Tablas de hash 1

Fernando Schapachnik[1,2]

[1]En realidad... push('Fernando Schapachnik', push('Esteban Feuerstein', autores))
[2]Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Algoritmos y Estructuras de Datos II,
segundo cuatrimestre de 2018

(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 α 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.
- 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 β son todos los datos de la persona.

(5) Relajemos (cont.)

- Entonces, en lugar de tener un arreglo: $A[1 \ldots N]$ de β...
- ...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 α 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 $\alpha \to [0 \ldots k-1]$.
- Tarea: pensar en invariante y función de abstracción.
- (Cada posición del arreglo se suele llamar *bucket*.)
- Si h distribuye "bien" a los α 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:
  $\forall c_1, c_2 \in \alpha, c_1 \neq c_2 \Rightarrow h(c_1) \neq h(c_2)$
- 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"?
- Supongamos que conocemos la distribución de frecuencias de α 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 α 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 $h(c) = c \bmod 5$.
- 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?
- (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.
- 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.
- 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 $fc = n/k$.
- 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.
- 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:
  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 $h(c, i)$,
- los problemas que se pueden presentar.
- y cómo solucionarlos.