

2015年10月5日 (月)

## Diccionarios en memoria secundaria.

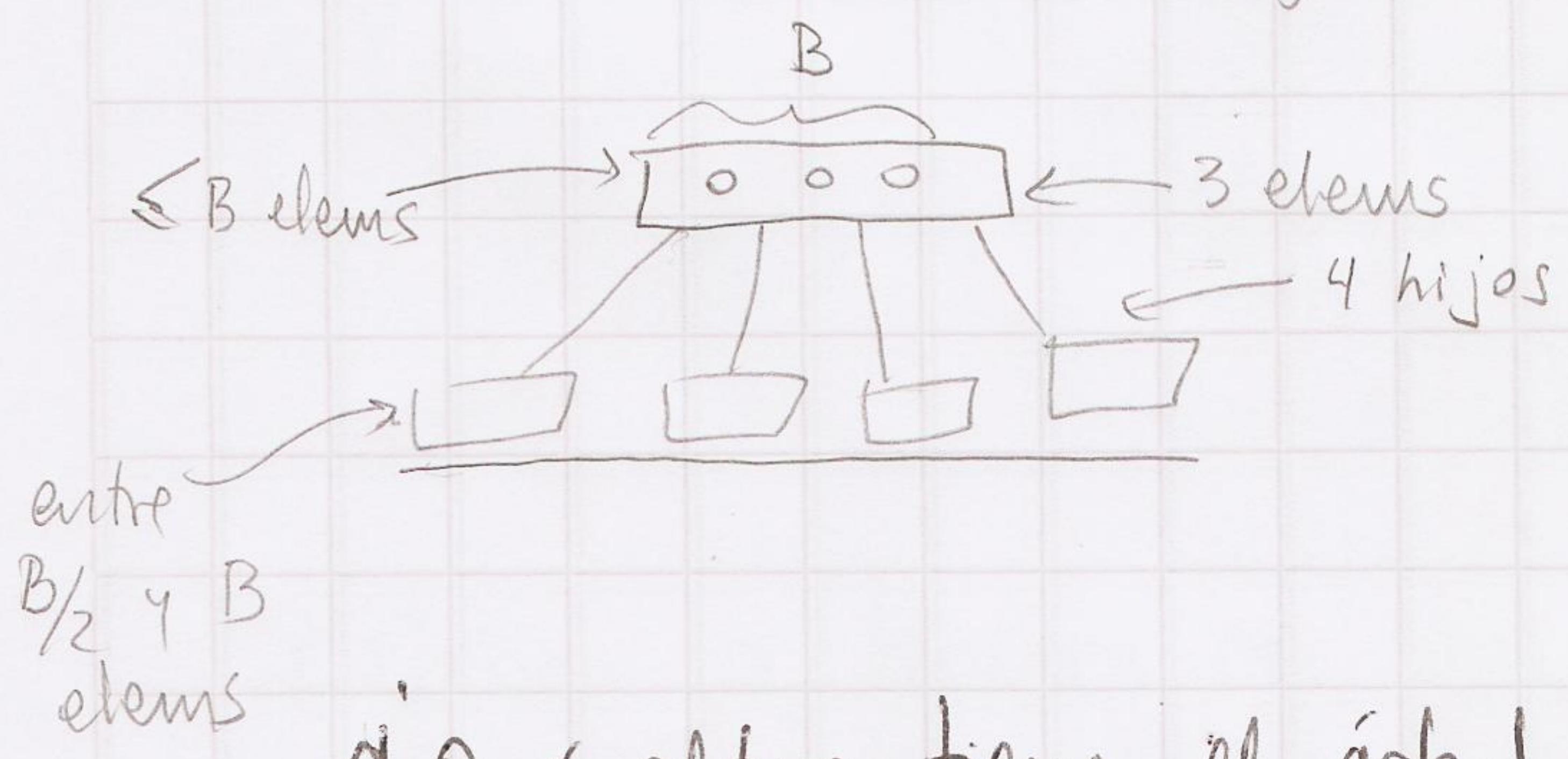
B-Tree : insertar sucesor  
borrar predecesor  
buscar iterar en orden...

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:

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

- ¿Qué altura tiene el árbol?

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

- Los algoritmos:

- a) Busqueda - (x)

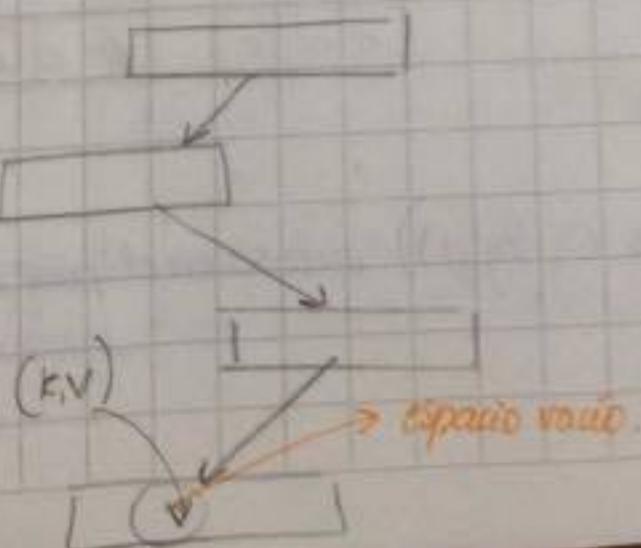
↳ Bajar por el árbol eligiendo el hijo adecuado en cada vez (toma  $O(\log_B N)$ ).

Lo complicado es insertar / Borrar.

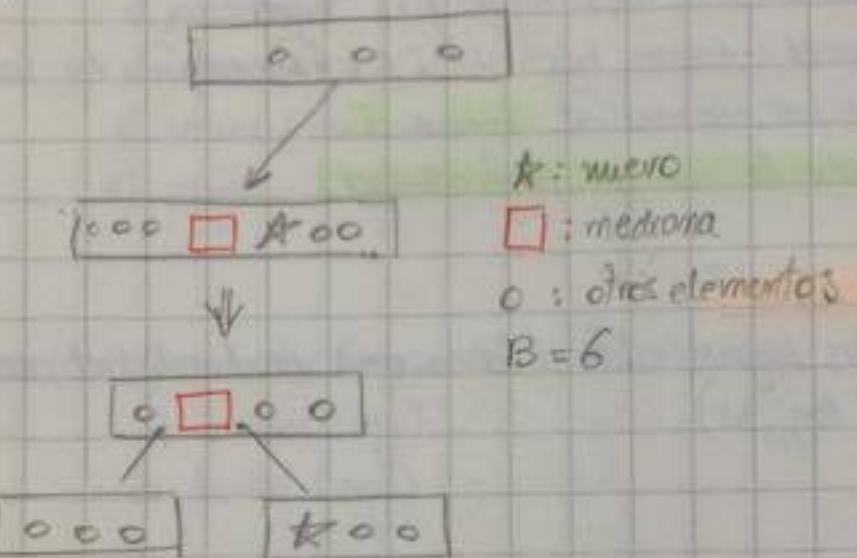
- a) insertar : (k,v)

↳ Se busca la clave a insertar; al no encontrarse, encontramos el lugar donde debería estar (en una hoja).

$\{O(\log_B N)\}$



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



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

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

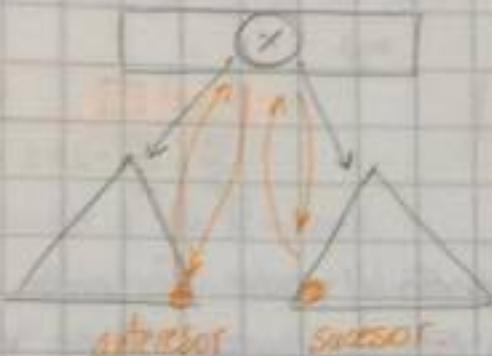


- ⇒ Tenemos que insertar es  $O(\log_B N)$  incluso en el peor caso.

### 3) Borrado (e):

Buscar y encontrar la clave.

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

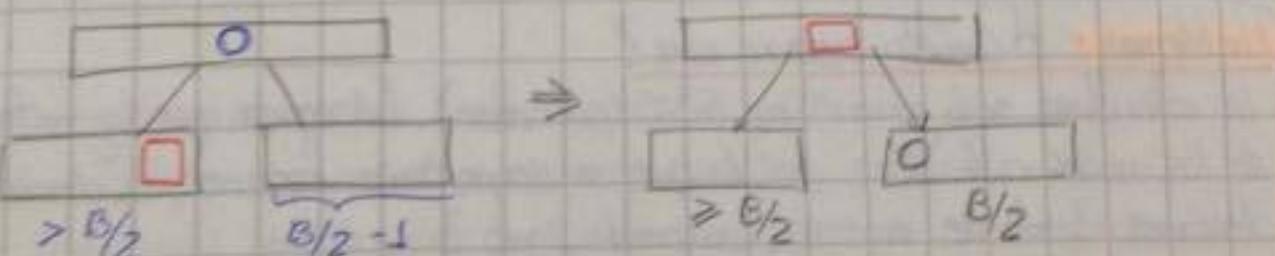


⇒ sin pérdida de generalidad, el borrado ocurre en las hojas.

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

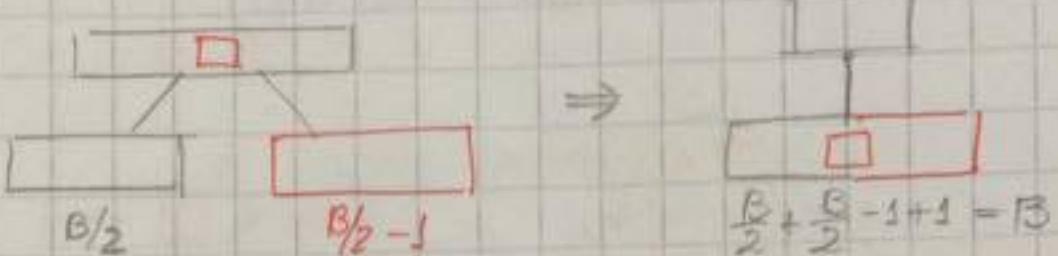
↳ Hago los vecinos y le pido un elemento a alguno:

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  $B/2$  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, ocupará 2 mb.]

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

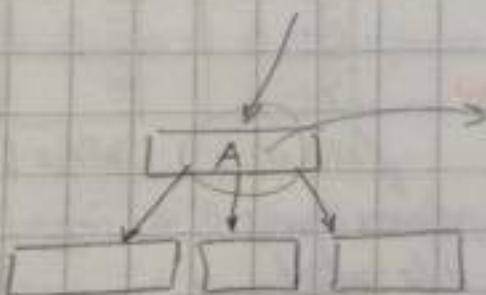
- Una variante útil son los Bt-trees.

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

### R-trees

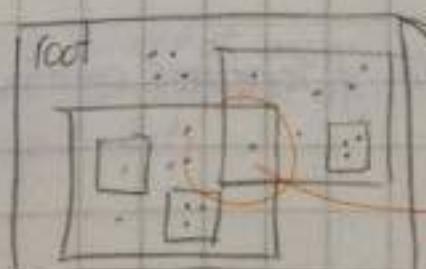
- ↳ Son muy parecidos a los R-trees.

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



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

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



↳ Contiene a todos los demás rectángulos.

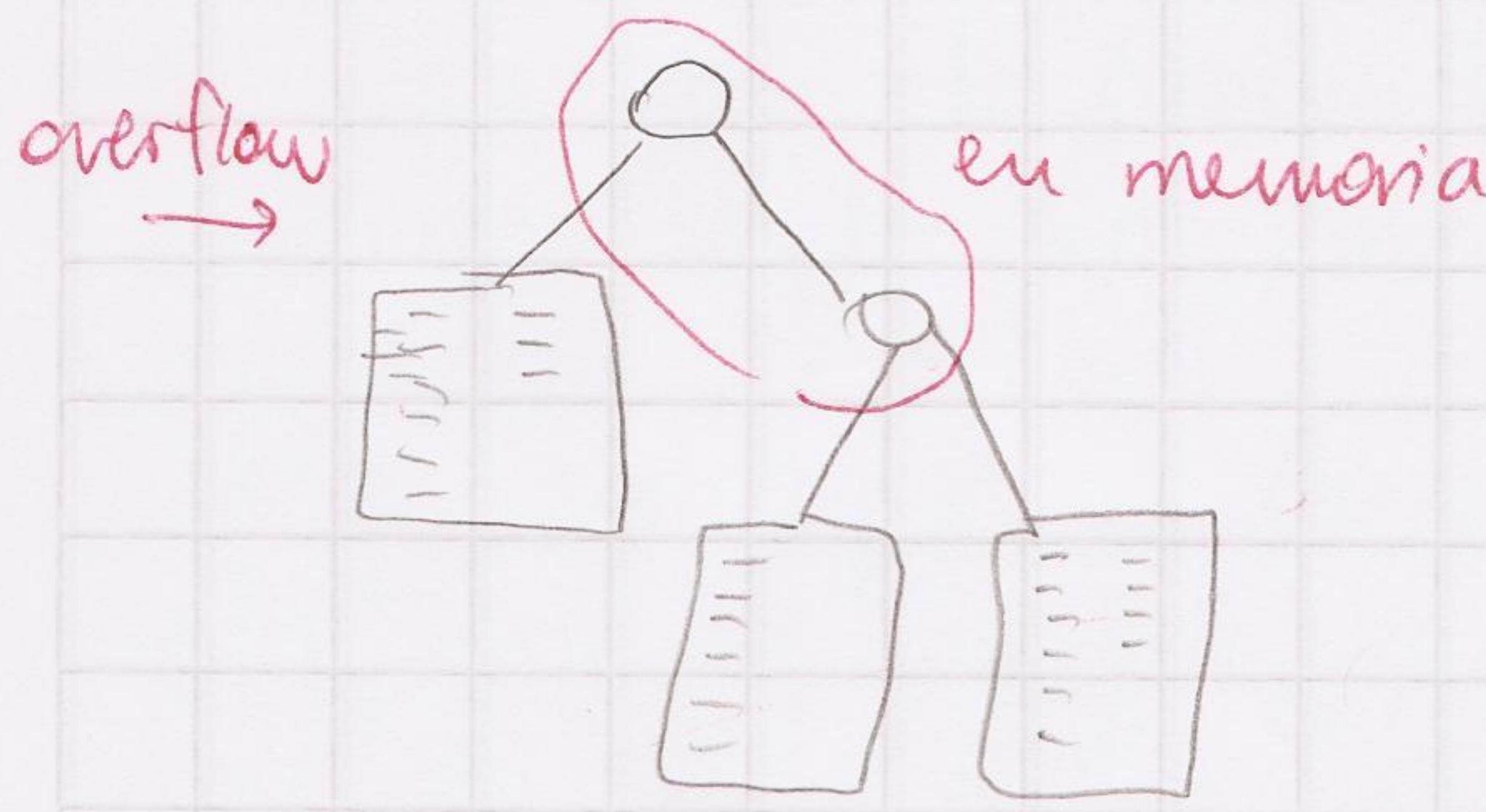
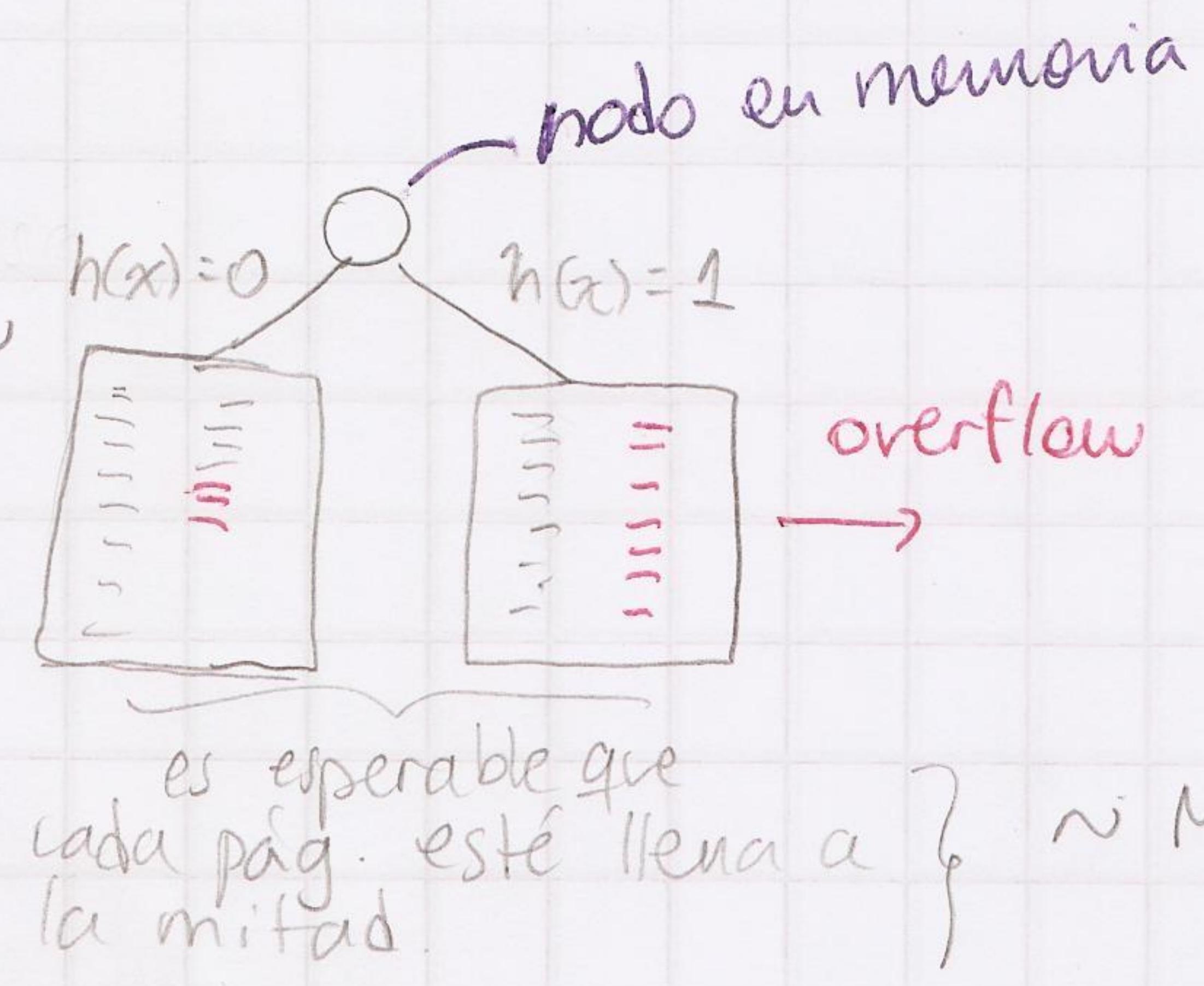
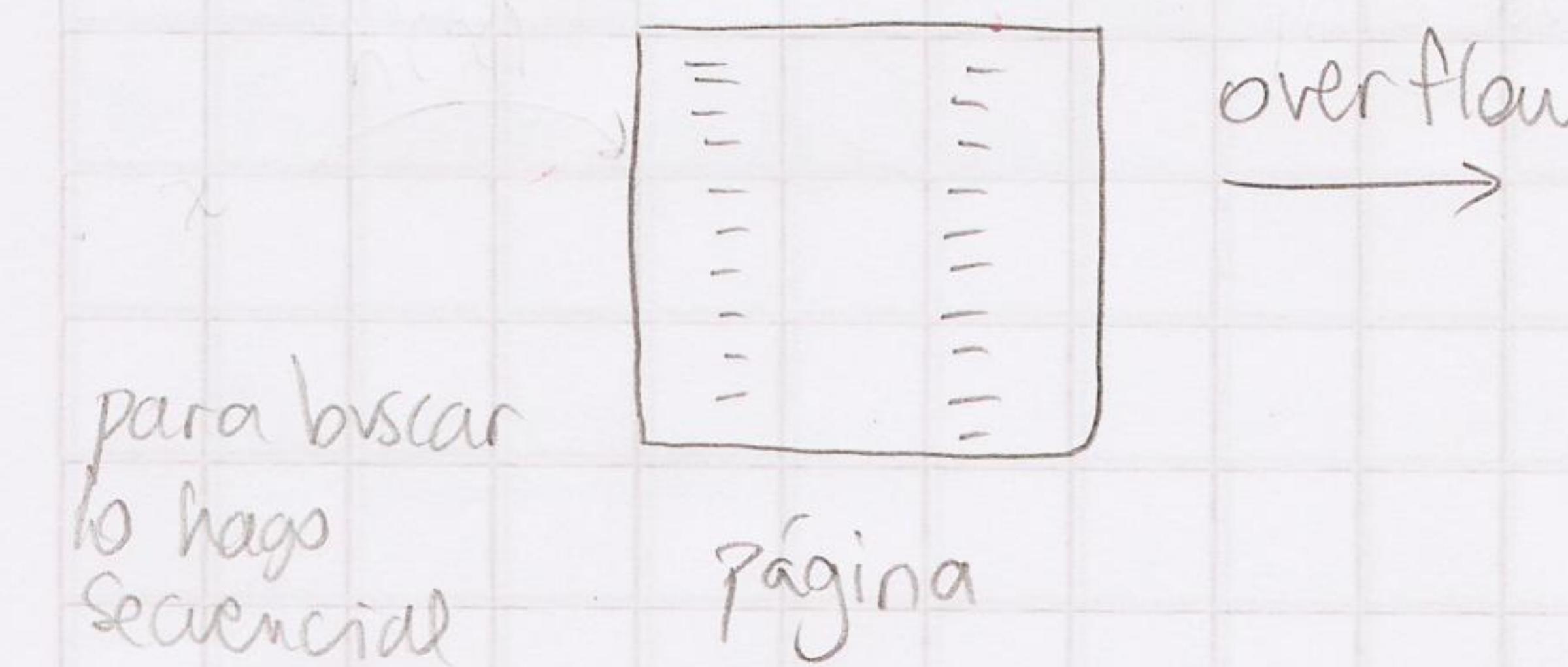
Se pueden solapar.

# Hashing en Disco

2015年10月6日 (X)

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

## • HASHING EXTENDIBLE:



(pues los nodos están en memoria.)

$\Rightarrow$  inserción = 3 accesos (?)

2 sin overflow: leer y escribir

3 con " " : leer y escribir 2 págs.

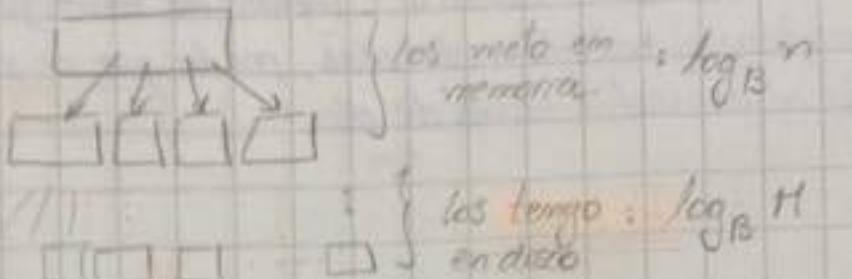
Si hay varios overflows seguidos, las páginas vacías las dejo NULL.

$\Rightarrow$  garantizado 50% ocupación

## Hashing en disco

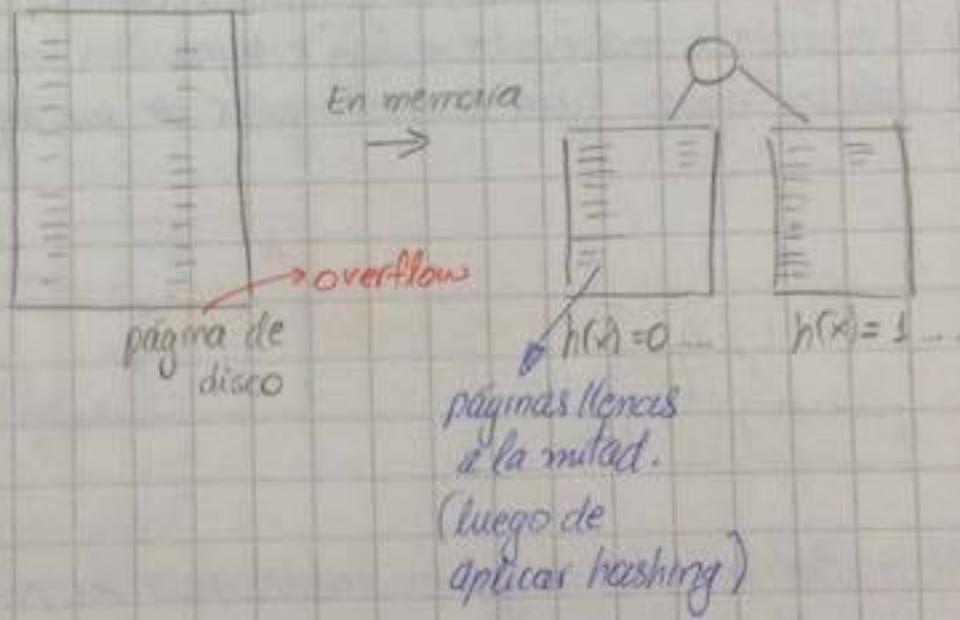
06/octubre/2015

- Hashing extendible
- Hashing lineal



$$\Rightarrow \log_{10} n - \log_{10} M = \log_{10} \frac{n}{M}$$

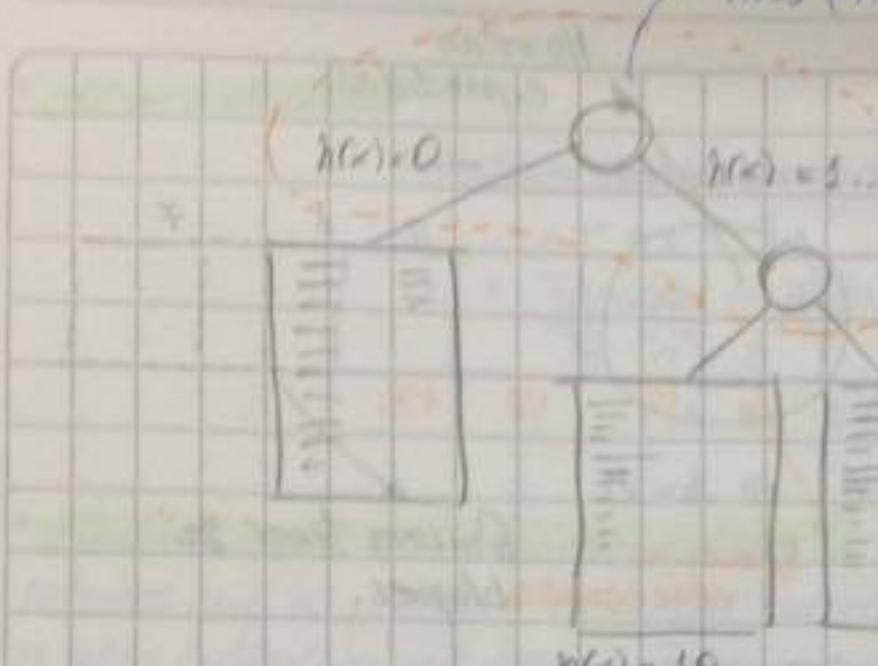
### 1. • Hashing extendible.



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)



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

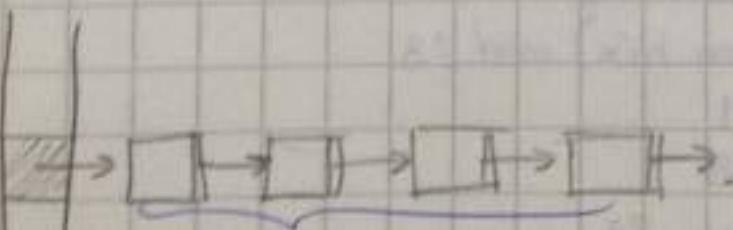
(En promedio es ~67%)

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

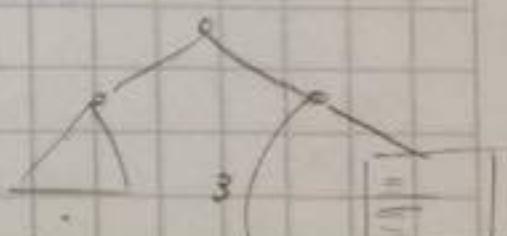
- Búsqueda en esta estructura es 1 acceso.
- Inserción son 3 accesos. (leer / escribir si no hay overflow son 2 accesos, y si leemos / 2 escribimos con overflow, pues tenemos 2 páginas que escribir )

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

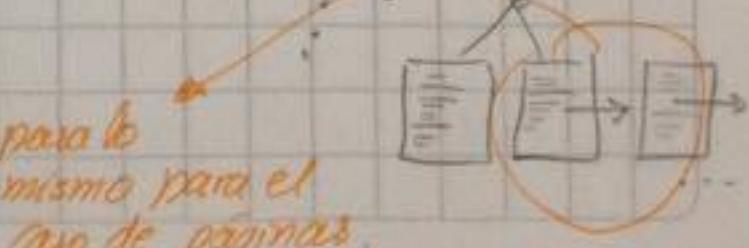
Si la función de hash es mala, entonces hay colisiones. No hay mucho que hacer, al igual que en estructura de datos.



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



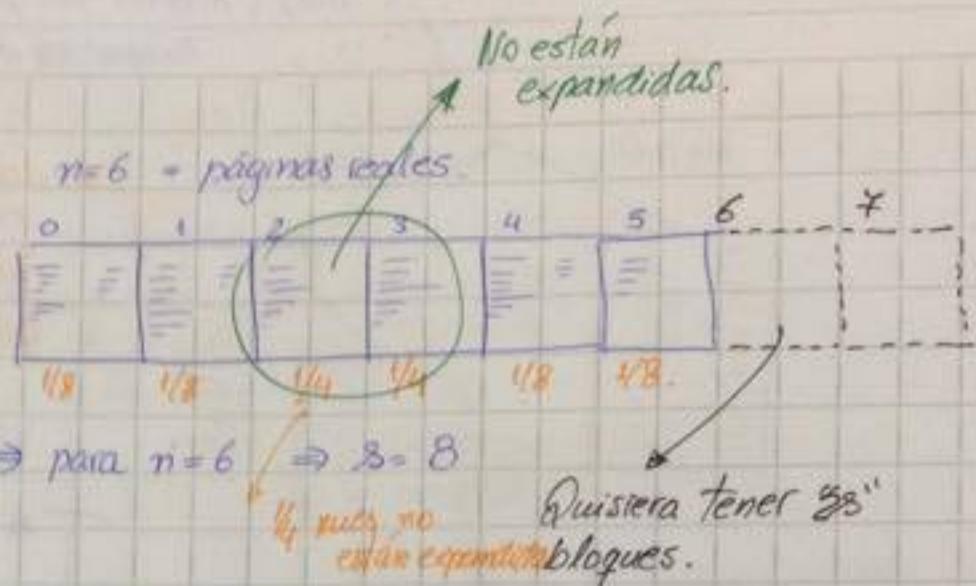
para lo mismo para el caso de páginas.



## 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) mod s < n mod s
    return h(x) mod 2s
    return h(x) mod s
  } supondremos s ≤ n < 2s
  }
```

## • Cómo expandir?

Expandir Tabla :

```

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

### • ¿Como contraer?

Contraer - Tabla :

$$n \leftarrow n-1$$

agregar la página  $n$  a la  $n-1$

$$s: n = s$$

$$s \leftarrow s/2$$

### • ¿Cuando expandir? ¿Cuando 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 enlazarán en una lista, esperando su turno. Cuando toque la expansión, se leerá toda la lista y se hashearán los elementos (todos los de la lista enlazada) en las nuevas posiciones (incluidas las celdas extendidas).

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

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

$$\square^* = \frac{1}{1-\square}$$

$$\sum_{i=0}^n a^i = \frac{1-a^{i+1}}{1-a}$$

2015年10月8日 (木)

## Análisis amortizado.

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

→ 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 =  $kn$ ,  $n = 2^k$

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

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

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

k bits	
000	000
000	001
000	010
000	011
000	100
000	101
000	110
000	111
1000	0000

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

⇒ costo amortizado de incrementar es 2.

yo resuelvo esto  
los otros

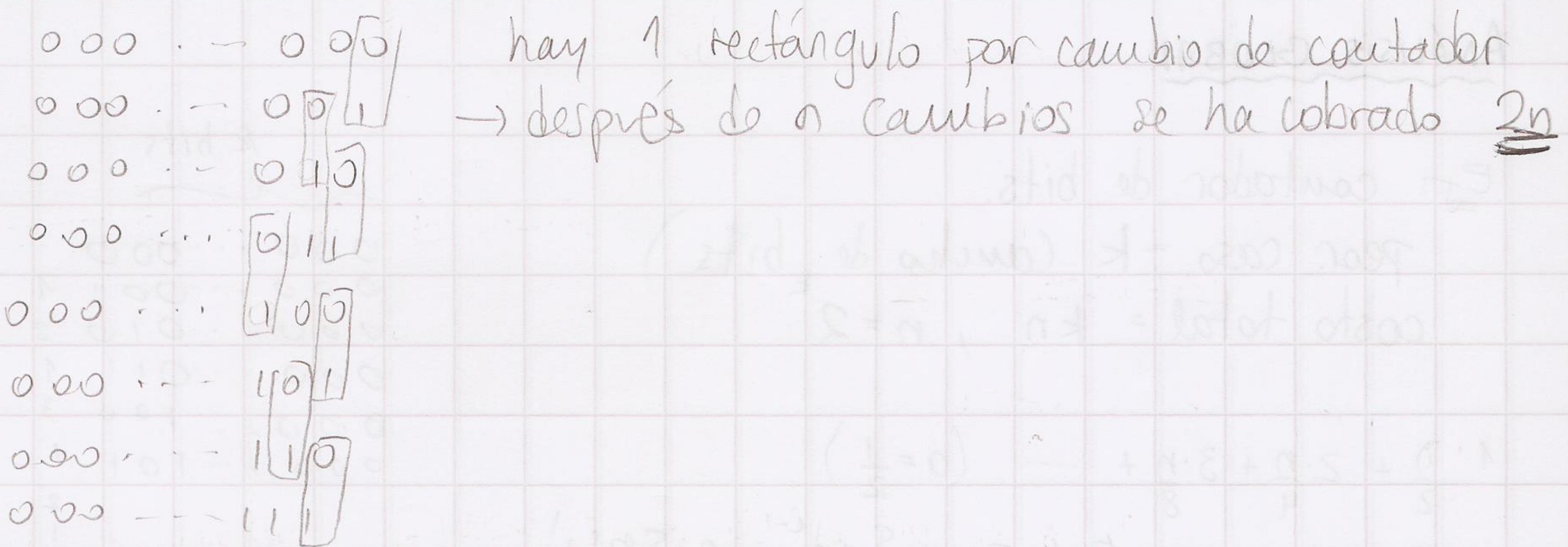
BR RUEZIAS Δ Δ

## dos formas zetaona

### CONTABILIDAD DE COSTOS

Ej: Contador de bits de nro.

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



→ alcaucía

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

$c_n$  = costo real de la  $n$ -ésima operación

$\phi_n$  = el valor de  $\phi$  (la alcaucía) después de la  $n$ -ésima op.

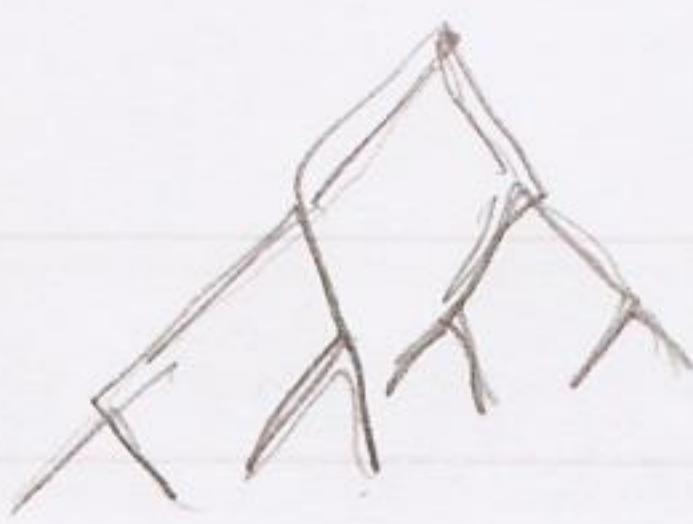
$\hat{c}_n$  = costo amortizado de la  $n$ -ésima operación.

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

Luego

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

cota superior al costo real.



Depth

Ej: Contador de bits.

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

0111

↓ +1

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

1000

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

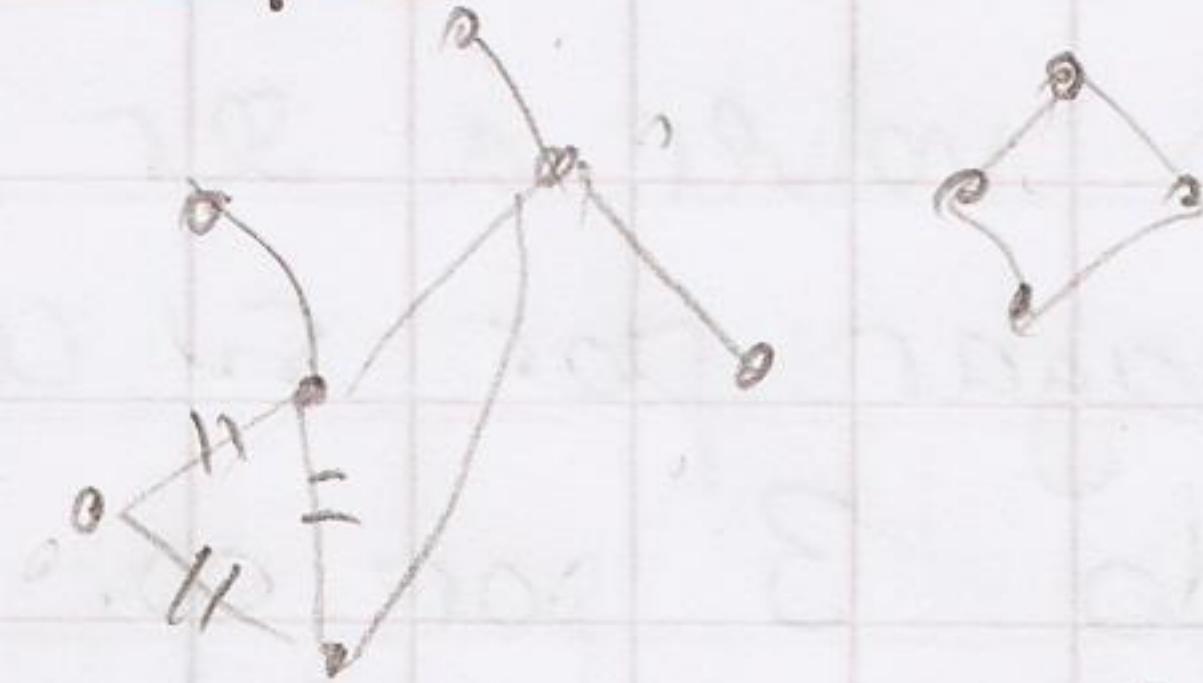
(-c<sub>i-1</sub>)

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

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

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

Otro ejemplo PFS en un grafo para detectar componentes conexas.

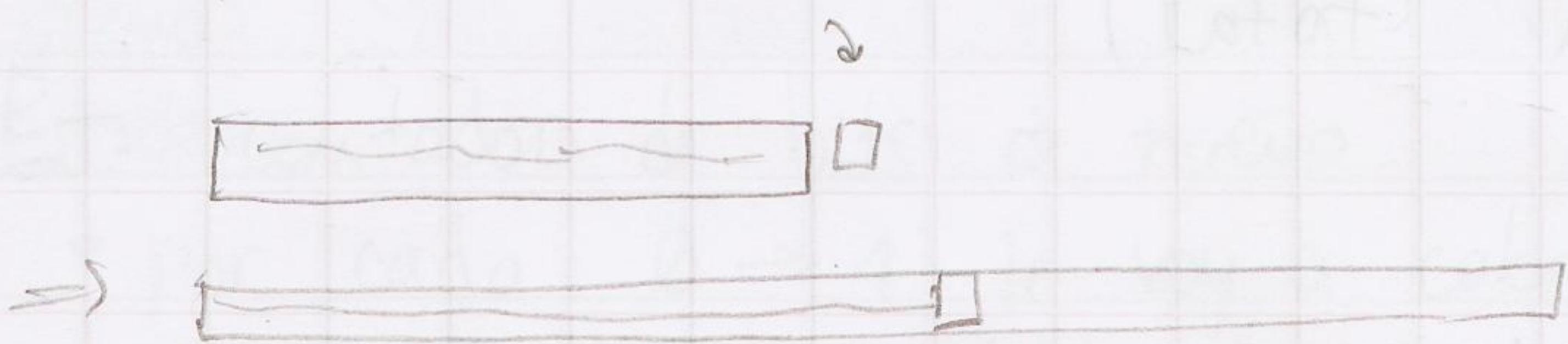


$$\sum_{v \in V} \text{arity}(v) = 2e$$

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

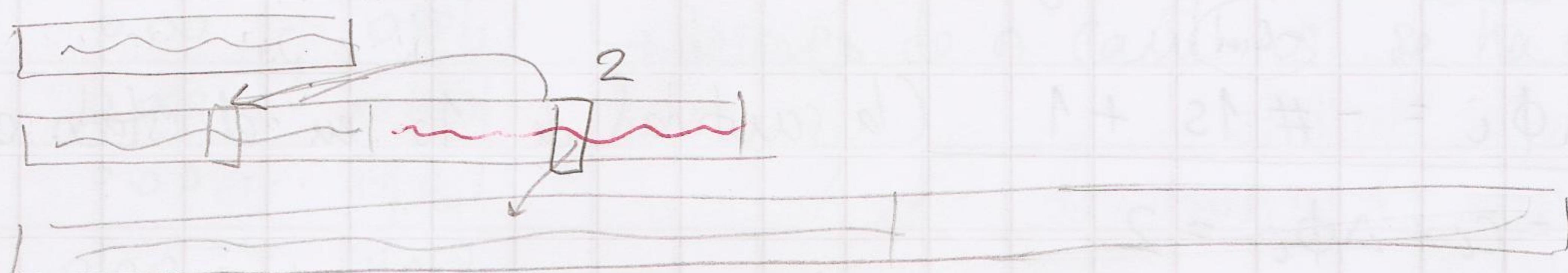
→ hemos usado una técnica de análisis amortizado para calcular costo global  $\Theta(n+e)$

Ej: reallocar duplicando el tamaño.



• Contabilidad de costo.

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

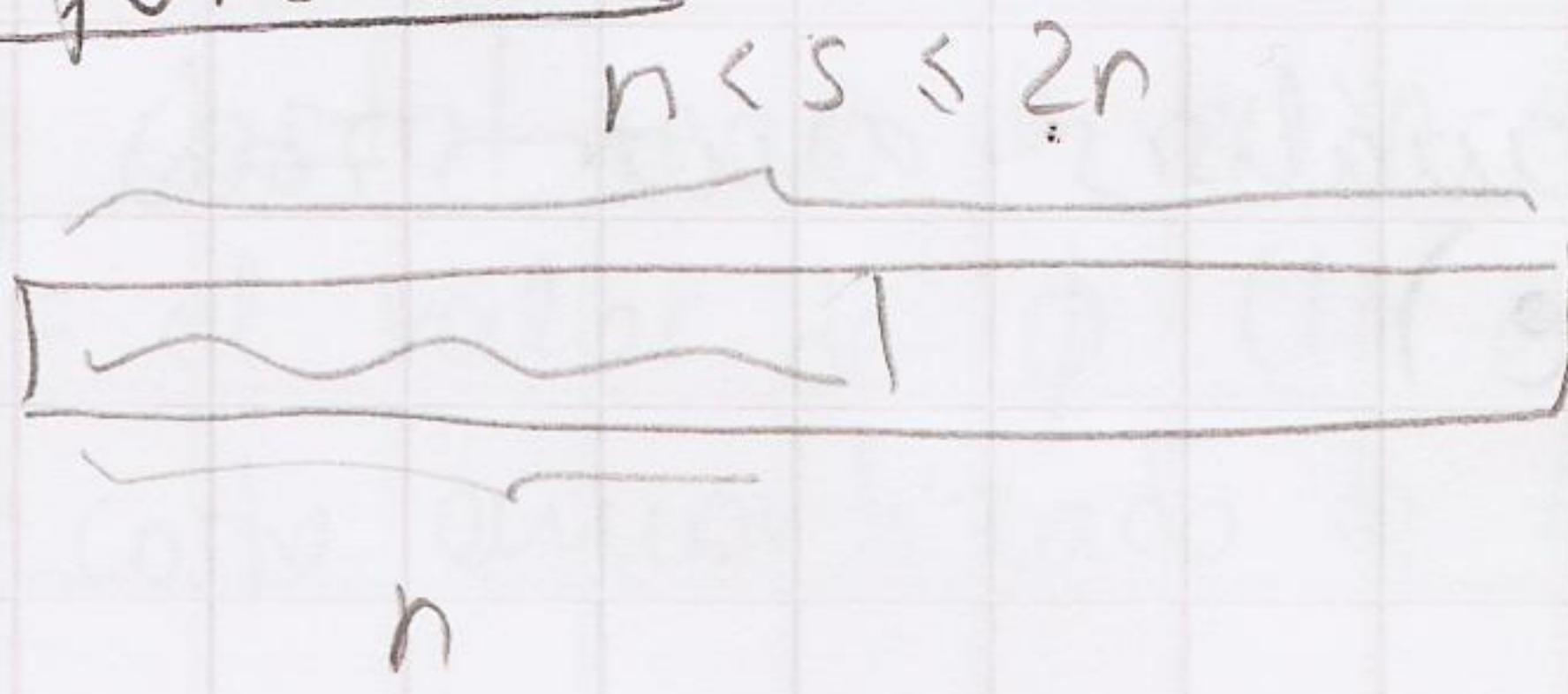


A los elementos que son insertados en la segunda mitad le cobras 1 por insertar + 2 por su futura copia y por copiar el otro elemento simétrico en la primera mitad.

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

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

• Función potencial



s: tamaño de lo que alcancé

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

$$c_i = 1$$

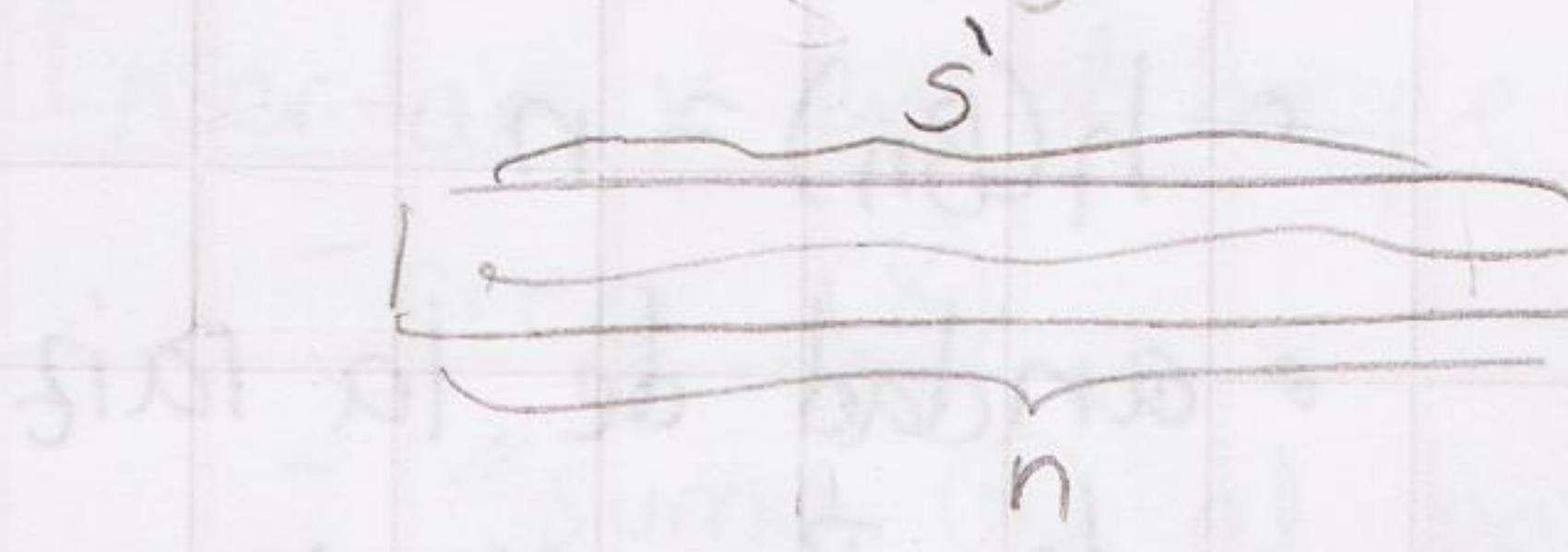
$$\Delta c_i = 2 \quad \because s \text{ no varía}$$

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

Ba Rotulos

admonit cosa

Cuando el gasto se realoca



$$C_i = n$$

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

=>

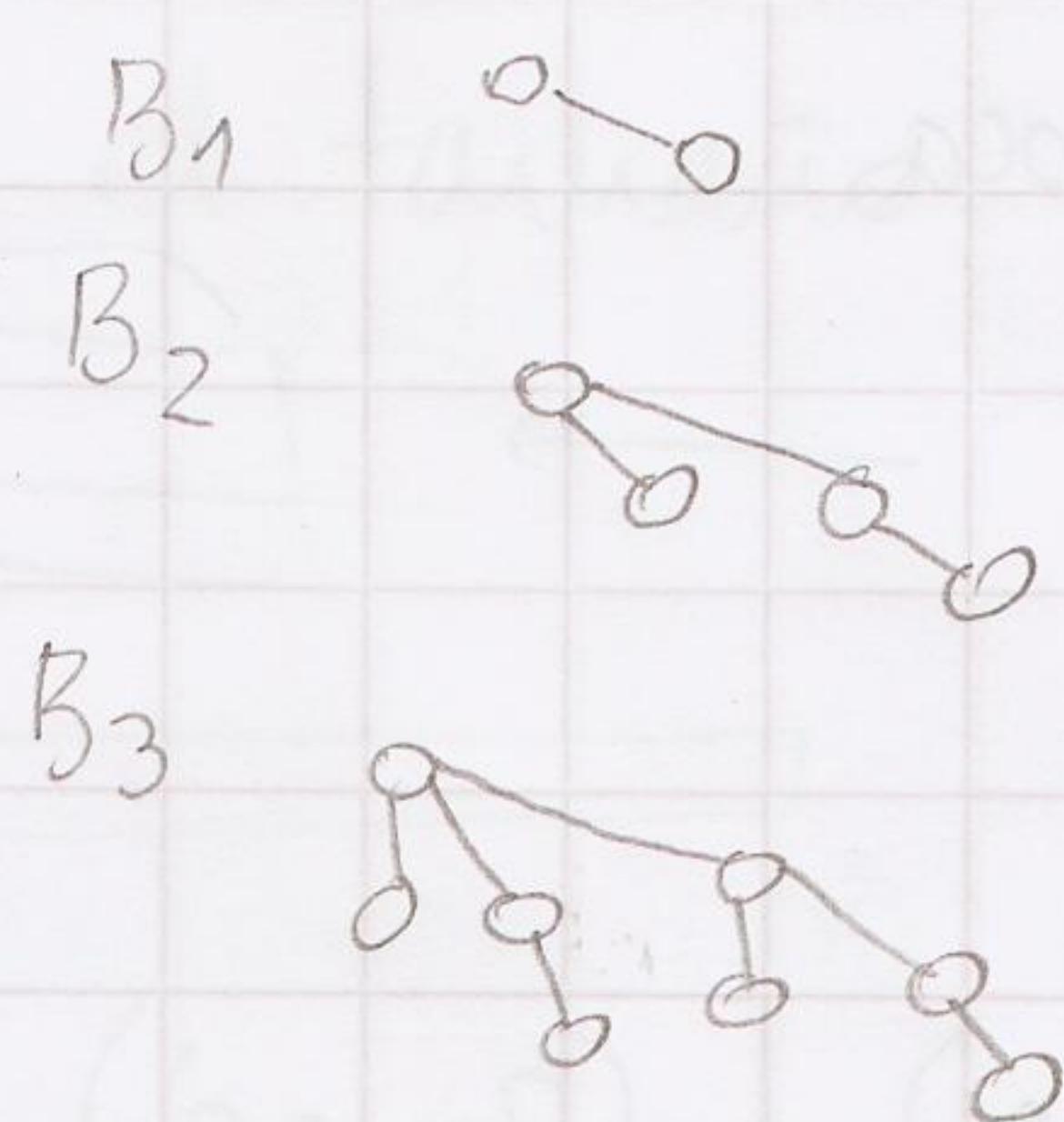
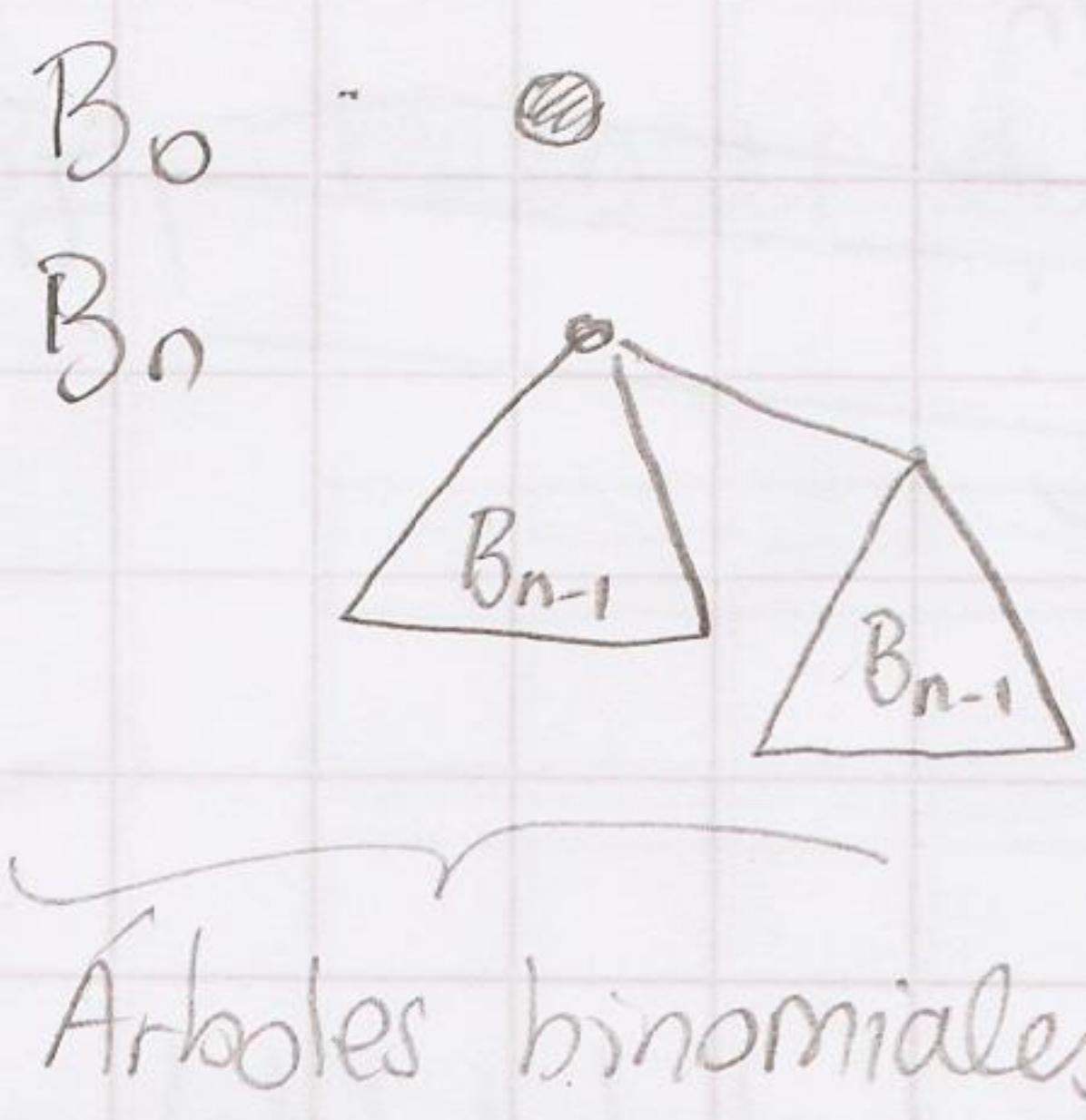
$$\hat{C}_i = 0 \quad \text{Pago con todos los ahorros.}$$

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

En el fondo  $\phi$  está relacionado al espacio libre que queda

# Colas Binomiales

2015年10月13日 (K)

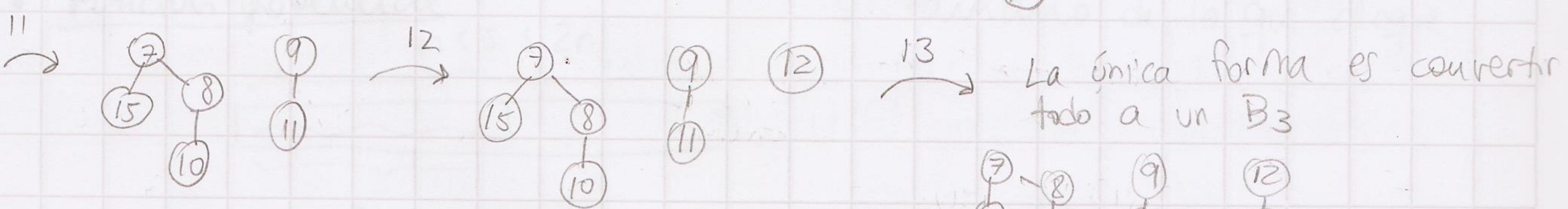
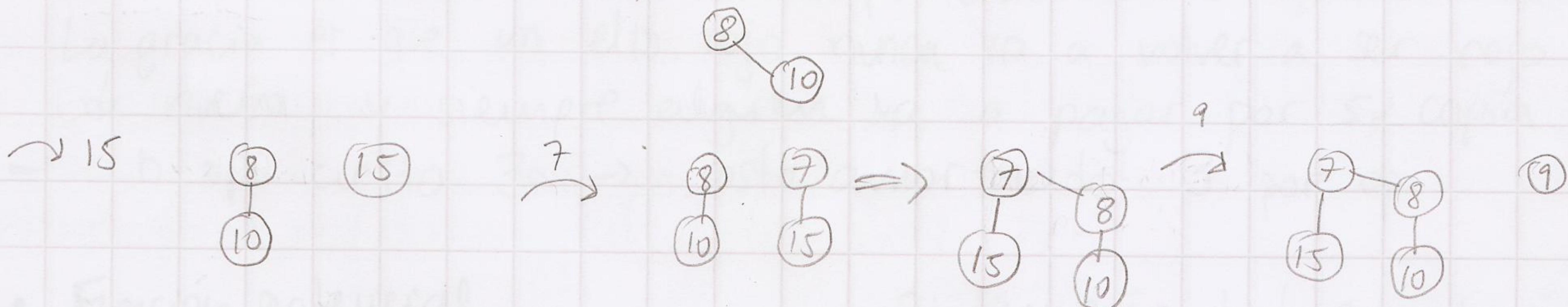


- $B_n$  tiene  $2^n$  nodos
- $h(B_n) = n$
- anchura de la raíz de  $B_n$  es  $n$

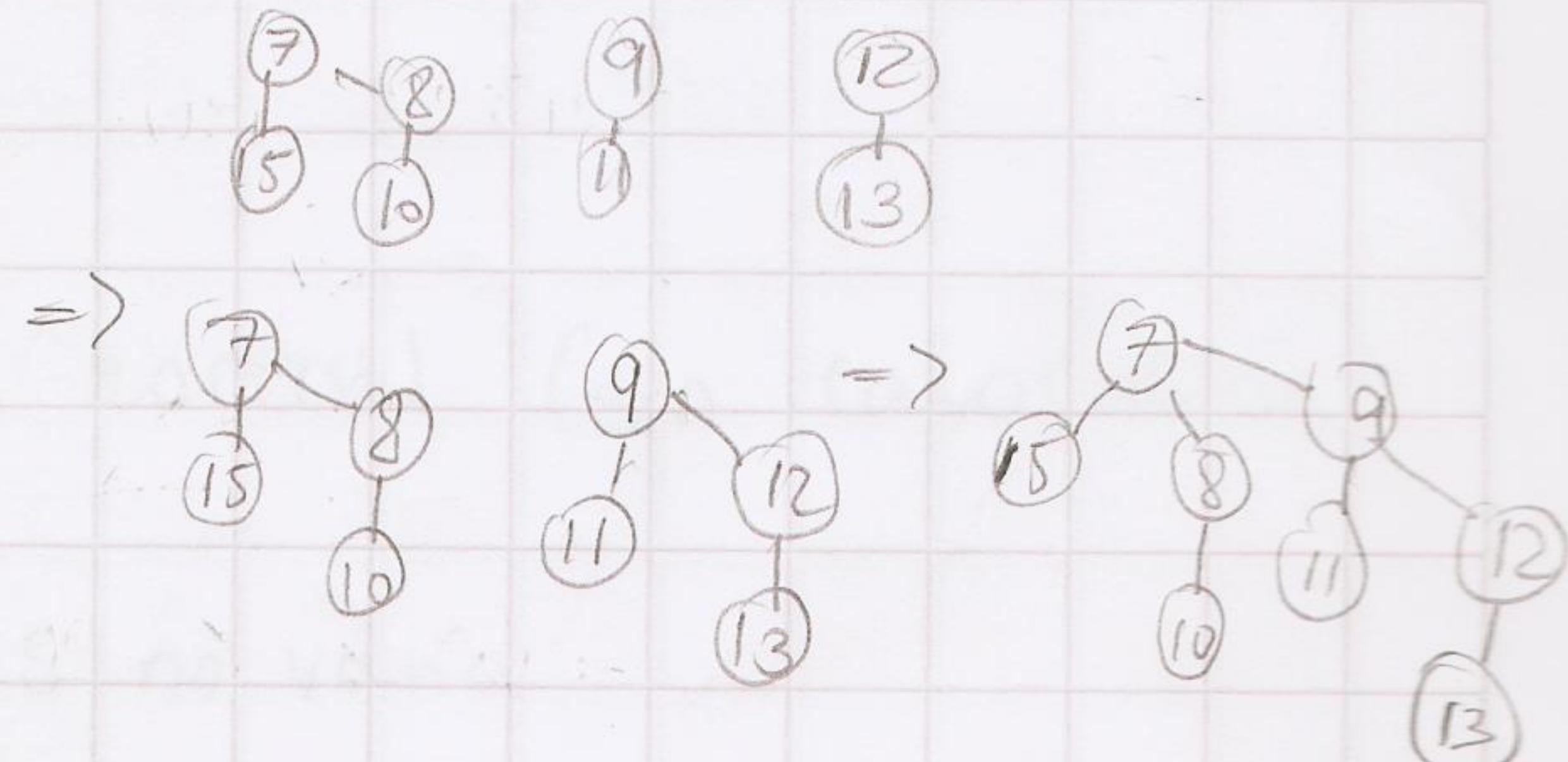
→ Un bosque binomial es un conjunto de árboles binomiales, todos de distinto tamaño.

→ 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       $\circlearrowleft 10$        $\circlearrowleft 8$        $\rightarrow$  bosque binomial. No puede haber 2 árboles de = tamaño



La única forma es convertir todo a un  $B_3$



Entonces ...

- > Insertar  $x$  en el bosque
  - creo  $\times$
  - "sumo"  $\times$  al bosque
    - ↳ si no hay un árbol del tamaño de  $\times$   
agregamos  $\times$
    - ↳ 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.

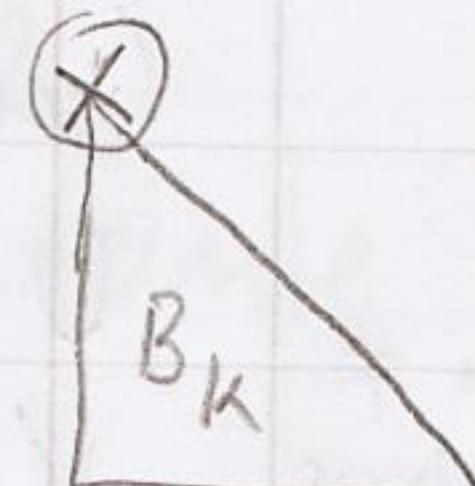
∴ La inserción cuesta  $\Theta(\log n)$

~~bien~~ también es  
peor caso

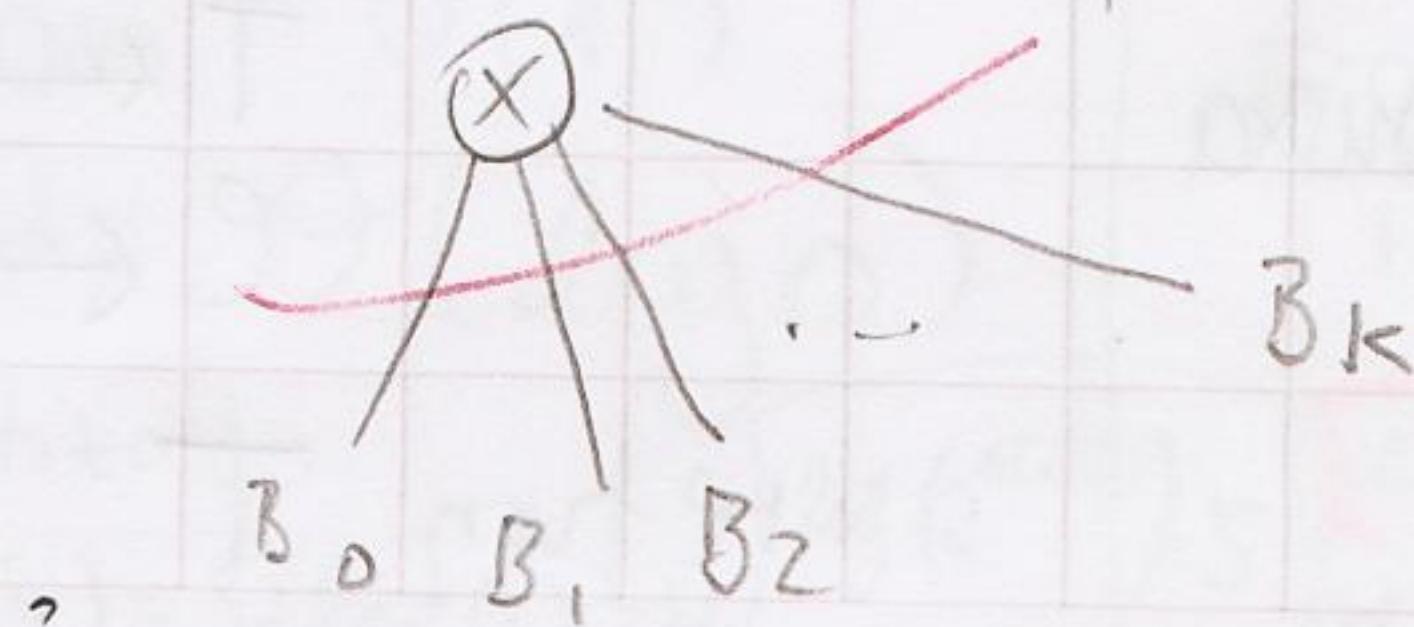
OBS: podemos hacer heapify mediante insertar  $n$  elementos, a costo  $\Theta(n)$   
(producir esta estructura)

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

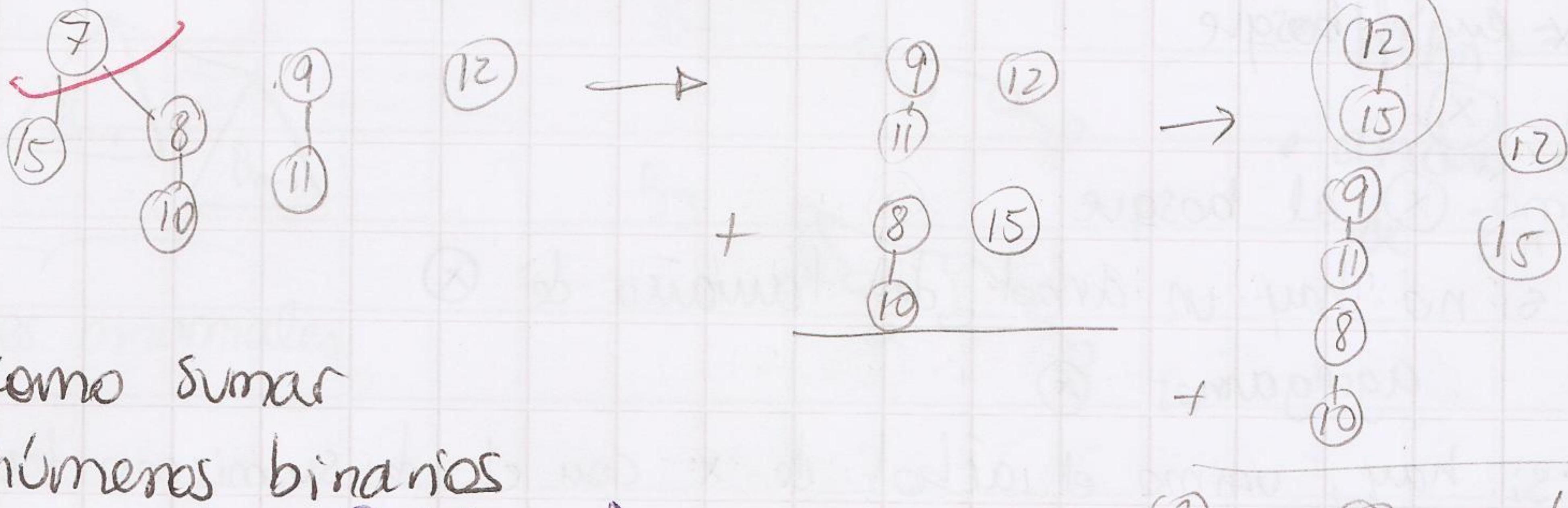


- Eliminar  $X$  y quedarse con los  $k$  subárboles



## Colas Binomiales

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



es como sumar  
dos números binarios

$\Theta(\log n)$

(o también  
podía dejar  
otra pareja  
a fiesta)

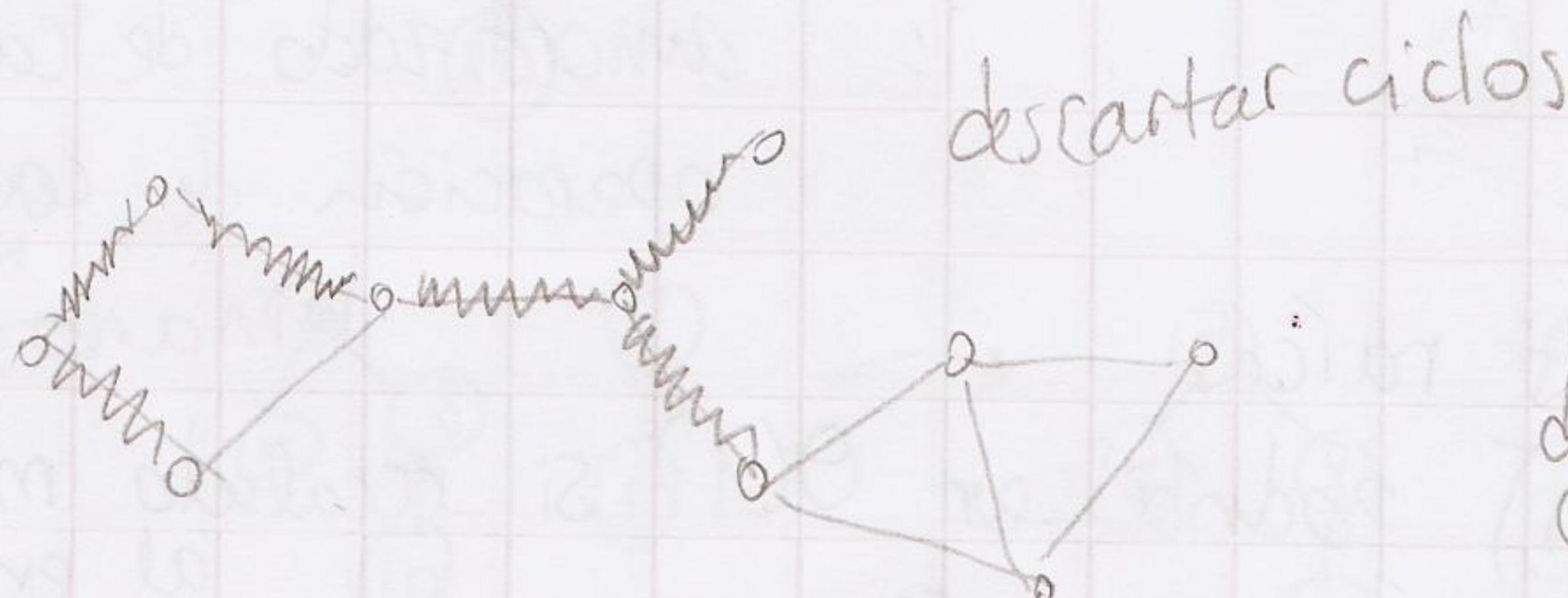
- Merge ( $T_1, T_2$ )

"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 costos  
de arista

heapify (E)  
 $\rightarrow \mathcal{C} \leftarrow \{\{v\}, v \in V\}$   
 $T \leftarrow \emptyset$   
 $\rightarrow$  while  $|\mathcal{C}| > 1$

$(u, v) \leftarrow \text{extraMin}(E)$   
 $U \leftarrow \text{componente de } u \text{ en } \mathcal{C}$   
 $V \leftarrow \text{componente de } v \text{ en } \mathcal{C}$

$\rightarrow$  if  $U \neq V$

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

$\mathcal{C} \leftarrow \mathcal{C} - \{U, V\} + \{U \cup V\}$

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.
- $\text{Find}(x)$  me da el representante de la clase a la que  $x$  pertenece.
- $\text{Union}(x,y)$  une las clases representadas por  $x$  e  $y$ .

Kruskal:  $(V, E)$

$\text{heapify}(E)$

$C \leftarrow \text{clases}(V)$

$T \leftarrow \emptyset$

while  $|C| > 1$

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

$x \leftarrow \text{Find}(u)$

$y \leftarrow \text{Find}(v)$

if  $x \neq y$

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

$\text{Union}(x, y)$

return  $T$

$\Theta(e \log n)$   
en cada iteración

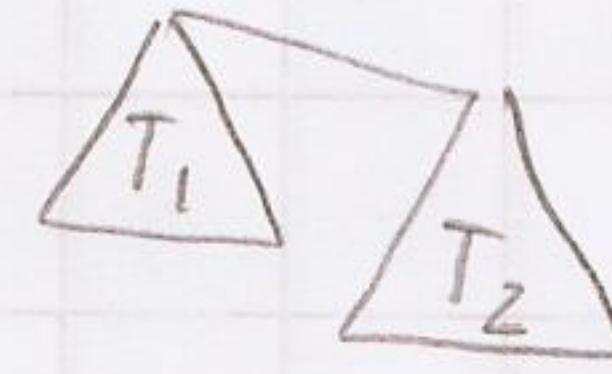
Lema: todo árbol de altura  $h$  tiene al menos  $2^h$  nodos

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)  $h(T) = h(T_1)$

$$h = h_1 + h_2 \geq h_1 \geq 2^{h(T_1)} = 2^{h(T)}$$

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

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

Se puede ver como árboles

cuyos nodos apuntan a

su padre → representante

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

$\text{Union}: \Theta(1)$

$\text{Find}: \Theta(\log n)$

$m \text{ finds} \rightarrow \Theta(m \log^* n)$  amortizado

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

$\text{Find}(u)$

if  $u.\text{padre} = \text{null}$

return  $u$

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

return  $u.\text{padre}$

va aprendiendo  
modifica la estructura  
al leer.

Evidencia de tipo de la fibra de los elefantes

Algunas de las evidencias más conocidas

• Caben 10 billetes en monedero

