

CC4102 - Diseño y Análisis de Algoritmos.

Prof: Gonzalo Navarro
of. 312
gnavarro@dcc.uchile.cl

1 [Cotas inferiores
Experimentación

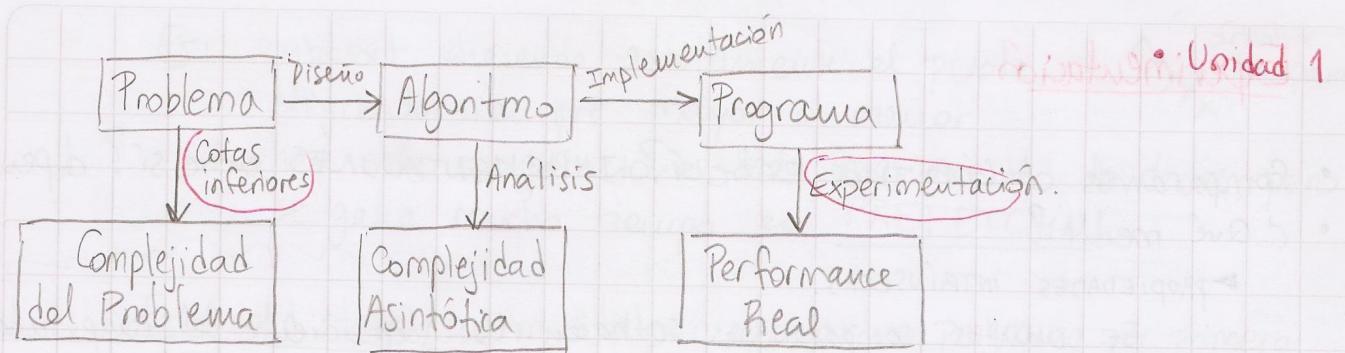
2 [Memoria Secundaria
- ordenar
- colas de prioridad
- hashing

~3 [Algoritmos avanzados
- amortización
- universos discretos
- competitividad

A [Algoritmos no convencionales
- probabilísticos / aleatorizados
- aproximados
- paralelos
Examen

- 2/3 • 2 controles y 1 Examen
- 1/3 • 3 Tareas (en pareja o de a 3)
Si reprobaban tareas
→ Habrá una tarea recuperativa
(habiéndole hecho de nuevo las tareas reprobadas)
- Ejecución con 5.0

2015年9月1日 (火)



2015年9月3日(木)

Experimentación

- Comparamos ALGORITMOS entre sí? IMPLEMENTACIONES entre sí? depende
- ¿Qué medir?

► PROPIEDADES INTRÍNSECAS:

Ej: cuántas comparaciones se hacen. Es independiente del computador en el que trabaja.

[MERGESORT] vs [QUICKSORT]

en prom $n \log_2 n$

$\approx 1.44n \log_2 n$

pero quicksort es más rápido en la práctica, pues mueve menos datos.

Ej: cantidad de accesos a discos.

► TIEMPOS

I/O + - usuario → CPU

- sistema → page-faults y seeks. El proceso no está corriendo realmente
- elapsed → tiempo medido por reloj. No se usa mucho. Se pueden contar también otros procesos que se ejecutan al mismo

Cosas que pueden afectar los tiempos:

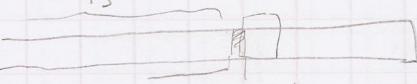
tiempo

- cache

- garbage collection

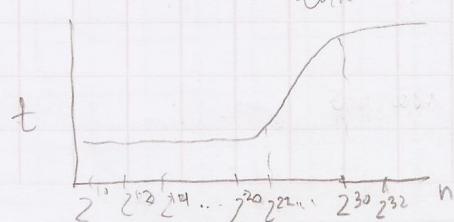
- prefetching de instrucciones (adivinar por dónde irse en las branches de los ifs, cuando incluso estadística)

- traducción de direcciones (memoria virtual a real)

Ej: rank 

$O(1) = 3$ accesos a disco

$n > 2^{20}$ el cache ya no sirve tanto.

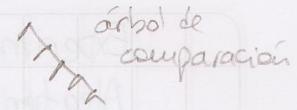


$n > 3^0$ se hacen 3 accesos a disco si o sí

Ej: quicksort eligiendo astutamente el pivote

→ eligiendo siempre mayor o menor

hay más comparaciones, más movimiento de datos, pero se gana mucho tiempo en PREFETCHING



Ej: cold state / warm state : ejecutar un programa por primera vez, versus ejecutarlo muchas veces seguidas. El SO se acuerda del mapa de memoria, y se demora mucho menos en warm state.

• ¿Qué inputs?

Ej: árboles binarios aleatorios. } altura prom \sqrt{n}

Es $1/5$? elegir uno de los 5 anteriores con prob uniforme.

Tomar todos los árboles distintos y darles la misma prob.

O es $1/6$? elegir todas las permutaciones de elementos a insertar. Son 2 modelos distintos.

altura prom	{ 123 132 213 } \rightarrow $1/3$ prob.
log n	{ 231 312 321 } \rightarrow

Ej: listas invertidas (inverted index). Intersección

[P1] → uuuuuu l_1 } $\Theta(l_1 + l_2)$

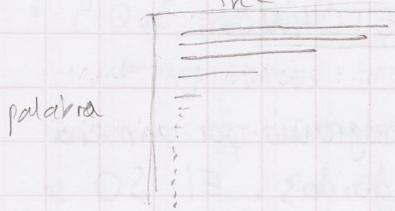
[P2] → uuuuuu l_2 } recorrer secuencialmente con l_1 y l_2 de largo similar

[P3] → uu l_3 con $l_3 \ll l_2$

es mejor hacer búsqueda binaria de todos los eltos en l_3 en l_2 . $\rightarrow \Theta(l_3 \log l_2)$



Experimentalmente como elijo cual de los 2 métodos usar?
Aleatoriamente? pero pocas palabras aparecen MUCHO y
muchas palabras aparecen POCO (Zipf's law).



En este caso aleatoriamente no es muy buena idea experimentar para decidir qué método elegir, pues en la realidad la cuestión no es nada aleatoria!

Otro ejemplo es elegir una buena función de hashing para palabras, p.e. suma de carac ascii, tomar los primeros 4 carac... problema: comparten prefijos, palabras que tienen las mismas letras, etc. Probar permutaciones de letras aleatoriamente es péssima idea, porque el desempeño de mis funciones de hashing pararía que se comportan bien, pero en la realidad NO!

Otro ejemplo: elegir nodos alazar en un árbol... 50% de prob. de elegir una hoja! a veces eso puede ser muy poco realista.

2015年9月10日 (木)

Cotas inferiores de problemas

- Adversario (combate naval contra alguien que pone los barcos al final)
- Reducción (transformar un problema a otro)
- Teoría de la información (permite incluso deducir cotas inf promedio)
(analiza redundancia de la información)

- BÚSQUEDA EN UN ARREGLO DESORDENADO. $A[1, n]$
 - min # accesos en el peor caso = n
¿por qué? hay que razonar en base a cualquier algoritmo. No importa el orden en que lo haga, si reviso $n-1$ celdas, en el peor caso siempre el número va a estar en la celda que falta.
(como el ejemplo de combate naval)
 - min # accesos en arreglo ordenado? con búsq. binaria sabemos que el peor caso es $\log n \rightarrow$ el adversario tiene más restricciones con respecto a donde poner el elemento maliciosamente.

- ENCONTRAR EL MÍNIMO DE $A[1, n]$.

comparaciones $n-1$

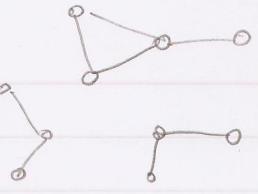
```
m ← A[1]
for i ← 2 to n
    if A[i] < m
        m ← A[i]
```

return m

} También podemos pensar en el TORNEO...
se ve lindo, es un buen algoritmo paralelo
pero también se realizan $n-1$ comp.

¿Podemos hacer menos de $n-1$ comp?

podríamos usar un argumento similar al del problema anterior pero... ¡en el primer paso del torneo TODOS los elementos ya se compararon con otro!



(+) BIBLIOGRAFÍAS

los arcos son comparaciones

Argumentaremos con grafos. Debemos hacer suficientes comparaciones para producir un grafo CONEXO \rightarrow si tengo n nodos, necesito $n-1$ arcos para producir un grafo conexo.

Lo importante es que el adversario puede seguir siendo consistente cuando coloca el mínimo en una componente conexa que he descartado (pues el adversario solo dice respuestas SÍ-NO)

Vamos a modelar el conocimiento del algoritmo sobre el input de otra forma

(a, b, c)

$a = \#$ eltós nunca comparados.

$b = \#$ eltós comparados alguna vez y siempre menores

$c = \#$ eltós comparados alguna vez y alguna vez Mayores

Cualquier algoritmo correcto parte de $(n, 0, 0)$ y llega a $(0, 1, n-1)$

¿Cómo es la evolución $(n, 0, 0) \rightarrow (0, 1, n-1)$?

	a	b	c	
a	$(a-2, b+1, c+1)$ para los 2 casos	$(a-1, b, c+1)$	$(a-1, b+1, c)$	
b	lo mismo	$(a, b-1, c+1)$	(a, b, c) $(a, b-1, c+1)$	en el peor caso el adv. puede hacer que ocurra eso sin entrar en inconsistencias
c	lo mismo	lo mismo	(a, b, c)	→ NO SIRVE, no me conviene

¡Es directo!

Para pasar de $c=0$ a $c=n-1$, como en cada paso c aumenta a lo sumo en 1, necesito hacer $n-1$ comparaciones.

Otra cosa interesante es que deduzco las comparaciones que más me convienen! \rightarrow Puedo deducir un algoritmo ó

Alternativa 1: comparar a con a $\rightarrow (n-2, 1, 1)$ y luego hacer el resto de las comparaciones a con b
 ↳ algoritmo for!

Alternativa 2: comparar a con a $\frac{n}{2}$ veces $(n, 0, 0) \rightarrow (0, \frac{n}{2}, \frac{n}{2})$
 y luego seguir comparando b con b.

- ENCONTRAR EL MÁX Y EL MÍN DE $A[1, n]$.
 $n-1 + n-2 = 2n-3$ comp?

Extendamos el modelo anterior...

(a, b, c, d) $a = \#$ ejtos nunca comparados

ganadores $b = \#$ ejtos comparados algunas vez y siempre menores
 perdedores $c = \#$ " " " y siempre mayores
 mediocres $d = \#$ " " " y alguna vez mayores y otras menores.

$$(n, 0, 0, 0) \rightarrow (0, 1, 1, n-2)$$

	a	b	c	d
a	$(a-2, b+1, c+1, d)$ si "a" gana	$(a-1, b+1, c, d+1)$ si "c" gana	$(a-1, b, c, d+1)$	$(a-1, b+1, c, d)$
b		$(a, b-1, c, d+1)$ "b" gana	(a, b, c, d) si "b" gana	$(a, b-1, c, d+1)$
c			$(a, b, c-1, d+1)$	(a, b, c, d)
d				(a, b, c, d)

- Notemos que la gran masa va a d.
- d crece a lo sumo de a 1 (adversario) y en ese caso a NO decrece.
 $\rightarrow n-2$ comp necesarias.

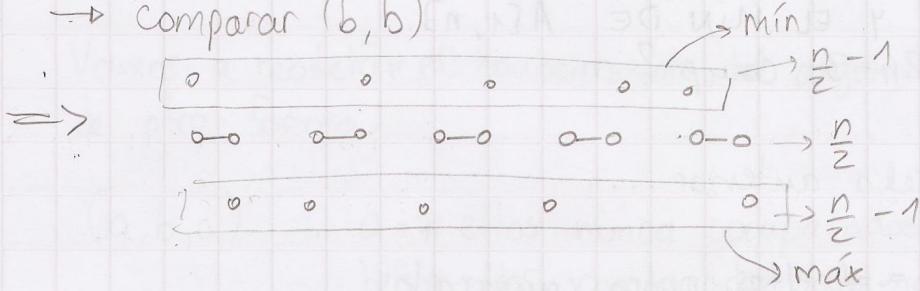
• Para hacer pasar a de n a 0, hago $\lceil \frac{n}{2} \rceil$ comp.

\Rightarrow Cota inferior $\lceil \frac{3}{2} n \rceil - 2$

¿Podemos derivar un algoritmo que nos lleva a la cota inferior?
Veamos...

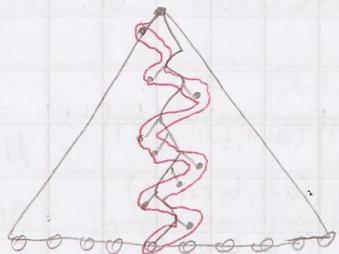
- (a, b) no me conviene mucho... es como el for.
- es mejor empestar a comparar pares desconocidos (a, a)
 $(n, 0, 0, 0) \xrightarrow{n/2} (0, \frac{n}{2}, \frac{n}{2}, 0)$ ↳ más rápido

→ comparar (b, b)



Como encontré un algoritmo, ahora SÉ que no hay cota inferior más alta. (A veces hay gap)

• MÁX y $2^{\text{º}} \text{MÁX}$ $n-1 + \lceil \log_2 n \rceil$



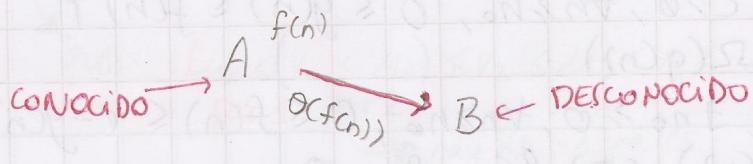
¿Es el óptimo? Aux.

Hacer la tablita del adversario es bien complicado.

Esto fue ADVERSARIO.

★ Técnicas de Reducción:

Si A tiene una cota inferior de $f(n)$ y se puede reducir a B en tiempo $\Theta(f(n))$ entonces B tiene una cota inferior $\Omega(f(n))$

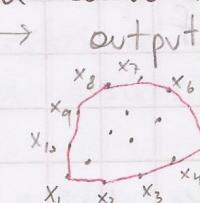


Si se hace lo único que hago es complicar más mi problema desconocido.

Ej. Ordenar es $\Omega(n \log n)$

→ Cápsula convexa "convex hull"

input



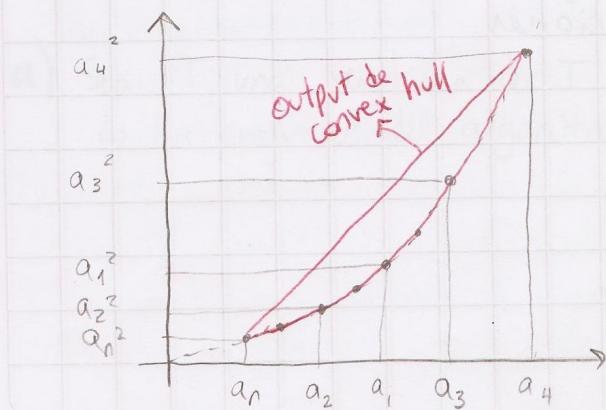
output
 $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}$

(ejemplo, encontrar el mínimo, reducirlo a ordenar y extraer el mínimo)

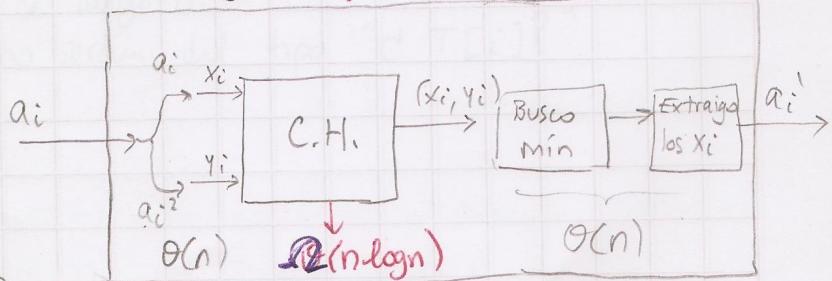
mínimo (en área) polígono que contiene a todas las puntos. Entrega una lista ORDENADA de los puntos (recorre los puntos en sentido antihorario)

Tomemos un problema de ordenamiento y lo transformamos a un problema convex hull. Luego veremos cómo, de la solución de convex hull podemos traducir "fácilmente" a la solución del problema de ordenamiento.

$$(a_1, a_2, a_3, \dots, a_n) \xrightarrow{\quad} (a_1, a_1^2), (a_2, a_2^2), (a_3, a_3^2), \dots, (a_n, a_n^2).$$



Sort $\Theta(n \log n)$



Si C-H. Se demorara menos, contradicción con SORT.

Auxiliar 1 - Cotas Inferiores

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

14 de Septiembre del 2015

1. Demuestre que se requieren al menos $2n - 1$ comparaciones, en el peor caso, para fusionar dos listas ordenadas de tamaño n en una de tamaño $2n$ ordenada.
2. El problema de *búsqueda aproximada en texto* consiste en, dado un string $T[1, n]$, llamado *texto*, otro string más corto $P[1, m]$, llamado *patrón*, y un entero $0 \leq k < m$, determinar si P ocurre en T permitiendo a lo más k *errores* (inserciones, borrados o reemplazos de caracteres en P (o T). Ambos T y P son secuencias de caracteres de un alfabeto $[1, \sigma]$.
Yao demostró en 1976 que no es posible resolver el problema de búsqueda exacta ($k = 0$) en menos de $\Omega(n \log_\sigma(m)/m)$ inspecciones de caracteres de T *en promedio* (el promedio supone que los caracteres de T se eligen al azar, uniformemente en $[1, \sigma]$).
 - (a) Demuestre que un adversario impide resolver el problema general inspeccionando menos de $k + 1$ caracteres en cualquier ventana posible de T de largo m .
 - (b) Deduzca de lo anterior una cota inferior de la forma $\Omega(kn/m)$ para el problema, incluso en el caso promedio.
 - (c) Deduzca la cota para el problema $\Omega(n(k + \log_\sigma m)/m)$ en promedio, demostrada por Chang y Marr en 1994. Ellos también diseñaron un algoritmo con esa complejidad promedio, $O(n(k + \log_\sigma m)/m)$. ¿Qué puede decir entonces de la complejidad promedio del problema?
3. Considere el problema de encontrar el máximo y el segundo máximo de un arreglo de n números. Se sabe que en el peor caso, $n + \lceil \lg n \rceil - 2$ comparaciones son suficientes. Utilizando la técnica del adversario, demuestre que $n + \lceil \lg n \rceil - 2$ comparaciones también son necesarias en el peor caso (en el modelo de comparaciones).
4. Demuestre que determinar si un grafo no dirigido es o no conexo toma $\binom{n}{2}$ consultas del tipo “¿existe un arco entre los nodos u y v ?”, si la cantidad total de nodos es n .

2015年9月14日(月)

AUX # 1

"Cotas inferiores"

- $\Theta(g(n)) = \{f(n) : \exists n_0, c > 0, \forall n \geq n_0, 0 \leq f(n) \leq g(n)\}$
- $\Omega(g(n)) = \{f(n) : \exists n_0, c > 0, \forall n \geq n_0, 0 \leq g(n) \leq f(n)\}$
- $\Theta(g(n)) = \Theta(g(n)) \cap \Omega(g(n))$
- $\Theta(g(n)) = \{f(n) : \forall c > 0, \exists n_0 \geq 0, \forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n)\}$
↳ $f(n) \in \Theta(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f}{g} = 1$
- $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 \geq 0, \forall n \geq n_0, 0 \leq c \cdot g(n) \leq f(n)\}$
↳ $f(n) \in \omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{g}{f} = 0$

PI

Tenemos 2 listas:

Sea $l_1 := u_1, u_2, \dots, u_n$ siendo $u_1 \leq u_2 \leq \dots \leq u_n$

$l_2 := v_1, v_2, \dots, v_n$ siendo $v_1 \leq v_2 \leq \dots \leq v_n$

- Adversario responde todas las preguntas ($\text{comparar}(a, b)$) como si:

$$u_1 < v_1 < u_2 < v_2 < u_3 < \dots < u_n < v_n$$

↳ el intercalamiento es un mal caso para merge.

Contradicción: Supongamos que el algoritmo hizo $2n - 2$ (o menos) comparaciones.

Consideremos los pares:

$(u_1, v_1), (v_1, u_2), (u_2, v_2), (v_2, u_3), \dots, (u_n, v_n)$ } 2^{n-1} pares

PUES: $(u_1 < v_1 < u_2) < v_2 < u_3 < v_3 < \dots < u_n < v_n$
Par 1 Par 2
hay 2^{n-1} pares.

Luego existe un par que no fue comparado, por ejemplo, (v_k, u_{k+1})
sin pérdida de generalidad, supongamos que fue puesto así: (No hizo comp)
Adversario anuncia que $u_{k+1} < v_k$... $v_k | u_{k+1} \dots$
↳ las respuestas del adversario no
permiten discernir este caso \Rightarrow El algoritmo NO es correcto.

Nota: como existe un par no comparado, el algoritmo debió situar dichos elementos en la lista. Basta que el adversario diga que el orden no era el correcto (El inverso) para que el algoritmo falte.
↳ Además debe ser de la forma (u, v) o (v, u) pues no pueden ser situados 2 "u", o 2 "v" ya que ya están ordenados, y contradice el input.

[P2]

$T[1, n]$ texto

$0 \leq k \leq m$ errores permitidos

$P[1, m]$ patrón

Σ : alfabeto

a) Sea T una ventana de T de largo m .

↳ las preguntas del algoritmo serán del tipo "¿ $T[i]?$ "

AUX # 1

Adversario: Dado m y una ventana de largo m (igual al patrón). A las primeras " k " consultas distintas responderá con "errores" (como por ejemplo, caracteres que no están en el patrón).

- Si el algoritmo responde que encontró un "match" el adversario "rellena" la ventana con errores \Rightarrow Algoritmo incorrecto.
- Análogamente, si responde que no hay match, rellena con caracteres del patrón.

Supongamos que tenemos PIZARRA, y hasta 3 errores.

$\hookrightarrow \begin{matrix} \textcircled{n} & \text{i} & \text{z} & \text{a} & \textcircled{n} & \text{a} \\ \uparrow & & \uparrow & & \uparrow & \end{matrix}$

El adversario tira errores a las 3 consultas del algoritmo.

Entonces:

① Algoritmo dice que hay match: $\begin{matrix} \textcircled{n} & \text{ñ} & \text{ñ} & \textcircled{n} & \textcircled{n} & \textcircled{n} \end{matrix}$
Rellena con \textcircled{n} 's
 \hookrightarrow Resultado incorrecto.

② Algoritmo dice que no hay match: $\begin{matrix} \textcircled{n} & \boxed{\text{i}} & \text{z} & \text{a} & \text{ñ} & \text{ñ} & \boxed{a} \end{matrix}$
Rellena con las letras del patrón. Por ende sí hay match con tolerancia de 3 errores.
 \hookrightarrow Resultado incorrecto.

(Reducción)

- b) Tenemos P_{initial} : $P_i = \text{"encontrar patrones en cualquier lugar"}$ \leftarrow difícil
 Otro: $P_o = \text{"encontrar patrones que coincidan en } i \cdot m, i \geq 0$

$T: \boxed{\quad} \boxed{\quad} \boxed{\quad} \boxed{\quad} \boxed{\quad} \dots$ Dividimos el texto en fragmentos de largo "m". Nos limitaremos a buscar en cada ventana de largo "m".

Cualquier solución de P_i , es transformable en una de P_o . Luego, si toda solución de P_o toma tiempo $T \in \Omega\left(\frac{kn}{m}\right)$, entonces toda solución de P_i también.

Para P_o : (adversario)

- Se divide T en bloques de largo "m" (hay $\frac{n}{m}$ bloques)
- En cada uno necesito al menos $(k+1)$ "checks". (Los bloques son disjuntos) \Rightarrow el total toma al menos $\Omega\left(\frac{kn}{m}\right)$, $k > 0$.

a)

n bloques

$\Omega\left(\frac{kn}{m}\right)$, $k > 0$.

en realidad es $k+1$, pero la notación Ω permite borrar esas constantes.

Lo ideal es tener un texto con ventanas, que es un problema fácil en cada ventana. Luego, con esto podría haber una mejor solución para el problema general, lo que no puede ser. Por ende, si el problema fácil toma al menos $\Omega\left(\frac{kn}{m}\right)$, luego el problema difícil se hace al menos en $\Omega\left(\frac{kn}{m}\right)$.

- c) Cualquier algoritmo demora al menos: $\Omega\left(\frac{kn}{m}\right)$ con k errores y $\Omega\left(\frac{n \log_2(m)}{m}\right)$ con 0 errores por enunciado.
 \Rightarrow En particular demora al menos el máximo de los 2 tiempos. Además: $(f+g) \in \Omega(\max(f, g)) \Rightarrow \Omega\left(\frac{n \log_2(m)}{m}\right) + \Omega\left(\frac{kn}{m}\right) = \Omega\left(\frac{n(k + \log_2(m))}{m}\right)$
 \hookrightarrow concluimos que es: $\Theta\left(\frac{n(k + \log_2(m))}{m}\right)$

P3

Sean "n" números con A el máximo y B el segundo máximo.

- Para que un algoritmo responda bien debe saber:

1) $A > B$

2) Identificar los $n-2$ elementos que no son ni A ni B.

\Rightarrow cada uno de estos elementos debe haber perdido contra B, o contra otro de este conjunto.

¿Por qué? En caso contrario, un adversario puede tomar un elemento que no haya perdido así, e intercambiárselo con B.
 \Rightarrow hay $n-2$ comparaciones que no involucran A.

Para encontrar "k", construimos el siguiente adversario:

Adv: crea un arreglo L, donde \forall elemento del conjunto, se le asigna un conjunto con ese elemento.

$\Rightarrow L[x] \leftarrow \{x\}$, en donde $L[x]$ son los elementos que el algoritmo sabe que son $\leq x$.

Compare (a, b):

- Si: $a \in L[b]$ ó $b \in L[a]$
responder correctamente.

- Si no: si: $|L[a]| \geq |L[b]|$
 $L[a] \leftarrow L[a] \cup L[b]$
else:

$L[b] \leftarrow L[a] \cup L[b]$

Queremos que las listas vayan creciendo lo más lento posible, pues sabemos que cuando $|L[x]| = h$, entonces sabremos qué x es el máximo, por la definición de $L[x]$ (todos los elementos $\leq x$)

• En cada comparación que involucra a x , $|L[x]|$ a lo más se duplica. Claramente $|L[x]| < 2^k$, con k número de comparaciones en que ha participado x . Si el algoritmo es correcto, al final de la ejecución $|L[A]| = n$ (máximo).

$$\Rightarrow A \text{ estuvo en } k \text{ comparaciones, con } 2^k \geq |L[A]| = n$$
$$\Rightarrow k \geq \lceil \log_2(n) \rceil$$

∴ # de comparaciones $\geq h - 2 + \lceil \log_2(n) \rceil$,

$$0 \leq 3 \cdot 4^{(3-5)n} \leq$$

2015年9月15日 (K)

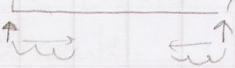
Colas inferiores - Reducción

Ej: colas de prioridad

heapify	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	-
insert	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
extract Min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
Heap binario		Fibonacci Heap	?	?
			no existe	

Simplemente usando argumentos de reducción, esas 2 alternativas no pueden existir pues en tal caso se podría ordenar en tiempo lineal.

Ej: 3 SUM (puedo deducir complejidades no conocidas.)

- Dados n números x_1, \dots, x_n , ¿puedo elegir a, b, c tq $a+b+c=0$?
- n^3 fuerza bruta
- $n^2 \log n$ buscar todos los pares y ordenar y buscar.
- n^2 buscar $a+b=0$? → 

para $a+b+c=0$, hacer el algoritmo anterior n veces para $-x_1, -x_2, \dots$

Conjetura: 3 SUM es $\Omega(n^2)$

Pero este año se demostró que 3SUM es $\Omega(n^{2-\epsilon})$ $\forall \epsilon > 0$

Gronlund & Pettie

$$\left(\frac{n^2}{\log n / \log \log n} \right)^{2/3}$$

Y 3SUM se ocupaba para reducir problemas 3SUM-hard ☺

Ej: 3 Colineal: Dados n puntos en el plano, existen 3 sean colineales?
(no en vertical)

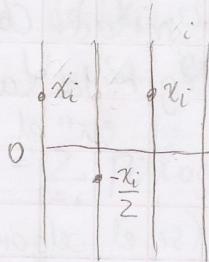
Sea $x_1 \dots x_n$ un problema de 3SUM

Por cada x_i creamos 3 puntos. (transformación en tiempo lineal)

$(0, x_i)$

$$(1, -x_i/2)$$

(2 x_i)



Sean 3 puntas colineales

(0,a) (1,b) (2,c)

$$\frac{1}{x_i} - \frac{x_j}{x_k}$$

$$b = \frac{a+c}{2} \quad b \text{ es el punto medio (Thales)}$$

$$-\frac{x_j}{2} = \frac{x_i + x_k}{2} \Rightarrow x_i + x_j + x_k = \emptyset.$$

¶ al resolver 3-colinear \neq soluciones 3SUM.

→ no es lineal, por adversario.

Teoría de la información

sirve incluso para encontrar cotas a casas promocionadas

Eg: sort

Para 2 permutaciones distintas, las secuencias de intercambios deben ser distintas, pues todos los intercambios son la permutación inversa para llegar a la "identidad". Y la inversa es **ÚNICA**.

Hay $n!$ permutaciones distintas, luego $n!$ transformaciones distintas.
 Pues si hay 2 permutaciones que tienen la misma transformación, una de ellas no llega a la identidad.

El algoritmo (determinista) debe ser capaz de distinguir entre $n!$ respuestas. Con 1 pregunta, descarta la mitad. Con 2, $\frac{3}{4}$.

$\Rightarrow \log_2 n!$ preguntas binarias para elegir los cambios a hacer en el arreglo.
 comp. binaria <

$\lceil \log_2 n! \rceil$ preguntas son NECESARIAS (si el algoritmo es bueno)
 siempre va a descartar la mitad del conjunto que me queda
 por adversario.

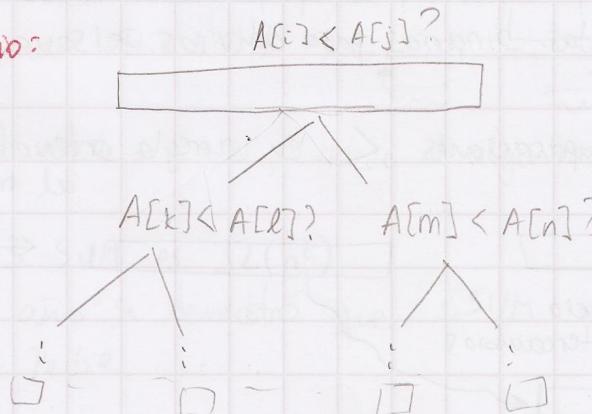
Recordar: STIRLING.

↳ analogía de cortar y elegir la torta.
 Minimizar el peor caso

$$n! = \frac{n^n}{e^n} \sqrt{2\pi n} \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$\Rightarrow \lfloor \log n! \rfloor = n \log n - \Theta(n)$ | Luego ordenar es al menos $\Theta(n \log n)$

Caso promedio:



Solo tengo información suficiente para entregar output ordenado en las hojas. Si lo codifico con bits, tengo una combinación ÚNICA para cada permutación (si no es única, el algoritmo entrega el mismo resultado para 2 perm. #s, y una de ellas no está ordenada).

AUX #2

Si tengo un algoritmo que hace q preguntas, codifico con q bits.

(Shannon 1948)

Sea U un conjunto de n objetos $U = \{x_1, \dots, x_n\}$ donde la probabilidad de x_i es p_i . Entonces cualquier codificación de objetos de U usa en promedio, al menos

$$H(U) = \sum_{i=1}^n p_i \log_2 \left(\frac{1}{p_i} \right) \text{ bits. (a los más probables les asigno codificaciones más pequeñas)}$$

En particular, si todos los eltos tienen la misma probabilidad, se necesitan $\log_2 n$ bits.

Supongamos transmiso una permutación al azar (uniforme entre los $n!$)

$$H = \sum_{i=1}^{n!} \frac{1}{n!} \log_2 \left(\frac{1}{1/n!} \right) = \log_2 n! = \Theta(n \log n)$$

Es decir, si uso un algoritmo para codificar permutaciones, cada una con igual prob., necesito en promedio $n \log n$ bits.

En teoría de la información, en general la cota inferior para el caso promedio es igual a la cota inferior para el peor caso, con distribución uniforme.

En cambio,

$$p_i = \frac{1}{2^{i+1}}$$

$$H = 2 \text{ bits}$$

Cota inferior,
independiente de la
forma de codificar

lanzar moneda muchas veces

$$\begin{array}{cccc} \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{16} \\ | & | & | & | \\ 01 & 001 & 0001 & \end{array}$$

Si al tener un arreglo con ese sesgo y quiero buscarlo, me conviene ordenar por sus p_i en orden decreciente y hacer búsqueda secuencial.

Si la probabilidad es uniforme, es conveniente usar un ABB.

Y cuando tengo un sesgo entre medio?

Auxiliar 2 - (Más) Cotas Inferiores

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

21 de Septiembre del 2015

Adversarios

- Demuestre que determinar si un grafo no dirigido es o no cíclico toma $\binom{n}{2}$ consultas del tipo “¿existe un arco entre los nodos u y v ?”, si la cantidad total de nodos es n .

Reducciones

- Demuestre que el problema de determinar si, dados n puntos sobre tres líneas horizontales en $y = 0, 1, 2$, existe una línea no horizontal que pase por tres puntos es 3SUM-hard.
- Demuestre que el problema de, dado un conjunto de rectas en el plano, determinar si existe un cruce de al menos 3 es 3SUM-hard.

Árboles de decisión

- Se tiene un arreglo A ordenado de tamaño n . Muestre una cota inferior sobre el número de comparaciones necesarias para encontrar la posición de un valor y en A (o retornar -1 si no se encuentra). Considere que cada comparación puede retornar una de tres posibles respuestas ($<$, $>$ o $=$). Realice esto para el peor caso y para el caso promedio.
- Se tiene un arreglo A ordenado de tamaño n . Suponga que sólo tiene disponible la función $\text{TEST}(i, j, k, y)$, que dados $0 < i < j < k \leq n$ entrega una de las siguientes respuestas:

$$\begin{array}{ll} y < A[i] & A[j] < y < A[k] \\ y = A[i] & y = A[k] \\ A[i] < y < A[j] & y > A[k] \\ y = A[j] & \end{array}$$

Muestre una cota inferior sobre el número de llamados a TEST para encontrar la posición de un valor y en A (o retornar -1 si no se encuentra).

2015年9月21日 (月)

Aux # 2

[P1] Adversario.

• Idea: construiremos un adversario que mantiene 2 grafos, C y A.

- 1) ambos serán consistentes con las respuestas dadas al algoritmo
- 2) C tiene un ciclo
- 3) A no tiene ciclo.

Estas props se mantienen ante $\leq^*(?)$ consultas

Adv:

$C \leftarrow kn$ // clique de n nodos

$A \leftarrow \emptyset$ // n nodos, 0 arcos.

adyacente (u, v):

Si $A + (u, v)$ es acíclico

$A \leftarrow A + (u, v)$

return si

else

$C \leftarrow C - (u, v)$

return no.

Veremos varias props de A y C para demostrar que C tiene 1 ciclo

a) $A \subseteq C$

b) $C - A$ es el conjunto de los arcos no preguntados

(todo arco preguntado fue agregado a A o fue quitado de C)

c) Si A es desconexo, C tiene todos los arcos que conectan las componentes conexas de A

$\Sigma \neq XVA$

- Sea un arco de este tipo. Ese arco no puede crear un ciclo en A, luego no puede haber sido quitado de C.

d) C es conexo.

Dem: Por contradicción. Supongamos en un paso se quita el arco e y C queda desconexo. Luego A

- sería cíclico de tener e.

Pero C queda desconexo \Rightarrow e no es parte de un ciclo en C

Pero por a), $A \subseteq C$

e) sea parte de un ciclo en A

\Leftrightarrow luego C es conexo.

e) Si $A \neq C$, C tiene un ciclo.

Dem: Por contradicción sea C acíclico.

• Es conexo por d) \Rightarrow es árbol.

• Como $A \neq C$ y $A \subseteq C \Rightarrow A$ es desconexo.

• Sea e $\in G(C-A)$

- e conecta 2 componentes de A.

- como C es árbol, e debe ser el único que conecta estos 2 componentes.

- Para $n \geq 3$, en K_n existe al menos un e' que las une, $e \neq e'$.

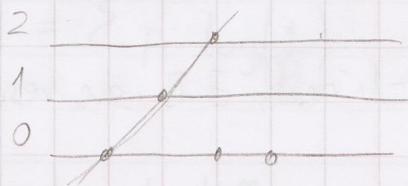
Por b), $e' \in C-A \Rightarrow \Leftarrow$

\Rightarrow Si $A \neq C$, C tiene ciclo.

* La gracia es que el adv. da las mismas respuestas para 2 instancias \neq 's del problema. El algoritmo no sabe en cuál de los 2 esté si no hasta hacer $\binom{n}{2}$ preguntas.

P2

a)



No es trivial modelar

- b) Reduciremos 3-colineal a esto. (idea, cambiar palabra "recta" por "línea")
- Tenemos una instancia de 3-colineal: n puntos, buscamos 1 recta tg

$$y_1 = ax_1 + b \quad (x_i, y_i) \rightarrow (a, b)$$

$$y_2 = ax_2 + b$$

$$y_3 = ax_3 + b$$

• buscamos convertir puntos en líneas y viceversa

• Definimos la i-ésima recta como: $(a_i, b_i) \rightarrow (x, y)$

$$y = (x_i) x + (-y_i)$$

Supongamos un alg encuentra (o determina que #) solución (x^*, y^*)

\exists pto (x^*, y^*)

$$\begin{aligned} &\text{en 3 rectas} (o) \quad y^* = a_1 x^* + b_1 \quad (1) \quad y^* = x_1 x^* - y_1 \\ &\Leftrightarrow y^* = a_2 x^* + b_2 \quad \Leftrightarrow \quad y^* = x_2 x^* - y_2 \\ &\quad y^* = a_3 x^* + b_3 \quad \quad \quad y^* = x_3 x^* - y_3 \end{aligned}$$

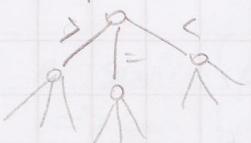
$$\begin{aligned} &\Leftrightarrow y_1 = x^* \cdot x_1 - y^* \quad \stackrel{(2)}{\Leftrightarrow} \quad \text{los puntos} \\ &y_2 = x^* \cdot x_2 - y^* \\ &y_3 = x^* \cdot x_3 - y^* \quad (x_1, y_1) \\ & \quad \quad \quad (x_2, y_2) \\ & \quad \quad \quad (x_3, y_3) \end{aligned}$$

son colineales.

- * Hay que verificar que la reducción tome $\Theta(n^{2-\epsilon})$. $\epsilon > 0$
 Bueno, el mapeo es lineal ($y \cdot -1$)

Llego para cualquier A que resuelva 3 -línneas, $\exists B$ que resuelve 3 colineal, tq: \sim reducción
 $T(B) = T(A) + \Theta(n)$

- P3 Todo algoritmo correcto debe tener al menos tantas hojas
 (a) como resp. posibles.



En este caso, los n elem. de A → posiciones
 deben estar en las hojas.
 \Rightarrow al menos n hojas.

- Cada nodo interno produce a lo más 2 nodos internos
 \Rightarrow hay a lo más 2^k nodos internos en el piso k .
 \Rightarrow hay a lo más $\sum_{i=0}^{k-1} 2^i$ nodos internos en los 1 's k pisos
 $(= 2^k - 1)$
- Como debe haber al menos n hojas.

$$2^k - 1 \geq n \Rightarrow k \geq \log_2(n+1) \quad (\text{peor caso})$$

Supongamos que solo busco los elem. que estén
 \Rightarrow la rta me da un código para el elemento.
 $A[i] >< >> <>>$ \Rightarrow puedo usar 1 bit por signo
 $\Rightarrow k_i$ bits para cada elem

$$\text{Pero } H = \sum p_i \log_2 \frac{1}{p_i}$$

Suponiendo una entrada uniforme, $p_i = \frac{1}{n} \Rightarrow H = \log_2 n$
 $\Rightarrow k_i \geq \log_2 n$ es el caso promedio.

2015年9月22日 (火)

Búsqueda Binaria con probabilidades de acceso.

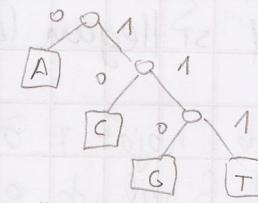
$$H = \sum P_i \log\left(\frac{1}{P_i}\right)$$

mín. número de bits por símbolo que puede usar una codificación si las prob. de los símbolos son P_1, \dots, P_n

• Algoritmo de Huffman

A	00	0.5	0
C	01	0.25	10
G	10	0.125	110
T	11	0.125	111

induce a errores. Un código no puede ser prefijo de otro.



$$H \leq \sum_{i=1}^n P_i l_i < H + 1$$

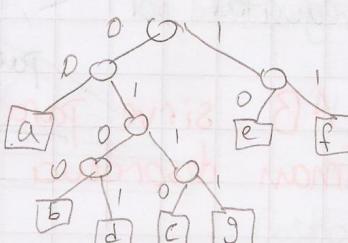
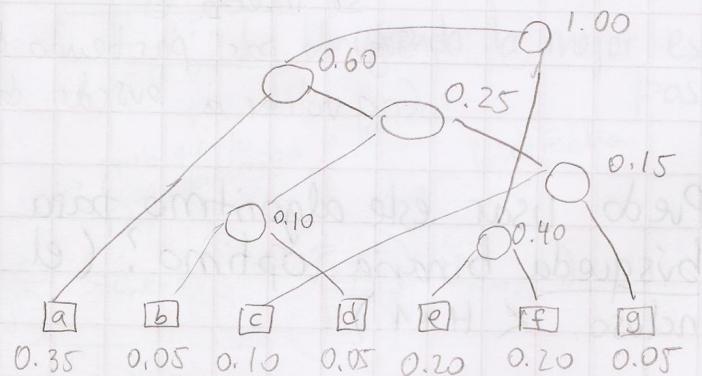
largo del código

largo promedio de un código

$(\sum_{i=1}^n P_i l_i) \cdot n =$ largo total del texto

$$\begin{aligned} 00 &= P(a) = 0.35 \\ 0100 &= P(b) = 0.05 \\ 0110 &= P(c) = 0.10 \\ 0101 &= P(d) = 0.05 \\ 10 &= P(e) = 0.20 \\ 11 &= P(f) = 0.20 \\ 0111 &= P(g) = 0.05 \end{aligned}$$

Árbol de peso mínimo



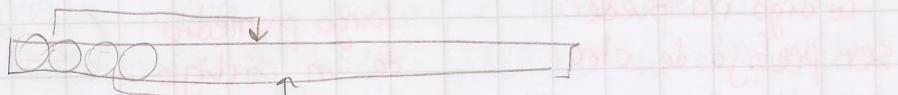
¿En qué tiempo se construye el árbol de peso mínimo?

Heapify: armo un heap, luego saco los 2 mínimos, los uno, los sumo, y el nuevo nodo lo reinserto en el heap.

$$\Theta(n \log n)$$

Otra manera? y si llegan los otros ordenados?

- 6



se inserta el segundo par partiendo de la última inserción!
(no volver a buscar del comienzo)

¿Puedo usar este algoritmo para crear un árbol de búsqueda binaria óptimo? (el costo esperado sería mínimo, incluso $< H+1$)

¿Qué hago? no pedo preguntar por <, =, >.

No es un ABB! es AB, sirve para codificar, pero no para buscar! Huffman desordena las hojas.

d) Entonces cómo puedo construir el mejor ABB con un algoritmo que no desordene las hojas?

Herramientas Secundarias

- Algoritmo de Hu-Tucker

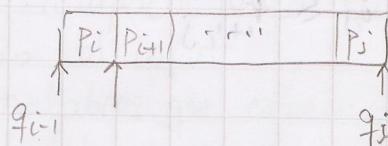
$\Theta(n \log n)$ → encuentra el mejor ABB. ($< H+2$)
 < $H+2$ (Garsia-Wachs) (con hojas ordenadas)

PROBLEMA MÁS GENERAL

		2.7	3.3				
p_1	p_2	p_3	p_4	p_5	p_6	p_7	
↑	↑	↑	↑	↑	↑	↑	
q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
							3.1

$$\sum p_i + \sum q_j = 1$$

$$P_{ij} = q_{i-1} + p_i + q_i + p_{i+1} + q_{i+1} + \dots + p_j + q_j$$



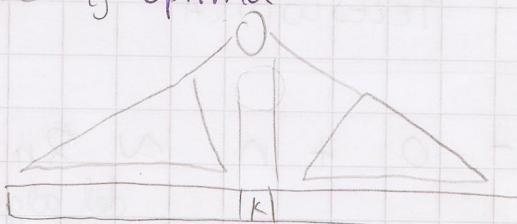
C_{ij} = costo de buscar en
 (cuantos a accesos)
 $C_{ii} = 1$

usando la mejor estrategia
 posible.

$$C_{ij} = 1 + \min_{1 \leq k \leq j} \left(\frac{p_{i,k-1}}{p_{ij}} C_{i,k-1} + \frac{p_{k+1,j}}{p_{ij}} C_{k+1,j} \right)$$

dónde poner
 raíz r_{ij} óptima

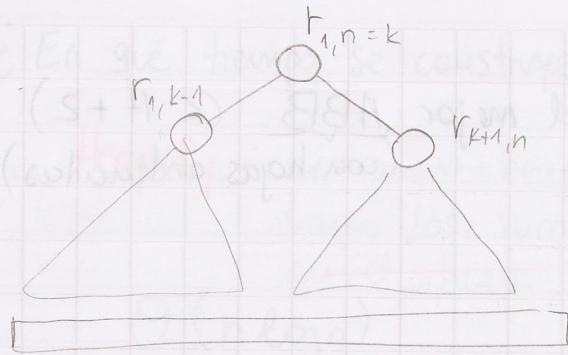
"dado que estoy
 buscando en $[i, j]$ "



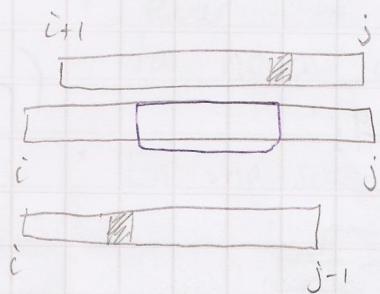
Luego de hacer todos estos
 cálculos, sabemos

Costo promedio de búsqueda:

$$C_{1,n}$$



¿Y $O(n^2)$?



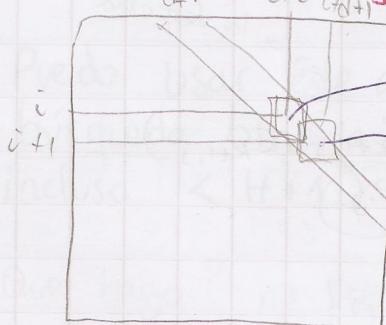
$$r_{i,j-1} \leq r_{ij} \leq r_{i+1,j}$$

con programación dinámica, el trabajo es rellenar esa matriz

$$\sum_{i=1}^n \sum_{j=i}^n j-i+1 = \frac{n^3}{6} = \Theta(n^3)$$

de j a i
buscando
el mínimo

Ahora $\rightarrow C_{ij} = 1 + \min_{d=1}^{r_{ij}} \left(\frac{P_{i,k-1}}{P_{ij}} C_{i,k-1} + \frac{P_{k+1,j}}{P_{ij}} C_{k+1,j} \right)$



$$r_{i,j-1} \leq k \leq r_{i+1,j}$$

$$- r_{i,i+d-1} + 1$$

$$r_{i+1,i+d} - r_{i,i+d} + 1$$

: TELESCÓPICA

$$r_{i+1,j} - r_{i,j-1} + 1$$

Sobrevive
algún
 $r \leq n$

- $O(n^2)$ $\sim 2n$ costo
del algoritmo

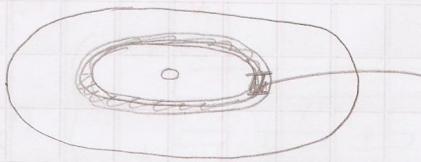
por diagonal

y son n diagonales
 $\Rightarrow O(n^2)$

Y si en vez de probabilidades de acceso
tengo COSTO de acceso?

Memoria Secundaria

2015年9月24日(木)



tiempo:

- seek → mover puntero a pista
- latencia → tiempo para que el bloque pase de ^{bajo del} cabezal.
- transferencia.

Para algoritmos en mem secundaria:

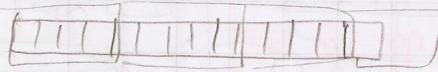
- minimizar los accesos a disco
- aprovechar localidad de bloques y entre bloques
(pasadas secuenciales)

I/O model: costo = # bloques leídos y escritos (no toma en cuenta que un alg. secuencial es mejor que uno salta.
Tampoco toma en cuenta trabajo en CPU.)

- n = tamaño del input
- B = n de palabras que caben en un bloque de mem secundaria
- M = n de palabras que caben en RAM.

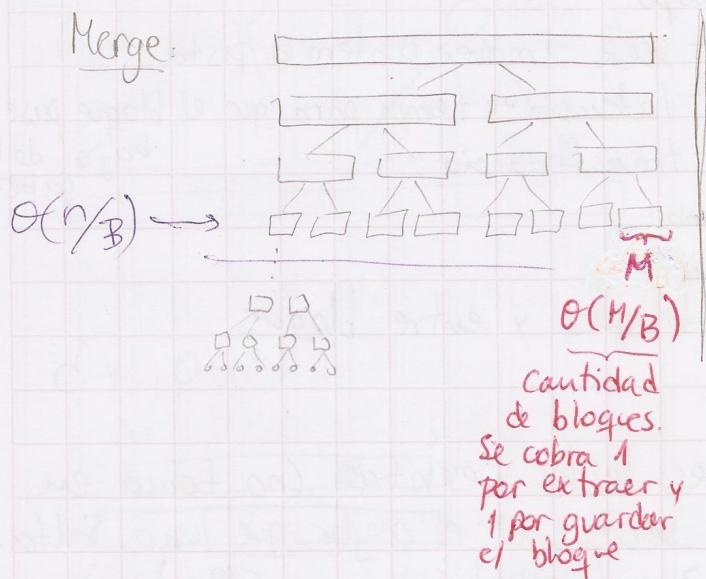
El costo en I/O model se puede definir/medir en fn. de n, B y M.

Ej: Si leo todos los n enteros de un arreglo en disco, el costo es,
 $\Theta\left(\frac{n}{B}\right)$ secuencial
 $\Theta(n)$ al azar.



Algoritmo Merge:

Merge:



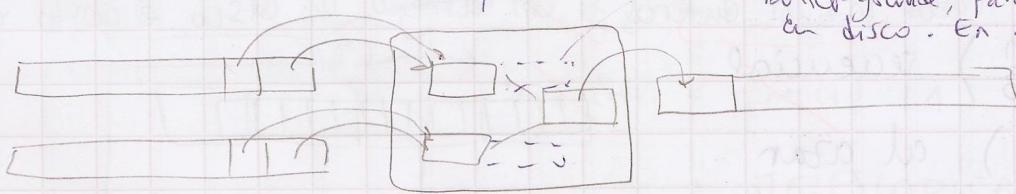
$$\Theta(n/B) \rightarrow$$

Cantidad de bloques.
Se cobra 1 por extraer y 1 por guardar el bloque

El algoritmo deja de dividir hasta que los pedazos son de tamaño M . Luego, en ese piso se hacen $\Theta(\frac{n}{B})$ accesos.

Costo total: $\Theta\left(\frac{n}{B} \cdot \underbrace{\log \frac{n}{M}}_{\text{pisos.}}\right)$

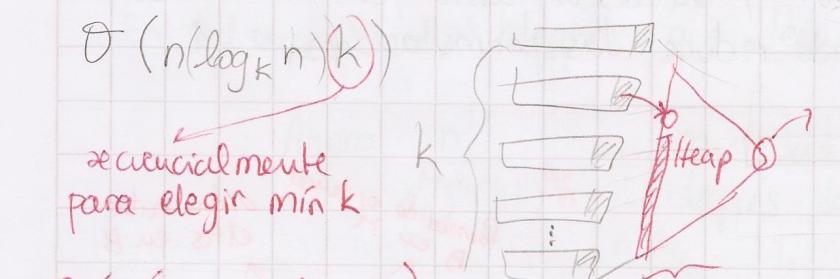
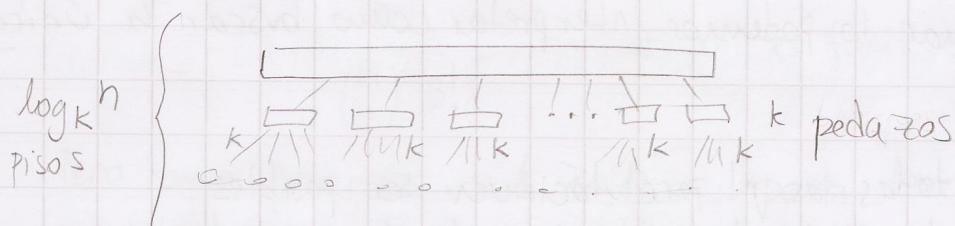
en la vida real es mejor tener un buffer grande, para leer secuencialmente en disco. En el modelo I/O da lo mismo



RAM
el merge es
asíncrono, luego
se lleva el bloque
de aptput en
cuálquier momento.

¿se puede hacer mejor?

¿Por qué merge divide de a 2? ¿Si dividimos de a más?



$$= \Theta(n(\log_2 n) \log_2 k)$$

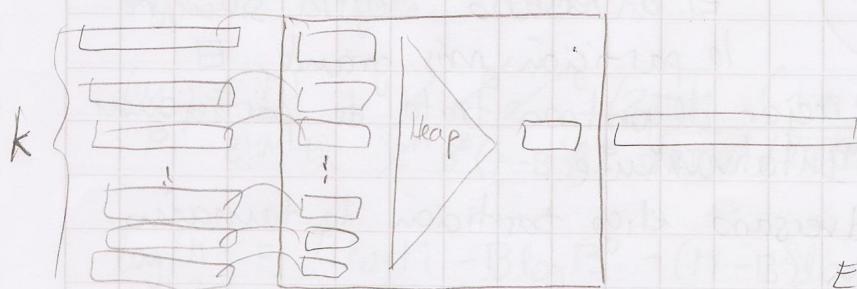
$$= \Theta(n \log_2 n)$$

:
no es mejor...

¿Cómo extraigo el mínimo de k eficientemente?
→ Usar heap

Se extrae el min y se inserta ello al heap del mismo arreglo del cual se extrajo min
→ $\Theta(\log k)$

Sirve este merge para el modelo I/O de ordenamiento en disco?



$$\rightarrow \Theta\left(\frac{n}{B} \log k \left(\frac{n}{m}\right)\right)$$

Pues las op. de heap se hacen en memoria
⇒ costo Θ .

El buen k es $\leq \frac{M}{B}$ (los máximos de bloques que caben en memoria)

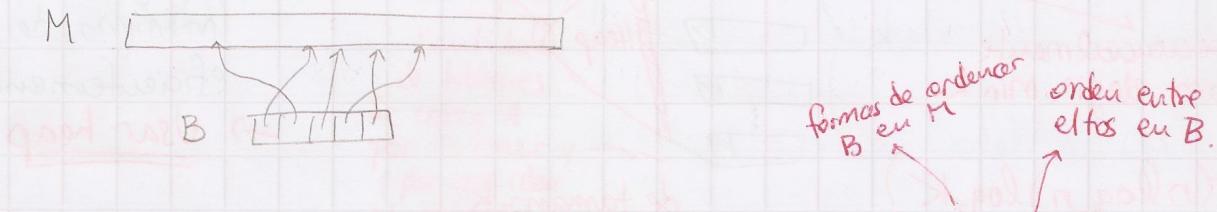
$$\Rightarrow \boxed{\Theta\left(\frac{n}{B} \log \frac{M}{B} \left(\frac{n}{M}\right)\right)}$$

Pero ese k óptimo en la vida real no b es tanto, pues conviene tener buffers más grandes.

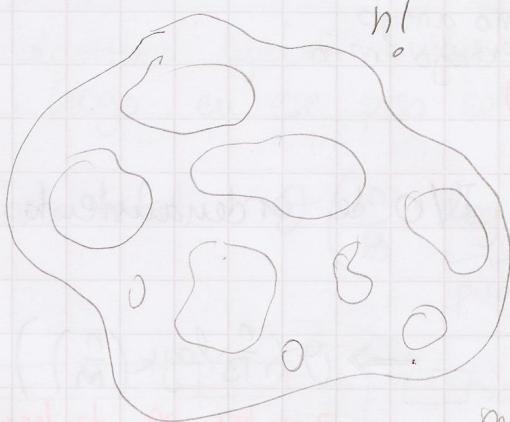
• Cota inferior para ordenar en disco

Recordar que ordenar lo podemos interpretar como buscar la única permutación.

- Inicialmente, todas las $n!$ permutaciones son posibles.
- El algoritmo lee el input y va descartando permutaciones.
- Seo puede terminar cuando reduce las permutaciones a 1.



En este procedimiento, el proceso puede resultar en $\binom{M}{B} B!$ casos



$$\binom{M+B-1}{B}$$

$\binom{M}{B} B!$ es la cantidad de clases en las que se particiona el conjunto de $n!$ perms.

El adversario elegirá siempre la partición más grande. El mejor algoritmo trata de particionar equitativamente.

En cualquier caso, el adversario elige partición de tamaño al menos $\frac{n!}{\binom{M}{B} B!}$

$$\left(\frac{n!}{\binom{M}{B} B!}\right) 0$$

Aux #3

Luego de t lecturas de bloques, el tamaño del conjunto es

$$\frac{n!}{(M)^t (B!)^t}$$

Pero somos generosos. $t \geq \frac{n!}{B}$ para leer todo.

En tal caso, hay bloques que leemos más de una vez.

Ahora $\frac{n!}{(M)^t (B!)^{\frac{n!}{B}}}$ cantidad de permutaciones admisibles después de leer t veces.

$$\frac{n!}{(M)^t (B!)^{\frac{n!}{B}}} \leq 1 \Rightarrow \frac{n!}{(B!)^{\frac{n!}{B}}} \leq \left(\frac{M}{B}\right)^t$$

$$\Rightarrow \log n! - \left(\frac{n!}{B}\right) \log B! \leq t \log \left(\frac{M}{B}\right)$$

$$n! = \frac{n^n}{e^n} \sqrt{2\pi n} \left(1 + O\left(\frac{1}{n}\right)\right)$$

$$\log n! = n \log n - O(n)$$

$$\left(\frac{M}{B}\right) = \frac{M!}{B!(M-B)!} = \frac{M^M e^B e^{M-B}}{e^M B^B (M-B)^{M-B}} \frac{\sqrt{2\pi M}}{\sqrt{2\pi B}} \frac{\sqrt{2\pi(M-B)}}{\sqrt{2\pi(M-B)}}$$

$$\log \left(\frac{M}{B}\right) = \frac{M \log M - B \log B - (M-B) \log(M-B) + O(\log M)}{B \log M + (M-B) \log M}$$

$$= B \log \frac{M}{B} + (M-B) \log \frac{B}{M-B} + O(\log M) = B \log \frac{M}{B} + O(B + \log M)$$

$$\ln 1 + x \leq x$$

$$\log \frac{(M-B)+B}{M-B} = \log 1 + \frac{B}{M-B} = O\left(\frac{B}{M-B}\right)$$

$$O\left(B \log \frac{M}{B}\right)$$

Luego $n \log n - n \log \frac{M}{B} \leq t$

$$B \log \frac{M}{B}$$

$$\Rightarrow \frac{n}{B} \log \frac{M}{B} \left(\frac{n}{B} \right) \leq t.$$

Cota Superior $\frac{n}{B} \log \frac{M}{B} \left(\frac{n}{B} \right) = 2 \left(\frac{n}{B} \log \frac{M}{B} n \right)$

↳ no es tan temible.

$$M > n^\alpha \quad \forall \alpha < 1$$

$$\frac{n}{B} \log \frac{n}{B} \left(\frac{n}{B} \right)$$

M tendría que ser demasiado grande para cambiar el orden de magnitud.

Tan grande que no tendría sentido práctico.

Auxiliar 3 - Memoria Externa

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

28 de Septiembre del 2015

1. Dados dos arreglos $A[1, N]$ y $B[1, N]$ de enteros que no caben en memoria principal, se desea construir (también en disco) el arreglo $C[i] = A[B[i]]$. Diseñe un algoritmo eficiente de memoria externa para construir C y analícelo.
2. Diseñe un algoritmo eficiente para memoria secundaria que, dada una relación binaria R sobre el dominio $\{1\dots n\}$, determine si R es simétrica. Suponga que la relación R se presenta en disco como un archivo secuencial de todos sus pares $(i, j) \in \{1\dots n\} \times \{1\dots n\}$.
3. Considere un arreglo de N elementos que no caben en memoria principal. Se desea encontrar un elemento que sea mayoría (es decir, que aparezca más de $\frac{N}{2}$ veces) y reportar su frecuencia. Si no existe tal elemento, se debe reportar **no**. Se dispone de una memoria de tamaño $M = \Theta(1)$. Diseñe un algoritmo eficiente que resuelva este problema.

Propuesto: Extender este problema a encontrar k elementos que aparezcan más de $\frac{N}{k+1}$ veces

Propuestirijillos

1. Dado un grafo dirigido $G = (V, E)$, con $|V| = n$ y $|E| = e$, ambos mucho mayores que M . El grafo es un archivo secuencial de pares (u, v) en disco, donde se listan las aristas. Se desconoce n y e . Considere $V = \{1...n\}$ para simplificar. Se pide construir y analizar algoritmos eficientes en el modelo de memoria secundaria para:

- Calcular el grado interior y exterior (número de arcos entrantes y salientes) de todos los nodos.
- Calcular el cuadrado del grafo, definido como $G^2 = (V, E')$, donde:

$$E' = \{(u, v) \mid u, v \in E \wedge \exists w \in V, (u, w) \wedge (w, v) \in E\}$$

Puede suponer, para este caso, que $M \geq n$, pero no que $M \geq e$.

- **Bonus track:** calcule cualquier potencia del grafo. Proponga un algoritmo eficiente en memoria y analícelo. No es aceptable una solución que cueste $k - 1$ veces o más que lo que obtuvo en el punto anterior.

2. Considere dos matrices esparsas de $N \times N$, almacenadas en disco en forma de una secuencia (i, j, v) para cada valor $A(i, j) = v \neq 0$. Se requiere obtener el producto de las dos matrices, pero se dispone solamente de una RAM de tamaño $M = \Theta(N)$. Diseñe un algoritmo eficiente en términos de p y p' , las cantidades de celdas no cero en las dos matrices. (Hint: se puede conseguir básicamente $O(pp')/(MB))$).

2015年9月28日 (月)

Aux #3

Memoria externa

$$\text{Ordenar: } \Theta(N \log N) \rightarrow \Theta\left(\frac{N}{B} \log \frac{\frac{N}{B}}{B}\right) = \Theta(n \log m n)$$

N: n° de elementos

B: unidad (de bloque) de I/O al disco

M: tamaño de la memoria

$$\frac{N}{B} = n, \quad \frac{M}{B} = m$$

[P1]

$$A[1..N] \Rightarrow C[i] = A[B[i]] \quad i=1..N$$

$\underbrace{B[1..N]}$
contienen
todos los números
 $\in \{1..N\}$

$\Theta(N)$ no queremos!

queremos que quede en
función de n y m
involucra bloques.

* Sea un arreglo $X[1..N]$

- Al ordenar, obtengo $[1..N]$
- Si considero a X como una permutación ϕ_X , ordenar aplica ϕ_X^{-1}

• Algoritmo:

- Copiar A a A' tq $A'[i] = (\text{id}, A[i])$ // $\Theta\left(\frac{N}{B}\right) = \Theta(n)$
- Ordeno A' por el segundo argumento $\Rightarrow (\phi_A^{-1}, \text{id})$ // $\Theta(n \log m n)$
- Copio B en el 2º argumento $\Rightarrow (\phi_A^{-1}, \phi_B)$ $\underbrace{A'}_{\text{aplicar } (\phi_A^{-1})} \quad \Theta(n)$
- Ordeno A' por el primer argumento $\Rightarrow (\text{id}, \phi_A \circ \phi_B)$ // $\Theta(n \log m n)$
- Copio el 2º argumento en C $\underbrace{\Rightarrow \phi_A \circ \phi_B}_{A[B[i]]} \quad \Theta(n)$

Tiempo total: $\Theta(n \log m n) \ll \Theta(n)$

Ejemplo

Hacer dibujo!! Ppt, revisar si $A \leftrightarrow B$ estuvieron bien.

$$A = [2 \ 4 \ 1 \ 5 \ 3] = (1) \quad A' = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 5 & 3 \end{bmatrix}$$

$$B = [3 \ 5 \ 2 \ 1 \ 4]$$

$$2) \quad A' = \begin{bmatrix} 3 & 1 & 5 & 2 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

$$\Rightarrow C = [1 \ 3 \ 4 \ 2 \ 5]$$

$$3) \quad A' = \begin{bmatrix} 3 & 1 & 5 & 2 & 4 \\ 3 & 5 & 2 & 1 & 4 \end{bmatrix}$$

$$4) \quad A' = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 1 & 3 & 4 & 2 \end{bmatrix}$$

5) Esto es $B[A \leftrightarrow C]$ X

P

P2

Relación Binaria R }
(R ⊆ {1..n} × {1..n}) } [(a₁, b₁), (a₂, b₂)...] }
|R| > M

Simetría: "(a, b) ∈ R ⇔ (b, a) ∈ R"

R R'

- ⇒ • Hacer copia de R: R'
- Ordenar R por el primer argumento → y luego por el 2º
- Ordenar R' por el segundo → luego por el 1º
- Comparar secuencialmente (invirtiendo los pares de R')
- Si una comp. falla ⇒ no es simétrica.

P3

A, |A|=N, A = a₁, a₂, ..., a_n.

Mayoría (a₁, ..., a_n)

X ← a₁

c ← 1

for y in a₂ ... a_n:

| if x == y

| c ← c + 1

| else c ← c - 1

| if c == 0:

| c ← 1

| x ← y

return X.

Dem: por contradicción.

Supongamos que $\exists a_k$ con $> \frac{N}{2}$ apariciones

Si $a_k \neq X$, todas las instancias de a_k deben haber sido "quemadas" por otros elementos

no necesariamente el mismo en todos los casos.

⇒ hay $> \frac{N}{2}$ a_i's ≠ a_k ⇒ c

Luego Mayoná retorna el a_k correcto si \exists ; si no

→ Verificar (x, a_1, \dots, a_N)

Compara cada a_i con x y cuenta el # de aciertos

⇒ si son más de N → retorna el conteo

⇒ else retorna "NO"

* ojo con el falso positivo del contador, en casos extremos

PPTOS :

- [P1] a) ordenar y contar
b) \sum pares de nodos entre los que existe un camino de largo $?$.
c)

2015年9月29日 (X)

Colas de prioridad en memoria secundaria

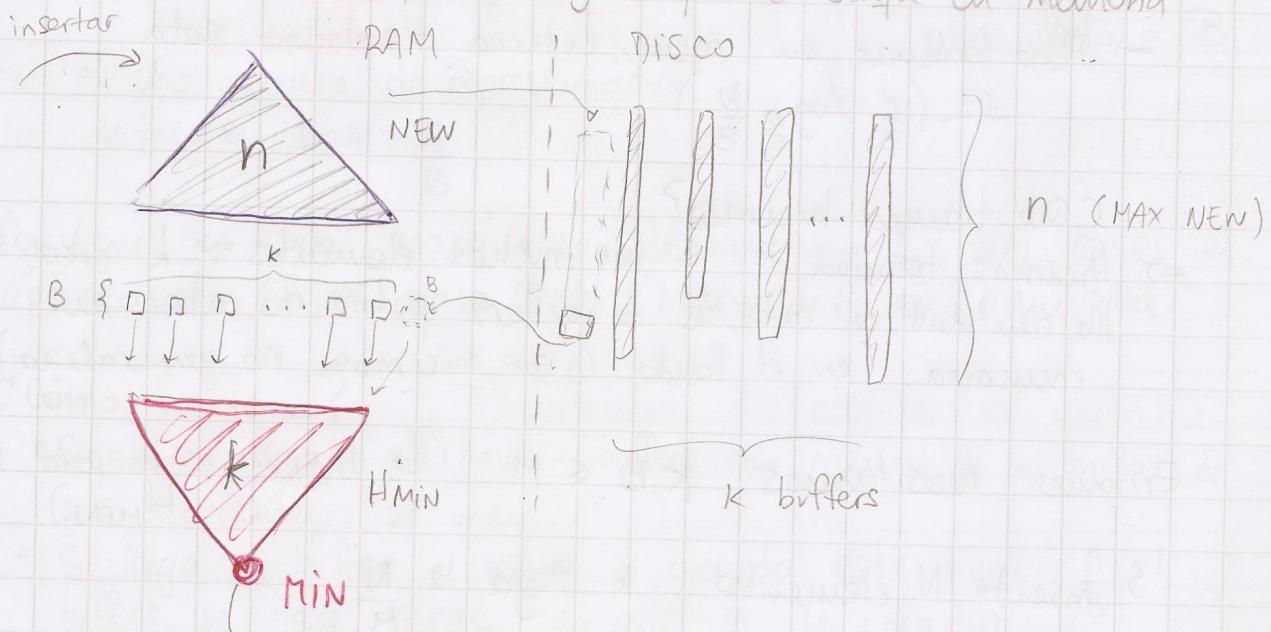
OPERACIONES:

- inicializar
- insertar elementos
- extraer mínimo.

En mem principal, tenemos el HEAP BINARIO

Idea:

- tener un heap en memoria principal
- Si se llena, ordeno sus elementos y los escribo a disco
en un arreglo como un arreglo: "buffer"
- De cada buffer, monto el primer bloque en memoria



- En otro heap en memoria se almacenan estos bloques

Ojo al insertar, comparar con el mínimo del H_{MIN}

Análisis:

Cuánto paga un elemento al insertarlo y eventualmente extraerlo?
(en I/O)

- $\Theta(\frac{1}{B})$ {
- entra a NEW $\rightarrow \emptyset$
 - viaja a un buffer en Disco $\rightarrow \frac{1}{B}$ (promediado cuando entran B elementos a la vez)
 - Pasa a un minibloque $\rightarrow \frac{1}{B}$
 - llega a H_{\min} $\rightarrow \emptyset$
 - Sale $\rightarrow \emptyset$

no se pondera por M (tamaño buffer)
Pues en el modelo no se considera secuencialidad.

Cuál pasa si inserto N elementos y dp los saco todos?

$$\Rightarrow \Theta\left(\frac{1}{B}\right) \cdot N = \Theta\left(\frac{N}{B}\right), \text{ pero salen en orden}$$

\rightarrow Pero ordenar en mem. externa tiene una cota

$$\Omega\left(\frac{N}{B} \log \frac{M}{B} \frac{N}{B}\right)$$

Cuál trampa hicimos?

\Rightarrow Hicimos trampa si: Si hay muchos elementos $\Rightarrow k$ crece y la cantidad de cabezas de buffers podría no cabrer en memoria. (en el fondo, lo que hicimos no generaliza)

Entonces necesitamos: $kB \leq M$ (los bloques usados para rellenar H_{\min})

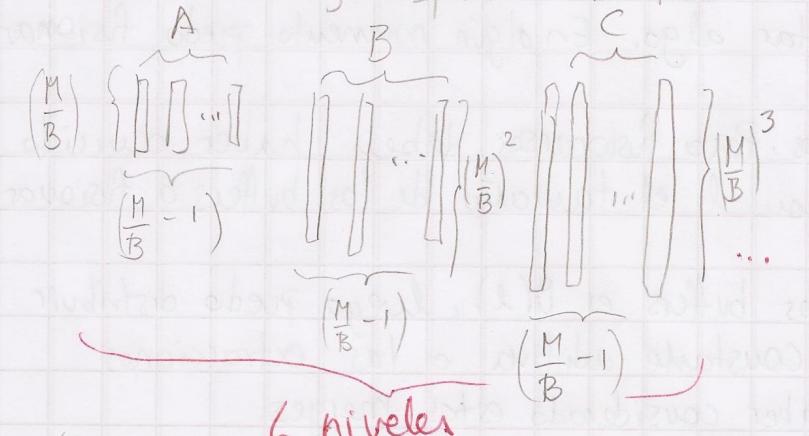
Si inserto N elementos, k llega a $\frac{N}{M}$

$$\Rightarrow \frac{N \cdot B}{M} \leq M \Rightarrow N \leq \frac{M^2}{B}$$

• En este caso $\log \frac{N}{B} \approx 1$, luego $\Theta\left(\frac{N}{B} \log \frac{N}{B}\right) \approx \Theta\left(\frac{N}{B}\right)$

¿Qué hacemos si $N > \frac{M^2}{B}$? es decir, los bloques para $HMIN$ no caben en mem. ppal.

La idea es limitar el número de cabezas de buffers en mem. ppal.
Haremos una jerarquía de buffers.



Esto es una manera de representar los datos en base $\frac{M}{B}$

Conviene que cada jerarquía sea de tamaño $\frac{M}{B}$ pues ese es el óptimo de "filas" para hacer Mergesort en mem. ppal.

¿Cómo funciona?

- Si en un mismo nivel hay 2 buffers con menos de la mitad de sus elementos, se une.
- Si llega un buffer al nivel i cuando éste tiene $\frac{M}{B} - 1$ buffers, se hace merge de éstos y se crea un buffer del nivel $i+1$ (posiblemente) repitiéndose la operación.

¿Cuántos niveles hay?

El máx número de niveles es tal que

$$\frac{N}{B} = \Theta\left(\left(\frac{M}{B}\right)^L\right) \Rightarrow L = \Theta\left(\log_{\frac{M}{B}} \frac{N}{B}\right)$$

Por ejemplo, si llega un M -ésimo buffer a A,

se hace merge de todos los buffers para crear un nuevo buffer de tamaño $\left(\frac{M}{B}\right)^2$ y mandarlo a B.

el ipc del elemento



Evaluemos el costo de la vida de un elemento.

- Supongamos que pasa por todos los niveles

- Entrar al 1^{er} nivel: $1/B$

- Pasar del $i \rightarrow i+1$: $2/B \Rightarrow O\left(\frac{L}{B}\right)$ (leer, merge en mem ppal, escribir)

- Salir a mem ppal: $1/B$

Wait! Nos faltó contar algo. En algún momento puedo fusionar buffers semivacíos.

* Los buffers hacen llenos. Para fusionarse, deben haber ocurrido $\frac{l}{2} + \frac{l}{2}$ extracciones (con l el tamaño de los buffers a fusionar)

Hacer el merge de los buffers es $O(l)$, luego puedo distribuir este costo como una constante aditiva a las extracciones
 \Rightarrow "está bien" no haber considerado estos merges.

- Insertemos y luego extraigamos N elementos

$$O\left(\frac{NL}{B}\right) = O\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$$

es L !

¿Caben los bloques en memoria?

Arboles de prioridad (Reposo)

01/Octubre/2015

- insertar
- Extraer mínimo



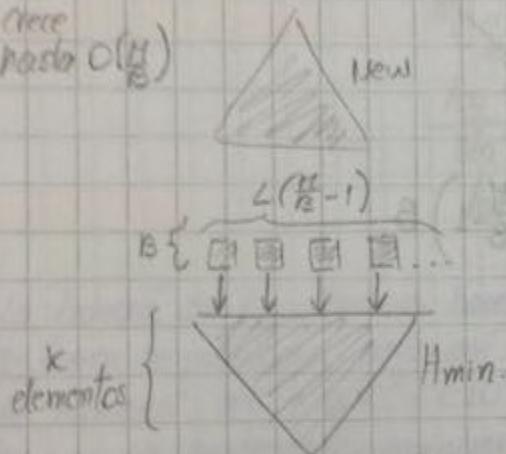
árboles binarios

[] = arreglos
(ordenados)

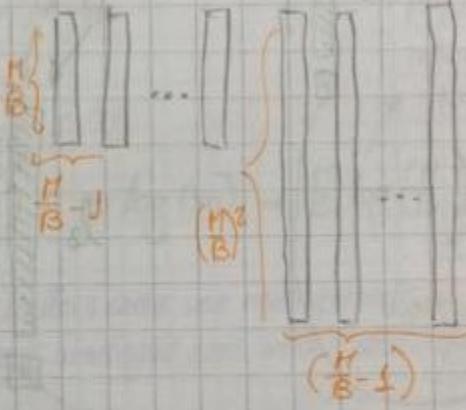
█ = bloques

Memoria

tiene
tamaño $O(\frac{H}{B})$



Disco



6 niveles
de tamaño $(\frac{H}{B}-1)$

- ¿Almacenar los bloques en memoria?

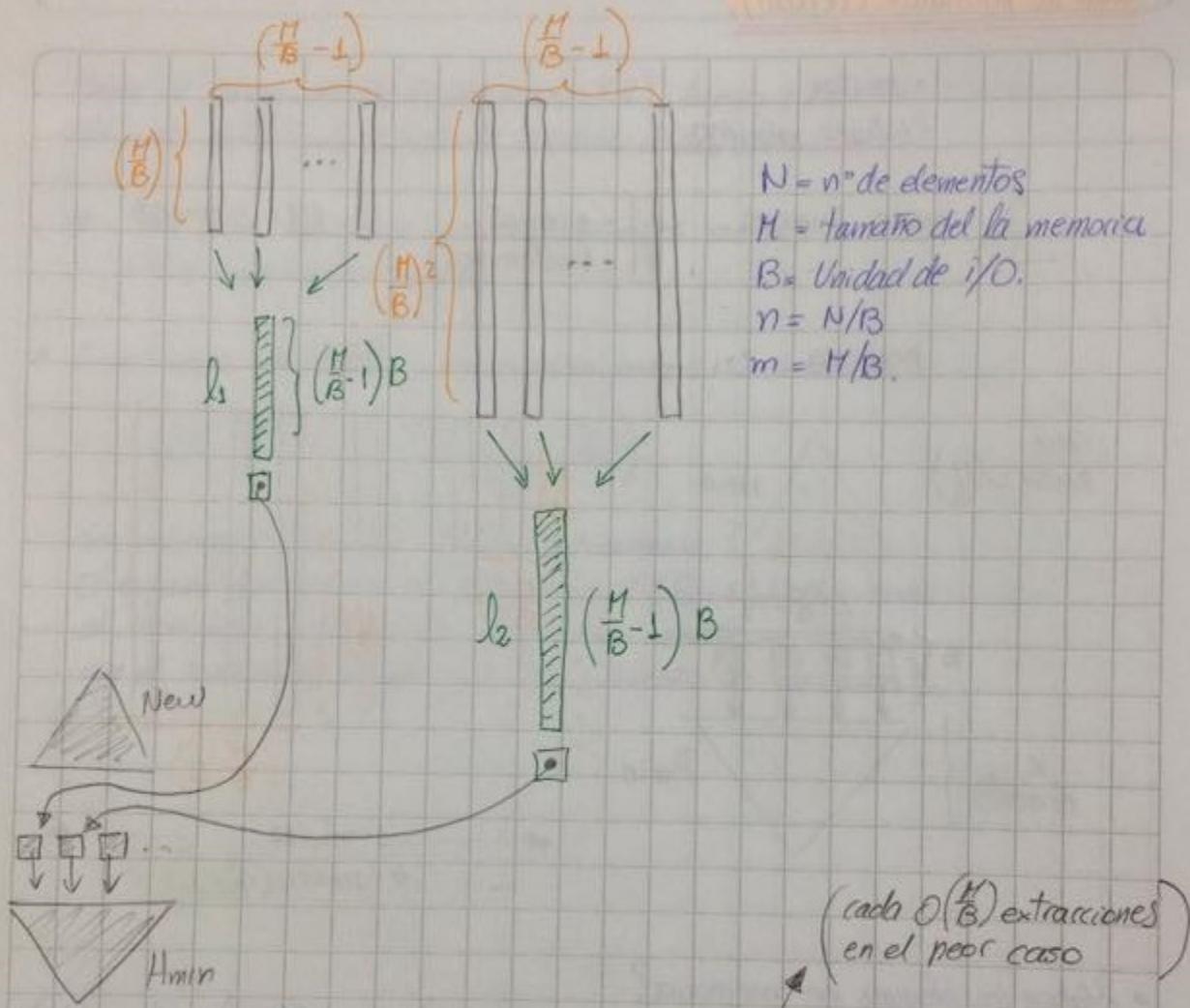
Espacio ocupado por los bloques :

$$L \cdot \left(\frac{H}{B}-1\right) B = \left(\frac{H}{B}-1\right) B \log_{\frac{H}{B}} \left(\frac{H}{B}\right)$$

$$= O(H \log_{\frac{H}{B}} \left(\frac{H}{B}\right))$$

La idea es aprovechar los niveles que creamos.

Idea: para cada nivel, guardaremos sólo 1 bloque en memoria.



En disco, agregamos 1 arreglo (buffer) ordenado l_i , para cada nivel i

→ Cada l_i contiene el merge de tamaño $(\frac{H}{B}-1) \cdot B$ de los buffers de nivel i .

- Al heap Min lo alimentan las cabezas de $l_i + i$
- Hay que reciclar l_i si se vacía (tiempo lineal $O(\frac{H}{B})$)
- Al crear un nuevo buffer en el nivel i , hay que hacer un merge parcial con l_i .

Entonces, para el espacio ocupado por los bloques:

$$LB = B \log_B \left(\frac{N}{B} \right) \leq M \quad \text{imponemos/ necesitamos...}$$

$$B \left(\log_B \left(\frac{N}{B} \right) - \log_B (B) \right) \leq B \cdot m \quad n \cdot \frac{N}{B} \Rightarrow N = n \cdot B$$

$$\log_B (n) + \log_B \left(\frac{m}{B} \right) - \log_B (B) \leq m$$

$$\log_B \left(\frac{n}{B} \right) \leq m$$

El primer intento costaba

$$\frac{\log(n)}{\log \left(\frac{n}{B} \right)} \leq m \quad \rightarrow N \leq \frac{M^2}{B}$$

$$\log(n) \leq m \log(m) - \frac{M}{B} \log \left(\frac{m}{B} \right)$$

- * La "trampa" del logaritmo es hacer buffers cada vez más grandes, de forma de potencias, para así hacer analogía con una "distribución de números" $a10 + b10^2 + c10^3 + \dots$. Así uno va haciendo formaciones de tiempo logarítmico, asumiendo que los buffers son ordenados. (Y luego tener una representante por cada familia de buffers y hacer merge de disco).

Auxiliar/Clase 4 - B-Trees y algo de R-Trees

CC4102 - Diseño y Análisis de Algoritmos
Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

5 de Octubre del 2015

B-Trees

Un *B – Tree* de orden m es un árbol que satisface las siguientes propiedades:

- Cada nodo tiene $\leq m$ hijos.
 - Cada nodo, excepto el nodo raíz y las hojas, tienen $\geq m/2$ hijos.
 - Todas las hojas se encuentran en el mismo nivel.
 - Un nodo que no es hoja y tiene k hijos contiene $k - 1$ llaves.
1. Especifique el algoritmo de inserción en un B-Tree.
 2. Especifique el algoritmo de borrado en un B-Tree.
 3. Especifique el algoritmo de búsqueda en un B-Tree.

R-Trees

Un *R – Tree* es una estructura de datos jerárquica, basada en una variante de B-Trees que sólo contiene datos en las hojas (a esta variante se le llama B+ -Trees); se les usa para organizar dinámicamente un conjunto de objetos geométricos. Cada nodo de un R-Tree representa el MBR (*minimum bounding rectangle*) que engloba a sus hijos. Los MBRs de diferentes nodos pueden solaparse. Adicionalmente, un R-Tree tiene las siguientes propiedades:

- Cada nodo hoja tiene hasta M elementos, donde el mínimo número de entradas es $m \geq M/2$.
 - El número de entradas que cada nodo interno no-raíz almacena está entre $m \geq M/2$ y M .
 - Todas las hojas del R-Tree están en el mismo nivel.
 - El número mínimo de entradas en el nodo raíz es 2, a menos que sea una hoja.
1. Especifique el algoritmo de inserción en un R-Tree.
 2. Especifique el algoritmo de borrado en un R-Tree.
 3. Especifique el algoritmo de búsqueda en un R-Tree.

2015年10月5日 (月)

Diccionarios en memoria secundaria.

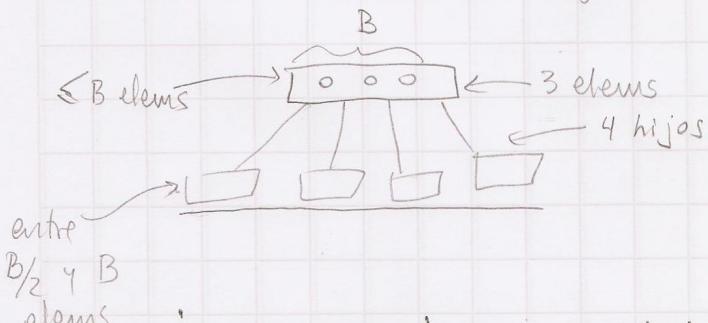
B-Tree : insertar sucesor
 borrar Predecesor
 buscar ↳ iterar en orden...

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 elem. 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 elem. $i-1$ e i .
- todas las hojas tienen la misma profundidad.



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

• Invariantes de la estructura:

- 1) Cada nodo almacena a lo más B elementos.
- 2) Salvo la raíz, cada nodo tiene al menos $B/2$ elementos.
- 3) Un nodo interno con k elementos tiene $k+1$ hijos.
- 4) 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 .
- 5) 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:

① Búsqueda : (κ)

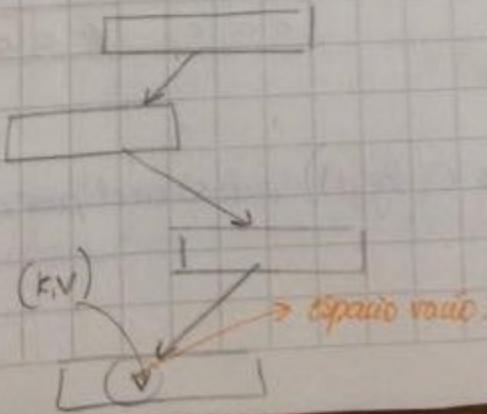
↳ Bajar por el árbol eligiendo el hijo adecuado en cada vez (toma $O(\log_B N)$).

Lo complicado es insertar / Borrar.

② insertar : (k, v)

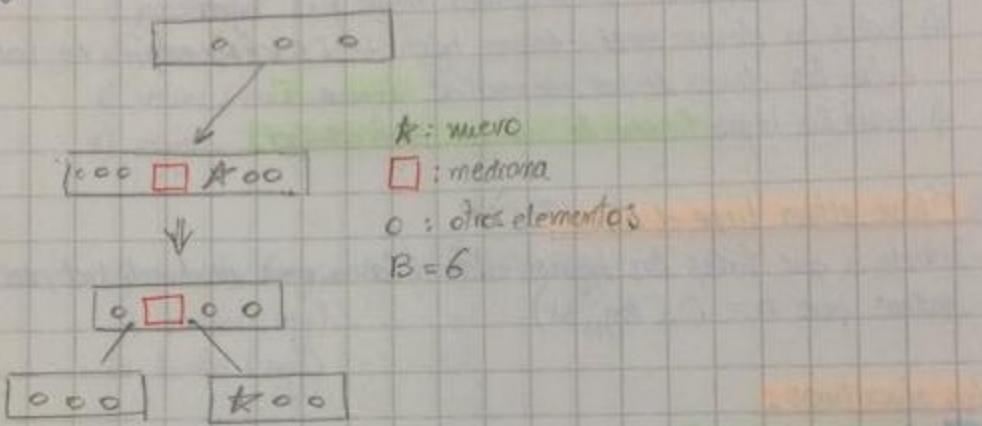
↳ Se busca la clave a insertar; al no encontrarse, encontramos el lugar donde debería estar. (en una hoja). $\{O(\log_B N)\}$

a)



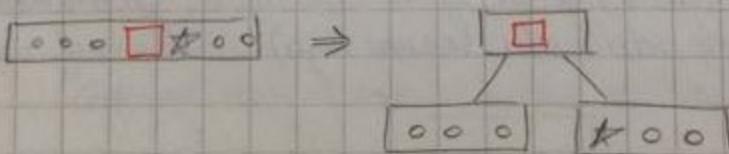
$O(\log_8 N)$

- 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.

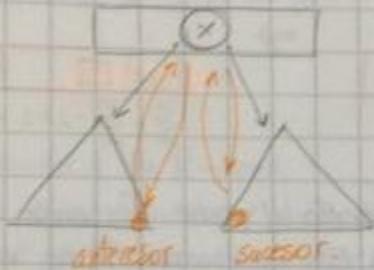


⇒ Tenemos que insertar es $O(\log_8 N)$ incluso en el peor caso.

3) Borrado (\times):

Buscar y encontrar la clave.

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



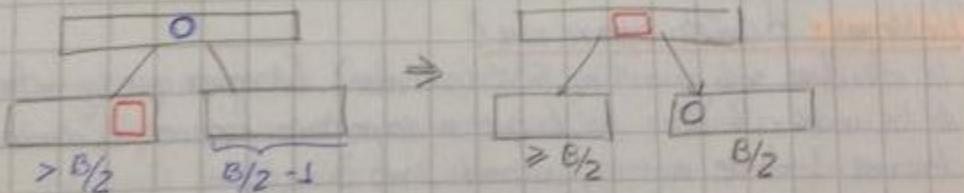
⇒ sin pérdida de generalidad, el borrado **ocurre en las hojas**.

- En cambio, si tenemos que borrar en una hoja.

Se borra, y si la hoja queda con menos de $B/2$ elementos.

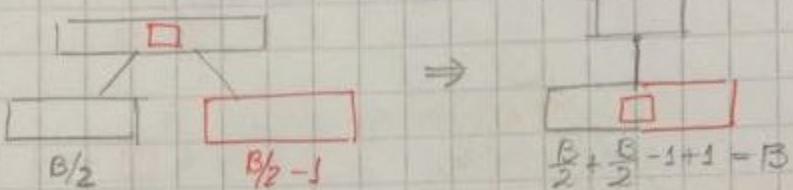
↳ Hago los vecinos y le pido un elemento a alguno:

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 Bst trees.

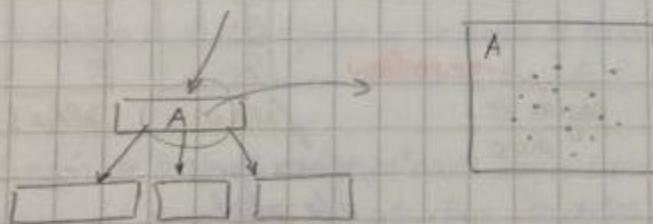
↳ Todos los elementos (datos) sólo están en las hojas. En los nodos internos tenemos sólo punteros ("copias de llaves").

- ↳ las hojas tienen púnticos entre ellas, por lo que se puede hacer un recorrido secuencial a través de ello.

R-trees

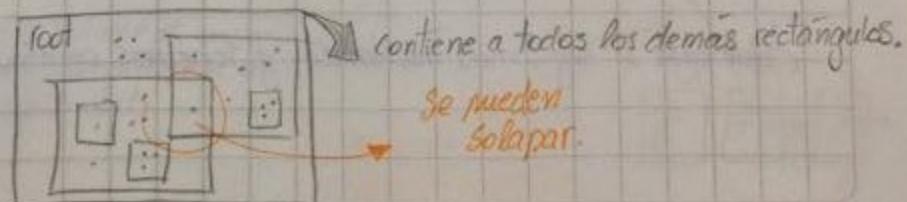
↳ Son muy parecidos a los R-trees.

- 1) Cada nodo interno almacena hasta 13 rectángulos.
 - 2) Salvo la raíz, todo nodo interno tiene al menos $13/2$ 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 crezca menos en área, o bien, el que me hace más overlapping, es decir, el que me contenga más).

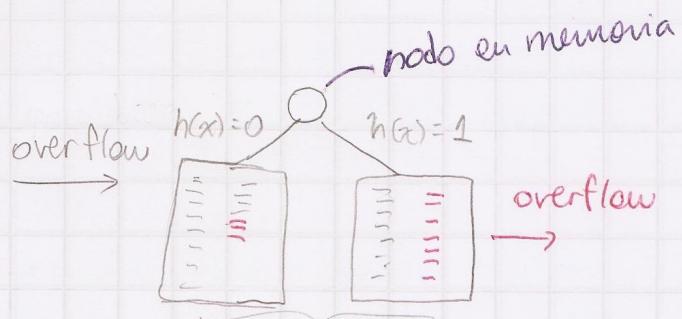
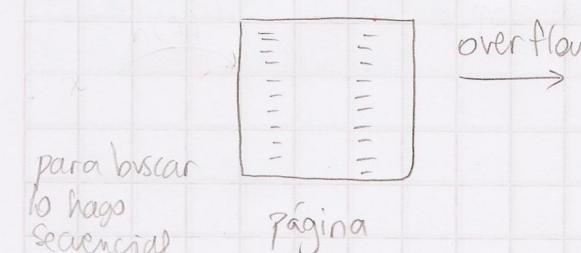


Hashing en Disco

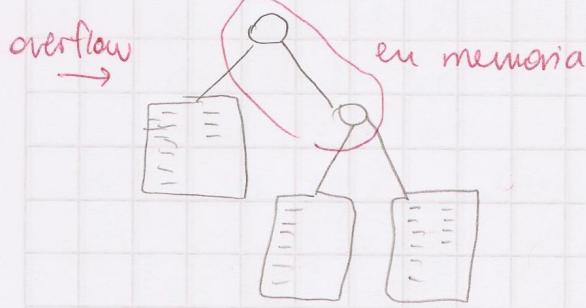
2015年10月6日 (X)

- 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 (?)

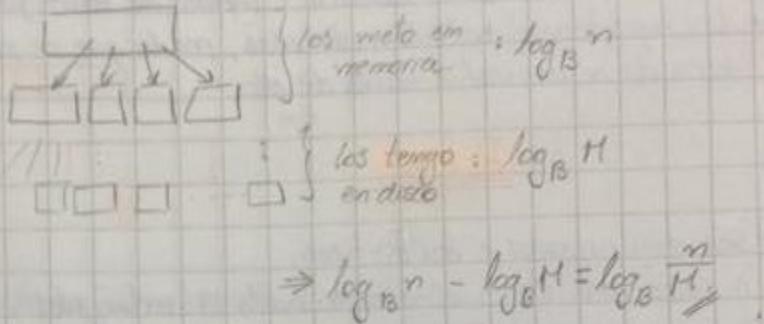
2 sin overflow: leer y escribir

3 con " · " : leer y escribir 2 págs.

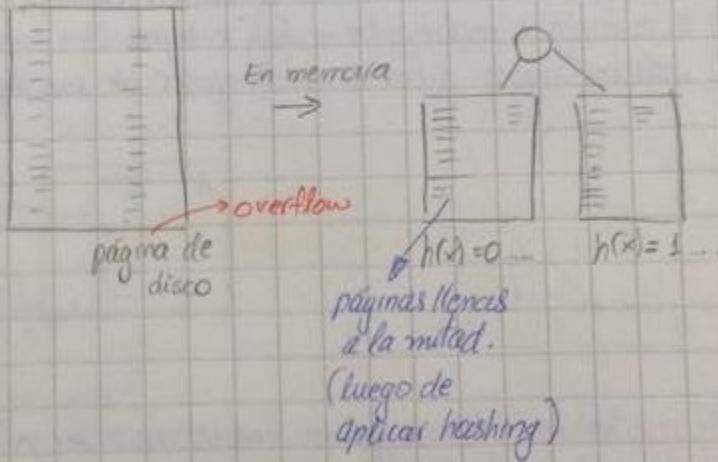
Si hay varios overflows seguidos, las páginas vacías las dejo NUL.

⇒ garantizado 50% ocupación

- Hashing extendible
- Hashing lineal



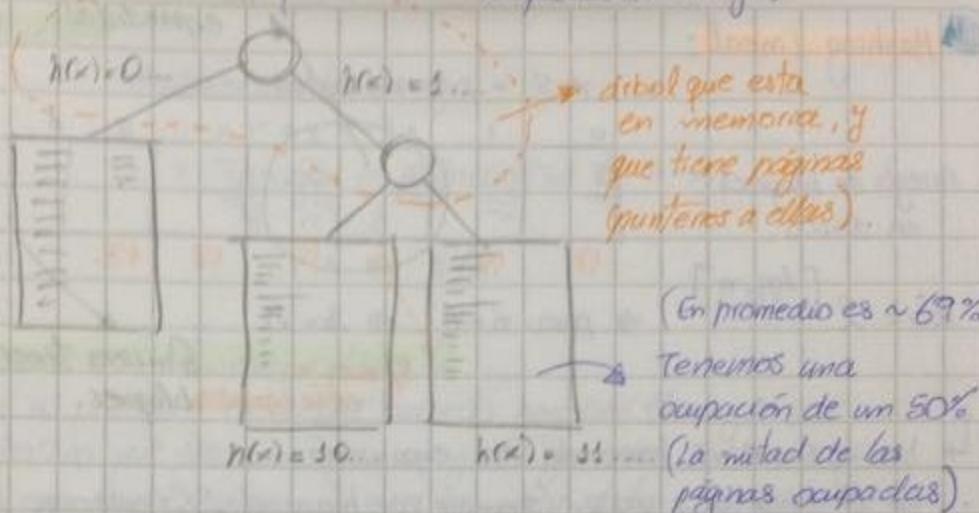
• Hashing extendible.



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)



→ dato que está en memoria, y que tiene páginas (punteros a ellas).

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

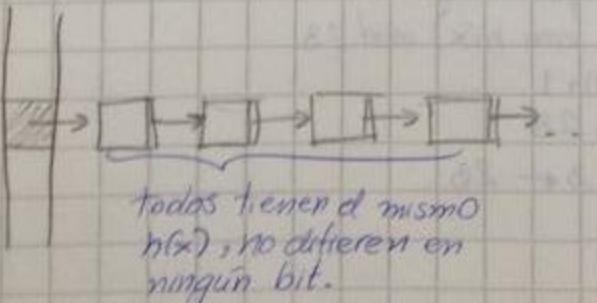
→ Tenemos una ocupación de un 50% (la mitad de las páginas ocupadas).

- 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)

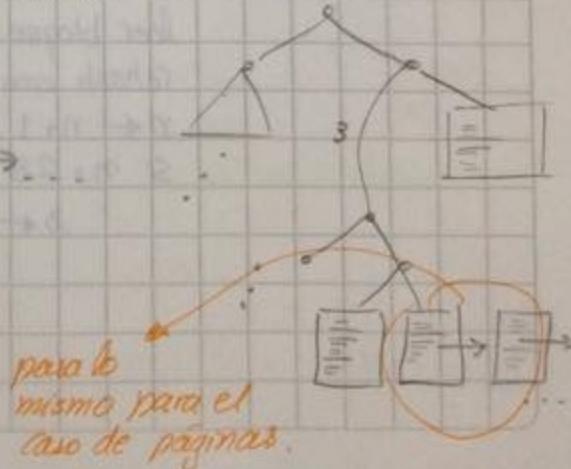
→ Sin embargo : 1) ¿Qué pasa si se aruba el hash?

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.

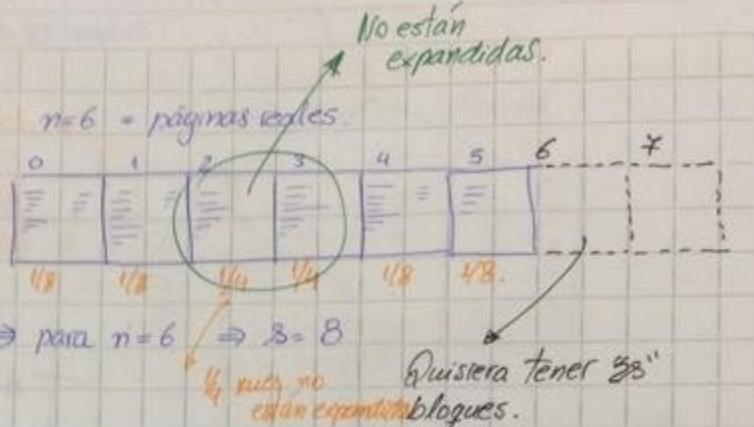


para lo mismo para el caso de páginas.

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 aún no expandida. El algoritmo es:

S: $h(x) \bmod s < n \bmod s$
 return $h(x) \bmod s$

return $h(x) \bmod s$

} supondremos $s \leq n < 2s$

• ¿Cómo expandir?

Expandir Tabla:

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

• ¿Cómo contraer?

Contraer - Tabla :

$$n \leftarrow n-1$$

agregar la página n a la $n-s$

si $n = 3$

$$s \leftarrow s/2$$

• ¿Cuándo expandir? ¿Cuándo contraer?

Notemos que no podemos expandir cuando hay overflow, ya que es algo reiterativo que trae consigo un gasto grande. Entonces las páginas con overflow se enlazan en una lista, esperando su turno. Cuando toque la expansión, se leerá toda la lista y se harán los cambios (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.

$$\square^* = \left(\frac{1}{1-\square} \right) \quad \sum_{i=0}^n \alpha^i = \frac{1-\alpha^{i+1}}{1-\alpha}$$

2015年10月8日 (木)

Análisis amortizado.

Promedio de una secuencia de operaciones \neq Promedio sobre inputs posibles

→ Análisis de costo de una secuencia de operaciones. Algunos casos se contrarrestan con otros. Se puede llegar a hablar de costos 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 , $n=2^k$

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

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

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

k bits	
0	000 ... 000
1	000 ... 001
2	000 ... 010
3	000 ... 011
4	000 ... 100
5	000 ... 101
6	000 ... 110
7	000 ... 111
8	1000 ... 0000 (K)

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

⇒ costo amortizado de incrementar es 2.

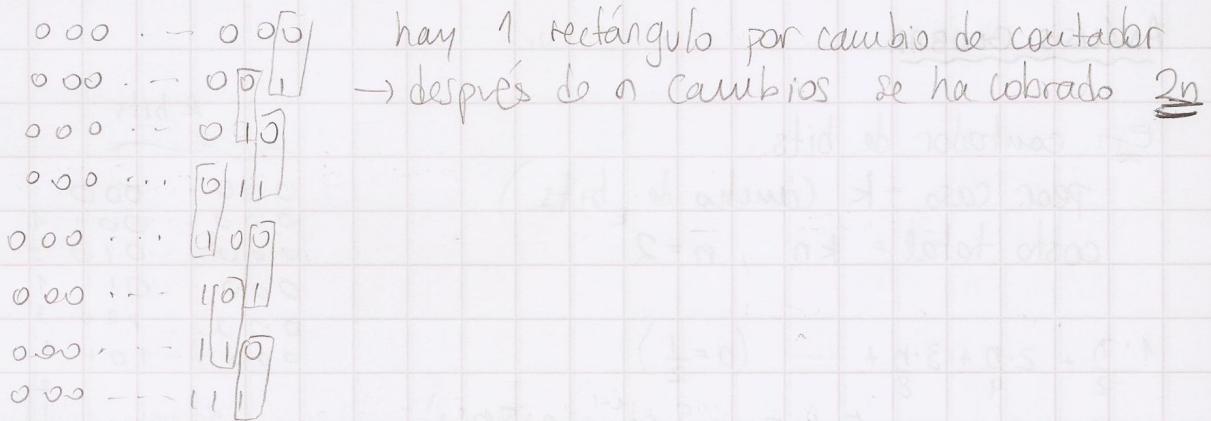
lo resolví esto
los otros

dos formas distintas

CONTABILIDAD DE COSTOS

Ej: Contador de bits de neros.

- 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.



→ 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

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

cota superior al costo real.



Depth

Ej: Contador de bits.

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

 . . . 0111

↓ +1

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

 . . . 1000

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

$-c_{i-1}$

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

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

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

Otro ejemplo DFS en un grafo para detectar componentes conexas.

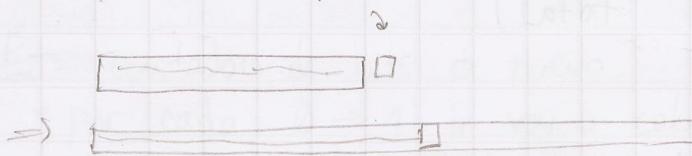


$$\sum_{v \in V} \text{arity}(v) = 2e$$

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

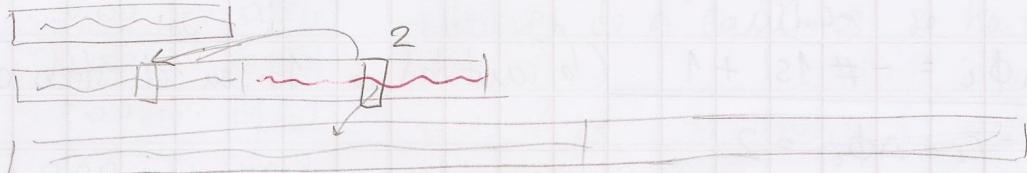
→ hemos usado una técnica de análisis amortizado para calcular costo global $\Theta(n+e)$

Ej: reallocar duplicando el tamaño.



• Contabilidad de costo.

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



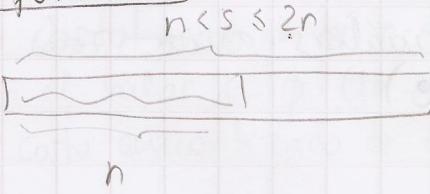
A los elementos que son insertados en la segunda mitad

le cobras 1 por insertar + 2 por su futura copia y por copiar el otro elemento simétrico en la primera mitad.

La gracia es que un alto rojo nunca va a volver a ser rojo de nuevo, y siempre alguien va a pagar por su copia posterior.

→ n ops cuesta $3n \rightarrow$ costo amortizado 3 por op.

• Función potencial



s: tamaño de lo que alcancé

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

$$c_i = 1$$

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

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

• La ruptura

Moneda rota

Cuando el anillo se realoca



$$c_i = n$$

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

=>

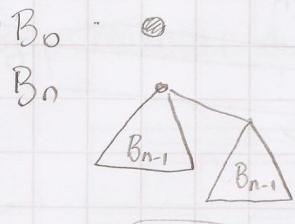
$$\hat{c}_i = \emptyset \quad \text{Pago con todos los ahorros.}$$

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

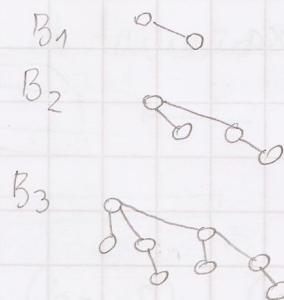
En el fondo ϕ está relacionado al espacio libre que queda.

2015年10月13日 (K)

Colas Binomiales



Árboles binomiales

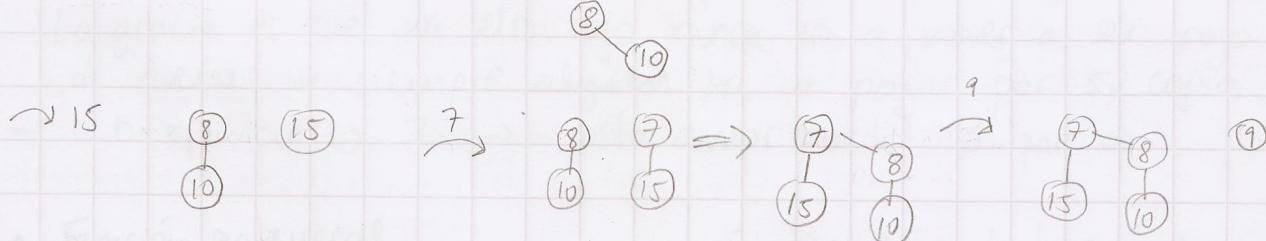


- B_n tiene 2^n nodos
- $h(B_n) = n$
- anchura de la raíz de B_n es n

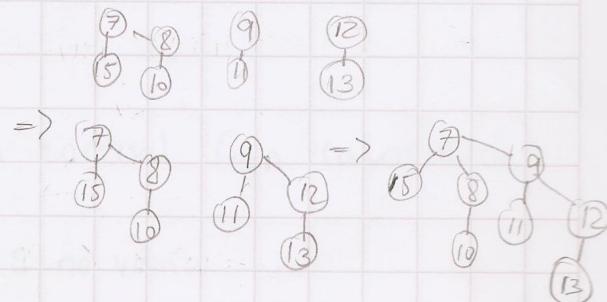
→ Un bosque binomial es un conjunto de árboles binomiales, todos de distinto tamaño.

→ 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.

→ 10 ⑩ → bosque binomial. No puede haber 2 árboles de = tamaño



↑ 11 ⑦ ⑧
↑ 12 ⑨ ⑪
↑ 13 ⑤ ⑯ ⑩ ⑪ ⑫ ⑬
La única forma es convertir todo a un B_3



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 $\Theta(\log n)$

bien también es peor caso

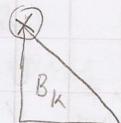
OBS: podemos hacer heapfy mediante insertar n elementos, a costo $\Theta(n)$
(producir esta estructura)

(pensar en el costo amortizado de cada operación de contador binario)

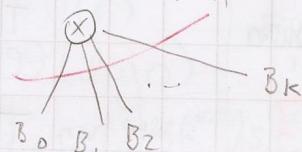
• Encontrar min: Buscar mínimo entre las raíces

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

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

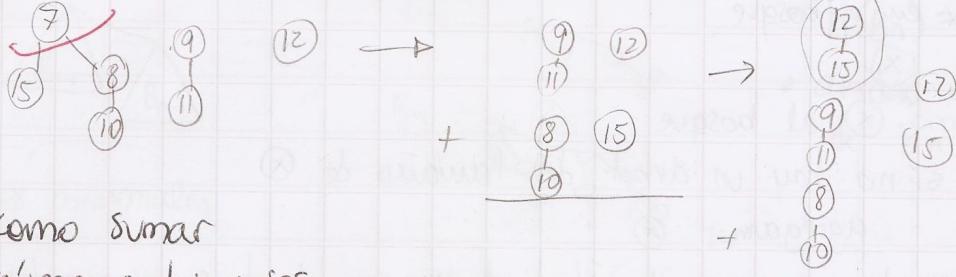


- Eliminar x y quedarse con los k subárboles



Colas Binomiales

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



es como sumar
dos números binarios

$\Theta(\log n)$

(o también
podía dejar
otra pareja
a fieras)

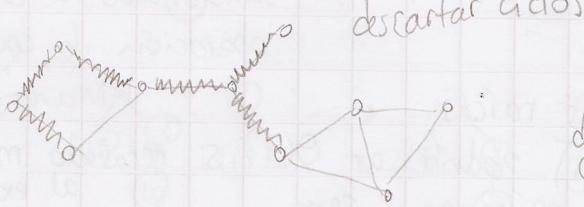
- Merge (T_1, T_2)

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

Union Find

Ej: MST (Minimum Spanning Tree)

Kruskal



Conjunto
de comp
conexas

todavía
no tenemos
árbol que
conecte
el grafo

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

Kruskal (V, E) dr a costo
de arista

heapify (E)

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

$T \leftarrow \emptyset$

\rightarrow while $|E| > 1$

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

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

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

si son \neq
comp conexas
agrega la
arista al grafo

\rightarrow if $U \neq V$

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

$E \leftarrow E - \{U, V\} + \{U \cup V\}$

return T



MST se puede ver de manera más general.

Problema: mantener un conjunto de clases de equivalencia sobre n elementos

- inicialmente, cada elemento forma una clase separada.
- en cada clase, elegiremos un representante.
- Find(x) me da el representante de la clase a la que x pertenece.
- Union(x,y) une las clases representadas por x e y.

Kruskal (V,E)

heapify(E)

C ← classes(V)

T ← \emptyset

while |E| > 1

(u,v) ← extractMin(E)

x ← Find(u)

y ← Find(v)

if x ≠ y

T ← T ∪ {(u,v)}

Union(x,y)

return T

$\Theta(\log n)$
oristas →

Lema: todo árbol de altura h tiene al menos 2^h nodos

Dem: Caso base, h=0 y n=1 ✓

Caso inductivo: Unimos T_1 y T_2

$n_1 \geq n_2$.

∴ Colgamos T_2 de T_1 .

T =



$$h(T) = \max(h(T_1), 1 + h(T_2))$$

2 casos: a) $h(T) = h(T_1)$

$$h = h_1 + n_2 \geq h_1 \geq 2^{h(T_1)} = 2^{h(T)}$$

b) $h(T) = 1 + h(T_2)$

$$n = n_1 + n_2 \geq 2n_2 \geq 2 \cdot 2^{h(T_2)} = 2^{1+h(T_2)} = 2^{h(T)}$$

Se puede ver como árboles

cuyos nodos apuntan a

su padre → representante

Conclujo: Al haber n nodos, todo árbol tiene altura a lo más $\lceil \log n \rceil$

Union: $\Theta(1)$

Find: $\Theta(\log n)$

m finds → $\Theta(m \log^+ n)$ amortizado

$$\log^*(65536) = 1 + \log^*(16) = 2 + \log^*(4) = 3 + \log^*(2) = 4$$

Find(u)

if u.padre = null

return u

i.e. $u \leftarrow \text{Find}(u.\text{padre})$

return u.padre.

va aprendiendo
modifica la estructura
al leer.

Evaluación del costo de la vida de los trabajadores
en el sector público en Argentina. La inflación
y el costo de vida en Argentina se han
incrementado drásticamente en los últimos años.
Este informe evalúa el costo de vida de los trabajadores
en el sector público en Argentina, considerando tanto la inflación
como el costo de vida en general. Se analizan las tendencias
de precios y se comparan con los niveles de vida y los salarios
de los trabajadores en Argentina. Se evalúan las estrategias
que se están implementando para abordar el problema
y se recomiendan cambios para mejorar la situación.

Costo de vida en Argentina

El costo de vida en Argentina es un tema que ha sido objeto de mucha atención en los últimos años. La inflación ha sido un factor clave en el aumento del costo de vida, lo que ha llevado a una disminución en el poder adquisitivo de los trabajadores. Los precios de los bienes y servicios básicos como la comida, el agua y el gas han aumentado significativamente, lo que ha puesto en evidencia la vulnerabilidad de los trabajadores al impacto de la inflación.

El costo de vida en Argentina es particularmente alto en comparación con otros países de la región. Los precios de los bienes y servicios básicos son más altos que en otros países, lo que ha llevado a una disminución en el poder adquisitivo de los trabajadores. Los precios de los bienes y servicios básicos como la comida, el agua y el gas han aumentado significativamente, lo que ha puesto en evidencia la vulnerabilidad de los trabajadores al impacto de la inflación.

El costo de vida en Argentina es particularmente alto en comparación con otros países de la región. Los precios de los bienes y servicios básicos son más altos que en otros países, lo que ha llevado a una disminución en el poder adquisitivo de los trabajadores. Los precios de los bienes y servicios básicos como la comida, el agua y el gas han aumentado significativamente, lo que ha puesto en evidencia la vulnerabilidad de los trabajadores al impacto de la inflación.

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

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro Auxiliar: Jorge Bahamonde

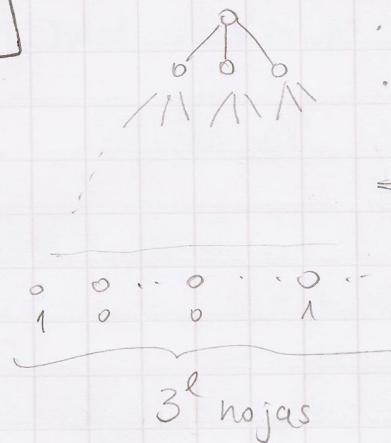
19 de Octubre del 2015

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
 †'s pero ambas
- Inicializo todas las hojas de A_0 con 0's
 " " " " " A_1 con 1's
 Son VÁLIDAS

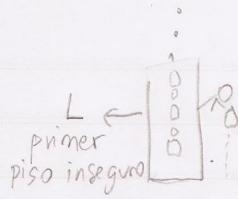
• Ante cada pregunta

- 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 itera desde el primer piso:

Se deja caer.

→ si se rompe, ese piso era L

→ si no, subo 1 piso

$\Theta(L)$

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

Adversario:

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

Si no, no.

$\Omega(L)$

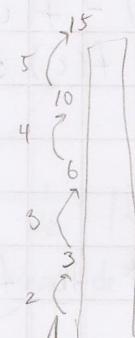
Si $f(i) > i$, el algoritmo no puede distinguir si $L = f(i)$ o $L = f(i) - 1$
 \Rightarrow el algoritmo no puede ser correcto,

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, $L = \frac{k(k+1)}{2}$



$$\binom{t}{2} + t = \binom{t+1}{2}$$

FLUX # 5

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

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

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

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

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

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

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

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

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

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 $f(0) := \emptyset$)

Dividimos el análisis en 2 casos:

• $f(i) - f(i-1) \leq i + i$

$\Rightarrow f(i) \leq \underbrace{i(i+1)}_{\sum_{j=0}^{2} j} + i$ (Veo si $f(i) \geq L$

$i = \sqrt{2}(\sqrt{L})$

$$\sum_{j=0}^{2} j$$

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

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

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

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

$$= i^*(i^*-1) + t$$

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

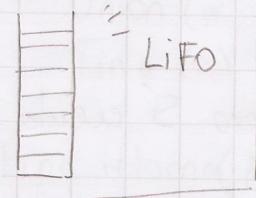
\Rightarrow el 2º huevo se bota $\Omega(\sqrt{L})$ veces.

Non-Block

2023/09/20 10:42

P3

Stacks en memoria secundaria.



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.

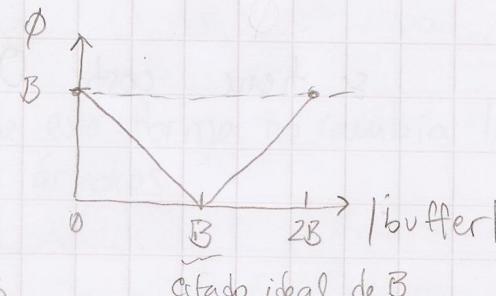
- Si se llena, escribo solo B
- Si se vacía, pido solo B .

ANÁLISIS: fn. potencial.

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

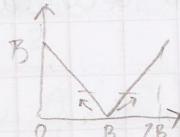
$$\Phi : \begin{cases} |buffer| - B & \text{si } |buffer| \geq B \\ B - |buffer| & \text{si } |buffer| < B \end{cases}$$

$$\Rightarrow \Phi = |buffer| - B$$



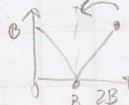
Costos

Push y Pop sin tocar disco
 $\Rightarrow \pm 1$ (operaciones en memoria no cuentan, pero sí cuenta el cambio en la fn potencial, $\Delta\phi$)



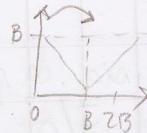
Push con overflow

$$\text{Costo: } B + \Delta\phi = B + -B = 0$$



Pop con underflow

$$\text{Costo: } B + \Delta\phi = B + -B = 0$$

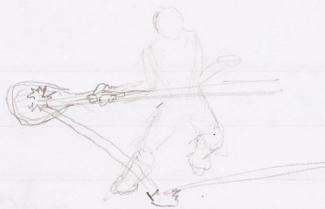


\Rightarrow Costo amortizado constante. Cuesta B mover un bloque
debería ser

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

se tiene costo $O(\frac{1}{B})$ por op.

Union Find



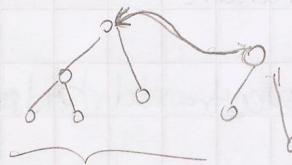
2015年10月20日 (X)

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

Union(x, y) → une los conjuntos de x e y

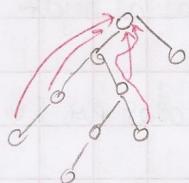
Find(x) → retorna "label" del conjunto de x .

- Find retorna la raíz
- Union une 2 árboles



→ 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 ⇒ $\text{rank}(x) \geq h(x)$. (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 .

①, 2, 4 y 5 son triviales

Le asigna un padre con rank mayor
no es que le cambie el rank al nodo
padre

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

3: P.C. sólo 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 fueron 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)

"Definamos" $\lg^* n$ como "cuántas veces debo aplicar \lg para llegar a un $n^o \leq 1$

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

Consideremos los ranks de la estructura y hagamos grupos
 $(\lg^{*-1}(i), \lg^{*-1}(i+1)]$

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

• Contabilidad

- Cuando un nodo deja de ser raíz, le asignamos dulces :)
- 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)

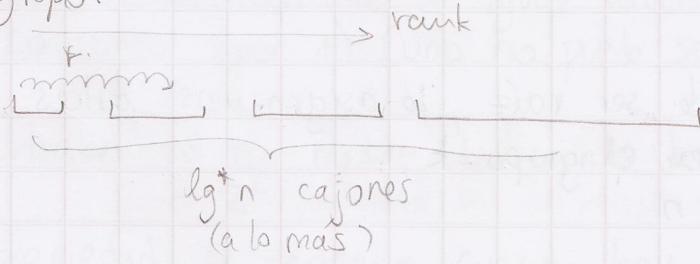
1: $\text{rank}(\text{padre}(x))$ está en un grupo \neq al de $\text{rank}(x)$

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 $\lg^* n \cdot \text{cajones}$.

• 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 $\ll 2^{65,536}$

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.

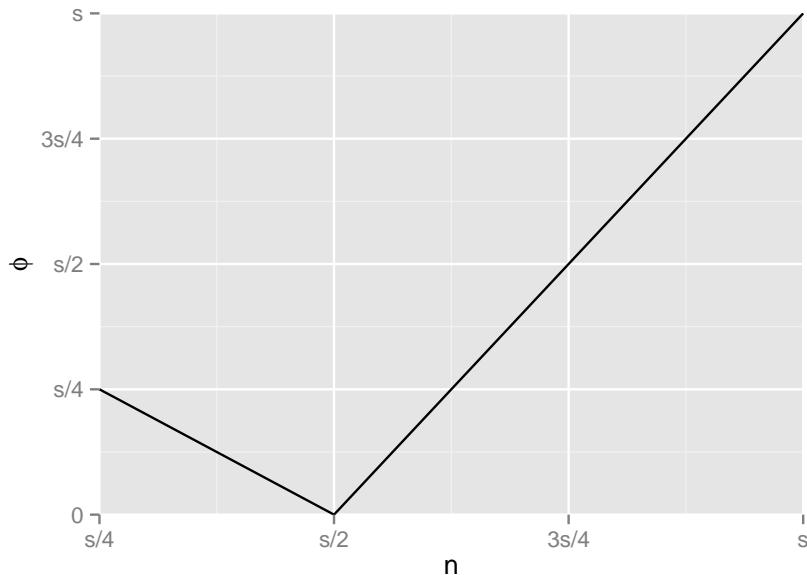
Así, cumplimos con todo lo que se nos pedía :)

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 .

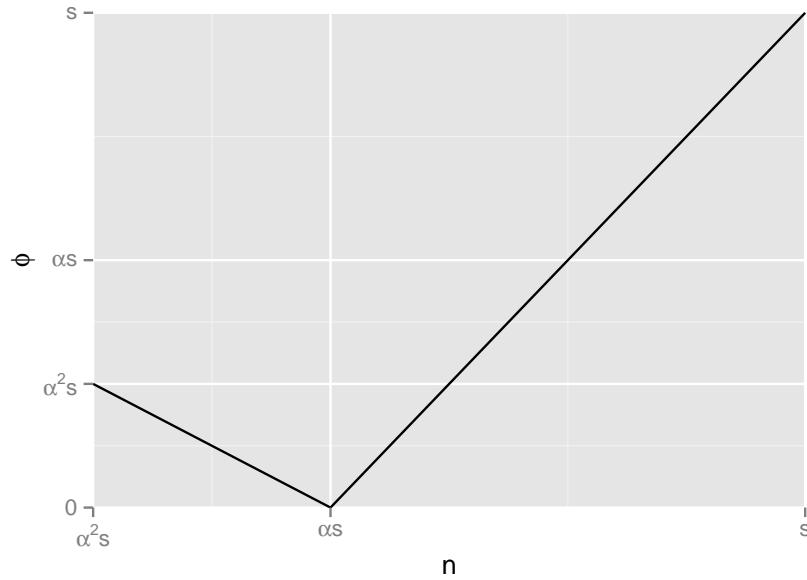
- Una inserción que no hace crecer la tabla tiene costo 1. El potencial puede cambiar en +2 o en -1, dependiendo de en qué tramo de la función potencial estemos, por lo que el costo amortizado puede ser 0 o 3, siendo constante en ambos casos.
- Un borrado que no hace crecer la tabla tiene costo 1. El 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

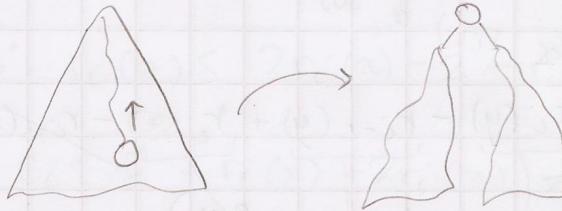
o decremento *debe* terminar. Luego al incrementar o decrementar pagamos 2 por el cambio del trit 0 a otro valor, que ocurre (a lo más) sólo una vez; los demás cambios en los trits son pagados por las monedas antes ahorradas.

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

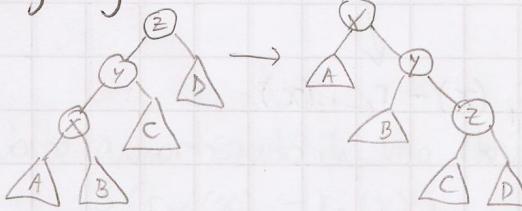
2015年10月27日 (K)

- 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 éltro quede en la raíz.
- Una secuencia de n operaciones tiene costo amortizado de $\Theta(\log n)$ amortizado, si todos los éltros tienen prob uniforme $1/n$.
- El costo es $\Theta(H)$ entropía de Huffman.

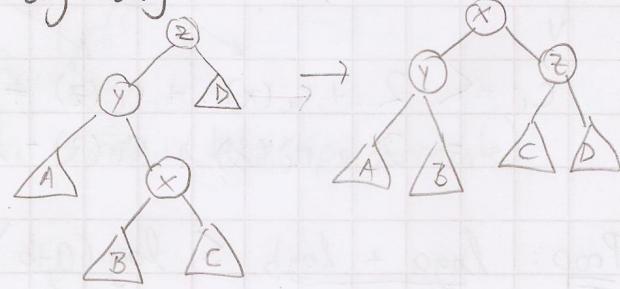


Operaciones de rotación (recordar AVL)

zig-zig



zig-zag

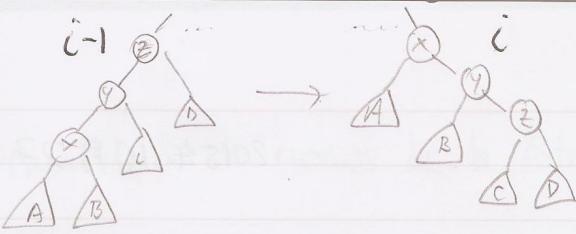


• Zag-zig

• Zag-zag

• zig } a dist 1 de
• tag } la raíz

Véamos costo de zig-zig ...



Fuimos a buscar x.

ESTA YAHG

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

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

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

En un zig-zig

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

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

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

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

• $r_{i-1}(y) > r_{i-1}(x)$

• $r_i(y) < r_i(x)$

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

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

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

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

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

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

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

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

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

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

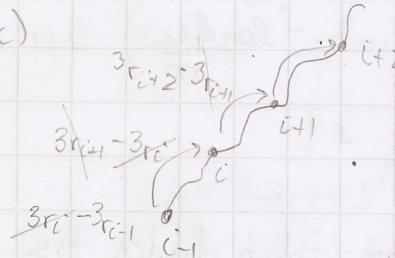
$$\Rightarrow r_{i-1}(x) + r_i(z) \leq 2 \log \frac{S_i(x)}{2} = 2r_i(x) - 2$$

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

$$\hat{c}_i \leq 2 + r_i(x) + 2r_i(x) - 2 - r_{i-1}(x) - 2r_{i-1}(x)$$

$$\hat{c}_i \leq 3(r_i(x) - r_{i-1}(x))$$

huele a 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 $q(x)$ veces (de las m) entonces el costo amortizado es $\Theta\left(\sum_{x \in T} \frac{q(x)}{m} \log \frac{m}{q(x)}\right)$

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

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

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

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

Si promediáramos sobre todos los x , con su probabilidad

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

$$1 + 3H$$

2015年10月29日 (木)

UNIVERSOS DISCRETOS

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

• Counting Sort

- las claves están en $[1 \dots U]$
- n claves no hay registros asociados
- 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

$C: 1 \quad \square$

 2 \square

 3 \square

 4

Funciona porque los números iguales son indistinguibles

Los elementos SON las claves

Complejidad $\Theta(n + U)$

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

ll

DATA STRUCTURES

2021-2022 2022-2023

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'

$$A = \underline{3} \underline{2} \underline{1} \underline{1} \underline{5} \underline{2} \underline{1} \underline{3} \underline{1} \underline{2} \underline{2}$$

$P = 2 3 4 5$

C:	1	□	pos 1	
	2	□	pos 8	6 7 8 9
	3	L	pos 9	10 11
	4		pos 11	
	5	I	pos 11	12

$$A' = \underline{\underline{1}} \underline{\underline{1}} \underline{\underline{1}} \underline{\underline{2}} \underline{\underline{2}} \underline{\underline{2}} \underline{\underline{2}} \underline{\underline{3}} \underline{\underline{3}} \underline{\underline{5}}$$

for $i \leftarrow 1$ to U

$C[i] \leftarrow 0$

for $j \leftarrow 1$ to n

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

$P[i] \leftarrow 1$

for $i \leftarrow 2$ to U

$P[i] \leftarrow P[i-1] + C[i-1]$

for $j \leftarrow 1$ to n

$A'[P[A[j]]] \leftarrow A[j]$

$P[A[j]] \leftarrow P[A[j]] + 1$

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

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 usando 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
17	1 0 0 0 1
5	0 0 0 0 1

Ahora por el segundo

6	0 0 1 1 0
8	0 1 0 0 0
17	1 0 0 0 1
5	0 0 0 0 1

Ahora estan ordenados por los 2 últimos bits

8	0 1 0 0 0
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

ALMISMO TIEMPO PUES bucket sort es ESTABLE

→ 8 0 1 0 0 0

17	1 0 0 0 1
2	0 0 0 1 0
5	0 0 1 0 1
14	0 1 1 1 0
6	0 0 1 1 0

→ 17 1 0 0 0 1

2	0 0 0 1 0
5	0 0 1 0 1
6	0 0 1 1 0
8	0 1 0 0 0
14	0 1 1 1 0

→ 2 0 0 0 1 0
5 0 0 1 0 1
6 0 0 1 1 0
8 0 1 0 0 0
14 0 1 1 1 0
17 1 0 0 0 1

cuántos bits escribir se ocupan para los elementos

$\Theta(n \log U)$

→ por qué no mejor elegir chunks más grandes que 1?

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

⇒ $\Theta(n \log U) = \Theta(n \log_n U)$

OTJ cuando U es muy grande

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

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

esa 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) = $\Theta(n)$ + estable = Bucket Sort / Counting Sort . (clases)

$\boxed{\quad}$ \rightarrow tomar arreglo del tamaño del universo.

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

espacio: $\Theta(n+U)$ \rightarrow arreglo para contar $\boxed{\quad}$,

\rightarrow arreglo del tamaño de la entrada (copia ordenada)

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

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

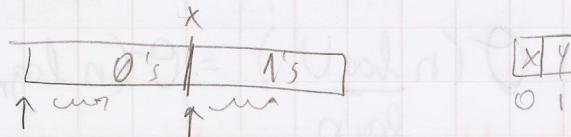
(al parecer hay un mergesort in-place que tiene tiempo estable $\Theta(n \log^2 n)$)

(Quicksort, es estable?)

a) + c) { Hago una pasada por el arreglo contando cuántos 0's y 1's hay: n_0 y n_1 ($n_0 + n_1 = n$)

espacio $\Theta(1)$ { Inicializo 2 punteros en 0 y n_0

tiempo $\Theta(n)$ { Ambos punteros avanzan hasta que el primero encuentra un 1 y el segundo un 0
los intercambian / swap



P2

n números
 $U = n^2$

Intentemos usar Radix Sort.

- 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 $\Theta(\log n \cdot (n + 2)) = \Theta(n \log n) \Rightarrow \tilde{\Theta}(n)$

* Usamos base n

\Rightarrow cada número tendrá 2 dígitos

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

\uparrow \uparrow
1 bucketsort buckets
por dígito elems
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 $\Theta(k(n + n)) = \Theta(kn)$

P3

si z es el límite teórico del número de bits necesarios para almacenar la información de una estructura, la estructura es sencilla si usa $\geq + \Theta(z)$

↳ Θ chico!

($\Theta(z) \Rightarrow$ compacta; $z + \Theta(1) \Rightarrow$ implícita)

se usan
harto
común
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 \times $((4)_1 = 1111)$

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

Con un 0 al final.

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

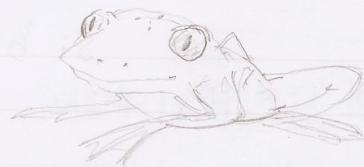
↳ tamaño arreglo

\Rightarrow Sum(r) = Rank₁ (Select₀ (r))

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

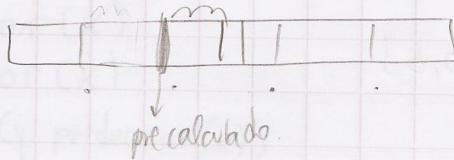
• Search(s) = Rank₀ (Select₁ (s)) + 1

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

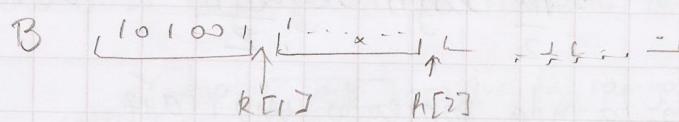


ÁRBOLES EN ERRORES

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 pregunten 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 $\frac{n}{b}$ elementos.

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

$\underbrace{\text{elems}}_{\text{bits}}$ $\lceil \log n \rceil$ 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

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

$$T[1011, 1] = 1$$

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

- Espacio de T : $2^b \cdot b \cdot \log b = 2^{\frac{\log n}{2}} \cdot \frac{\log n}{2} \cdot \log \frac{\log n}{2}$ o chica
 $= \frac{\sqrt{n}}{2} \log n \cdot \log \frac{\log n}{2} \in o(n)$

$$b = \frac{\log n}{2}$$

(2) → ese 2 nos salvó la vida poniendo una raíz.

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

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

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
 de tamaño $S = (\frac{\log n}{2})^2$

• En $S[1, \lceil \frac{n}{S} \rceil]$ guardaremos lo que antes iba en R , pero usando S en vez de b .

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

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

$|T| \in o(n)$ 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: $\frac{\log n}{\log n} = 1$

• $|T| = \frac{1}{b} \cdot \log((\frac{\log n}{2})^2)$ bits = $2 \log \log n \cdot \frac{2n}{\log n} = 4n \cdot \frac{(\log \log n)}{\log n} \in o(n)$

2015年11月3日 (K)

ÁRBOLES VAN EMDE BOAS

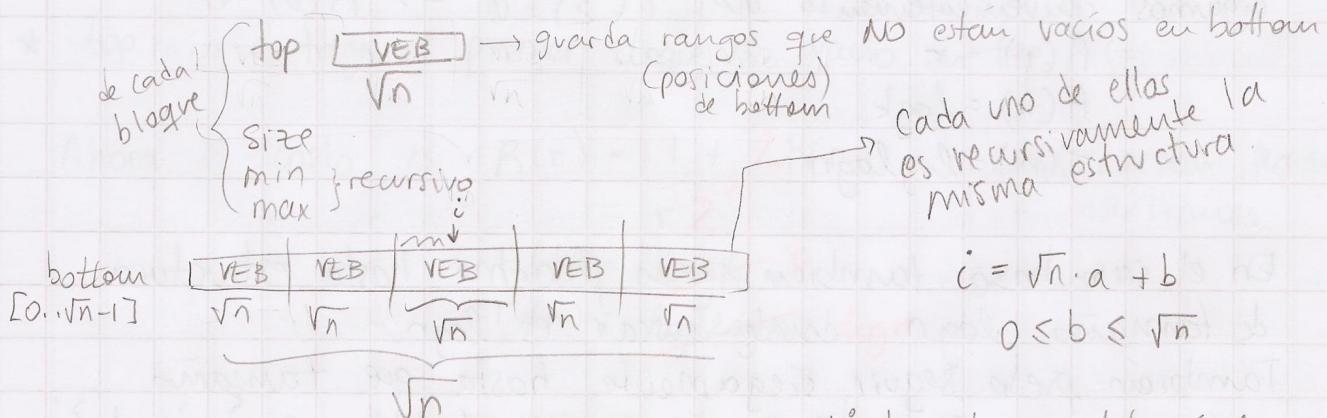
- insertar(x)
- borrar(x)
- sucesor(x)
(y predecesor(x))

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

Universo $[1..n]$

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

Lineal en el tamaño del UNIVERSO.



¿Cómo buscamos?

- $\text{succ}(i)$ devuelve el primer entero mayor igual que i en

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

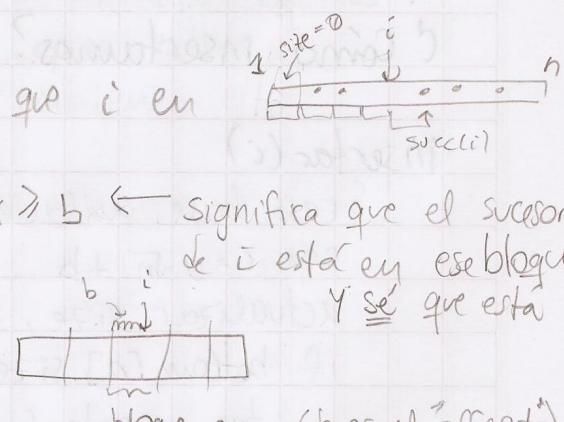
if $\text{bottom}[a].\text{size} > 0 \& \text{bottom}[a].\text{max} \geq b$ ← Significa que el sucesor
return $a\sqrt{n} + \text{bottom}[a].\text{succ}(b)$

$c \leftarrow \text{top}, \text{succ}(a+1)$ } encontrar el primer bloque No vacío
return $c \cdot \sqrt{n} + c.\text{min}$

podría haber un caso

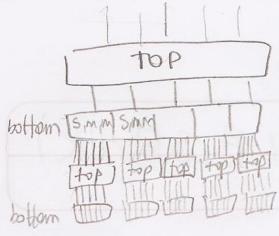
base donde la estructura es

un árbol binario o algo, vs búsqueda binaria (en una hoja de tamaño $\log n$, luego buscar en $\log \log n$ también)



conveniente

log n, luego buscar en log log n también)



size, min, max

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

$n \rightarrow \infty$

¿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 $T(2) = 0 \Rightarrow R(0) = 0$

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

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

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

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

insertar(i)

{caso base, cualquier

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

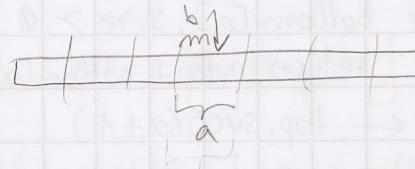
actualizar size, min, max

if 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 top.min: \sqrt{n} + bottom[top,min].min

* top.min entrega el primer bloque no vacío en top.

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

$$= r2^r$$

Cada inserción produce

2 inserciones

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

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

¿Solución? Cambiar invariante de la estructura

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

Ahora en SVCC(i) se agrega linea al comienzo

SVCC(i)

if i < min return min

:

insertar(i)

{ if size = 0 : min ← i , max ← i , size = 1
else if i < min : i ← min
Sea $i = a\sqrt{n} + b$
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, $\lceil \frac{n}{2} \rceil + \text{bottom}[\text{top}.min].min$
 i ← min
Sea $i = a\sqrt{n} + b$
bottom[a].borrar(b)
if bottom[a].size = 0
 top.borrar(a)
actualizar size, max

Espacio

$$\begin{aligned} T(n) &= 1 + (\lceil \sqrt{n} \rceil + 1) T(\lceil \sqrt{n} \rceil) \\ &\left(\begin{array}{l} T(n) \leq 2n - 3 \\ \vdots \\ T(n) \leq 1 + (\lceil \sqrt{n} \rceil + 1)(2\lceil \sqrt{n} \rceil - 3) \\ = 1 + 2n - \lceil \sqrt{n} \rceil - 3 \\ \leq 2n - 3 \end{array} \right) \end{aligned}$$

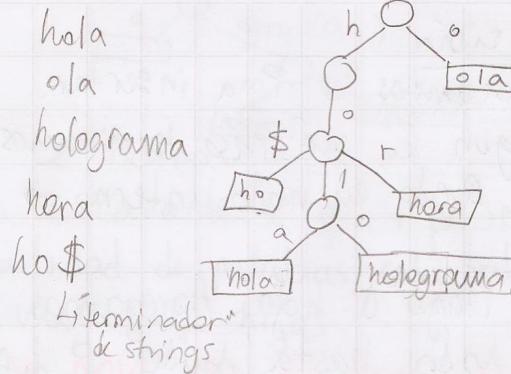
2015年11月5日 (木)

TRIES (árboles digitales)

Conjunto de n strings
largo total L .

- espacio $\Theta(L)$
- tiempo de búsqueda de string de largo m
- $\Theta(m)$
- $\Theta(m \log \sigma)$

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?
 $\hookrightarrow \Theta(m) \circ \Theta(m \log \sigma)$

- 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 (puedo guardar el num de hijos en cada nodo, o recorrerlo entero)

鳥 翅 ポリヨ

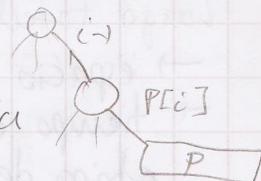
フライ
Fu ra i

• Inserción

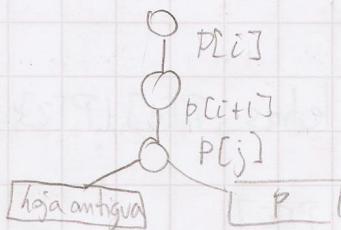
Buscamos el P a insertar.

Según en qué caso terminemos.

1) Llego 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 "mín"
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,

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

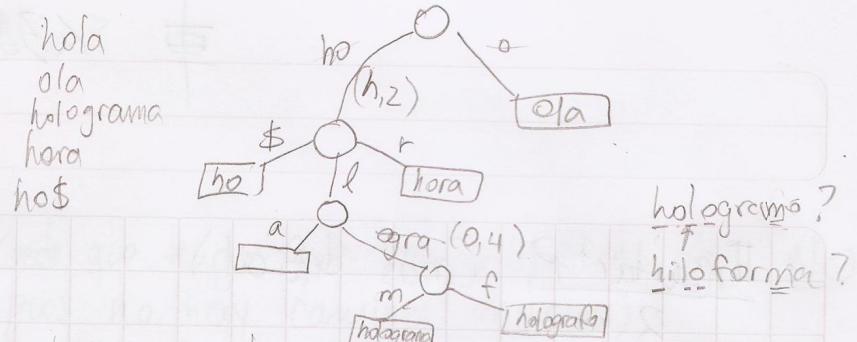
$\text{padre}(\text{hoja}) \leftarrow u$

borrar V

$V \leftarrow u$

} me gustaría hacer esto de una

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 extremo. 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. blind

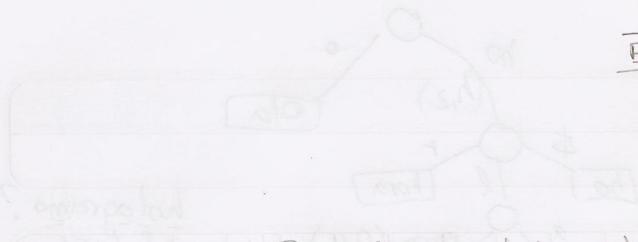
Buscar:

- 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 くろ 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 (algunas)
y encastrar 1º diferencia

caso b) buscar la 1º diferencia con esa hoja

- 2) colgar la nueva hoja a la profundidad correcta

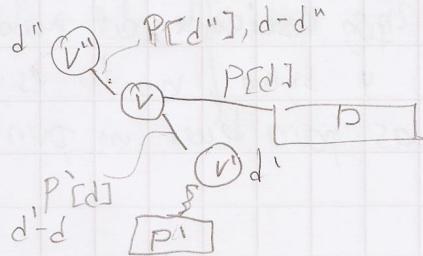
Volver a la raíz y bajar nuevamente hasta llegar a la profundidad buscada. Sea d' la profundidad deseada.

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

La distancia entre nodo y hoja no se pone

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

→ corto la anista con un nuevo nodo v.



(ojo que puede que haya que arreglar esto por error de ±1)

en d, d', d''

馬

ハタリシヤ

APR 48

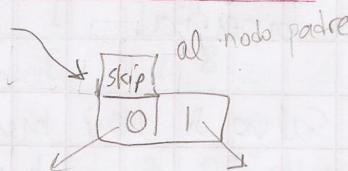
Borrar (más fácil). Puedo que tenga que borrar padre, pero NO abuelo(s), pues no hay caminos varios)

- 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 $\Sigma=2$, las cadenas son secuencias de bits.



⇒ Caminos de a lo más largo.

a seguir para Normalmente 8 veces si son 256 caract. es harto malo.

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

ÁRBOL DE SUFIJOS

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

(Suffix tree) (Weiner, 1973)

- 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 de un sufijo

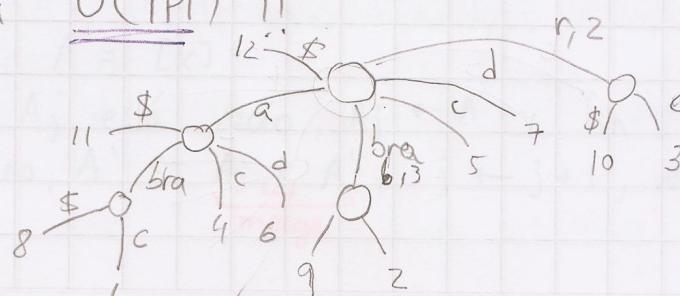
todos los substrings = $P \equiv$ todos los sufijos que empiezan con P .

→ 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 $\Theta(|P|)$

y no en orden del tamaño del texto



espacio $\Theta(n)$

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

Cadena más larga?
que se repite?
buscar nodo interno
más profundo

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

Arreglo de sufijos (suffix array) Salvo las hojas son suficientes!

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

buscar: $O(m \log n)$

→ 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* 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

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

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

• INSERCIÓN:

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

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, $A' \leftarrow \underline{\underline{A_j + A'}}$; $j \leftarrow j+1$, volver a a)

merge

8 # XVA

En este caso, el "+" es una op. de merge
Peor caso: hago merge de todos los arreglos.

$\Rightarrow k$ merges. Cada merge es de 2^i elemos

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

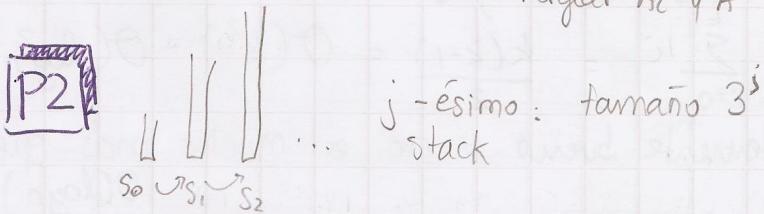
Mejor hagamos análisis amortizado.

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

Notemos que A_0 participa en un merge cada 2 inserciones.

$A_1 \dots$ 4 inserciones
 $A_r \dots$ 2^{r+1} inserciones

$$\Rightarrow \text{Costo total de merges: } \sum_{i=0}^{k-1} 2^{i+1} \left[\underbrace{\frac{m_{i+1}}{2}}_{\substack{\text{Cuánto cuesta} \\ \text{mergear } A_i \text{ y } A'}} \right] \leq k \cdot m = m \cdot \Theta(k) \Rightarrow m \cdot \underbrace{\Theta(\log n)}_{\substack{\text{en cuántos} \\ \text{merges participa } A_i \\ \text{por op}}}$$



• Peor caso: mover todo $\Theta(n)$

• n ops en m m.s. vacío

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

$\underbrace{n_j}_{\substack{\text{n de eltos en } S_j}}$

Vamos a calcular el costo de vaciar el stack S_i en el S_{i+1}

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

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

$$i > 0 \wedge j > 0 \quad ?$$

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

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

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

solo considero los éltos 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

$$\hat{C}_{total} = C_{push_{S0}} + \cancel{C_{vaciar stacks}} + \Delta\phi_{push_{S0}} + \cancel{\Delta\phi_{vaciar stacks}}$$

$$= 1 + (\phi_f - \phi_i)$$

$$= 1 + 2 \log_3 n = \mathcal{O}(\log n)$$

$$\Delta\phi_{push_{S0}}: \phi_f = 2 \cdot \log_3 n$$

$$\phi_i = 0$$

$$= 2 \log_3 n$$

y para n pushes $\rightarrow \mathcal{O}(n \log n)$

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?

$$\bullet i = 0$$

• Se hace pop de S_i

• Si se vacía, se llena con éltos 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 vacio) quedan en $1/3$ (justo) y S_i queda al menos $1/3$ lleno.

$$\begin{array}{c} S_0 \rightarrow S_1 \\ \downarrow S_1 \rightarrow S_2 \\ \downarrow S_2 \rightarrow S_3 \end{array}$$

Para un pop relevante:

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

$$S_i \text{ al menos con } 1/3 \quad S_i \text{ a lo más } 2/3$$

La primera op. relevante debe ser un push.

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

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

Abreviamos Select Next = SN

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

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

Si no hay next
 $SN = n+1 - 1$

algo fuera de rango.

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

(Nx son tablas precalculadas)

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

↳ selectNext del primer elto del superbloque.

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

Para cada bloque de un superbloque,

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

Espacio: $\Theta(\frac{n \log \log n}{\log n})$ ~ 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 $S = \emptyset$ (solo 0's), $N_p[S, i] = b+1$)

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

↓ el más chico primero

Se sale de rango.

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

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 un cierto juego, pero no sé cuántos días jugaré antes de aburrirme

- Una tienda lo arrienda con costo de $\$A/\text{día}$.
- 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

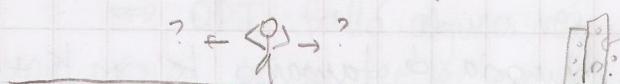
$$C(n) = \max_{\{I \in I\}} \frac{A(I)}{OPT(I)}$$

\leftarrow 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 $\frac{A}{B} = \frac{B}{A}$

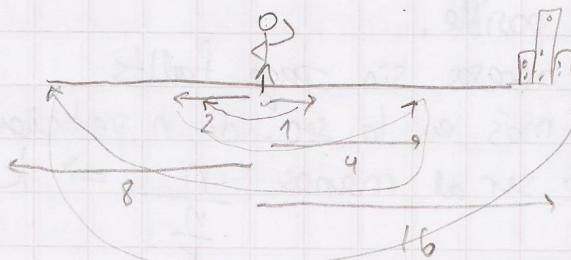
$\Rightarrow C_n(I) \leq 2 \Rightarrow$ la estrategia es 2-competitiva

- Otro problema Imagine despertar en una larguísima carretera

 \Rightarrow quiero minimizar los pasos dados.

\rightarrow No sabemos en qué dirección está la ciudad \Rightarrow 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 recorrida 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.

$$< 2 \cdot (2^{\lfloor \log_2 n \rfloor + 2}) + n$$

$$\leq 2 \cdot (4n) + n = 9n \Rightarrow A(I) \leq 9n$$

$$\Rightarrow C(n) \leq 9$$

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 comparamos 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 tendrá al menos K accesos sin page faults.

\Rightarrow $\frac{n}{K}$ page faults a lo más en la sec. de n peticiones

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

¿Qué algoritmos hay?

• LRU: last recently used → k-competitivo

• FIFO

→ k-competitivo

• LFU last freq. used → NO es k-competitivo
:

LRU

Dividamos 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 todas las demás fueron consultadas \Rightarrow OPT falló alguna vez.

• Si una página p' se repite en los pf's

\Rightarrow entre los 2 pf's deben haberse pedido todos.

\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 c_i

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

Aplicación: compresión

codificad \rightarrow símbolos $s_1 \dots s_N$ (ordenados en tamaño)
elementos $s_1 \dots s_N$

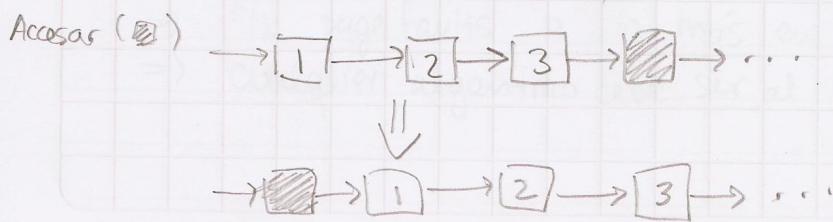
Para comprimir s : lo busco en la lista. Si está en la posición i , uso s_i
 \Rightarrow Si el costo de los accesos se minimiza voy a minimizar el
largo del texto comprimido. La idea es tener una estrategia que
logre hacer que las búsquedas resulten en el rango más cercano,
más al comienzo de la lista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

Al buscar pedo 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
 - x está después de y en B
- o viceversa

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}

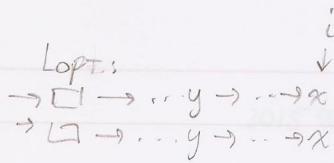
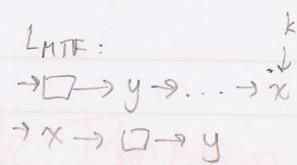
$k =$ posición de x en L_{MTF}

Consideraremos a parte los swaps que OPT puede hacer.

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

$$C_{MTF} = 2k - 1$$

- 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.

\rightarrow A MTF le cuesta 0 (cr a este swap, MTF no está haciendo nada)

\rightarrow A OPT le cuesta 1

Un swap de OPT crea o destory 1 inversión

$$\Rightarrow \Delta\phi = \pm 2 \quad (\phi = 2 \cdot \# \text{ inversions})$$

$$\Rightarrow C_{MTF} \leq C_{MTF} + \Delta\phi \leq 2 = 2 \cdot 1$$

$$= 2 \cdot C_{OPT}$$

$$\leq 4 \cdot C_{OPT}$$

\Rightarrow MTF es 4 competitivo

Analicemos por partes

{ 1) OPT retorna elto

el MTF retorna elto

{ 3) MTF hace swaps

4) OPT hace swaps

Los alg.
no dependen
del otro

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

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.

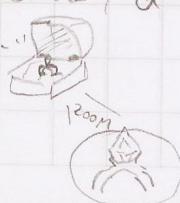
El online conoce n .

Cuál quiero optimizar?

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

Estrategia:

- Permuto aleatoriamente los participantes.
- Calibraremos usando las primeras t citas
 $(\text{elegimos "Adiós para siempre" en todas ellas})$
 Recordamos el mejor puntaje P
- En las siguientes $n-t$ citas, a penas 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.

- $P \leftarrow \min \pi[1] \dots \pi[t]$

- j^* es el 1^{er} $j \geq t+1$ tq $\pi[j] < p$

• d'P($\pi[j] = 1$)?

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

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

Fallamos si entre las cifas $t+1$ 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.

$$\Rightarrow P[\pi[j] = 1]$$

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

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

d'por qué?

$$P[\pi[j] = 1 \text{ y } (\min \dots)] \Rightarrow t = \frac{n}{2}$$