coincidan colisionarán...
- Y si $|A| = 2^p$ para algún $p$ lo mismo sucederá con los $p$ bits menos significativos.
- En general, la función debería depender de todas las cifras de la clave, cualquiera sea la representación.

# (9) Basadas en división

- $h(c) = c \bmod |A|$
- Tiene baja complejidad pero...
- Si $|A| = 10^p$ para algún $p$ todas las claves cuyos últimos digitos coincidan colisionarán...
- Y si $|A| = 2^p$ para algún $p$ lo mismo sucederá con los $p$ bits menos significativos.
- En general, la función debería depender de todas las cifras de la clave, cualquiera sea la representación.
- Una buena elección en la práctica: un número primo no demasiado cercano a una potencia de 2 (eg, $h(c) = c \bmod 701$ para $|A| = 2048$ valores posibles).

# (10) Partición y extracción

- Pueden ser útiles cuando la clave es muy larga.

# (10) Partición y extracción

- Pueden ser útiles cuando la clave es muy larga.
- La idea es pensar a $c$ como compuesto por segmentos $c_1 c_2 \ldots c_n$.

# (10) Partición y extracción

- Pueden ser útiles cuando la clave es muy larga.
- La idea es pensar a $c$ como compuesto por segmentos $c_1 c_2 \ldots c_n$.
- En el caso de partición, se busca calcular $h(c) = h'(c_1, c_2, \ldots, c_n)$

# (10) Partición y extracción

- Pueden ser útiles cuando la clave es muy larga.
- La idea es pensar a $c$ como compuesto por segmentos $c_1 c_2 \ldots c_n$.
- En el caso de partición, se busca calcular $h(c) = h'(c_1, c_2, \ldots, c_n)$
- Ejemplo: partir el número de la tarjeta de crédito en cuatro partes y luego hacer $(c_1 + c_2 + c_3 + c_4) \bmod 701$.

# (10) Partición y extracción

- Pueden ser útiles cuando la clave es muy larga.
- La idea es pensar a $c$ como compuesto por segmentos $c_1 c_2 \ldots c_n$.
- En el caso de partición, se busca calcular $h(c) = h'(c_1, c_2, \ldots, c_n)$
- Ejemplo: partir el número de la tarjeta de crédito en cuatro partes y luego hacer $(c_1 + c_2 + c_3 + c_4) \bmod 701$.
- En el caso de extracción, la idea es quedarse sólo con algunos de los $c_i$.

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.
- Se usan mucho en seguridad.

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.
- Se usan mucho en seguridad.
- En cripto se utilizan hashes *de una vía*.

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.
- Se usan mucho en seguridad.
- En cripto se utilizan hashes *de una vía*.
- Es decir, la idea es que sea prácticamente imposible obtener la preimagen.

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.
- Se usan mucho en seguridad.
- En cripto se utilizan hashes *de una vía*.
- Es decir, la idea es que sea prácticamente imposible obtener la preimagen.
- Se suele pedir que cumplan con:

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.
- Se usan mucho en seguridad.
- En cripto se utilizan hashes *de una vía*.
- Es decir, la idea es que sea prácticamente imposible obtener la preimagen.
- Se suele pedir que cumplan con:
  - Resistencia a la preimagen. Dado $h$ debería ser difícil encontrar un $m$ tal que $h = \text{hash}(m)$.

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.
- Se usan mucho en seguridad.
- En cripto se utilizan hashes *de una vía*.
- Es decir, la idea es que sea prácticamente imposible obtener la preimagen.
- Se suele pedir que cumplan con:
  - Resistencia a la preimagen. Dado $h$ debería ser difícil encontrar un $m$ tal que $h = \text{hash}(m)$.
  - Resistencia a la segunda preimagen. Dado $m_1$ debería ser difícil encontrar un $m_2 \neq m_1$ tal que $\text{hash}(m_1) = \text{hash}(m_2)$.

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.
- Se usan mucho en seguridad.
- En cripto se utilizan hashes *de una vía*.
- Es decir, la idea es que sea prácticamente imposible obtener la preimagen.
- Se suele pedir que cumplan con:
  - Resistencia a la preimagen. Dado $h$ debería ser difícil encontrar un $m$ tal que $h = \text{hash}(m)$.
  - Resistencia a la segunda preimagen. Dado $m_1$ debería ser difícil encontrar un $m_2 \neq m_1$ tal que $\text{hash}(m_1) = \text{hash}(m_2)$.
  - Etc.

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.
- Se usan mucho en seguridad.
- En cripto se utilizan hashes *de una vía*.
- Es decir, la idea es que sea prácticamente imposible obtener la preimagen.
- Se suele pedir que cumplan con:
  - Resistencia a la preimagen. Dado $h$ debería ser difícil encontrar un $m$ tal que $h = \text{hash}(m)$.
  - Resistencia a la segunda preimagen. Dado $m_1$ debería ser difícil encontrar un $m_2 \neq m_1$ tal que $\text{hash}(m_1) = \text{hash}(m_2)$.
  - Etc.
- Muy útiles para almacenar contraseñas (conviene que las contraseñas no se puedan leer). ¿Cómo hago?

# (11) No sólo de tablas vive la función de hash

- Las funciones de hash tienen interés más allá del mundo de las estructuras de datos.
- Se usan mucho en seguridad.
- En cripto se utilizan hashes *de una vía*.
- Es decir, la idea es que sea prácticamente imposible obtener la preimagen.
- Se suele pedir que cumplan con:
  - Resistencia a la preimagen. Dado $h$ debería ser difícil encontrar un $m$ tal que $h = \text{hash}(m)$.
  - Resistencia a la segunda preimagen. Dado $m_1$ debería ser difícil encontrar un $m_2 \neq m_1$ tal que $\text{hash}(m_1) = \text{hash}(m_2)$.
  - Etc.
- Muy útiles para almacenar contraseñas (conviene que las contraseñas no se puedan leer). ¿Cómo hago?
- ¿Y para asegurar que el software bajado es el correcto?

# (12) Bonus: un problema de seguridad

Ejemplo:

- Requerimiento: antender 1000 conexiones por segundo.

# (12) Bonus: un problema de seguridad

Ejemplo:

- Requerimiento: antender 1000 conexiones por segundo.
- Almacenamiento de las conexiones: tabla de hash, h= IP mod 1000

# (12) Bonus: un problema de seguridad

Ejemplo:

- Requerimiento: antender 1000 conexiones por segundo.
- Almacenamiento de las conexiones: tabla de hash, h= IP mod 1000
- Preparo el ataque: genero montones de pedidos de conexión que hasheen al mismo bucket.

# (12) Bonus: un problema de seguridad

Ejemplo:

- Requerimiento: antender 1000 conexiones por segundo.
- Almacenamiento de las conexiones: tabla de hash, h= IP mod 1000
- Preparo el ataque: genero montones de pedidos de conexión que hasheen al mismo bucket.
- Complejidad: pasó de $O(1)$ a $O(n)$ para las búsquedas.

# (12) Bonus: un problema de seguridad

Ejemplo:

- Requerimiento: antender 1000 conexiones por segundo.
- Almacenamiento de las conexiones: tabla de hash, h= IP mod 1000
- Preparo el ataque: genero montones de pedidos de conexión que hasheen al mismo bucket.
- Complejidad: pasó de $O(1)$ a $O(n)$ para las búsquedas.
- Ataque: fuerzo a la aplicación a realizar búsquedas en el bucket ($O(n)$), consumiendo todo el tiempo de CPU.

# (13) Más bonus: Universal Hashing

Concepto teórico: *Universal Hashing*

Si $k$ la cantidad de celdas de la tabla de hash y $H = \{h_1 : \alpha \to [1 \dots k], \dots, h_n : \alpha \to [1 \dots k]\}$ es un conjunto de funciones de hash, definimos la cantidad de colisiones de $H$ en los valores distintos $i$ y $j$ ($\#C_{i,j}^H$) como $\#\{h \in H \mid h(i) = h(j)\}$, es decir, la cantidad de funciones de $H$ que hashean al mismo valor los elementos $i$ y $j$.

# (13) Más bonus: Universal Hashing

Concepto teórico: *Universal Hashing*

Si $k$ la cantidad de celdas de la tabla de hash y $H = \{h_1 : \alpha \to [1 \dots k], \dots, h_n : \alpha \to [1 \dots k]\}$ es un conjunto de funciones de hash, definimos la cantidad de colisiones de $H$ en los valores distintos $i$ y $j$ ($\#C_{i,j}^H$) como $\#\{h \in H \mid h(i) = h(j)\}$, es decir, la cantidad de funciones de $H$ que hashean al mismo valor los elementos $i$ y $j$.

$H$ es *universal* ssi $\forall i \neq j \in \alpha : \#C_{i,j}^H \leq \frac{|H|}{k}$.

# (14) Más bonus: Universal Hashing (cont.)

$H$ es *universal* ssi $\forall i \neq j \in \alpha : \#C_{i,j}^{H} \leq \frac{|H|}{k}$.
¿Qué ventaja tiene que $H$ sea universal?

# (14) Más bonus: Universal Hashing (cont.)

$H$ es *universal* ssi $\forall i \neq j \in \alpha : \#C_{i,j}^H \leq \frac{|H|}{k}$.
¿Qué ventaja tiene que $H$ sea universal?

Analicemos la probabilidad de que dos funciones de $H$ colisionen en todos sus valores:

$$\frac{favorables}{totales = |H|} \leq \frac{min_{i,j}\ \#C_{i,j}^H}{|H|} \leq \frac{\frac{|H|}{k}}{|H|} = \frac{1}{k}$$

Cuando $k \to \infty \ \Rightarrow\ \frac{1}{k} \to 0$.

# (14) Más bonus: Universal Hashing (cont.)

$H$ es *universal* ssi $\forall i \neq j \in \alpha : \#C_{i,j}^H \leq \frac{|H|}{k}$.
¿Qué ventaja tiene que $H$ sea universal?

Analicemos la probabilidad de que dos funciones de $H$ colisionen en todos sus valores:

$$\frac{favorables}{totales = |H|} \leq \frac{min_{i,j}\ \#C_{i,j}^H}{|H|} \leq \frac{\frac{|H|}{k}}{|H|} = \frac{1}{k}$$

Cuando $k \to \infty \ \Rightarrow\ \frac{1}{k} \to 0$.

Entonces: selecciono la función de hash de una familia universal, al azar en tiempo de ejecución. El atacante no sabe cuál es la seleccionada (aunque conozca $H$). Aunque sus "chances" de adivinar la función son $\frac{1}{n}$, la probabilidad de que todos sus pedidos de conexión hasheen a la misma celda tiende a cero con $k$.