

2015年10月5日 (月)

Diccionarios en memoria secundaria.

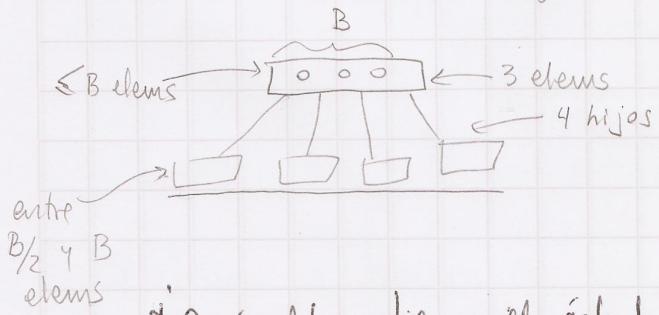
| | | |
|--------|------------|----------------------|
| B-Tree | : insertar | sucesor |
| | : borrar | predecesor |
| | : buscar | ↳ iterar en orden... |

Idea básica: generalizar un árbol binario

• Cada nodo tiene tamaño B (unidad mínima de I/O)

• Invariantes:

- cada nodo almacena al menos B elementos
- salvo la raíz, cada nodo tiene al menos $B/2$ hijos
- un nodo interno con k elementos tiene $k+1$ hijos
- todas las claves en el i -ésimo hijo ($1 \leq i \leq k+1$) están, en valor, entre las claves de los elementos $i-1$ e i .
- todas las hojas tienen la misma profundidad.



¿Qué altura tiene el árbol? $h = \Theta(\log_B N)$

- Invariantes de la estructura:

- a) Cada nodo almacena a lo más B elementos.
- b) Salvo la raíz, cada nodo tiene al menos $B/2$ elementos.
- c) Un nodo interno con k elementos tiene $k+1$ hijos.
- d) Todas las claves en el i -ésimo hijo ($1 \leq i \leq k+1$) están, en valor, entre las claves de los elementos $i-1$ e i .
- e) Todas las hojas tienen la misma profundidad.

- ¿Qué altura tiene el árbol?

Debido a que todas las hojas están a la misma profundidad, se deduce que $h = O(\log_B N)$.

- Los algoritmos:

- a) Busqueda : (k)

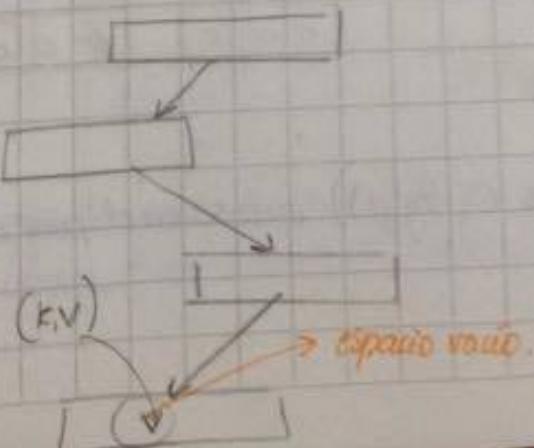
↳ Bajar por el árbol eligiendo el hijo adecuado en cada vez (toma $O(\log_B N)$).

Lo complicado es insertar / Borrar.

- a) insertar : (k,v)

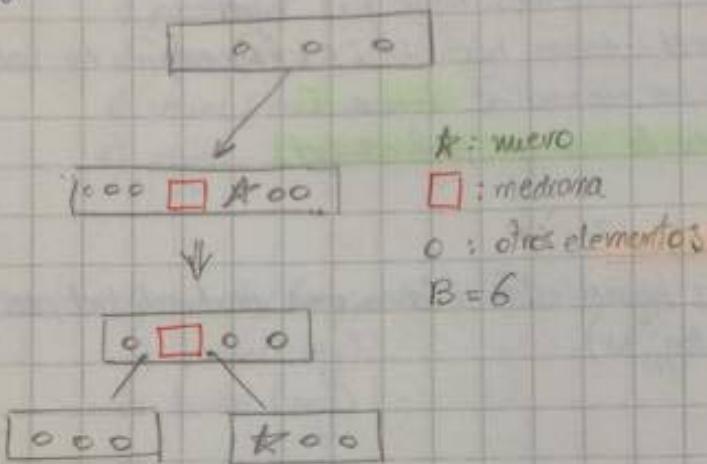
↳ Se busca la clave a insertar; al no encontrarse, encontramos el lugar donde debería estar (en una hoja). $\{O(\log_B N)\}$

a)



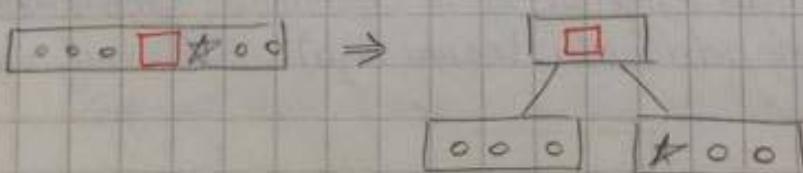
$O(\log n)$

- b) Se inserta el dato en la hoja. Si la hoja queda con $B+1$ elementos, tenemos un "overflow". Entonces dividimos la hoja en dos y se promueve la mediana al padre.



- ↳ Se agregó un elemento al padre, luego, puede recursivamente haber overflow en los ancestros de la hoja que tuvo problemas.

Si la raíz sufre overflow, se crea una nueva raíz de 1 sólo elemento, y por ende, la altura del árbol crece en 1.

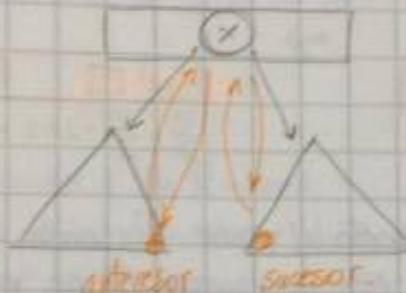


⇒ Tenemos que insertar es $O(\log_B N)$ incluso en el peor caso.

3) Borrado (e):

Buscar y encontrar la clave.

- Si es un nodo interno, se intercambia el elemento con su sucesor o antecesor (que estará en una hoja).

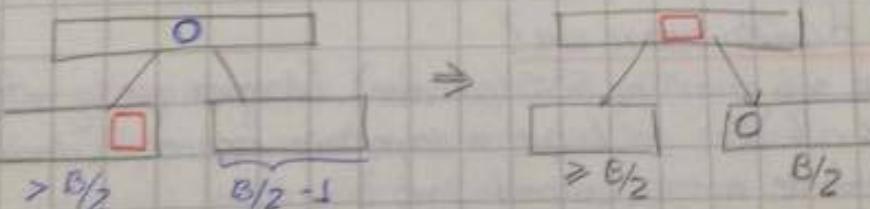


⇒ sin pérdida de generalidad, el borrado **ocurre en las hojas**.

- En cambio, si tenemos que borrar en una hoja.
Se borra, y si la hoja queda con menos de $B/2$ elementos.

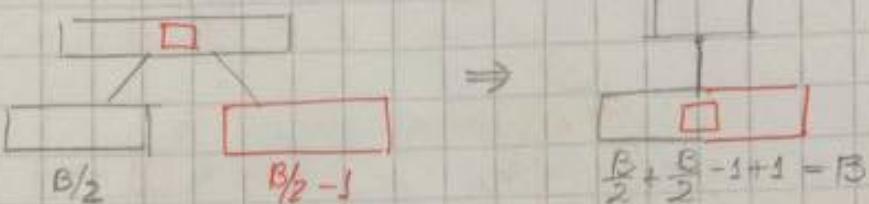
↳ Pido los vecinos y le pido un elemento a alguno :

En el caso del izquierdo:



Análogo en el caso de vecino derecho. Esto funciona siempre y cuando cada bloque tenga más de $B/2$ elementos.

- Si ambos tienen $\frac{B}{2}$ elementos, no puedo pedirle elementos. Entonces elijo uno de ellos y me fusiono con él (y con el elemento "separador" en el padre).



Notemos que le quitamos un elemento al padre. Por ende, el padre puede sufrir underflow. \Rightarrow Esto recursivamente se va propagando a la raíz.

- Como la raíz no tiene restricciones, fusionamos y creamos una raíz más grande.



\Rightarrow De nuevo es $O(\log_B N)$.

• Problemas: ¿Cuánto espacio usa?

la estructura sólo garantiza 50% (por caso). [Asegura que la mitad de los nodos está llena, por lo que se desperdicia espacio, es decir, si tenemos 1 mb de números, ocuparía 2 mb.]

El caso promedio $\approx 69\%$ (similar a logaritmo (2)).

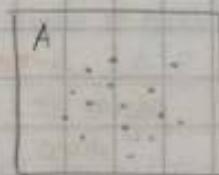
• Una variante útil son los Bt árboles.

- ↳ Todos los elementos (datos) sólo están en las hojas. En los nodos internos tenemos sólo punteros ("copias" de llaves).
- ↳ Las hojas tienen punteros entre ellas, por lo que se puede hacer un recorrido secuencial a través de ello.

R-trees

- ↳ Son muy parecidos a los R-trees.

- 1) Cada nodo interno almacena hasta B rectángulos.
- 2) Salvo la raíz, todo nodo interno tiene al menos $B/2$ rectángulos.
- 3) Todas las hojas tienen igual profundidad.
- 4) Las hojas almacenan la info geométrica (valores).
- 5) Si tenemos k rectángulos \rightarrow hay k hijos.
- 6) Cada rectángulo cubre los rectángulos de sus hijos. (totalmente)



↳ La raíz tendrá el bounding box (el rectángulo mayor) que cubre a todos los demás.

- Al insertar, uno debe escoger un criterio para ver por qué hijo hay que descender. (Puede ser por el que crece menos en área, o bien, el que me hace más overlapping, es decir, el que me contenga más).



↳ Contiene a todos los demás rectángulos.

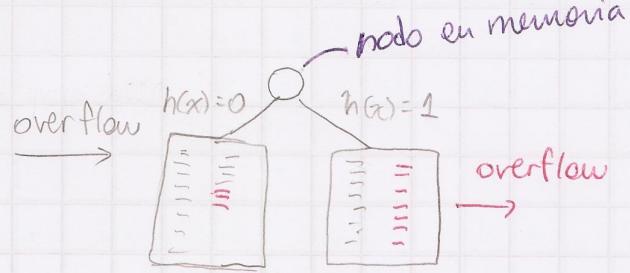
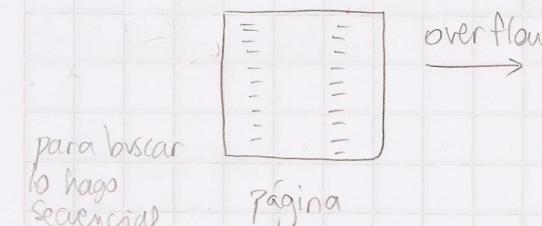
Se pueden solapar.

Hashing en Disco

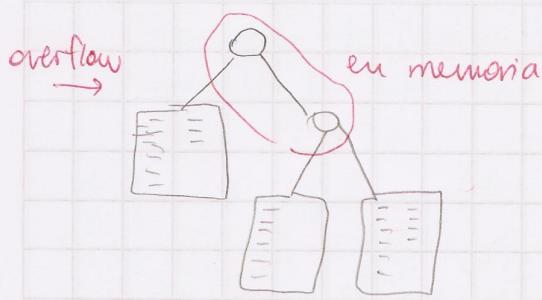
2015年10月6日(木)

- Hashing extendible → exige cierto espacio en mem. (desv: puede haber mucho espacio vacío en los bloques)
- Hashing lineal → no exige RAM

Hashing Extendible:



es esperable que cada pág. esté llena a la mitad

$$\sim M > \frac{2^n}{B}$$


Las hojas son punteros a disco
⇒ búsqueda = 1 acceso
(pues los nodos están en memoria.)

⇒ inserción = 3 accesos (?)

2 sin overflow: leer y escribir

3 con " " : leer y escribir 2 págs

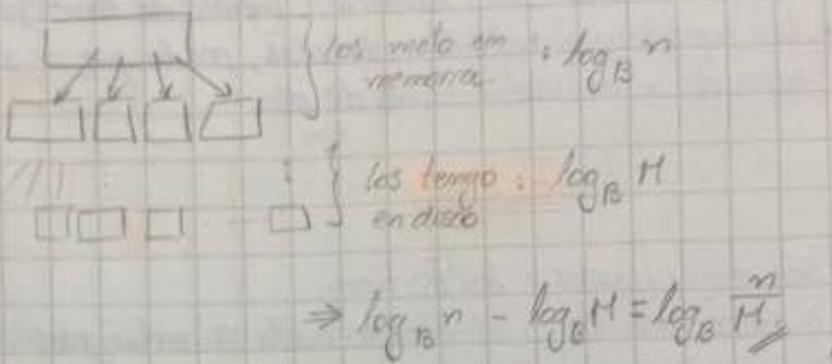
Si hay varios overflows seguidos, las páginas vacías las dejo NULL.

⇒ garantizado 50% ocupación

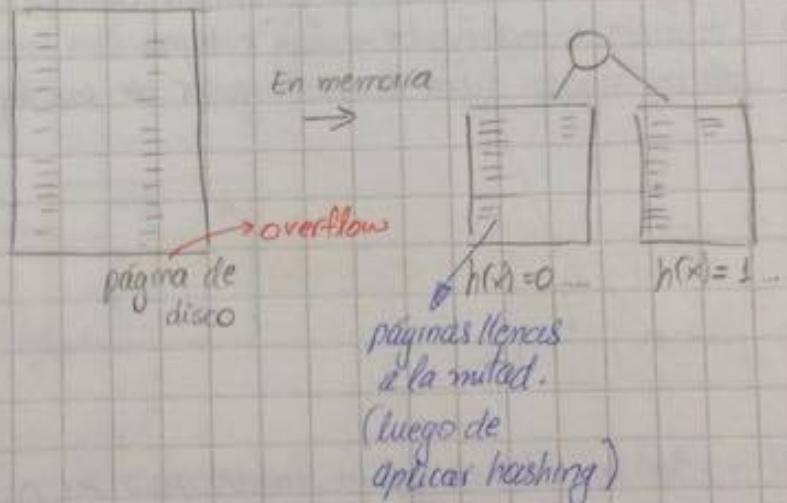
Hashing en disco

06/0ctubre/2015

- Hashing extendible
- Hashing lineal



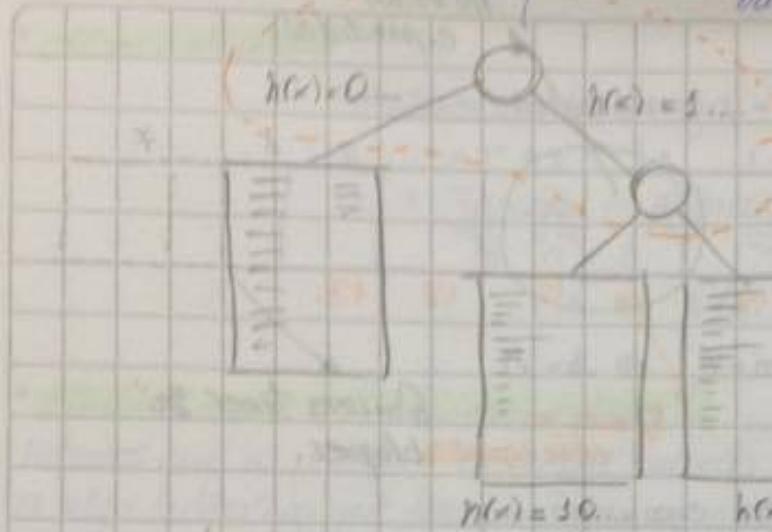
• Hashing extendible



Luego, para ello necesitamos tener $M \geq \frac{2}{B} n$.

¿Qué pasa si seguimos insertando, y tenemos overflow?

* Tries (Arboles que permiten búsquedas de strings)



* Índice que está en memoria, y que tiene páginas (punteros a ellas).

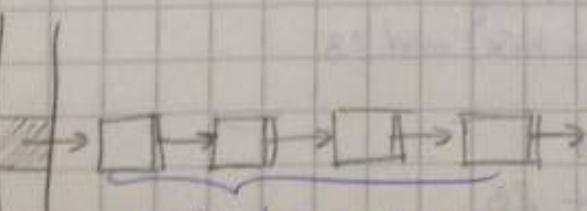
(En promedio es $\sim 67\%$).

* Tenemos una ocupación de un 50% (la mitad de las páginas ocupadas)

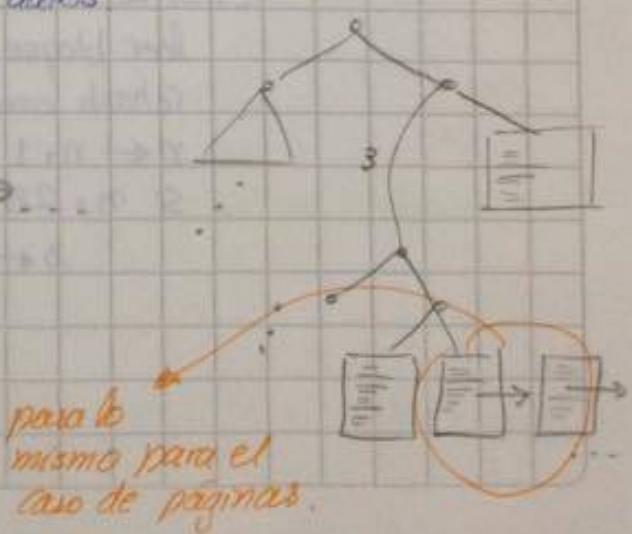
- Búsqueda en esta estructura es 1 acceso.
- Inserción son 3 accesos. (leer / escribir si no hay overflow son 2 accesos, y si leemos / 2 escribimos con overflow, pues tenemos 2 páginas que escribir)

↳ Sin embargo: 1) ¿Qué pasa si se acaba el hash?
 2) ¿Memoria interna suficiente?

Si la función de hash es mala, entonces hay colisiones. No hay mucho que hacer, al igual que en estructura de datos.



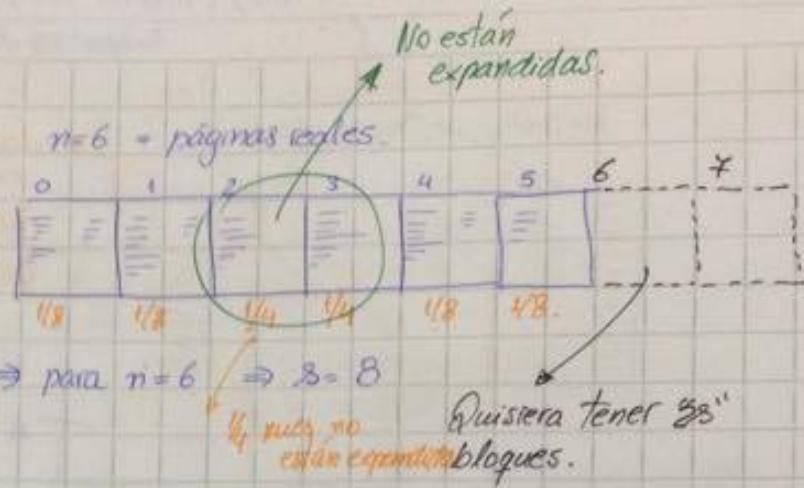
todos tienen el mismo $h(x)$, no difieren en ningún bit.



Hashing Lineal:

Arreglo de bloques en discos

$$2s = 2^{\lceil \log_2 n \rceil}$$



↳ hashing lineal va haciendo una expansión de la tabla de a poco. (Siempre voy buscando $2s$ =potencia próxima de dos). Supongamos que tenemos 8 celdas como arriba. Si queremos que un elemento caiga en una celda "virtual", lo envío a la celda aun no expandida. El algoritmo es:

| | | |
|--|---|-----------------------------|
| $S: h(x) \bmod s < n \bmod s$ $\quad \quad \quad \text{return } h(x) \bmod 2s$ $\text{return } h(x) \bmod s$ | } | supondremos $s \leq n < 2s$ |
|--|---|-----------------------------|

• ¿Cómo expandir?

Expandir Tabla:

```

leer bloque n-s
(rehash con  $h(x) \bmod 2s$ )
 $n \leftarrow n+1$ 
S:  $n=2s$ 
 $s \leftarrow 2s$ 

```

• **¿Cómo contraer?**

Contraer - Tabla :

$$n \leftarrow n-1$$

agregar la página n a la $n-3$

$$\text{si } n = 3$$

$$z \leftarrow z/2$$

• **¿Cuando expandir? ¿Cuando contraer?**

Notemos que no podemos expandir cuando hay overflow, ya que es algo restringente que trae consigo un gasto grande. Entonces las páginas con overflow se enlazarán en una lista, esperando su turno. Cuando toque la expansión, se leerá toda la lista y se trasladarán los elementos (todos los de la lista enlazada) en las nuevas posiciones (incluidas las celdas extendidas).

Estrategia (1): expandir cuando sube el tiempo promedio de búsqueda, y contraer en el caso contrario.

Estrategia (2): contraer cuando la tasa de ocupación es baja, y expandir en el caso contrario.

$$\square^* = \left(\frac{1}{1-\square} \right)$$

$$\sum_{i=1}^n a^i = \frac{1-a^{i+1}}{1-a}$$

2015年10月8日 (木)

Análisis amortizado.

→ Análisis de costo de una secuencia de operaciones ≠ ^{promedio sobre inputs posibles} constante amortizado

- ▶ Análisis global
- ▶ Contabilidad de costos
- ▶ Función potencial.

ANÁLISIS GLOBAL

Ej: contador de bits.

peor caso = k (ancho de k bits)

costo total = kn , $n=2^k$

$$1 \cdot \frac{n}{2} + 2 \cdot \frac{n}{4} + 3 \cdot \frac{n}{8} + \dots \quad (a=\frac{1}{2})$$

$$\leq \sum_{i \geq 1} i \cdot \frac{n}{2^i} = n \sum_{i \geq 1} \frac{i}{2^i} = an \sum_{i \geq 1} (a^i)^{-1} = an \sum_{i \geq 1} (a^i)^{-1}$$

$$= na \left(\sum_{i \geq 1} a^i \right)^{-1} = an \left(\frac{1}{1-a} - 1 \right)^{-1} = \frac{an}{(1-a)^2} = 2n$$

| k bits | |
|------------------|-------|
| 000 | 000 |
| 000 | 001 1 |
| 000 | 010 2 |
| 000 | 011 1 |
| 000 | 100 3 |
| 000 | 101 1 |
| ... | ... |
| 0111111111111111 | 4 |
| 1000000000000000 | (k) |

o también $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots = 2n$ (ver por columnas)

⇒ costo amortizado de incrementar es 2.

No importa esto
Los otros

(3) BARRA DE COSTOS

costos de cambio estrictos

CONTABILIDAD DE COSTOS

EJ: Contador de bits de nro.

- Por cada $0 \rightarrow 1$ le voy a cobrar 2 a la operación, adelantándose a su próximo cambio de $1 \rightarrow 0$
- los cambios $1 \rightarrow 0$ son gratis.

000 ... 000
000 ... 001
000 ... 010
000 ... 011
000 ... 100
000 ... 101
000 ... 110
000 ... 111

hay 1 rectángulo por cambio de contador
→ después de n cambios se ha cobrado $2n$

→ alcancía

Función potencial: ϕ es como hacer contabilidad también. Tengo un saquito de ahorro.. ϕ es un valor numérico, en función del estado de la estructura de datos.

c_n = costo real de la n -ésima operación

ϕ_n = el valor de ϕ (la alcancía) después de la n -ésima op.

\hat{c}_n = costo amortizado de la n -ésima operación.

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1} = c_i + \Delta\phi_i$$

Luego

$$\sum_{i=1}^n \hat{c}_i = \sum_{i=1}^n c_i + \underbrace{\phi_1 - \phi_0}_{\geq 0} \geq \sum_{i=1}^n c_i$$

cota superior al costo real.



Depth

Ej: Contador de bits.

$$\phi = \sum 1s \text{ en la cadena (total)}$$

0111

↓ +1

$c_i = \# 1s$ seguidos de der a izq + 1.

1000

Luego $\Delta \phi_i$ es la variación de 1's

($-c_{i-1}$)

$$\Delta \phi_i = -\# 1s + 1 \quad (\text{la cantidad de } 1s \text{ en el estado anterior})$$

$$\hat{c}_i = c_i + \Delta \phi_i = 2$$

$$\sum \hat{c}_i = 2^n \geq \sum c_i$$

Otro ejemplo DFS en un grafo para detectar componentes conexas.

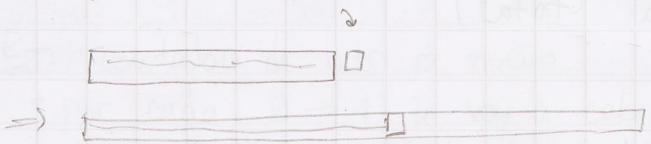


$$\sum_{v \in V} \text{arity}(v) = 2e$$

en vez de ver qué pasa en los nodos, es más fácil cobrar 2 (por adelantado) a cada arista que tenemos.

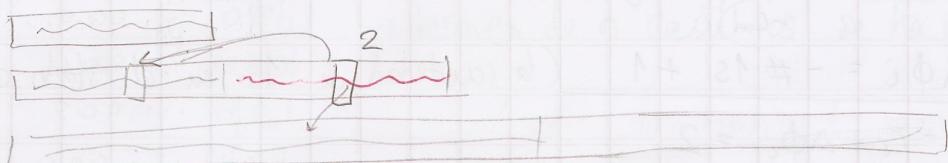
⇒ hemos usado una técnica de análisis amortizado para calcular costo global $\Theta(n+e)$

Ej: reallocar duplicando el tamaño.



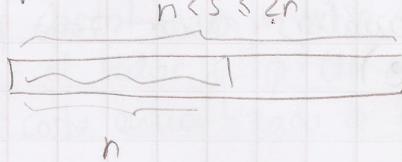
• Contabilidad de costo.

para n ops, cuántas veces se duplica el elemento? peor caso $\log n$...



A los elementos que son insertados en la segunda mitad le cobro 1 por insertar + 2 por su futura copia y por copiar el otro elemento simétrico en la primera mitad.
La gracia es que un elto rojo nunca va a volver a ser rojo de nuevo, y siempre alguien va a pagar por su copia posterior
⇒ n ops cuesta $3n$ → costo amortizado 3 por op.

• Función potencial



$$\phi = 2n - s \rightarrow \text{en una inserción normal (sin reallocación)}$$

$$c_i = 1$$

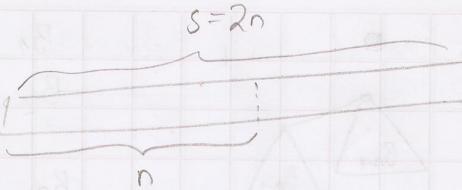
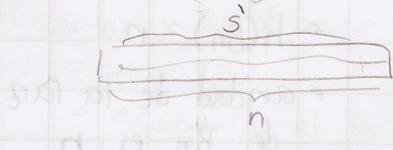
$$\rightarrow \Delta c_i = 2 \quad \text{s no varía}$$

$$\hat{c}_i = 3$$

s : tamaño de lo que alcancé

Admonitida

Cuando el arojo se realoca



$$C_i = n$$

$$\begin{aligned}\Delta\phi_i &= \phi_i - \phi_{i-1} = (2n - s) - (2n - s') \\ &= (2n - 2n) - (2n - n) \\ &= -n\end{aligned}$$

\Rightarrow

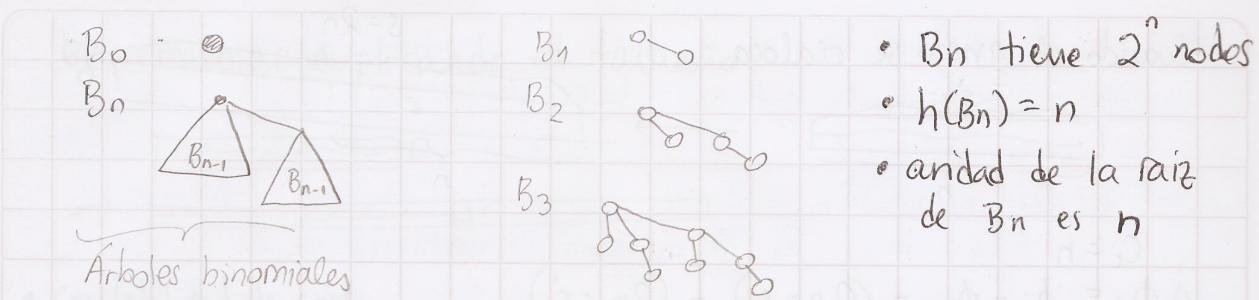
$$\hat{c}_i = 0 \quad \text{Pago con todos los ahorros.}$$

ya que me da 0 , no
debo preocuparme de
cuántas veces realoco
para

En el fondo ϕ está relacionado al espacio libre que queda

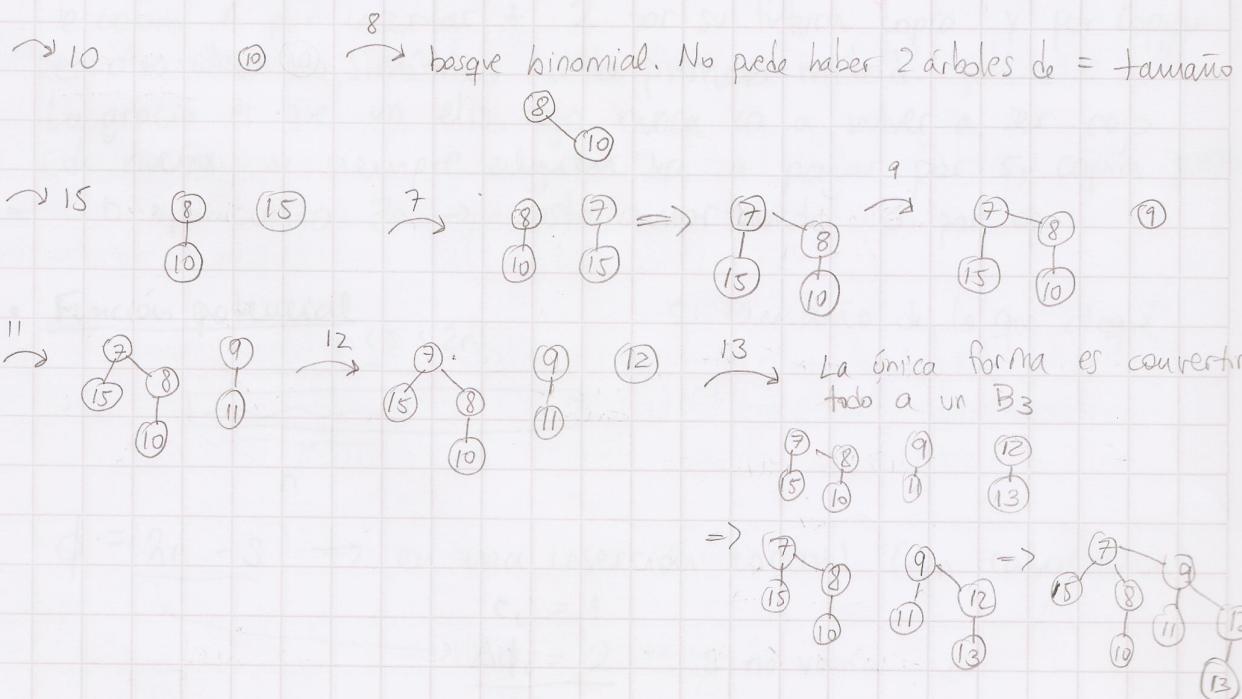
Colas Binomiales

2015年10月13日 (K)



→ Un bosque binomial es un conjunto de árboles binomiales, todos de distinto tamaño.

→ Una cola binomial es un bosque binomial con una clave asociada a cada nodo, donde la clave de un padre no puede ser mayor que la de un hijo.



Entonces ...

-> Insertar x en el bosque

- creo \otimes

- "sumo" \otimes al bosque

↳ si no hay un árbol del tamaño de \otimes
agregamos \otimes

↳ si hay, unimos el árbol de x con el de su mismo tamaño
colgando el de mayor raíz del de menor raíz.

(exactamente la misma mecánica de sumar 1 a un contador binario)

↳ Se obtiene un árbol del doble de tamaño, y se itera.

OBS: Una cola binomial con n claves tiene a lo sumo $\lceil \log_2 n \rceil$ árboles.

∴ La inserción cuesta $\Theta(\log n)$

OBS: podemos hacer heapify mediante insertar n elementos, a costo $\Theta(n)$

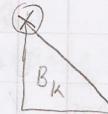
(producir esta estructura)

también es
peor caso
↳ pensar en el costo
amortizado de cada
operación de contador
binario)

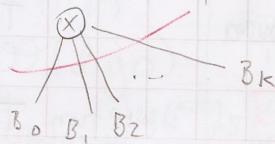
• Encontrar min: Buscar mínimo entre las raíces

↳ $\Theta(\log n)$ podría ser $\Theta(1)$ si recalcular min

• Extraer min: - Buscar raíz de clave mínima, sea

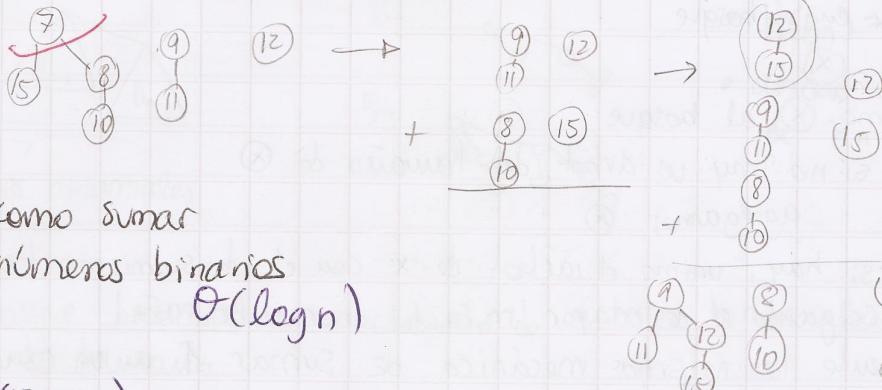


- Eliminar x y quedarse con los k subárboles



Colas Binomiales

- "Sumar" esos k subárboles al resto del bosque que → "reserva"



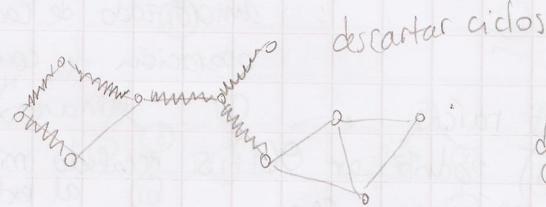
- Merge (T_1, T_2)

"Sumar" los bosques de T_1 y T_2
 $\Theta(\log(|T_1| + |T_2|))$

Union Find

Ej: MST (Minimum Spanning Tree)

Kruskal



Al principio, en C
cada nodo por si solo es una comp conexa

todavía
no tenemos
árbol que
conecte todos
el grafo

Kruskal (V, E)

dr a costa
arista

heapify (E)

$\rightarrow C \leftarrow \{\{v\}, v \in V\}$

$T \leftarrow \emptyset$

\rightarrow while $|C| > 1$

$(u, v) \leftarrow \text{extractMin}(E)$

$U \leftarrow \text{componente de } u \text{ en } C$

$V \leftarrow \text{componente de } v \text{ en } C$

\rightarrow if $U \neq V$

$T \leftarrow T \cup \{(u, v)\}$

$E \leftarrow E - \{(u, v)\} + \{(U \cup V)\}$

return T

A) BZ A01 #202

Jificaciones en numeros romanos

Los Agentes

1) BISQUEDA (E)

- Buscar el mejor resultado de acuerdo a la regla

- 2) (logística) buska en resultados random

2) INSECCION (el sistema basico) 3) observar cada uno

- 1) solo debe haber una insercion, ya no se consideran mas

- 2) solo se inserta una vez en la lista

- 3) solo se inserta una vez en la lista

- 4) solo se inserta una vez en la lista

- 5) solo se inserta una vez en la lista

- 6) solo se inserta una vez en la lista

- 7) solo se inserta una vez en la lista

- 8) solo se inserta una vez en la lista



$$(n_{\text{coll}}) \odot = d \quad ? \quad \text{Lo que se ve es que } d \text{ es}$$

Auxiliar 5 - Repaso para el control y algo de análisis amortizado

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

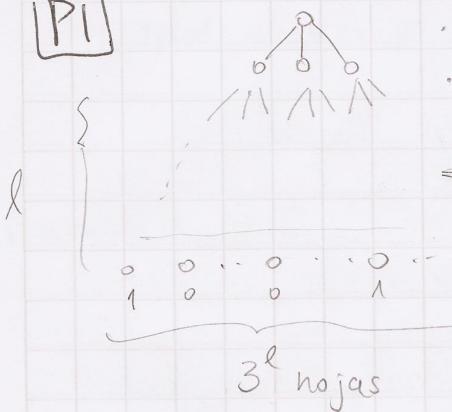
19 de Octubre del 2015

1. Considere un árbol ternario completo de altura l , en el que cada hoja contiene un valor en $\{0, 1\}$. Se define, para cada nodo interno, su valor como el mayoritario de sus hijos. Demuestre que cualquier algoritmo determinístico que determine el valor de la raíz debe examinar todas las 3^l hojas en el peor caso.
2. En Torre 15 están muy aburridos y decidieron determinar cuál es el primer piso inseguro del edificio. Un piso se considera seguro si se puede lanzar un estudiante huevo por la ventana sin que se rompa. Si se rompe el huevo, el piso se considera inseguro. Suponga para este problema que el primer piso inseguro es el L -ésimo.
 - Muestre que si sólo tiene un huevo, se puede determinar el primer piso inseguro con L tests.
 - Demuestre que si sólo se tiene un huevo, deben realizarse al menos L tests en el peor caso.
 - Muestre que si tiene dos huevos, puede encontrar el primer piso inseguro usando $O(\sqrt{L})$ tests.
 - Demuestre que si tiene dos huevos, debe realizar al menos $\Omega(\sqrt{L})$ tests en el peor caso.
 - **Propuesto:** Generalice al caso en que tiene k huevos.
3. Implemente stacks en memoria secundaria, de modo de obtener un costo de $O(1/B)$ escrituras a disco para las operaciones de **push** y **pop**.

Aux # 5

2015年10月19日 (A)

PI



- Árbol binario
- valor del padre es el mismo que la mayoría de los hijos
⇒ Cuál es el valor del padre?
- El algoritmo pide hojas en orden ALEATORIO
⇒ debo trabajar en función del orden en que me las preguntan.

Adv: La idea es que si el ALG no pregunta todos, A_0 y A_1 son respuestas

- Mantenemos 2 Árboles A_0 y A_1 \neq 's pero ambas
- Inicializo todas las hojas de A_0 con 0's Son VÁLIDAS
- " " " " " A_1 con 1's

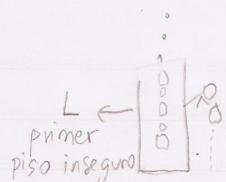
• Ante cada pregunta

- Si es el primer hijo de su trío en ser preguntado, respondo 0.
y actualizo A_1
- Si es el segundo hijo de su trío en ser preguntado, respondo 1
y actualizo A_0

- Si es el tercer hijo, subir un nivel y repetir

↳ nunca llega a la raíz si hay menos de 3^l preguntas

* Luego si el algoritmo responde 0, muestro A_1 y viceversa.
Luego un algoritmo que haga $< 3^l$ preguntas, no puede ser correcto.



P2 Buscar primer piso inseguro para tirar un huevo por la ventana

a) Si tengo 1 solo huevo

→ Se intenta desde el primer piso:

Se deja caer.

→ Si se rompe, ese piso era L

→ Si no, subo 1 piso

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} \Theta(L)$$

b) Definamos $f(i)$ el piso desde el que el algoritmo bota el huevo en el paso i

Adversario:

Si en el paso i , $f(i) > i$ → se rompió

Si no, no.

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} \Omega(L)$$

Si $f(i) > i$, el algoritmo no puede distinguir si $L = f(i)$ o $L = f(i) - 1$

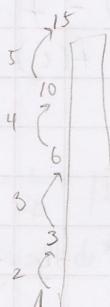
⇒ El algoritmo no puede ser correcto,

c) 2 huevos $\Rightarrow \Theta(\sqrt{L})$

• para el primer huevo:

for $i=1\dots$ hasta que se rompe:

bota el huevo desde el piso $\frac{i(i+1)}{2}$



• Para el 2º huevo:

Sea k la iteración en que se rompió el primer huevo.

for $i = \frac{k(k-1)}{2} + 1 \dots \frac{k(k+1)}{2} - 1$:

bota el huevo desde i

• Si el 2º huevo se rompe en un piso, ése será L .

• Si no, $L = \frac{k(k+1)}{2}$

$$\boxed{\begin{array}{|c|} \hline \text{ } \\ \hline \end{array}} \quad \binom{t}{2} + t = \binom{t+1}{2}$$

nivel anterior
 $\frac{(k-1)(k-1+1)}{2}$

Por def. del algoritmo, $L > \frac{k(k-1)}{2} \rightarrow k = \Theta(\sqrt{L})$

el 1º huevo se bota k veces $\Rightarrow \Theta(\sqrt{L})$

$$\text{el 2º huevo: } \frac{k(k+1)-1}{2} = \frac{k(k-1)+1}{2}$$

$$= \frac{k^2+k}{2} - \frac{k^2}{2} + \frac{k}{2} - 2 \\ = k-2$$

$$\Rightarrow \underline{\Theta(\sqrt{L})}$$

$$\binom{t}{2} = \frac{t!}{2!(t-2)!} + t$$

$$= \frac{t! + 2t(t-2)!}{2(t-2)!} \\ = \frac{t \cdot t-1 + 2t}{2(t-1)!} = \frac{t!(t+1)}{2(t-1)!}$$

d) El adversario elige un valor "grande" de L

De nuevo, $f(i)$ son los pisos testeados por el algoritmo en el piso i
(definimos $f(0) := 0$)

Dividimos el análisis en 2 casos:

$$\bullet f(i) - f(i-1) \leq i \forall i$$

$$\Rightarrow f(i) \leq \underbrace{i}_{2} \binom{i+1}{2} + i \text{ (Veo si } f(i) \geq L \text{)}$$

$$\sum_{j=0}^{i-1} j$$

\Rightarrow el algoritmo hace $\Omega(\sqrt{L})$ tests

$$\bullet \exists i^* \text{ tq } \underbrace{f(i^*) - f(i^*-1)}_t > i^*$$

El adversario declara que el huevo se rompió en $f(i^*)$
y define $L = f(i^*)$

De la parte 2, sabemos que entre $f(i^*-1)$ y $f(i^*)$ hay que hacer $t-2$ tests

$$L = f(i^*) = f(i^*-1) + t, \text{ pero } f(i^*-1) \geq \frac{i^*(i^*-1)}{2}$$

$$= \frac{i^*(i^*-1)}{2} + t$$

$$< \frac{t(t+1)}{2} + t = \binom{t}{2} + t = \binom{t+1}{2} \Rightarrow t = \Omega(\sqrt{L})$$

\Rightarrow el 2º huevo se bota $\Omega(\sqrt{L})$ veces.

Union Find

[P3]

Stacks en memoria secundaria.



LIFO

Naive:

↳ buffer de tamaño B en memoria.

• Pushes y pops van al buffer

→ si se llena, envío el bloque a disco

→ si se vacía, pido un bloque de disco

PROBLEMA: una secuencia de pushes y pops con buffer lleno resultan en muchos I/O

Idea: $2B$ en vez de B para el buffer.

• Si se llena, escribo solo B

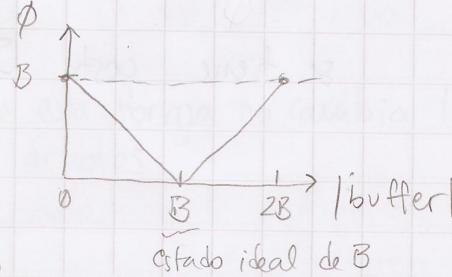
• Si se vacía, pido solo B .

ANÁLISIS: fn. potencial

Φ : • siempre positiva ($\Phi_k \geq \Phi_0 \forall k$)

$$\Phi: \begin{cases} |buffer| - B & \text{si } |buffer| \geq B \\ B - |buffer| & \text{si } |buffer| < B \end{cases}$$

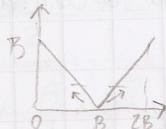
$$\Rightarrow \dot{\Phi} = |buffer| - B$$



Costos

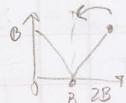
Push y Pop sin tocar disco

$\Rightarrow \pm 1$ (Operaciones en memo no cuentan, pero sí cuenta el cambio en la fn potencial, $\Delta\phi$)



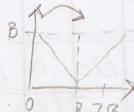
Push con overflow

$$\text{Costo: } B + \Delta\phi = B + -B = \emptyset$$



Pop con underflow

$$\text{Costo: } B + \Delta\phi = B + -B = \emptyset$$



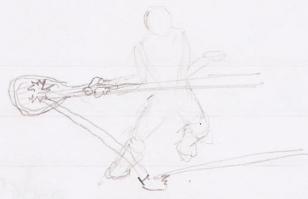
\Rightarrow Costo amortizado constante. Cuesta B mover un bloque

\rightarrow debe ser

$$\phi = |\text{buffer}| - B$$

se tiene costo $\mathcal{O}\left(\frac{1}{B}\right)$ cor op.

Union Find



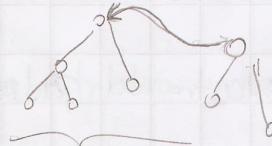
2015年10月20日 (X)

Ops: Make-set(x) \rightarrow crea conjunto $\{x\}$

Union(x, y) \rightarrow une los conjuntos de x e y

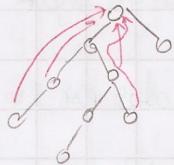
Find(x) \rightarrow retorna "label" del conjunto de x .

- Find retorna la raíz
- Union une 2 árboles



\rightarrow Link-by-rank: uno el de "menor altura"
($\text{rank}(x) = h(x)$) como hijo del de mayor
se actualiza

Path Compression:



Importante: al hacer compresión de caminos NO vamos a actualizar $\text{rank}(x)$ en el momento de la compresión $\Rightarrow \text{rank}(x) \geq h(x)$. (No como en el caso anterior)

Propiedades:

0: Agregar p.c. (path compression) de esta forma no cambia los ranks, raíces o elementos de los árboles

1: Si x no es una raíz,

$$\text{rank}(x) < \text{rank}(\text{padre}(x))$$

2: Si x no es una raíz, $\text{rank}(x)$ no cambia.

3: Si $\text{padre}(x)$ cambia, $\text{rank}(\text{padre}(x))$ crece

4: Toda raíz de $\text{rank}(k)$ tiene $\geq 2^k$ nodos en su árbol.

5: El rank más alto es $\leq \lfloor \lg n \rfloor$

6: Si $r \geq 0$, hay $\leq \frac{n}{2^r}$ nodos con rank r .

PROBLEMAS

①, 2, 4 y 5 son triviales

Le asigno un padre con rank mayor o igual
no es que le cambie el rank al nodo

1: P.C. solo incrementa el rank de los padres de los nodos, pues los enlaza a sus antecesores.

3: P.C. solo asigna un ancestro del parente original \Rightarrow por prop 1 se tiene

6: Si no hay p.c., la prop 4 también se cumple para nodos no-raíz.
(el rank solo aumenta cuando uno 2 árboles con raíces de igual rank... se "duplica" la cantidad de nodos, al menos)
(pues alguna vez fisionan raíces, y sus hijos quedan fijos al dejar de ser raíz)

\Rightarrow si no hay p.c. todo nodo de rank k tiene $\geq 2^k$ nodos en su árbol.

A demás, nodos distintos de igual rank no pueden tener descendientes comunes (por 1) (uno no puede ser antecesor del otro) \uparrow

\Rightarrow debe haber $\leq \frac{n}{2^k}$ nodos de rank k para cada k

(la propiedad se conserva cuando hay p.c., pues no cambia los ranks)

"Definimos" $\lg^* n$ como cuantas veces debo aplicar \lg para llegar a un $n^o \leq 1$.

$$\begin{array}{l} 1 \mapsto 0 \\ 2 \mapsto 1 \\ 2^2 \mapsto 2 \end{array} \quad \begin{array}{l} 2^2 \mapsto 3 \\ 2^{2^2} \mapsto 4 \end{array} \quad \begin{array}{l} 2^{2^2} = 2^{65536} \mapsto 5 \end{array}$$
$$\lg^* n = \begin{cases} n & \text{si } n \leq 1 \\ 1 + \lg^*(\lg(n)) & \text{si } n > 1 \end{cases} \sim$$

Consideremos los ranks de la estructura y hagamos grupos:

$$[\lg^{*-1}(i), \lg^{*-1}(i+1)]$$

- 1 $(0, 1]$ Prop: todo rank > 0 cae en los primeros $\lg^* n$ grupos.
- 2 $(1, 2]$
- 3 $(2, 2^2]$ (pues ranks $\leq \lfloor \lg n \rfloor - 1$)
- 4 $(2^2, 2^{2^2}]$
- 5 $(2^{2^2}, 2^{2^{2^2}}]$

• Contabilidad

- Cuando un nodo deja de ser raíz, le asignamos dulces \ddagger
Le damos 2^k si cae en el grupo k
 \Rightarrow pagamos $\leq n \lg^* n$

¿Por qué? por (b), el nº de nodos con rank $\geq k+1$ es

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \dots \leq \frac{n}{2^k} \Rightarrow \text{los nodos del grupo } k \text{ necesitan a lo más } n \text{ dulces}$$

Luego, como hay $\leq \lg^* n$ grupos, pago un total de $n \lg^* n$ en este punto.

n = cantidad de nodos de la estructura.

¿Cómo Analizar Find?

- Find está acotado por el n^o de punteros que se siguen hasta llegar a una raíz.

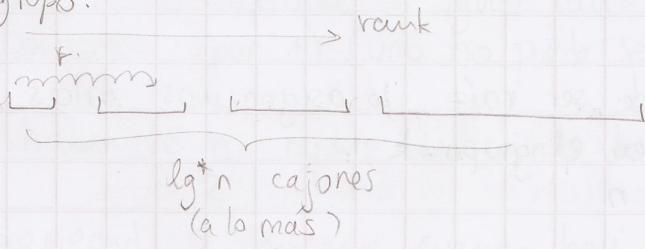
Dado $\text{Find}(x)$, hay 3 casos sobre x :

- 0: $\text{padre}(x)$ es una raíz (solo pasa 1 vez por Find)
- 1: $\text{rank}(\text{padre}(x))$ está en un grupo \neq al de $\text{rank}(x)$
- 2: $\text{rank}(\text{padre}(x))$ está en el mismo grupo que $\text{rank}(x)$

1: en este caso, a lo más $\lg^* n$ nodos pueden estar en un grupo más alto.

2: Se nos pide 1 para subir al padre. Cada vez que subimos, rank del "nodo presente" debe crecer

\Rightarrow Antes de pagar 2^k , vamos a llegar a un nodo de otro grupo.



- todos los saltos dentro del mismo cajón son GRATIS
- Si cambio de cajón, me cuesta 1.

\Rightarrow Luego el costo está acotado por $\lg^* n$ cajones.

• Una vez que estoy en un nodo con un rank de un grupo más alto, esto se puede mantener así, pues $\text{rank}(\text{padre})$ no baja

$\text{rank}(x)$ queda constante

$\Rightarrow x$ puede pagar hasta ser un caso 1.

Partiendo de una estructura vacía, $m \geq n$ finds y $n-1$ unions toman

$\Theta(m \lg^* n)$ ops.

≤ 5 cualquier cosa es $\ll 2^{65,522}$

Auxiliar 6 - Análisis Amortizado

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

26 de Octubre del 2015

1. Suponga que se le entrega una implementación de un *stack*, en la cual las operaciones PUSH y POP toman tiempo constante (¿cómo implementaría esto?). Utilice estructuras de este tipo para implementar una cola (*queue*) para la cual las operaciones ENQUEUE y DEQUEUE tienen un costo amortizado constante.
2. Un *quack* es una mezcla entre queues y stacks. Puede ser visto como una lista de elementos, escrita de derecha a izquierda, que soporta las siguientes operaciones:
 - QUACKPUSH(x) agrega x en el extremo izquierdo.
 - QUACKPOP(x) remueve y retorna el elemento más a la izquierda.
 - QUACKPULL(x) remueve y retorna el elemento más a la derecha.

Implemente un quack usando tres stacks (idénticos a los del problema anterior) de modo que cada operación tenga un costo amortizado constante. Puede utilizar $O(1)$ memoria. Almacene los elementos sólo una vez en cualquiera de los 3 stacks.

3. Considere una tabla de tamaño s que almacena n elementos, que se va llenando con inserciones y debemos reallocarla cuando no queda espacio para insertar ($s = n$). Esta vez, también ocurren borrados y no queremos que s sea mucho mayor que n , para no desperdiciar espacio. Al reallocar la tabla para agrandarla o reducirla debemos pagar un costo de $O(n)$.
 - (a) Muestre que duplicar la tabla cuando se llena, y reducirla cuando $n = s/2$ no consigue un costo amortizado constante.
 - (b) Considere la estrategia de duplicar cuando la tabla se llena, y reducirla a $s/2$ cuando $n = s/4$. Demuestre que esta estrategia obtiene un costo amortizado constante utilizando la siguiente función potencial:
$$\Phi = \begin{cases} 2 \cdot n - s & \text{si } n \geq s/2 \\ s/2 - n & \text{si } n < s/2 \end{cases}$$
 - (c) Analice el caso general en que queremos obtener un factor de carga α , con $0 < \alpha < 1/2$.
4. Suponga que tenemos un contador de modo que el costo de incrementar el contador es igual al número de dígitos (en el caso binario, bits) que deben ser modificados. Vimos en clases que si el contador comienza en 0, el costo amortizado de un incremento es $O(1)$. En este problema buscamos implementar un contador que permite *incrementos* y *decrementos*. Sólo consideraremos secuencias de operaciones que mantienen el contador en un valor no negativo.
 - (a) Muestre que incluso en secuencias que mantienen el contador con un valor no negativo, es posible que una secuencia de n operaciones que comience con el contador en 0 y que permita incrementos y decrementos tenga un costo amortizado de $\Omega(\log n)$ por operación (equivalentemente, un costo total de $\Omega(n \log n)$).

- (b) Para arreglar este problema, considere el siguiente *sistema de números ternario redundante*. Un número se representa como una secuencia de *trits*, que, como su nombre lo indica, pueden tomar tres posibles valores: 0, +1 o -1. El valor de un número t_{k-1}, \dots, t_0 (donde cada t_i es un trit) se define como

$$\sum_{i=0}^{k-1} t_i 2^i$$

El proceso de incrementar un número de este tipo es análogo al de la operación en números binarios. Se añade 1 al trit menos significativo; si el resultado es 2, se cambia el trit a 0 y se propaga una *reserva* hacia el siguiente trit. Se repite el proceso hasta que no exista carry.

El proceso de decrementar es similar: se resta 1 al trit menos significativo; si el resultado es -2, se reemplaza por 0 y se “pide” al siguiente trit.

Mediremos el costo de un incremento o decremento como el número de trits que se ve alterado. Comenzando de 0, se realiza una secuencia de n incrementos y decrementos (sin un orden particular).

Demuestre que, utilizando esta representación, el costo amortizado por operación es $O(1)$ (equivalentemente, que el costo total para las n operaciones es $O(n)$). Hint: Puede utilizar el método de contabilidad o de función de potencial para llegar al resultado.

Antes de Empezar...

El análisis amortizado nos sirve cuando ciertas operaciones son mucho más costosas que otras. Esto significa que analizar el peor caso (como hacíamos anteriormente) es bastante injusto con el algoritmo. Queremos una especie de costo promedio de una operación; sin embargo, no es el costo promedio usual: analizaremos *secuencias* de operaciones.

Recordemos que tenemos (al menos) tres formas de abordar el análisis amortizado:

- Hacer un **análisis global**, sumando todos los costos de alguna forma.
- Hacer **contabilidad** de costos, moviendo el costo de ciertas operaciones a otras. En otras palabras, reorganizamos los costos que queremos contar, de modo de que el conteo sea más sencillo. Una forma típica es asignar un costo adicional a operaciones pequeñas, que luego es cobrado al momento de realizar una operación costosa.
- Utilizar una **función potencial** ϕ . Luego estimamos el costo como el costo verdadero más las variaciones de potencial. Si la función de potencial está bien diseñada, típicamente cancelará el costo de las operaciones costosas. Es necesario, sí, que la función potencial que se utilice nunca tome valores menores al inicial (pues queremos siempre sobreestimar el costo del algoritmo a analizar).

Con esto en mente, resolvamos los problemas.

Soluciones

1. Usaremos dos stacks para simular la cola. El primer stack, IN, será la *entrada* de la cola, y el otro, OUT, será la *salida*. De esta forma, los algoritmos funcionan así:

- ENQUEUE(x): Hacemos IN.PUSH(x).
- DEQUEUE(x): Si OUT está vacío, repetimos OUT.PUSH(IN.POP()) hasta que IN esté vacío. Retornamos OUT.POP().

Notemos, primero, que tenemos que mostrar que la estructura se comporta como una cola. Como IN y OUT son stacks, si x entra a la estructura antes que y , saldrá *después* que y de IN, y por lo tanto saldrá *antes* de OUT. Por lo tanto, la estructura se comporta como una cola.

Ahora es necesario ver que el costo amortizado de las operaciones es constante. Notemos que DEQUEUE es la operación “problemática”, pues el caso en que OUT está vacío puede resultar en muchas operaciones. La gracia está en notar que si bien pueden moverse muchos elementos de IN a OUT, cada uno de esos elementos *entró de a uno* a la estructura. Utilizamos entonces una estrategia de contabilidad.

Cada vez que un elemento entra a la estructura (a través de un ENQUEUE) le asignamos una ficha. De esta forma, ENQUEUE tendrá un costo amortizado $O(1)$ (es necesario sumar el costo de la ficha, pero $O(1) + 1 = O(1)$).

Los elementos utilizan esta ficha para pagar el costo de ser movidos a OUT, por lo que la operación de DEQUEUE sólo cuesta hacer el POP de OUT, que es $O(1)$. Es importante que los elementos sólo son movidos a OUT una vez en su “vida”: si no fuera así, no nos alcanzarían las fichas para pagar.

También se puede hacer el análisis utilizando $\phi = 2 \times |IN|$ como función potencial y considerando que se parte de una estructura vacía. Se los dejo propuesto, debería ser fácil :)

2. La forma de construir esta estructura es muy parecida a la parte anterior. En adición a los stacks IN y OUT, tendremos un stack TEMP. Mantendremos la cuenta de cuántos elementos tiene cada stack, lo que usa $O(1)$ de memoria adicional. Por simplicidad, consideraremos que las operaciones sobre los stacks cuestan exactamente 1.

Las operaciones son similares al caso anterior: QUACKPULL es básicamente un DEQUEUE, mientras que QUACKPOP es hacer un POP de IN. Es necesario, sí, definir lo que se hace si alguno de los stacks está vacío al momento de realizar un POP.

Si se hace un QUACKPULL de un OUT vacío, se mueve la mitad de IN a TEMP haciendo operaciones de PUSH y POP. Luego se mueve el resto de IN a OUT. Finalmente, se mueve el contenido de TEMP a IN. El caso de hacer un QUACKPOP cuando IN está vacío se trata de forma simétrica. La idea de esto es mantener elementos tanto en IN como en OUT, ya que nos pueden pedir un QUACKPULL o un QUACKPOP en el futuro.

Para el análisis utilizaremos una función potencial. A veces es útil pensar en la función potencial como “¿cuán indefensa está mi estructura ante las operaciones que puedan venir?”. Nuestro quack se vuelve vulnerable (en el sentido de que pueden aparecer operaciones costosas) cuando IN o OUT están vacíos. Al contrario, si ambos stacks tienen un tamaño similar, estamos súper preparados para cualquier operación que nos lancen. Usamos entonces la siguiente función potencial, que va con esta idea metida:

$$\phi = c||\text{IN}| - |\text{OUT}||$$

con c una constante que determinaremos en el camino.

Analicemos las operaciones con esta función potencial. Para esto, hay que agregar el $\Delta\phi$ al costo de cada operación.

- Un QUACKPUSH cuesta $1 + \Delta\phi = 1 \pm c = O(1)$ (pues $|\text{IN}|$ crece en 1 y $|\text{OUT}|$ queda igual).
- Un QUACKPOP que se da cuando IN no está vacío cuesta $1 + \Delta\phi = 1 \pm c = O(1)$, pues $|\text{IN}|$ disminuye en 1 y $|\text{OUT}|$ permanece igual. El mismo análisis se da en el caso de un QUACKPULL con un OUT no vacío.
- Si se hace un QUACKPULL cuando OUT está vacío, el costo original es $1 + 3|\text{IN}|$ debido a los traspasos de elementos entre IN, OUT y TEMP (no es difícil de verificar).

Notemos que partimos con OUT vacío y terminamos con la misma cantidad de elementos en IN y OUT. En otras palabras, $\Delta\phi = -c|\text{IN}|$. El costo amortizado, entonces, es $1 + 3|\text{IN}| - c|\text{IN}|$. En este punto somos astutos y elegimos $c = 3$, con lo que el costo amortizado resulta ser constante. El análisis de un QUACKPOP con un IN vacío es análogo, y el mismo valor de c sirve.

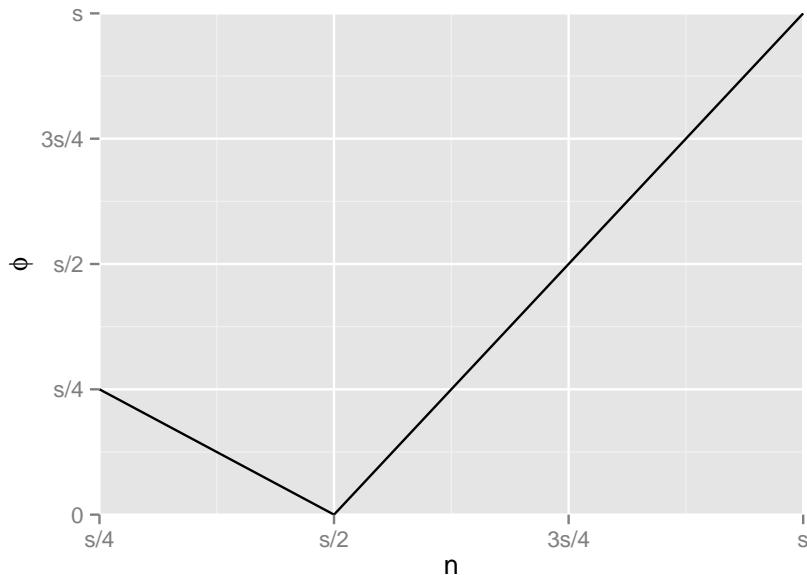
Así, cumplimos con todo lo que se nos pedía :)

3. (a) Duplicar la tabla cuando se llena y reducirla cuando llega a la mitad no es una buena estrategia. Consideremos una tabla que se encuentra llena con n elementos. La inserción de un nuevo elemento fuerza a duplicar la tabla, con un costo de n . Si a continuación se remueve un elemento, se fuerza a reducir la tabla, nuevamente con un costo de n . Podemos repetir estas dos operaciones de forma alternada, con el mismo costo cada vez. En otras palabras, encontramos una secuencia de inserciones y borrados que tiene un costo de $O(n)$ por operación, lo que es bastante malo.
- (b) Nos entregan la función potencial a utilizar. Sólo tenemos que calcular el costo de cada operación usándola, agregando los cambios en la función potencial. Consideremos por simplicidad que el costo de duplicar y el de reducir la tabla es n .

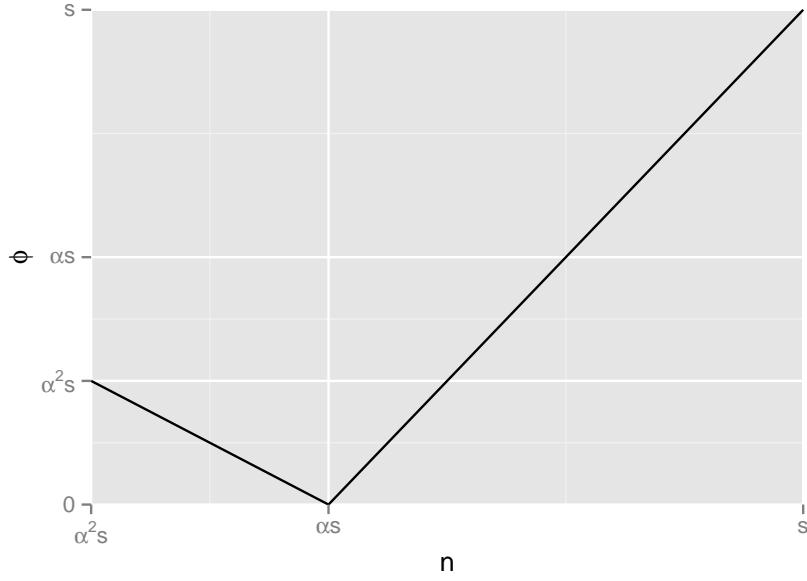
- Una inserción que no hace crecer la tabla tiene costo 1. El potencial puede cambiar en +2 o en -1, dependiendo de en qué tramo de la función potencial estemos, por lo que el costo amortizado puede ser 0 o 3, siendo constante en ambos casos.
- Un borrado que no hace crecer la tabla tiene costo 1. El potencial puede cambiar en -2 o en +1. Nuevamente, el costo amortizado resulta constante en ambos casos.
- Una inserción que duplica la tabla tiene costo $n + 1$. El potencial pasa de n a 0, por lo que $\Delta\phi = -n$. Así, el costo amortizado es sólo 1: el de insertar el nuevo elemento.
- Un borrado que reduce la tabla nuevamente tiene costo $n + 1$. El potencial pasa de $s/2 - n = 2n - n = n$ a 0, por lo que $\Delta\phi = -n$. Así, el costo amortizado es sólo 1: el de insertar el nuevo elemento.

Luego hemos demostrado que las operaciones toman un costo amortizado constante.

- (c) En el caso anterior las operaciones de duplicar y reducir la tabla siempre nos dejaban la tabla con $s/2$ elementos. Ahora queremos obtener un factor de carga α . Nos gustaría que la función potencial pagara por todo, como en el caso anterior. Dibujemos cómo se veía la función potencial antes, para ver cómo la modificaremos.



Lo que queremos ahora es que el llenado de la tabla esté en torno a α en vez de $1/2$. En otras palabras, cuando la tabla se llena, la agrandamos en un factor $1/\alpha$ (de modo de que quede llena en una fracción α), lo que nos cuesta $n = s$. Cuando la cantidad de elementos en la tabla disminuya a un nivel $\alpha^2 s$, la achicamos en un factor α , lo que nos cuesta $n = \alpha^2 s$. Entonces, nos gustaría una función potencial que tome la siguiente forma:



Esta función es la siguiente (es cosa de hacer un poco de ecuación de la recta):

$$\Phi = \begin{cases} n/\alpha - s & \text{si } n \geq \alpha s \\ \frac{-1}{\alpha(1-\alpha)}(n - \alpha s) & \text{si } n < \alpha s \end{cases}$$

Con esto, queda hacer el análisis: en particular, es necesario mostrar los valores que $\Delta\phi$ toma al agrandar y achicar la tabla.

Al agrandar la tabla, $\Delta\phi = 0 - s = -n$, pues $n = s$ en este caso (la tabla está llena). Luego una inserción que agranda la tabla tiene un costo amortizado de 1 (el cambio de potencial cancela el costo de mover los elementos a la nueva tabla).

Al achicar la tabla, $\Delta\phi = 0 - \alpha^2 s = -n$ pues $n = \alpha^2 s$ en este caso. Luego un borrado que achica la tabla tiene un costo amortizado de 1 (el cambio de potencial cancela el costo de mover los elementos a la nueva tabla).

Adicionalmente, las operaciones que no causan modificaciones en el tamaño de la tabla a lo más sufren un costo adicional constante (de $1/\alpha$ o de $1/\alpha(1 - \alpha)$). En conclusión, todas las operaciones tienen un costo amortizado constante.

4. (a) Sea $2^{b+1} \leq n \leq 2^{b+2}$. Usamos 2^b operaciones para llevar el contador a un 1 seguido de $b - 1$ ceros. Esto cuesta al menos 2^b . Luego alternamos decrementos e incrementos del contador. Cada una de estas operaciones cuesta $b - 1$, pues hay que flippear los últimos $b - 1$ bits del contador. Luego el costo total de estas operaciones es al menos $b \cdot 2^b$; el costo amortizado por operación será al menos $b \cdot 2^b/n \geq b/4 = \Omega(\log n)$.
- (b) Usaremos el método de contabilidad. Notemos que cuando el contador se incrementa o decremente, los 1s cambian a 0 o siguen en 1, y los -1 cambian a 0 o siguen igual. Luego podemos pagar 2 cada vez que un trit se convierte desde un 0 a otro valor. Con esto, cada trit tendrá suficiente para pagar cuando se convierta de nuevo en un 0.

Para acotar el costo amortizado, entonces, sólo nos faltaría demostrar que cada incremento no cambia demasiados trits que tengan 0's a otros valores (de lo contrario, tendríamos que pagar demasiadas monedas adicionales).

De hecho, lo que sucede es que cada incremento o decremento convierte **a lo más un trit que estaba 0 en otro valor**. ¿Por qué se cumple esto? Notemos que sólo traspasamos un carry cuando un 1 se convierte en un 2 o un -1 se convierte en un -2. Esto significa que cuando cambiamos un 0 a un 1 o a un -1, la operación de incremento

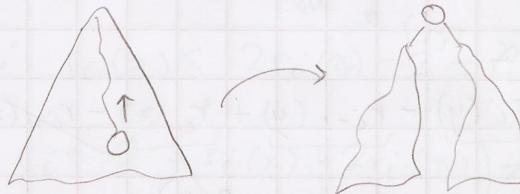
o decremento *debe* terminar. Luego al incrementar o decrementar pagamos 2 por el cambio del trit 0 a otro valor, que ocurre (a lo más) sólo una vez; los demás cambios en los trits son pagados por las monedas antes ahorradas.

Una forma de hacer el análisis usando una función de potencial es tomando ϕ = (número de trits en -1 o 1). Se los dejo propuesto.

SPLAY TREES

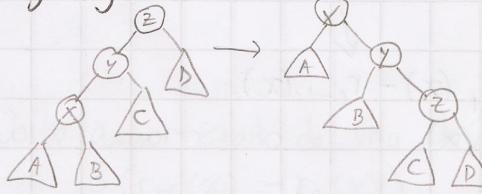
2015年10月27日 (K)

- Son árboles que se balancean de manera amortizada.
- Implica que en cada consulta el árbol cambia, en particular, al consultar un elemento, se harán balanceos de tal forma que el élto quede en la raíz.
- Una secuencia de n operaciones tiene costo amortizado de $\Theta(\log n)$ amortizado, si todos los éltos tienen prob uniforme $1/n$.
- El costo es $\Theta(H)$ entropía de Huffman.

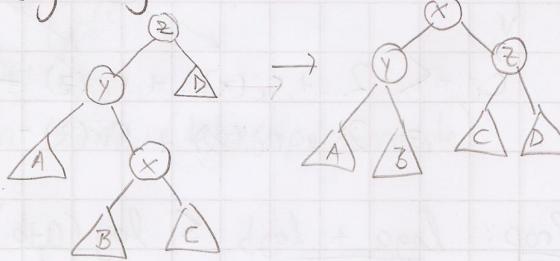


Operaciones de rotación (recordar AVL)

zig-zig

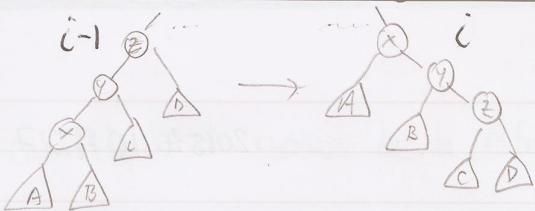


zig-zag



- zig-zig
- zig-zag
- zig } a dist 1 de la raíz
- tag }

Véamos costo de zig-zig ...



Fuimos a buscar x .

ESTA Y ATRÁS

$$S(x) = \text{tamaño} (\# \text{ de nodos}) \text{ del subárbol con raíz } x \text{ (contando } x\text{)}$$

$$r_i(x) = \log_2 S(x) \text{ luego de la operación } i$$

$$\phi_i = \sum_{x \in T} r_i(x)$$

En un zig-zig

$$c_i = 2 \quad (\text{dos rotaciones})$$

$$\hat{c}_i = c_i + \phi_i - \phi_{i-1}$$

$$\hat{c}_i = 2 + r_i(x) - \underbrace{r_{i-1}(x)}_{\text{los } r_i(\cdot) \text{ de los nodos en A,B,C y D no cambian.}} + \underbrace{r_i(y) - r_{i-1}(y)}_{\cdot} + \underbrace{r_i(z) - r_{i-1}(z)}_{\cdot}$$

Notar que: $\cdot r_{i-1}(z) = r_i(x) \quad (\text{tienen el mismo } S(z))$

$$\cdot r_{i-1}(y) > r_{i-1}(x)$$

$$\cdot r_i(y) < r_i(x)$$

$$\begin{aligned} \hat{c}_i &< 2 + r_i(x) + r_i(z) - r_{i-1}(x) - r_{i-1}(x) \\ &= 2 + r_i(x) + r_i(z) - 2r_{i-1}(x) \end{aligned}$$

$$\underline{\text{Prop: }} \frac{\log a + \log b}{2} \leq \log \frac{a+b}{2}$$

$$\log ab \leq 2 \log(a+b)$$

$$\log ab \leq 2 \log(a+b)^2 - 2$$

$$\log 4ab \leq \log (a+b)^2$$

$$4ab \leq a^2 + 2ab + b^2$$

$$0 \leq a^2 + 2ab + b^2$$

$$0 \leq (a-b)^2$$

UNIVERSOS DIFERENTES

Luego $r_{i-1}(x) + r_i(z) = \log S_{i-1}(x) + \log S_i(z)$ babilomida
 $\leq 2 \log \frac{S_{i-1}(x) + S_i(x)}{2}$ (por prop)

Como $S_{i-1}(x) + S_i(z) < S_i(x)$

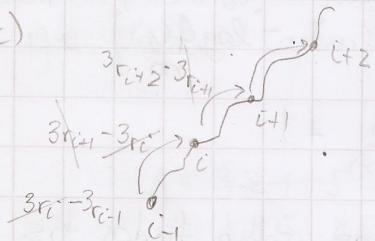
$$\Rightarrow r_{i-1}(x) + r_i(z) < 2 \log \frac{S_i(x)}{2} = 2r_i(x) - 2$$

$$\therefore r_i(z) < 2r_i(x) - 2 - r_{i-1}(x)$$

$$c_i < 2 + r_i(x) + 2r_i(x) - 2 - r_{i-1}(x) = 2r_i(x) + r_{i-1}(x)$$

$$c_i < 3(r_i(x) - r_{i-1}(x))$$

huelga telescopica



Costo amortizado de una búsqueda Suma (telescopicamente)

$$3(r_m(x) - r_0(x)) \\ \leq 3r_m(x) = 3\log n$$

Analizar para zig-zag, zag-zag, etc es análogo... en zig y zag da como $3(\dots) + 1$ o algo así pero solo 1 vez pues está debajo de la raíz.

Optimalidad estática.

Si el elemento x se busca $g(x)$ veces (de las m) entonces el costo amortizado es $\Theta\left(\sum_{x \in T} \frac{g(x)}{m} \log \frac{m}{g(x)}\right)$

Le daremos un peso $w(x) = \frac{g(x)}{m}$ a x .

$$w = \sum_{x \in T} w(x) = \sum_{x \in T} \frac{g(x)}{m} = 1$$

$$S(x) = \sum_{v \text{ desde } x} w(v), r_i(x) = \log_2 S_i(x)$$

$$\begin{aligned}\hat{c}_i &\leq 3(r_m(x) - r_0(x)) + 1 \\ &= 3(\log(w) - \log \frac{g(x)}{m}) + 1 \\ &= 3 \log \frac{m}{g(x)} + 1\end{aligned}$$

Si promediáramos sobre todos los x , con su probabilidad

$$3 \frac{g(x)}{m} \log \frac{m}{g(x)} + 1$$

$$1 + 3H$$

2015年10月29日 (木)

UNIVERSOS DISCRETOS

- Ordenar en $\Theta(n)$
- Predecesor en $\Theta(\log \log U)$
- Tries y árboles de sufijos $\Theta(m)$

• Counting Sort

- las claves están en $[1 \dots U]$
- n claves
- no hay nada más que las claves (no hay punteros. Dos números con el mismo valor son indistinguibles)
 - inicializar contadores $C[i] \leftarrow 0$
 - acumular el # de ocurrencias de cada clave
 - output el # de veces que aparece cada clave.

for $i \leftarrow 1$ to U

$C[i] \leftarrow 0$

for $j \leftarrow 1$ to n

$C[A[i]] \leftarrow C[A[i]] + 1$

$j \leftarrow 1$

for $i \leftarrow 1$ to U

 for $k \leftarrow 1$ to $C[i]$

$A[j] \leftarrow i$

$j \leftarrow j + 1$

$$A = 32115213122$$

$$11112222335$$

$$C: \begin{array}{l} 1 \\ 2 \\ 3 \\ 4 \end{array}$$

$$\square$$

$$\square$$

$$L$$

$$4$$

Funciona porque los números iguales son indistinguibles.
Los elementos SON las claves

Complejidad $\Theta(n + U)$

Conviene solo cuando U es razonable. Gasta mucho espacio extra.

10

④ BUCKET SORT

2023/2024 2023/2024

Cuando los elementos sí son distinguibles (cuando tienen = valor
uso (los elementos son algo más que solo claves))

• Bucket Sort

1º cuento

2º inicializo punteros a la zona de A' donde se escriben los
distintos claves

3º paso por A copiando cada valor a su posición definitiva en A'

$$A = \begin{matrix} 3 & 2 & 1 & 1 & 5 & 2 & 1 & 3 & 1 & 2 & 2 \end{matrix}$$

$$\begin{matrix} C: & 1 & \square & \text{pos } 1 \\ & 2 & \square & \text{pos } 8 \\ & 3 & \square & \text{pos } 8 \\ & 4 & & \text{pos } 11 \\ & 5 & 1 & \text{pos } 11 \end{matrix}$$

$$A' = \begin{matrix} 1 & 2 & 2 & 2 & 3 & 3 & 3 & 5 \end{matrix}$$

```
for i ← 1 to U  
    C[i] ← 0  
for j ← 1 to n  
    C[A[j]] ← C[A[j]] + 1  
P[1] ← 1  
for i ← 2 to U  
    P[i] ← P[i - 1] + C[i - 1]  
for j ← 1 to n  
    A'[P[A[j]]] ← A[j]  
    P[A[j]] ← P[A[j]] + 1
```

Conserva la identidad de los eltos y el alg. es estable (preserva el
orden relativo original entre claves iguales)

Complejidad $\Theta(n + U)$ también

↳ inicializar contadores

Luego podemos ordenar en tiempo lineal si $|U|$ es $\Theta(n)$

Bajo un modelo NO basado en comparaciones

¿Podemos ir más lejos?

se

EJERCICIOS

FINA

| | | | | | |
|----|---|---|---|---|---|
| 17 | 1 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 1 | 0 | 1 |
| 14 | 0 | 1 | 1 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 1 | 1 | 0 |
| 8 | 0 | 1 | 0 | 0 | 0 |

Vamos a ordenar por el último bit cuando bucket sort ($|U|=2$!)

| | | | | | |
|----|---|---|---|---|---|
| 14 | 0 | 1 | 1 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 1 | 1 | 0 |
| 8 | 0 | 1 | 0 | 0 | 0 |

Ahora por el segundo

| | | | | | |
|----|---|---|---|---|---|
| 2 | 0 | 0 | 0 | 1 | 0 |
| 6 | 0 | 0 | 1 | 0 | 0 |
| 8 | 0 | 1 | 0 | 0 | 0 |
| 17 | 1 | 0 | 0 | 0 | 1 |
| 5 | 0 | 0 | 0 | 1 | 0 |

ahora están
ordenados
por los 2
últimos
bits
ALMISMO

\rightarrow 8 01000
 17 10001
 2 00010
 5 00101
 14 01110
 6 00110

\rightarrow 17 10001
 2 00010
 5 00101
 6 00110
 8 01000
 14 01110

que bucket
sort es
ESTABLE

\rightarrow 2 00010
 5 00101
 6 00110
 8 01000
 14 01110
 17 10001

$\Theta(n \log U)$

cuántos bits escribir
& cuantos para los elementos

por qué no mejor elegir chunks más grandes que 1?

Puedo elegir chunks de $\log n$ bits para que el universo sea n
(Recordar que Bucket sort funciona en $\Theta(n + U)$ si $|U|$ es $\Theta(n)$)

$$\Rightarrow \Theta(n \frac{\log U}{\log n}) = \Theta(n \log_n U)$$

OJO cuando U es muy grande

$$\text{ej: } U = \Theta(n^6)$$

$$\rightarrow \Theta(n \log n^6) = \Theta(6n)$$

$\log n^6 = \Theta(n)$
esta constante puede llegar a competir con Quicksort.

Auxiliar 7 - Dominios Discretos

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

2 de Noviembre del 2015

1. Se desea ordenar un arreglo de llaves cuyos valores pueden ser 0 o 1. Algunas características deseables son las siguientes:
 - (a) El algoritmo toma tiempo $O(n)$.
 - (b) El algoritmo es estable.
 - (c) El algoritmo ordena de forma *in-place*: es decir, el espacio adicional utilizado para ordenar es constante.

Diseñe algoritmos que cumplan (a) y (b); (a) y (c); (b) y (c).

2. Ordene n números en el rango $[0, n^2 - 1]$ en tiempo $O(n)$. Generalice su resultado para dominios de la forma $[0, n^k - 1]$, para k constante.
3. Sea B una secuencia de bits de largo n . Supongamos que acceder a un bit $B[i]$ se define $\text{RANK}_b(B, i)$ como el número de bits con valor b en $B[1, i]$. En particular:

$$\text{RANK}_1(B, i) = \sum_{0 < j \leq i} B_j, \quad 1 \leq i \leq n$$

Se define, además, $\text{SELECT}_b(B, i)$ como la posición de la i -ésima repetición del valor b en B .

- (a) Sea $A[1, t]$ un arreglo de t enteros no negativos que suman n . Muestre cómo realizar las siguientes consultas usando RANK y SELECT :
 - $\text{SUM}(r)$: el valor de $\sum_{j=1}^r A[j]$.
 - $\text{SEARCH}(s)$: el mínimo valor de r para el cual $\sum_{j=1}^r A[j] \geq s$.
 - (b) Construya una estructura que permita calcular $\text{RANK}(B, i)$ en tiempo constante, utilizando $2n + o(n)$ **bits** de espacio.
 - (c) Resuelva el mismo problema, esta vez utilizando $o(n)$ bits de espacio.
 - (d) **Propuesto** (fácil): Construya una estructura que permita calcular $\text{SELECT}(B, i)$ en tiempo $\Theta(\log \log n)$ que utilice $o(n)$ bits de espacio.
 - (e) **Propuesto** (no-fácil): Construya una estructura que permita calcular $\text{SELECT}(B, i)$ en tiempo constante, usando $o(n)$ bits de espacio.
4. Describa un algoritmo que, dados n enteros en $[0, \dots, k]$, preprocese su entrada y responda cuántos de estos enteros se encuentran en el rango $[a, \dots, b]$ en tiempo constante. Su algoritmo debería tomar tiempo $\Theta(n + k)$ en el preprocesamiento.

Aux #7

2015年11月2日(月)

[PI]

- a) Tiempo $\Theta(n)$
- b) Estable \Rightarrow Orden de los "repetidos" se conserva
- c) In-place \Rightarrow usa espacio adicional cte.

Pedir los 3 es absurdo.

a) + b) = $\Theta(n) + \text{estable} = \text{Bucket Sort / Counting Sort}$. (clases)

→ tomar arreglo del tamaño del universo.

estable, $\Theta(n+U) = \Theta(n+2) = \Theta(n)$

espacio: $\Theta(n+U) \rightarrow$ arreglo para contar $\boxed{\square}$
→ arreglo del tamaño de la entrada (copia ordenada)

b) + c) = Bubblesort (y parece que Insertion Sort)

estable, tiempo $\Theta(n^2)$, espacio $\Theta(1)$

(al parecer hay un mergesort in-place que tiene tiempo estable $\Theta(n \log^2 n)$)

(Quicksort, es estable?)

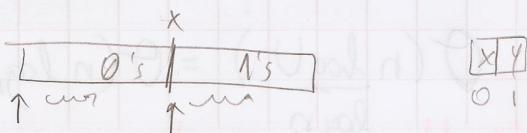
a) + c) { Hago una pasada por el arreglo contando cuántos 0's y

espacio $\Theta(1)$ 1's hay: $n_0 + n_1$ ($n_0 + n_1 = n$)

• Inicializa 2 punteros en 0 y n_0

• Ambos punteros avanzan hasta que el primero encuentra

tiempo $\Theta(n)$ { un 1 q el segundo un 0
• los intercambian / swap



P2

n números

$$U = n^2$$

Intentemos usar Radix Sort + 5 adv de atravesar

- Para cada dígito, desde el menos significativo al más significativo, ordeno usando BucketSort.

$$\Rightarrow \Theta(d \cdot (n+b))$$

\downarrow \downarrow \downarrow
nº dígitos elems base

n+b : counting/bucketsort

Si usamos $b = \log n$

\Rightarrow cada número tiene $\log(n^2)$ dígitos $\rightarrow \Theta(\log n)$

$$\Rightarrow$$
 tiempo $\Theta(\log n \cdot (n+2)) = \Theta(n \log n) \rightarrow \frac{n}{n}$

* Usamos base n

\Rightarrow cada número tendrá 2 dígitos

$$\Rightarrow$$
 radix sort toma $\Theta(2 \times (n+n)) = \Theta(2n) = \Theta(n)$

\uparrow \uparrow \uparrow
1 bucketsort por dígito elems buckets
0 0 1

Generalización

Si el universo es $[0, n^{k-1}]$

\Rightarrow en base n tienen a lo más k dígitos ...

\Rightarrow Radix Sort toma $\Theta(k(n+n)) = \Theta(kn)$

Aux #7

[P3]

si z es el límite teórico del número de bits necesarios para almacenar la información de una estructura, la estructura es **suscrita** si usa $\geq z + \Theta(z)$
↳ chica!
 $(\Theta(z) \Rightarrow$ compacta; $z + \Theta(1) \Rightarrow$ implícita)

se usan
nortos
comis
ops
básicas

{ Rank_b (B, i) : # de bits b en B[1, i]
Select_b (B, i) : Pos. de la i-ésima repetición de b.

a) • $\sum_{j=1}^r A[j]$

- el min valor de r tq $\sum_{j=1}^r A[j] \geq s$

Idea: usar representación unaria $\Leftrightarrow (4)_1 = 1111$

→ cada numero x pasa a ser una secuencia de x 1's

con un 0 al final.

=> Arreglo de $\Theta(n + t)$ bits.

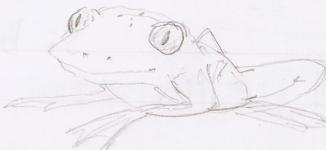
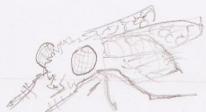
↳ tamaño arreglo

=> • Sum (r) = Rank₁ (select₀ (r))

• elegir el r-ésimo separador y contar 1's

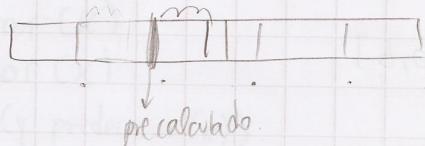
• Search (s) = Rank₀ (select₁ (s)) + 1

b) Como queremos una estructura suscrita entonces siempre vamos a trabajar en función de la cantidad de bits necesarios para representar la info.

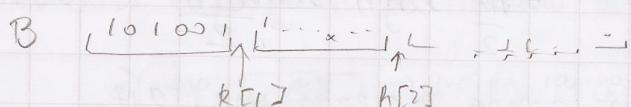


ÁRBOLES VENDEMAN BOS

La idea es dividir B , en bloques



Para cada bloque, guardaremos el valor de Rank "al final del bloque" en un arreglo R .



Si los bloques tienen tamaño b , cuando nos preguntén Rank (B, i), descompondremos $i = q \cdot b + r \Rightarrow$ el n° de 1's hasta $q \cdot b$ es $R[q]$.

¿Qué b usar? $b = \lceil \log_2 n \rceil$ (en algunos controles lo han dado como hint)

¿Qué tamaño tiene R ? R tiene $\frac{n}{b}$ elem.

$$\Rightarrow |R| = \frac{n}{b} \cdot \log n = 2n \text{ bits}$$

$\frac{n}{b}$ elems $\log n$ bits
para representarlos.



Queda calcular el r (del $qb + r$):

O sea, el n° de 1's en $B[qb + 1, qb + r]$
Haremos una tabla con todos los posibles valores

$$\text{Si } b=4 : \quad T[1011, 3] = 2 \\ T[1011, 1] = 1$$

- $T[x, r] = n^{\circ}$ de bits en $x[1, r] \Rightarrow b \times 2^b$ elems
 ↑
 Patrón de bits de largo b
 $\in [0, 2^b - 1]$

- Espacio de T : $2^b \cdot b \cdot \log b = 2^{\frac{\log n}{2}} \cdot \log \frac{n}{2} \cdot \log \log \frac{n}{2}$ & chica
 $b = \frac{\log n}{2}$
 $(\textcircled{2}) \rightarrow$ ese 2 nos salvó la vida poniendo una raíz.

$$\text{Rank}(B, i) = R[q] + T[B[qb+1, qb+r], r]$$

$$\begin{cases} q = \lfloor \frac{i}{b} \rfloor \\ r = i \mod b \end{cases}$$

es $i = qb + r$ descomposición en tiempo constante?

$$\text{Espacio: } 2n + o(n)$$

- c) la tabla R es muy grande, por eso aparece $2n$

- Mantendremos T igual. Es $o(n)$
- Agregaremos un nivel más grande de bloques: superbloques \circlearrowleft
 de tamaño $S = (\frac{\log n}{2})^2$

- En $S[1, n/S]$ guardamos lo que antes iba en R , pero usando S en vez de b .

- Ahora R solo almacena el rank dentro del superbloque correspondiente.

$$R[q] = \text{rank}(B, qb) - S(\lfloor \frac{q}{\log n} \rfloor)$$

$|T| \leq o(n)$ igual que antes.

• $|S|: \frac{n}{S}$ elems, con valores hasta $n = \frac{n}{S} \log n = \frac{2n}{\log n} \in o(n)$

• $|R|: \frac{n}{b}$ elems, pero ahora los valores son más pequeños, solo hasta $O((\log n)^2)$

$$= \frac{n}{b} \cdot \log((\log n)^2) \text{ bits} = 2 \log \log n \cdot \frac{2n}{\log n} = 4n \cdot \frac{\log \log n}{\log n} \in o(n)$$

ÁRBOLES VAN EMDE BOAS

2015年11月3日 (K)

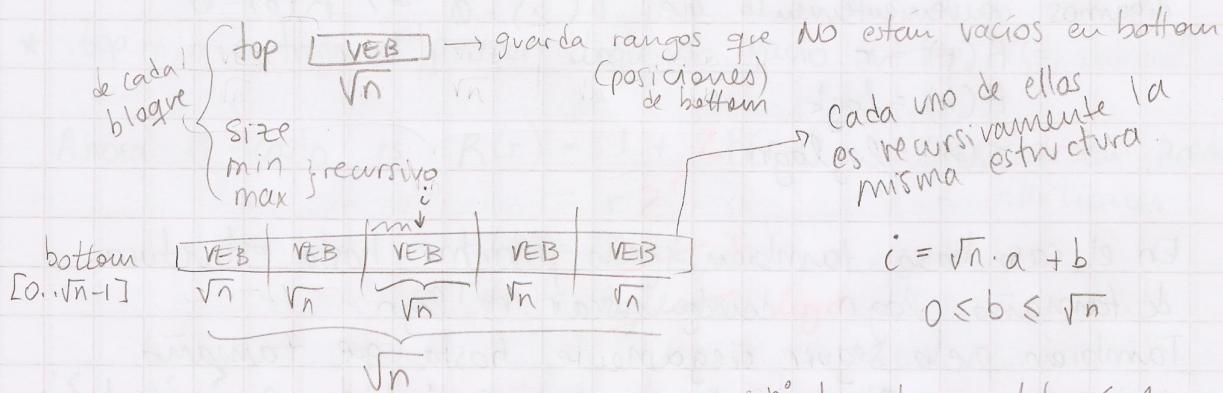
- insertar(x)
- borrar(x)
- sucesor(x)
(y predecessor(x))

Universo $[1 \dots n]$

tiempos $\Theta(\log \log n)$
espacio $\Theta(n)$

Es razonable usar esta
estructura cuando la cantidad de
elementos es similar al tamaño del universo

Lineal en el tamaño del
UNIVERSO



Cómo buscamos?

- `succ(i)` devuelve el primer alto mayor igual que i en

$$\text{Sea } i = a \cdot \sqrt{n} + b$$

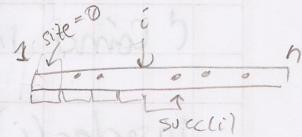
if $\text{bottom}[a].\text{size} > 0 \& \text{bottom}[a].\text{max} \geq b$ \leftarrow Significa que el sucesor

return $a\sqrt{n} + \text{bottom}[a].\text{succ}(b)$

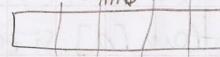
$$c \leftarrow \text{top}.succ(a+1)$$

$$\text{return } c \cdot \sqrt{n} + c.\text{min}$$

? encontrar
el primer
bloque NO
vacío



b : de i está en ese bloque
Y se que está



bloque a (b es el 'offset')

podría haber un caso

base donde la estructura es

un árbol binario o algo, uso búsqueda binaria (en una hoja de tamaño Θ también puedo llegar a universo de tamaño 1).

conveniente

$\log n$, luego buscar
en $\log \log n$ también)



size, min, max

$$\sqrt[n]{a} = a^{\frac{1}{n}} \rightarrow 1$$

$n \rightarrow \infty$

ARMAR BLOQUE

¿Cuánto toma? todo el costo a parte de la llamada recursiva

$$T(n) = 1 + T(n)$$

$$n=2^k \Rightarrow T(2^k) = 1 + T(2^{k/2})$$

$$H(k) = T(n) \Rightarrow H(k) = 1 + H(\frac{k}{2})$$

$$k=2^r$$

$$R(r) = H(k) \Rightarrow R(r) = 1 + R(r-1)$$

digamos convenientemente que $T(2) = 0 \Rightarrow R(0) = 0$

$$\Rightarrow R(r) = r$$

$$H(k) = \log k$$

$$T(n) = \log \log n$$

En el caso base también puedo pasarme hasta estructura de tamaño $\log n$, donde buscar es $\log n$. También puedo seguir ciegamente hasta que tamaño universo es 1, pero no es recomendable.

¿Cómo insertarlos? si el bloque estaba vacío, además de insertar en bottom hay que insertar en top.

insertar(i)

caso base, cualquier

sea $c = a\sqrt{n} + b$

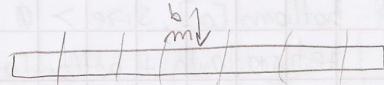
actualizar size, min, max

if $\text{bottom}[a].size = 0$

top.insertar(a)

bottom[a].insertar(b).

de toda
la estructura



top.size es el número de bloques.

borrar(i)

{ caso base cualquiera }

Sea $i = a\sqrt{n} + b$

bottom[a].borrar(b)

if bottom[a].size = 0

top.borrar(a)

actualizar size, min, max.

$$\min \leftarrow \text{top}.min \cdot \sqrt{n} + \text{bottom}[\text{top}.min].min$$

* top.min entrega el primer bloque no vacío en top. (i) correcto

Ahora el costo es $\cdot R(r) = 1 + 2R(r-1)$

$$= r2^r$$

Cada inserción produce
2 inserciones

$$\cdot H(k) = \cancel{\log k}$$

$$\cdot T(n) = \cancel{\log \log n} \log n$$

¿Solución? Cambiar invariantes de la estructura

Ahora \rightarrow el min de la estructura NO se almacena en los bottom (ni top), no guardo copias extras.

Ahora en SVCC(c) se agrega línea al comienzo

SVCC(i)

if $i \leq \min$ return \min

:

insertar(i)

{ if size = 0: min ← i, max ← i, size = 1
else if i < min: i ← min
sea $i = \sqrt{n} + b$
actualizar size, min, max
if bottom[a].size = 0
 top.insertar(a)
bottom[a].insertar

borrar(i)

{ if size = 1: size ← 0
else if i = min:
 min ← top.min, $\sqrt{n} + \text{bottom}[\text{top}.min].min$
 i ← min
 sea $i = \sqrt{n} + b$
 bottom[a].borrar(b)
if bottom[a].size = 0
 top.borrar(a)
actualizar size, max.

Espacio

$$\begin{aligned} T(n) &= 1 + (\sqrt{n} + 1) T(\sqrt{n}) \\ T(n) &\leq 2n - 3 \\ T(n) &\leq 1 + (\sqrt{n} + 1)(2\sqrt{n} - 3) \\ &= 1 + 2n - \sqrt{n} - 3 \\ &\leq 2n - 3 \end{aligned}$$

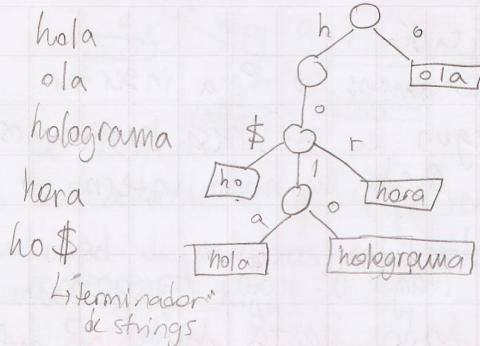
2015年11月5日 (木)

TRIES (árboles digitales)

Conjunto de n strings
Largo total L .

- espacio $\Theta(L)$
- + tiempo de búsqueda de string de largo m
- $\Theta(m)$
- o $\Theta(m \log n)$

tamaño alfabeto



- Para buscar $P[1..m]$, $P[m+1] = \$$
voy bajando desde la raíz por las letras $P[1], P[2], \dots$
hasta que
 - 1) no existe un hijo con label $P[i]$
→ P no está en el conjunto.
 - 2) llego a una hoja
→ comparo el resto de P con el resto del string en la hoja.

* Esto toma tiempo $\Theta(m)$... ¿cómo busco por cuál hijo bajar?

hash de hijos? ABB? lista ordenada y búsqueda bin?

↳ $\Theta(m)$ o $\Theta(m \log n)$

- Búsqueda de prefijos $P[1..m]$ (no le agrego "\$")
voy bajando desde la raíz por $P[1], P[2], \dots$ hasta que
 - 1) = a 1) anterior
 - 2) llego a una hoja → P solo puede ser prefijo de la cadena en esa hoja ≠ verificar
 - 3) se termina P en un nodo interno (como "ho" en el trié anterior)
→ P es prefijo de todas las hojas que descienden de ese nodo (predo guardar el númer de hijos en cada nodo, o recorrerlo entero)

鳥 イリ ポリヨ

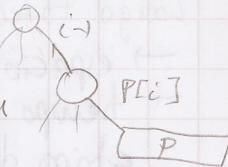
フライ
Fu ra i

Inserción

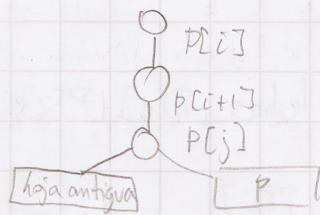
Buscamos el P a insertar.

Según en qué caso terminemos.

1) Quedo en nodo interno \rightarrow agregamos una hoja



2) Llego a hoja. Agregamos una secuencia de nodos hasta donde P difiere de la hoja a la que habíamos llegado.



proporcional al largo
de lo que inserto/borro
pues estoy a lo más "m"
pisos más abajo de la raíz

Borrar

- Encuentro P en una hoja hija de V

- borro la hoja

- mientras V tenga un solo hijo y sea hoja,

$u \leftarrow \text{padre}(v)$

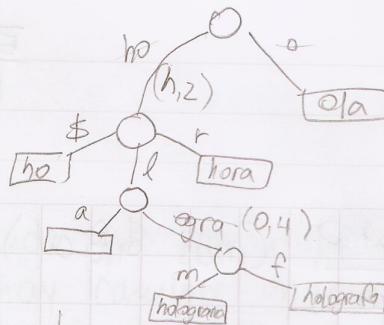
$\text{padre}(\text{hoja}) \leftarrow u$

borrar V

$v \leftarrow u$

} me gustaría hacer esto de una

hola
ola
holograma
hora
ho\$



holograma?
hiforma?

Blind Trie

- los nodos con un solo hijo se borran
- cada arista guarda el # de caracteres que representa.
- Espacio $\Theta(n)$ n cantidad de palabras, cada nodo tiene al menos 2 hijos.

- la distancia entre un nodo y su hoja no se pone. No es necesario.
- Para buscar bajo confiando en que calzan las letras entre medio. Al llegar a una hoja, debo comparar todo pues puede que no sea: Por ESO hay que guardar el string entero (en otro lado, asumimos que siempre es así) pues las aristas, el trie, no guarda información suficiente.

Buscar:

- igual que en trie, saltando los caracteres que no veo
- al llegar a una hoja, verificar todo P.

Buscar prefijo

- idem a buscar-prefijo de trie, pero en caso 3) comparar P con cualquier hoja que desciende del nodo. Si es prefijo de esa, es prefijo de todas, y sino, no lo es de ninguna.
- ★ Siempre pude arreglármelas para tener un puntero a una hoja desde cada nodo.

車 < 弟 犬 テニス
inu te ni su

• Insertar P. (más difícil)

2 pasos

- 1) descubrir la posición de la primera diferencia
buscar P (y no encontrarlo)

caso a) buscar una hoja descendiente ~~cualquier~~
y encontrar 1º diferencia

caso b) buscar la 1º diferencia con esa hoja

- 2) colgar la nueva hoja a la profundidad correcta

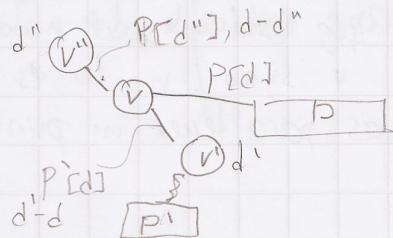
Volver a la raíz y bajar nuevamente hasta llegar a la profundidad buscada. Sea d' la profundidad deseada.

(caso a) hay un nodo explícito v a profundidad d'
→ a agregar una nueva hoja a v

La distancia entre ~~hijo~~
nodo y hoja no se pone

caso b) llego a un nodo v' de profundidad $d' > d$ y
su padre era v'' de profundidad $d'' < d$
(quedo en medio de una arista.)

→ corta la arista con un nuevo nodo v



Ojo que puede que haya que arreglar esto por error de ±1
en d, d', d''

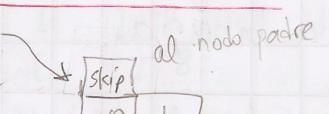
馬

パトリシア

Borrar (más fácil. Puede que fuga que borrar padre, pero No abuelo(s), pues no hay caminos varios)

- busco y borro la hoja
- Si al padre le queda un solo hijo, lo reemplazo por su abuelo

(Morrison 1968) ÁRBOLES PATRICIA = Blind Tries con $\underline{f=2}$; las cadenas son secuencias de bits.



→ Caminos de a lo más largo.
a seguir para Normalmente 8 veces si son 256 carac
búscar es hasta menos.

⇒ Tiene exactamente $n-1$ nodos. El anterior
tendrá un poquito menos, pero en ambos
es a lo sumo n .

ÁRBOL DE SUFIJOS (Suffix tree) (Weiner, 1973)

1 2 3 4 5 6 7 8 9 10 11 12
abracadabra \$

- Un string de largo n , tiene n sufijos (mas strings con \$)
- todo substring en patron de largo n es un PREFijo de un SUFIJO.

Substring = prefijo do m sufijo

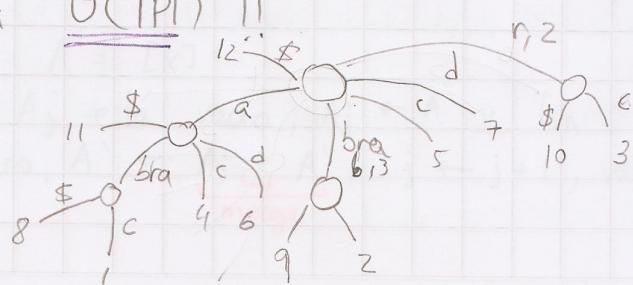
todos los substrings = P \equiv todos los sufijos que empiecen con P.

→ inserto todos los sufijos en un Patricia tree y hago búsqueda de prefijos, por P.

→ obtengo los sufijos que empiezan con P
⇒ los substrings = P del texto.

En orden
y no en orden
del tamaño
del texto

$O(|P|)$



espacio $\Theta(n)$

Patricia tiene
 $O(n)$ nodos y
un string de tamaño
 n tiene n sufijos

PAT arrays (Mumberg y Myers 1990)
Gonnet y Baeza-Yates 1993

Arreglo de fijos (suffix array) Solo las hojas son suficientes!

12, 11, 8, 1, 4, 6, [9, 2], 5, 7, 10, 3. Orden lexicográfico

buscar: $\Theta(m \log n)$

↳ aquí sí importa el largo del texto
Pues hay búsqueda binaria

Para construir se puede

hacer con quicksort !! (ordenar sufijos) Salvo si hay mucha
repetición de patrones largos. Las comparaciones se vuelven
muy costosas.

Auxiliar 8 - Más análisis amortizado y dominios discretos

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

9 de Noviembre del 2015

1. La búsqueda binaria en un arreglo ordenado toma tiempo logarítmico, sin embargo la inserción toma tiempo lineal. Veremos que podemos mejorar el tiempo de inserción manteniendo varios arreglos ordenados.

Específicamente, suponga que se desea implementar las operaciones de búsqueda e inserción en un conjunto de n elementos. Sea $k = \lceil \log(n+1) \rceil$ y asuma que la representación binaria de n es $n_{k-1} \cdots n_1 n_0$. Utilizaremos k arreglos ordenados A_0, \dots, A_{k-1} , donde cada A_i tiene largo 2^i . Cada arreglo está lleno o vacío dependiendo si $n_i = 1$ o $n_i = 0$, respectivamente. El número total de elementos contenidos en los k arreglos es entonces $\sum_{i=0}^{k-1} n_i 2^i = n$. Aunque cada arreglo A_i está ordenado, no existe ninguna relación particular entre los elementos de distintos arreglos.

Analice las operaciones de búsqueda e inserción en esta estructura.

2. Un *multistack* consiste en una serie (potencialmente infinita) de stacks S_0, \dots, S_{t-1} donde el j -ésimo stack puede almacenar hasta 3^j elementos. Todas las operaciones de *push* y *pop* se realizan inicialmente sobre S_0 . Cuando se desea *push*ear un elemento en un stack lleno S_j , se vacía este stack en el siguiente, S_{j+1} (posiblemente repitiéndose la operación de forma recursiva). Al hacer *pop* de un stack vacío, se llena éste a partir de *pops* del siguiente stack (posiblemente repitiéndose la operación de forma recursiva) y luego se hace el *pop* correspondiente. Considere que las operaciones de *push* y *pop* en los stacks individuales tiene costo 1.

- En el peor caso, ¿cuánto cuesta una operación de *push* en esta estructura?
- Demuestre que el costo amortizado de una secuencia de n operaciones de *push* en un multistack inicialmente vacío es de $O(n \log n)$. Utilice la siguiente función de potencial:

$$\Phi = 2 \sum_{j=0}^{t-1} N_j \cdot (\log_3 n - j)$$

donde N_j es el número de elementos en el j -ésimo stack.

- Demuestre lo mismo para cualquier secuencia de *pushes* y *pops*.
3. Una operación de interés sobre cadenas binarias es *SELECTNEXT*(B, i), que retorna la posición del siguiente 1 después de la posición i . Usando una metodología similar a la usada para *RANK* (uso de bloques y superbloques) diseñe una estructura que requiera $o(n)$ bits extra y que permita responder *SELECTNEXT* en tiempo constante. Note que este problema es similar al de encontrar el sucesor al elemento i .
 4. Describa un algoritmo que, dados n enteros en $[0, \dots, k]$, preprocese su entrada y responda cuántos de estos enteros se encuentran en el rango $[a, \dots, b]$ en tiempo constante. Su algoritmo debería tomar tiempo $\Theta(n + k)$ en el preprocesamiento.

Aux #8

2015年11月9日 (月)

[P1]

n elementos; $k = \lceil \log(n+1) \rceil$

↳ k arreglos ordenados A_0, \dots, A_{k-1}
pero no tienen orden entre sí.

$A_0 [1] 1$ elto
 $A_1 [xx] 2$
 $A_2 [\dots] 4$
 $A_3 [\dots] 8$
⋮
 $A_{k-1} [\dots] 2^{k-1}$ eltos

} completamente llenos o completamente vacíos.

• BÚSQUEDA:

→ búsqueda binaria - "secuencial", es decir, una por arreglo no vacío.

En el peor caso busco en todos los arreglos

$$\sum_{i=0}^{k-1} \log_2(2^i) = \sum_{c=0}^k c = \frac{k(k-1)}{2} = \Theta(k^2) = \Theta(\log^2 n)$$

Diremos que eso es suficientemente bueno... no es mucho más que $\Theta(\log n)$

• INSERCIÓN:

Idea: la estructura "representa n en base 2", donde un arreglo es lleno si 1 o vacío si 0, (ni)

Luego insertar un elemento podría hacerse como "sumar 1",
obviamente sin perder las invariantes (arreglos ordenados, llenos
o vacíos)

→ Creamos $A' = [x]$

a) Si A_j está vacío, $A_j \leftarrow A'$ y fin

b) Si no, $A' \leftarrow \underbrace{A_j + A'}_{\text{merge}}$; $j \leftarrow j+1$, volver a a)

8 # xA

En este caso, el "+" es una op. de merge

Pior Caso: hago merge de todos los arreglos.

⇒ k merges. Cada merge es de 2^i elementos

$$\Rightarrow T(n) = \Theta\left(\sum_{i=0}^{k-1} 2 \cdot 2^i\right) = \Theta(2^k) = \Theta(n)$$

Mejor hagamos análisis amortizado.

Sean m operaciones de inserción sobre una estructura con n elementos.

Notemos que A_0 participa en un merge cada 2 inserciones.

$A_1 \dots$

$A_r \dots$

4 inserciones

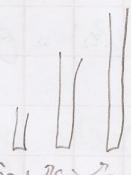
2^{r+1} inserciones

$$\Rightarrow \text{Costo total de merges: } \sum_{i=0}^{k-1} 2^{i+1} \left[\frac{m}{2^{i+1}} \right] \leq k \cdot m = m \cdot \Theta(k)$$

cuánto cuesta
mergear A_i y A'

$\Rightarrow m \cdot \Theta(\log n)$
en cuántos
merges participa A_i
 $\Theta(\log n)$
por op

P2



j-ésimo: tamaño 3^j
stack

$S_0 \cup S_1 \cup S_2$

• Pior caso: mover todo $\Theta(n)$

• n ops en un m.s. vacío

$$\Phi = 2 \sum_{j=0}^{t-1} n_j (\log_3 n - j) \quad (\text{dados...})$$

nº de elementos en S_j

Vamos a calcular el costo de vaciar el stack Si en el Siti

Costo

$$\text{amortizado} = \Delta \Phi + 2 \cdot 3^i$$

$$\phi = 2 \sum_{j=0}^{t-1} n_j (\log_3 j) \quad \begin{cases} \text{Esto puede verse como que cada} \\ \text{elemento tiene un potencial } \phi_{elem}. \\ \phi_{elem} = 2(\log_3 n - j), \text{ donde } j \text{ es el índice} \\ \text{del stack en el que está.} \end{cases}$$

$$\Rightarrow \phi > 0 \quad ?$$

$$\text{Costo amortizado} = \Delta\phi + 2 \cdot 3^i$$

$$\phi_f = 2 \cdot 3^i (\log_3 n - (i+1))$$

$$\phi_i = 2 \cdot 3^i (\log_3 n - i)$$

solo considero los elementos que se están "moviendo"

$$\Rightarrow \Delta\phi = -2 \cdot 3^i$$

$$\Rightarrow \hat{C} = \Delta\phi + C = -2 \cdot 3^i + 2 \cdot 3^i = 0$$

\Rightarrow vaciar stacks es "gratis"

C Cuánto cuesta un push entonces?

por análisis anterior

$$\begin{aligned} \hat{C}_{total} &= C_{push_{S_0}} + \underbrace{\Delta_{vaciar}_{stacks}}_{stacks} + \Delta\phi_{push_{S_0}} + \Delta\phi_{vaciar_{stacks}} \\ &= 1 + (\phi_f - \phi_i) \quad \Delta\phi_{push_{S_0}}: \phi_f = 2 \cdot \log_3 n \\ &= 1 + 2 \log_3 n = \Theta(\log n) \quad \phi_i = 0 \\ &\qquad\qquad\qquad = 2 \log_3 n \end{aligned}$$

Y para n pushes $\rightarrow \Theta(n \log n)$

También este problema se puede hacer con un método parecido al problema anterior.

- n ops de push y pop.

C Cómo se hace un pop?

- $i = 0$

- Se hace pop de S_i

- Si se vacía, se llena con elementos de S_{i+1} (se puede repetir recursivamente)

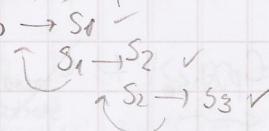
Consideremos un stack S_i . Un push o pop es **relevante** para S_i , si el mayor stack tocado es S_i .

Cada push relevante move $\sum_{j=0}^{i-1} 3^j = \Theta(3^i)$ elem al siguiente stack; Cada pop relevante move $\sum_{j=0}^{i-1} 3^j = \Theta(3^i)$ elem al stack anterior.

\Rightarrow cada op. come en tiempo $\Theta(3^i)$

Antes de un push relevante, todos los stacks antes de S_i están llenos y S_i está a lo mas $2/3$ lleno

Despues, todos excepto S_0 (que queda vacío) quedan en $1/3$ (justo) y S_i queda al menos $1/3$ lleno.



Para un pop relevante:

$S_0 \dots S_{i-1}$ vacío $\Rightarrow S_0 \dots S_{i-1} 2/3$

S_i al menos con $1/3$ $\Rightarrow S_i$ a lo más $2/3$

La primera op. relevante debe ser un push.

Antes, debe haber $\sum_{j=0}^{i-1} 3^j = \Theta(3^i)$ pustes (para llenar todo)

Entre 2 ops relevantes, debe haber al menos $\sum_{j=0}^{i-1} 3^{j-1} = \lceil 2(3^i) \rceil$ ops irrelevantes

Luego si pagamos $\Theta(1)$ por cada stack en cada op, el pago total es suficiente

\Rightarrow el costo total es $\Theta(\log n)$ (pues hay $\Theta(\log n)$ stacks)

Select: Posición de j-ésimo 1

rank: cuántos 1's hay hasta la j-ésima pos.

P3

Abriremos SelectNext \equiv SN

$$\rightarrow SN(j) = \text{Select}(1 + \text{rank}(j-1))$$

$$\rightarrow \text{bloques de tamaño } \frac{\log n}{2}, \text{ super bloques de tamaño } S = b \cdot \log n = \frac{\log^2 n}{2}$$

Para cada superbloque j, con $1 \leq j \leq \lceil \frac{n}{S} \rceil$ calculamos

(Nx son tablas precalculadas)

$$N_S[j] = SN(j \cdot s + 1)$$

\hookrightarrow selectNext del primer elto del superbloque.

Espacio: $\Theta\left(\frac{n}{\log n}\right)$ bits.

Para cada bloque de un superbloque,

$N_b[i] = SN$ del 1^{er} elto del bloque con respecto a ESE bloque

Espacio: $\Theta\left(\frac{n \log \log n}{\log n}\right)$ ~ los elemns de N_b son posiciones en un bloque de tamaño $< n$

Finalmente, para cada posible bloque S de tamaño b y posición i,

$N_p[S, i] = SN$ de i, para la secuencia S.

(notar que, por ejemplo, si $S = \Theta$ (solo 0's), $N_p[S, i] = b + 1$

$$\hookrightarrow 2^b \cdot b \cdot \log b = \sqrt{n} \log \log n \dots \text{el mismo truco de la clase aux anterior}$$

$\underbrace{\quad}_{\substack{\text{el más chico primero}}}$

Si fuera de rango (del futuro), voy a bloque más grande.

se sale de rango

Otro ejercicio de esto mismo, RMQ

range minimum query?

ALGORITMOS EN LÍNEA / ON LINE

2015年11月10日 (火)

Consideremos el siguiente problema:

Quiero jugar en cierto juego, pero no sé cuántos días jugaré antes de aburrirme

- Una tienda lo arrienda con costo de $\$A$ / día.
- Otra lo vende a $\$B$.

¿Qué hacer? Quiero minimizar mi gasto.

Si supiera el n° de días de uso del juego, d , la solución es comparar $A \cdot d$ con B , elijo comprar si $B < A \cdot d$ (y viceversa)
 \Rightarrow costo es $\min(A \cdot d, B)$

Si no conocemos d :

Idea: Estrategia "arriendo por \bar{d} días, luego lo compro"

- Si $d \leq \bar{d} \Rightarrow$ me aburro del juego mientras lo arrendaba
 \Rightarrow costo $A \cdot d$
- Si $A \cdot \bar{d} < B \Rightarrow$ el óptimo también arrienda
 \Rightarrow costo igual al óptimo $\Rightarrow \frac{\text{ALG(I)}}{\text{OPT(I)}} = 1$
- Si $A \cdot \bar{d} \geq B \Rightarrow$ el óptimo compraba $\Rightarrow \frac{\text{ALG(I)}}{\text{OPT(I)}} = \frac{A \cdot \bar{d}}{B}$

- Si $d > \bar{d} \Rightarrow$ voy a arrender \bar{d} y luego comprar
 \Rightarrow costo $A \cdot \bar{d} + B$

- Si $A \cdot \bar{d} < B \Rightarrow$ el óptimo arrendaba $\Rightarrow \frac{\text{ALG(I)}}{\text{OPT(I)}} = \frac{A \cdot \bar{d} + B}{A \cdot \bar{d}}$

- Si $A \cdot \bar{d} \geq B \Rightarrow$ el óptimo compraba
 $\Rightarrow \frac{\text{ALG(I)}}{\text{OPT(I)}} = \frac{A \cdot \bar{d} + B}{B} = \frac{A \cdot \bar{d}}{B} + 1$

$$\frac{\bar{d}}{d} + \frac{B}{A \cdot \bar{d}} \leq 1 + \frac{B}{A \cdot \bar{d}} \\ \leq 1 + \frac{B}{A \cdot \bar{d}}$$

El análisis se hace con respecto a un algoritmo óptimo que conoce "todo". Definimos "Competitividad de un algoritmo online A" como

$$C(n) = \max_{\{I|I|=n\}} \frac{A(I)}{OPT(I)} \leftarrow \text{algoritmo óptimo (y que sabe todo)}$$

Entonces se dice que A es $C(n)$ -competitivo.

Queremos limitar $\max \frac{ALG(I)}{OPT(I)} \Rightarrow$ nos conviene $\frac{A}{B} \approx 1$. Si $\frac{A}{B} = \frac{B}{A}$

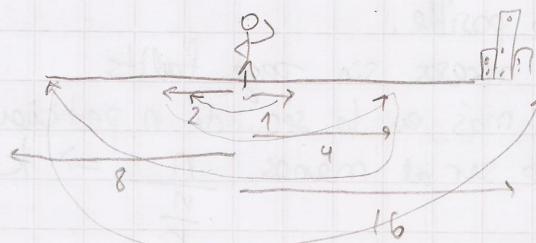
$\Rightarrow C_n(I) \leq 2 \Rightarrow$ la estrategia es 2-competitiva

- Otro problema Imagine despertar en una larguísima carretera
 $? \rightarrow ?$ \Rightarrow quiero minimizar los pasos dados.

→ No sabemos en qué dirección está la ciudad
 El algoritmo óptimo es caminar en la "dirección correcta"
 $\Rightarrow OPT(n) = n$

Estrategias como "caminar hacia la derecha" tienen costo "máximo" no acotado (puedo irme en la dirección incorrecta)

Idea: caminamos en una dirección; giro al haber recorrido 2^i , vuelvo y recorro 2^{i+1} hacia la otra dirección.



Algoritmos EN LÍNEA (ON LINE)

Supongamos encontramos la ciudad entre 2^i y 2^{i+1}
La distancia recomendada es $\Rightarrow i = \lfloor \log_2 n \rfloor + 1$

$$d = 2 \sum_{i=0}^{\lfloor \log_2 n \rfloor + 1} 2^i + (n) = ALG(I)$$

↳ lo encontré, en el peor caso, en la dirección opuesta a la que partí.

$$\begin{aligned} &< 2 \cdot (2^{\lfloor \log_2 n \rfloor + 2}) + n \\ &\leq 2 \cdot (4n) + n = 9n \Rightarrow A(I) \leq 9n \\ &\Rightarrow C(n) \leq 9 \end{aligned}$$

Otro Problema: Paginado.

- La información que no sabemos es la secuencia de operaciones
- El costo se da ante la petición de una página que está en disco (page fault)
- Nos limitaremos a una memoria de tamaño K y $K+1$ páginas de información.

Pior Caso: el "adversario" siempre pide **la** página que tenemos en disco.

\Rightarrow Si existe un algoritmo que produce n page faults $\Rightarrow ALG(I) = n$

Cómo comparemos con el óptimo? El óptimo "conoce el futuro".
 \Rightarrow el algoritmo óptimo envía a disco la página que será usada lo más adelante posible.

\Rightarrow requerirá al menos K accesos sin page faults.

\Rightarrow n/K page faults a lo más en la sec. de n peticiones

\Rightarrow cualquier algoritmo debe ser al menos $\frac{n}{K}$ $\Rightarrow K$ -competitivo

(本) はい。なぜなら

instacOPT global fail

¿Qué algoritmos hay?

- LRU: last recently used → k-competitivo
- FIFO → k-competitivo
- LFU last freq. used → NO es k-competitivo
- :

LRU

Dividiremos la secuencia de ops en bloques que terminan cada k page faults de LRU. En cada bloque, OPT sufre un pf. al menos 1 vez.

Sea p la página pedida antes de un bloque B (p es la MRU)

- Si uno de los pf's en B trae p a memoria, p fue el LRU en algún momento \Rightarrow todos los demás fueron consultados
 \Rightarrow OPT falló alguna vez.
- Si una página p' se repite en los pf's
 - \Rightarrow entre los 2 pf's deben haberse pedido todas.
 - \Rightarrow OPT falla 1 vez entre las 2 peticiones
- Si los k fallos son diferentes \Rightarrow OPT siempre falla 1 vez por cada k fallos. de LRU.
 \Rightarrow LRU es k-competitivo.

List Update Problem

2015年11月12日(木)



Accesar (i) \rightarrow costo i
Intercambiar ($i, i+1$) \rightarrow costo 1

Aplicación: compresión

codifico \rightarrow símbolos s_1, \dots, s_N (ordenados en tamaño)
elementos s_1, \dots, s_N

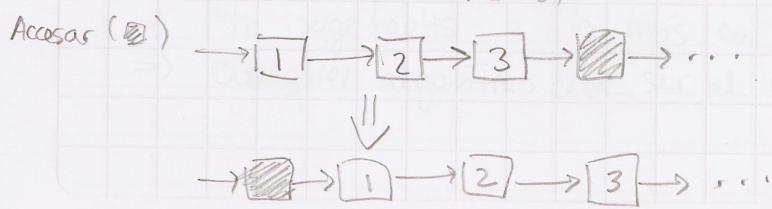
Para comprimir s : lo busco en la lista. Si está en la posición i , uso s_i
 \Rightarrow Si el costo de los accesos se minimiza voy a minimizar el
largo del texto comprimido. La idea es tener una estrategia que
logre hacer que las búsquedas resulten en el rango más cercano,
más al comienzo de la lista posible. De esa forma minimizo
bien, también, el largo del texto comprimido.

Al buscar puedo también intercambiar elementos s_j de acuerdo
a alguna estrategia bien definida (no aleatoria, p.e., para poder
usarlo al revés al descomprimir y tener resultados coherentes.)

Queremos una estrategia de reordenamiento (on-line) tal que dada
una secuencia de accesos s_1, s_2, \dots minimice el costo de acceso.

Recordemos que el óptimo conoce s_1, s_2, \dots, s_N y también puede
invocar intercambios.

Idea: mover el elemento accedido a la primera posición
(Move-to-Front)



Costo de acceder al k -ésimo
elemento: $2k-1$
(k para llegar,
 $k-1$ swaps)

La estructura cambia de forma dinámica según la secuencia de operaciones. \Rightarrow tiene sentido hacer un análisis amortizado

Mostraremos que el costo de MTF es a lo más 4 veces de OPT.

Def: Inversión

Una inversión en una lista A clr a una lista B es un par (x, y) tq:

- x está antes de y en A } o viceversa
- x está después de y en B

Usaremos la sgte. función potencial

$$\phi = 2 \cdot \# \text{ de inversiones de } L_{MTF} \text{ clr a } L_{OPT}$$

$$\phi_0 = 0 \text{ (al pp. } L_{MTF} \text{ y } L_{OPT} \text{ son iguales)}$$

$$\phi > 0 \checkmark$$

Demostraremos $C_{MTF} + \Delta\phi \leq 4 \cdot C_{OPT}$

Sea x un elemento en la secuencia de peticiones

$i =$ posición de x en L_{OPT}

$k =$ posición de x en L_{MTF}

Consideraremos a parte los swaps que OPT puede hacer.

$\Rightarrow C_{OPT} = i$ (Solo lo busca... el análisis de los swaps que hace los swaps veremos después)

- Hay $k-1$ elementos antes de x en L_{MTF} ; todos estos elementos quedan después de x después de la operación.

$$\Rightarrow \# \text{ de inv. creadas} + \# \text{ de inv. destruidas} = k-1$$



- Sea y un elemento antes de x en L_{MTF} antes de los swaps
 Para que se cree una nueva inversión, y debe estar antes de x en L_{OPT} \Rightarrow existen a lo más $(i-1)$ y 's
 \Rightarrow a lo más se crean $i-1$ inversiones
 \Rightarrow el resto deben ser destrucciones
 $\Rightarrow k-1-(i-1) = k-i$ deben ser destrucciones (a lo menos)
 $\Rightarrow \Delta\phi \leq 2(i-1 - (k-i))$
 $= 2(2i - k - 1)$
 $= 4i - 2k - 2$

$$\Rightarrow C_{MTF} \leq C_{MTF} + \Delta\phi \leq 2k-1 + 4i - 2k - 2 \\ = 4i - 3 \leq 4i = \boxed{4 \cdot C_{OPT}}$$

sobre secuencia de ops.

Queda ver qué pasa con los swaps que OPT pueda hacer

- A MTF le cuesta 0 (tr a este swap, MTF no está haciendo nada)
- A OPT le cuesta 1

Analizamos por partes

- Los alg. no dependen del otro
- 1) OPT retorna alto
 - 2) MTF retorna alto
 - 3) MTF hace swaps
 - 4) OPT hace swaps

Un swap de OPT crea o destro 1 inversión
 $\Rightarrow \Delta\phi = \pm 2$ ($\phi = 2 \cdot \# \text{ inversions}$)
 $\Rightarrow C_{MTF} \leq C_{MTF} + \Delta\phi \leq 2 = 2 \cdot 1$
 $= 2 \cdot C_{OPT}$
 $\leq 4 \cdot C_{OPT}$

\Rightarrow MTF es 4 competitivo

- Si se permite intercambiar pares de elementos arbitrarios con costo 1 = $\frac{1}{2}$ OPT "vence" a cualquier algoritmo outline por un factor de al menos $\frac{n}{2}$, sobre secuencias arbitrariamente largas y escogidas aleatoriamente ($n=|L|$)

Problema Tinder

- Tengo n participantes, quiero elegir a quien tenga el mejor puntaje.
- Se puede festejar a 1 participante a la vez.
- Luego de festejar y obtener su puntaje, pedo
 - a) "Me quedo con este participante"
 - b) "Adiós para siempre"

El algoritmo offline conoce n y el puntaje de cada participante; de antemano.

El online conoce n .

C) Qué quiero optimizar?

→ la probabilidad de encontrar al mejor participante
 $(\Rightarrow \text{OPT logra } P=1)$

Estrategia:

- Permuto aleatoriamente los participantes.
- Calibrámos usando las primeras t citas
 (elegimos "Adiós para siempre" en todas ellas)
 Recordamos el mejor puntaje P
- En las siguientes $n-t$ citas, si vea un puntaje mayor que P , me caso.



Sean $1 \dots n$ los participantes y t es la persona con mayor puntaje, etc, luego de permutar usando una permutación Π , el problema es buscar el mínimo de $\Pi[1] \dots \Pi[n]$

El algoritmo es entonces.

$$\cdot P \leftarrow \min \{\Pi[1], \dots, \Pi[t]\}$$

$$\cdot j^* \text{ es el } 1^{\text{er}} \ j \geq t+1 \text{ tq. } \Pi[j] < P$$

$$\circ \mathbb{P}(\Pi[j] = 1) ?$$

$$\sum_{j=t+1}^n \mathbb{P}[\Pi[j] = 1] \text{ y nos quedamos con la persona } j$$

• Si $\Pi[j] = 1$ para algún $j > t$, ¿cuándo fallamos?

Fallamos si entre las citas $t+1$ y $j-1$ apareció alguien mejor que entre 1 y t .

$\Rightarrow \min \{\Pi[1], \dots, \Pi[j-1]\}$ debe estar en una posición $\in (t+1, j-1)$

Si este min está en una posición $\in (1, t)$, vamos a elegir a la persona correcta.

$$\Rightarrow \mathbb{P}[\Pi[j^*] = 1]$$

$$= \sum_{j=t+1}^n \mathbb{P}[\Pi[j] = 1] \text{ y } \min(\{\Pi[1], \dots, \Pi[j-1]\}) \in \{\Pi[1], \dots, \Pi[t]\}$$

$$= \sum_{j=t+1}^n \frac{1}{n} \frac{t}{j-1}$$

¿Por qué?

$$\mathbb{P}[\Pi[j] = 1 \text{ y } (\min \dots)] \Rightarrow t = \frac{n}{\mathbb{E}}$$