Capítulo 3

Técnicas avanzadas de diseño y análisis de algoritmos

ANÁLISIS AMORTIZADO

Análisis amortizado

- Objetivo: medir el tiempo requerido para realizar una secuencia de operaciones
  - Operaciones en una estructura de datos
  - Promedio sobre todas las operaciones realizadas
- Sirve para mostrar que el *costo promedio* de una operación es pequeño (aunque pueden haber operaciones individuales costosas)

Análisis amortizado

- Tres técnicas comunes
  - Análisis global
    - Se muestra que para todo *n*, una secuencia de *n* operaciones toma tiempo T(*n*) (peor caso)
    - En el peor caso, el costo amortizado es T(*n*)/*n*
  - Contabilidad de costos
    - Se asignan diferentes costos a operaciones distintas (su costo amortizado)
    - A algunas operaciones se les cobra más de lo que realmente cuestan (dan crédito)
    - Crédito sirve para "pagar" otras operaciones costosas

Análisis amortizado

- Tres técnicas comunes
  - Función potencial
    - Crédito se representa como “energía potencial”
    - Se puede utilizar para pagar operaciones futuras

Incrementar contador binario

- Problema: implementar un contador binario
  - *k*-bits (rango: [0, 2^k-1]), cuenta a partir de 0
- Estructura de datos
  - Arreglo A[0..k-1] de bits, length(A)=*k*
  - Bit de menor orden en A[0], bit de mayor orden en A[k]

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

Incrementar contador binario

- Inicialmente:
  - x=0, A[i]=0 para todo i=1..k-1
- Procedimiento para incrementar contador

```
Increment(A)
1 i = 0
2 while i<length(A) and A[i]=1
3       A[i] = 0
4       i = i+1
5 if i < length(A)
6       A[i] = 1
```

Incrementar contador binario

- Ejemplo de ejecución:

```
Increment(A)
1 i = 0
2 while i<length(A) and A[i]=1
3       A[i] = 0
4       i = i+1
5 if i < length(A)
6       A[i] = 1
```

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Incrementar contador binario

- Costo de cada operación de *Increment* (acarreos) es lineal en el número de bits actualizados
- Análisis de peor caso
  - *Increment* toma O(k) acarreos en el peor caso en cada ejecución
  - Tiempo total: O($k\,2^k$)
- Cota correcta pero no ajustada

Incrementar contador binario

- Cota ajustada usando análisis global
  - Observación: no todos los bits cambian cada vez que se invoca a *Increment*
    - A[0] cambia en cada invocación a *Increment*
    - A[1] cambia cada dos invocaciones
    - A[2] cambia cada cuatro invocaciones
    - …
    - A[*i*] cambia floor($n/2^i$) veces en una secuencia de *n* operaciones de *Increment* (partiendo de 0)
    - Nota: $n = 2^k$

Incrementar contador binario

- Cota ajustada usando análisis global
  - Número total de cambios:

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

  - Peor caso para secuencia de *n* operaciones de *Increment*: O(*n*) (= O(2^k))
  - Costo amortizado: O(n)/n = O(1)

Incrementar contador binario

- Cota ajustada usando contabilidad de costos
  - Tiempo es proporcional al número de bits cambiados: se usará como costo
  - Un cambio de bit = una luca
  - Para el análisis amortizado
    - Se cobran 2 lucas para cambiar un bit a 1
      - 1 luca paga el cambio del bit
      - 1 luca queda de crédito (para cuando el bit cambie a 0)

Incrementar contador binario

- Cota ajustada usando contabilidad de costos
  - En cada instante de tiempo, cada ‘1’ en el contador tiene una luca de crédito
  - No se cobra nada para cambiar el bit a 0
  - Costo amortizado:
    - Costo de cambiar bits a 0 en el *loop* no se cobran (se paga con crédito, nunca hay crédito negativo)
    - Se cobran 2 lucas para cambiar un bit a 1
    - Costo amortizado de una operación de Increment: 2 lucas
    - Costo amortizado total: O(*n*)

Incrementar contador binario

- Cota ajustada usando función potencial
  - Sea c_i el costo real de la i-ésima operación
  - Sea D_i la estructura de datos que resulta de realizar la i-ésima operación sobre D_{i-1}
  - Funcion potencial ϕ
    - Mapea D_i al número real $\phi(D_i)$
    - Es el potencial asociado a D_i
    - Costo amortizado se define como:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_i - 1)$$

Incrementar contador binario

- Cota ajustada usando función potencial
  - Costo amortizado total:

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} c_i + \Phi(D_i) - \Phi(D_i - 1)$$

$$= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)$$

Incrementar contador binario

- Cota ajustada usando función potencial
  - Si $\phi(D_n)$ es mayor que $\phi(D_0)$, el costo amortizado total es cota superior del costo real total
  - Si no se sabe a priori el número de operaciones a realizar:
    - Si $\phi(D_i)$ es mayor que $\phi(D_0)$ para todo i, se garantiza que se “paga por adelantado”
    - Es conveniente definir $\phi(D_0)=0$ y luego mostrar que $\phi(D_i)>=0$ para todo i

Incrementar contador binario

- Cota ajustada usando función potencial
  - Potencial del contador después de la i-ésima invocación de *Increment*: número de 1's en el contador (b_i)
  - Costo amortizado de una operación de *Increment*
    - Supongamos que i-ésimo *Increment* resetea t_i bits
    - Costo real de la operación: a lo más $t_i + 1$ (resetear t_i bits y cambiar un bit a 1)

Incrementar contador binario

- Cota ajustada usando función potencial
  - Si $b_i = 0$, i-ésima operación reseteó todos los bits a 0, entonces $b_{i-1} = t_i = k$
  - Si $b_i > 0$, entonces $b_i = b_{i-1} - t_i + 1$
  - En ambos casos $b_i <= b_{i-1} - t_i + 1$
  - Por lo tanto, diferencia de potencial es

$$\Phi(D_i) - \Phi(D_{i-1}) = (b_{i-1} - t_i + 1) - b_{i-1}$$

$$= 1 - t_i$$

Incrementar contador binario

- Cota ajustada usando función potencial
  - Costo amortizado es:

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2\end{aligned}$$

Incrementar contador binario

- Cota ajustada usando función potencial
  - Costo amortizado de secuencia de *n* operaciones es cota superior del costo real:
    - $\phi(D_0)=0$ si el contador empieza en 0
    - $\phi(D_i)>=0$ para todo *i*
  - Costo amortizado total (peor caso): O(*n*)

Realocación de un arreglo

- En algunas aplicaciones no se sabe a priori cuántos elementos se insertarán en un arreglo (o tabla)
- Arreglo debe ser realocado (copiado) en uno mayor si éste se llena
- Estudiaremos el problema de expandir dinámicamente un arreglo

Realocación de un arreglo

- Arreglo se llena cuando todos sus casilleros han sido ocupados
- Si se intenta insertar un elemento en un arreglo lleno se produce *overflow*
- Solución: expandir el arreglo
  - Crear arreglo nuevo con más casilleros
  - Copiar casilleros del arreglo antiguo en el nuevo
  - Insertar el elemento nuevo

Realocación de un arreglo

- Heurística: crear arreglo nuevo del doble del tamaño del antiguo
  - Espacio no utilizado nunca es mayor a la mitad del tamaño del arreglo
- Sea
  - T: tabla, *x:* elemento
  - table[T]: puntero a la tabla
  - num[T]: número de elementos en la tabla
  - size[T]: tamaño de la tabla

Realocación de un arreglo

- Seudocódigo del método de inserción

```
Table-Insert(T,x)
1. if size[T]==0
2.     allocate table[T] with 1 slot
3.     size[T]=1
4. if num[T]==size[T]
5.     allocate new-table with 2*size[T] slots
6.     insert all items in table[T] into new-table
7.     free table[T]
8.     table[T]=new-table
9.     size[T]=2*size[T]
10.insert x into table[T]
11.num[T]=num[T]+1
```

Realocación de un arreglo

- Análisis del método de inserción
  - Número de inserciones en arreglo
  - En una secuencia de n operaciones
    - $c_i = 1$ si hay espacio en el arreglo
    - $c_i = i$ si se produce *overflow*
  - Peor caso de una operación: $O(n)$
  - Análisis de peor caso para n operaciones: $O(n^2)$
- Cota es correcta, pero no ajustada

Realocación de un arreglo

- Cota no es ajustada porque expansiones del arreglo no ocurren frecuentemente
  - i-ésima operación produce una expansión sólo cuando *i*-1 es una potencia de 2
- Utilizando análisis amortizado obtendremos cotas más ajustadas

Realocación de un arreglo

- Cota ajustada usando análisis global
  - Costo de i-ésima operación es

$$c_i = \begin{cases} i & \text{si } i-1 \text{ es potencia de 2} \\ 1 & \text{si no} \end{cases}$$

  - Costo total:

$$\begin{aligned} \sum_{i=1}^{n} c_i &\leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \\ &< n + 2n \\ &= 3n \end{aligned}$$

Realocación de un arreglo

- Cota ajustada usando análisis global
  - Costo total: $O(n)$
  - Costo amortizado por operación: $O(1)$

Realocación de un arreglo

- Cota ajustada usando contabilidad de costos
  - ¿Intuición detrás del costo amortizado ‘3’ por cada operación?
  - Cada objeto paga por 3 inserciones (3 lucas)
    - Su inserción en el arreglo
    - Moverlo cuando el arreglo se expanda
    - Mover otro objeto que ya haya sido movido anteriormente en una expansión

Realocación de un arreglo

- Cota ajustada usando contabilidad de costos
  - Tamaño del arreglo es *m* después de expansión
    - *m*/2 objetos en el arreglo
    - Se pagan 3*m*/2 lucas hasta próxima expansión
      - *m*/2 lucas para objetos insertados
      - *m* lucas para realocar los *m* objetos
  - Costo amortizado por operación: $O(1)$

Realocación de un arreglo

- Cota ajustada usando función potencial
  - $\phi(T)$=0 después de una expansión
  - Función potencial:

$$\phi(T) = 2 \cdot num[T] - size[T]$$

    - Después de expansión num[T]=size[T]/2 y $\phi(T)$=0
    - Justo antes: num[T]=size[T] y $\phi(T)$=num[T]
    - $\phi(D_0)$=0 y num[T]>=size[T]/2, esto implica $\phi(T)$>=0
    - Por todo esto, costo amortizado es cota superior del costo real

Realocación de un arreglo

- Cota ajustada usando función potencial
  - Sean:
    - num_i: objetos en arreglo después de i-ésima operación
    - $size_i$: tamaño arreglo después de i-ésima operación
    - ϕ_i: potencial después de i-ésima operación
    - Inicialmente: $num_0 = size_0 = \phi_0 = 0$

Realocación de un arreglo

- Cota ajustada usando función potencial
  - Si inserción no expande arreglo
    - $size_i = size_{i-1}$
  - Costo amortizado por operación:

$$\begin{aligned}\hat{c}_i &= c_i + \phi_i - \phi_{i-1} \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\ &= 1 + (2 \cdot num_i - size_i) - (2 \cdot (num_i - 1) - size_i) \\ &= 3\end{aligned}$$

Realocación de un arreglo

- Cota ajustada usando función potencial
  - Si inserción expande arreglo
    - $size_i = 2*size_{i-1} = 2*(num_{i-1}) = 2*(num_i - 1)$
  - Costo amortizado por operación:

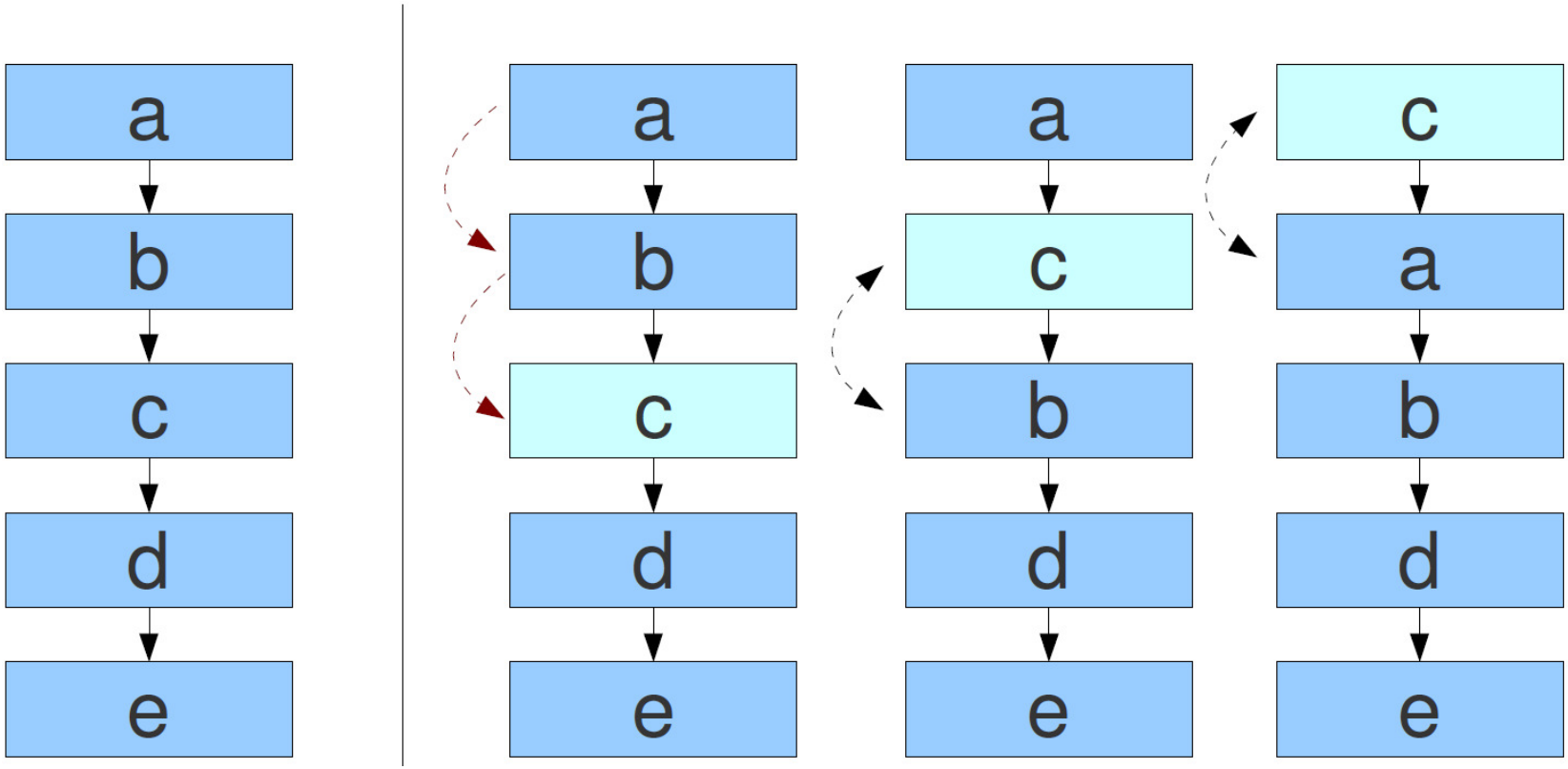
$$
\begin{aligned}
\hat{c}_i &= c_i + \phi_i - \phi_{i-1} \\
&= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
&= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2 \cdot (num_i - 1) - (num_i - 1)) \\
&= num_i + 2 - (num_i - 1) \\
&= 3
\end{aligned}
$$

Move-To-Front (MTF)

- Algoritmo de acceso a listas enlazadas desordenadas
- Idea: cada vez que se accesa un nodo de la lista, éste se lleva al comienzo de la lista
  - Costo de intercambiar dos nodos *contiguos* es constante
  - Costo de acceder nodo en posición *i*: *i*
  - Costo de mover al principio de la lista: *i*-1
  - Costo total: 2*i*-1

Move-To-Front (MTF)

- Ejemplo:



Move-To-Front (MTF)

- Usando análisis amortizado, se puede demostrar que
  - Costo MTF <= 4 * costo cualquier otro algoritmo
- Análisis:
  - Sea *A* un algoritmo arbitrario (e.g., el óptimo)
  - Se define el potencial de MTF en el tiempo *t* como dos veces el número de pares de nodos en orden inverso con respecto a *A*

Move-To-Front (MTF)

- Ejemplo de potencial:
  - *A*: (a,b,c,d,e) y MTF: (a,b,c,e,d) => potencial es 2 (por par (d,e))
- Notar que
  - Potencial al inicio=0 (ambas listas comienzan con el mismo orden por definición)
  - No es posible tener potencial negativo
  - => costo amortizado total es cota superior del costo actual de cualquier secuencia de acceso

Move-To-Front (MTF)

- Considere el análisis de acceder a *x*
  - Sea *k* la posición de *x* en MTF y sea *i* la posición de *x* en *A*
  - Costo de acceder y mover *x* en MTF es 2(*k*-1) y costo de acceder *x* en A es *i*
  - Mover *x* al frente de la lista invierte el orden de los pares (*x*,[1,*k*-1]) (*k*-1 pares en total); los otros pares mantienen su posición relativa

Move-To-Front (MTF)

- En la lista *A* hay *i*-1 nodos previos a *x*; éstos estarán después de *x* en MTF
  - A lo más hay min{*k*-1,*i*-1} inversiones *nuevas*
- Todas las otras inversiones (*k*-1 - min{*k*-1,*i*-1}) son inversiones que se *revierten*
- Cambio en potencial acotado superiormente por

$$\Delta\phi = 2\left(\min\{k-1, i-1\} - (k-1-\min\{k-1, i-1\})\right)$$
$$\Delta\phi = 4\min\{k-1, i-1\} - 2(k-1)$$

- Costo amortizado de acceder *x* es

$$\hat{c} = c + \Delta\phi = (2k-1) + 4\min\{k-1, i-1\} - 2(k-1) \leq 4\min\{k-1, i-1\} \leq 4i$$

Move-To-Front (MTF)

- Nota: *A* pudo realizar independientemente intercambios entre pares
  - Supongamos que *A* intercambia dos nodos
  - Esto podría incrementar/disminuir el potencial en 2 e incrementar el costo de acceso para *A* en 1
  - Costo amortizado aumenta a los más en dos, pero cota aumenta en cuatro => cota se mantiene
  - Esto es cierto para cualquier número de intercambios en *A*

Colas binomiales

- También conocido como “heaps mezclables”
- TDA que provee las operaciones de
  - Make-Heap()
  - Insert(H,x) (O(1) amortizado)
  - Minimum(H)
  - Extract-Min(H)
  - Union(H_1,H_2)
  - (Decrease-Key(H,x,k), Delete(H,x))

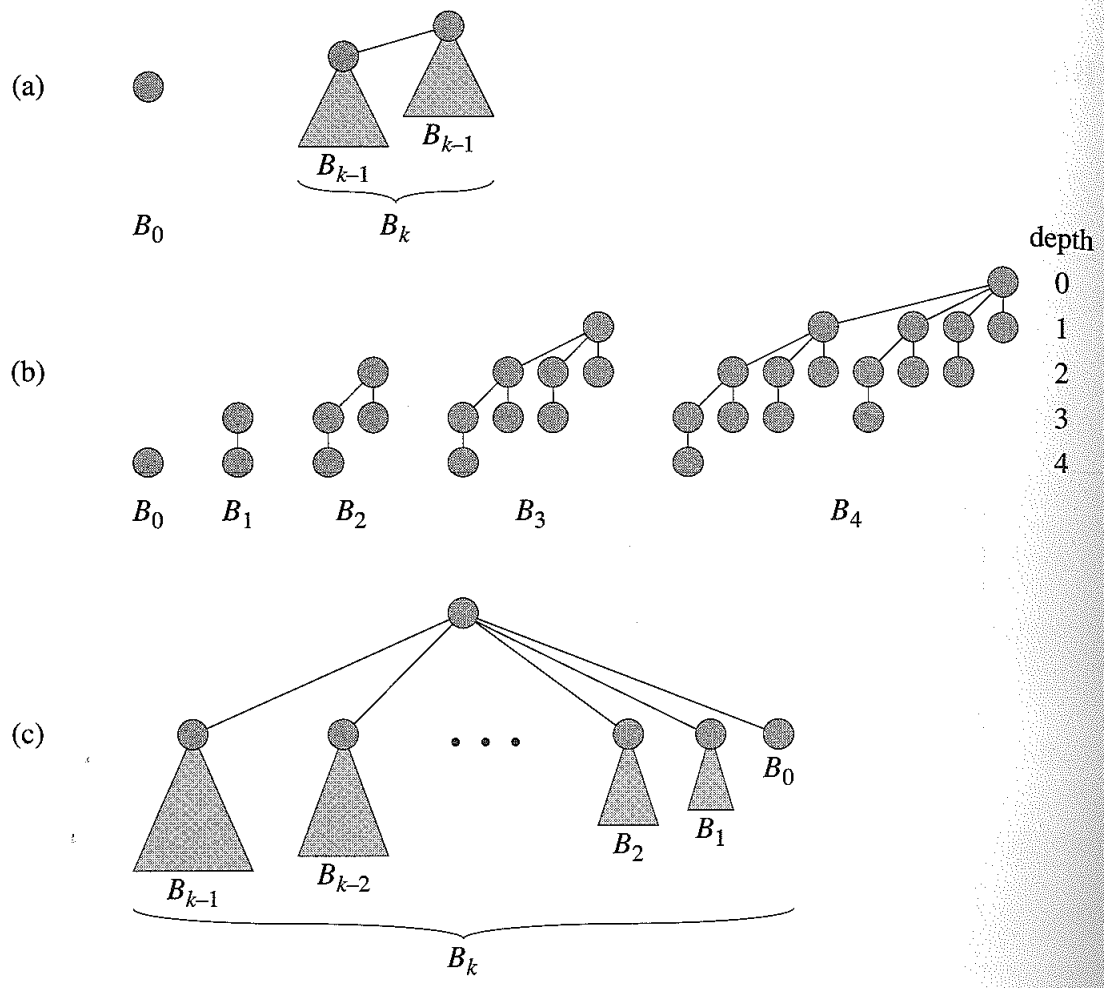
Colas binomiales

- Cola binomial es una colección de árboles binomiales
- Árbol binomial:
  - Árbol ordenado definido recursivamente
    - B_0: Un nodo solo
    - B_k: dos árboles binomiales B_{k-1} enlazados, la raíz de uno es el hijo más izquierdo del otro

Colas binomiales

- Ejemplo:

a) Definición recursiva
b) Ejemplo: B_0 a B_4
c) Árbol binomial B_k



Colas binomiales

- Propiedades del árbol binomial B_k:
  1. Tiene 2^k nodos
  2. Su altura es k
  3. Tiene exactamente $\binom{k}{i}$ nodos a profundidad i para i=0..k
  4. La raíz tiene grado k y es mayor que la de cualquier otro nodo
     - Si los hijos de la raíz se numeran de izq a der por k-1, k-2,…,0, el i-ésimo hijo es la raíz de un subárbol B_i

Colas binomiales

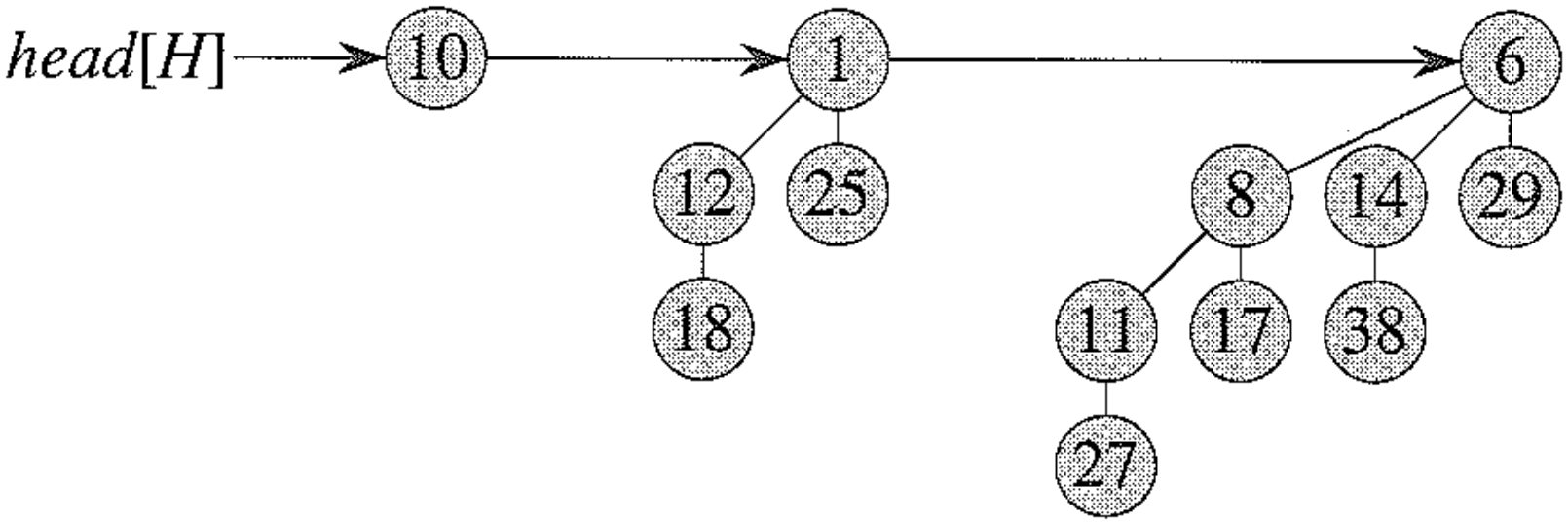
- Demostraciones (inducción sobre k)
  1. Árbol B_k consiste de dos copias de B_{k-1}, por lo tanto tiene $2^{k-1}+2^{k-1}=2^k$ nodos
  2. Profundidad máxima de B_k es uno más que de B_{k-1}. Por hipótesis de inducción esto es $(k-1)+1=k$
  3. Propuesto
  4. Único nodo con mayor grado en B_k que en B_{k-1} es la raíz, que tiene un hijo más que B_{k-1}. Raíz de B_{k-1} tiene grado $k-1$, entonces raíz de B_k tiene grado k
- Corolario: máximo grado de cualquier nodo en un árbol con n nodos es log n

Colas binomiales

- Cola binomial (*binomial heap*) *H*
  - Bosque de árboles binomiales que satisface:
    1. Cada árbol binomial en *H* obedece la propiedad de *min-heap*
    2. Para cada *k* no-negativo, hay a lo más un árbol binomial cuya raíz tiene grado *k*
- (1) indica que la raíz de cada árbol contiene la llave menor en ese árbol
- (2) indica que en una cola binomial de *n* nodos hay a lo más floor(log *n*)+1 árboles

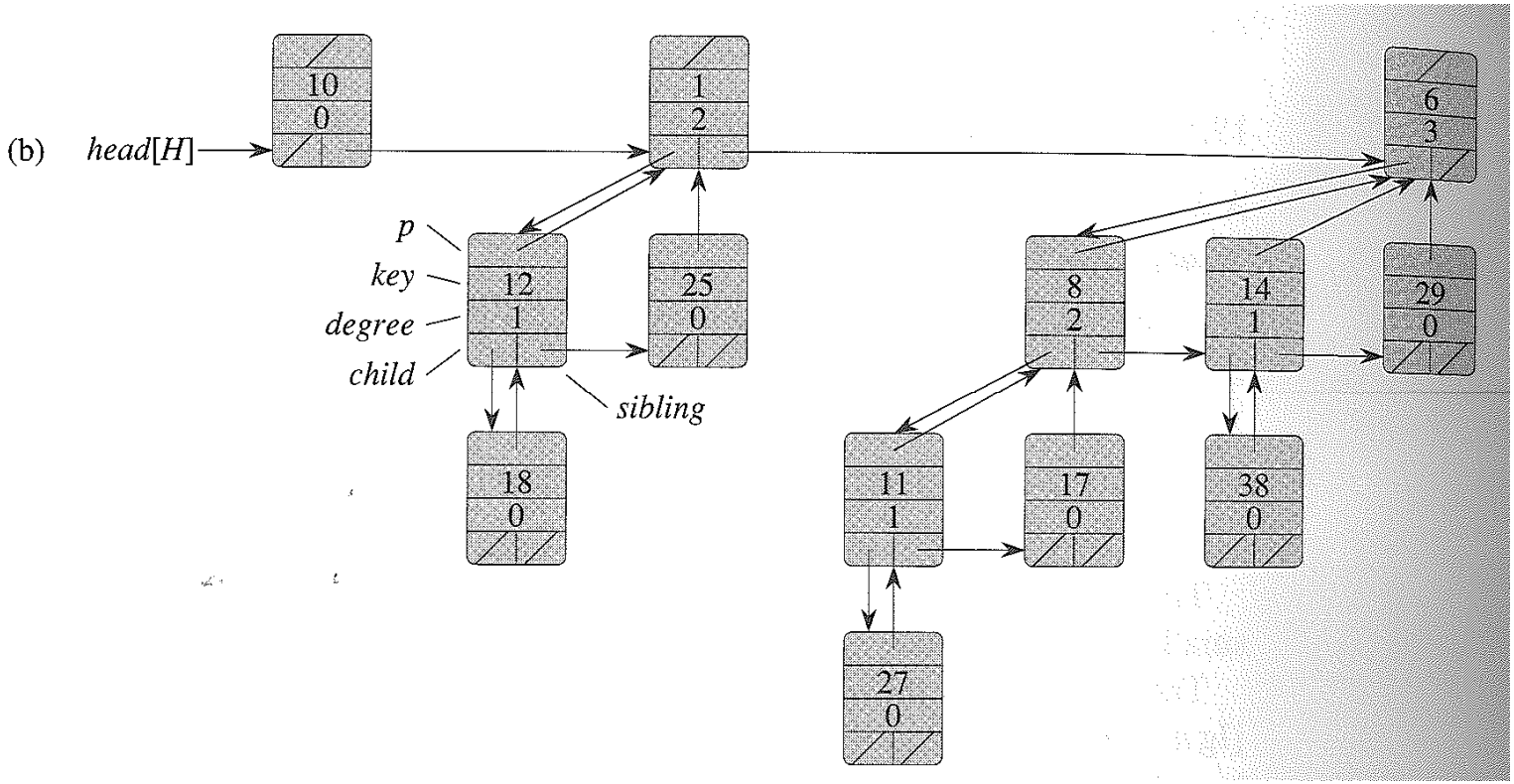
Colas binomiales

- Ejemplo (n=13, en binario 1101):



Colas binomiales

- Detalle de la estructura de datos



Colas binomiales

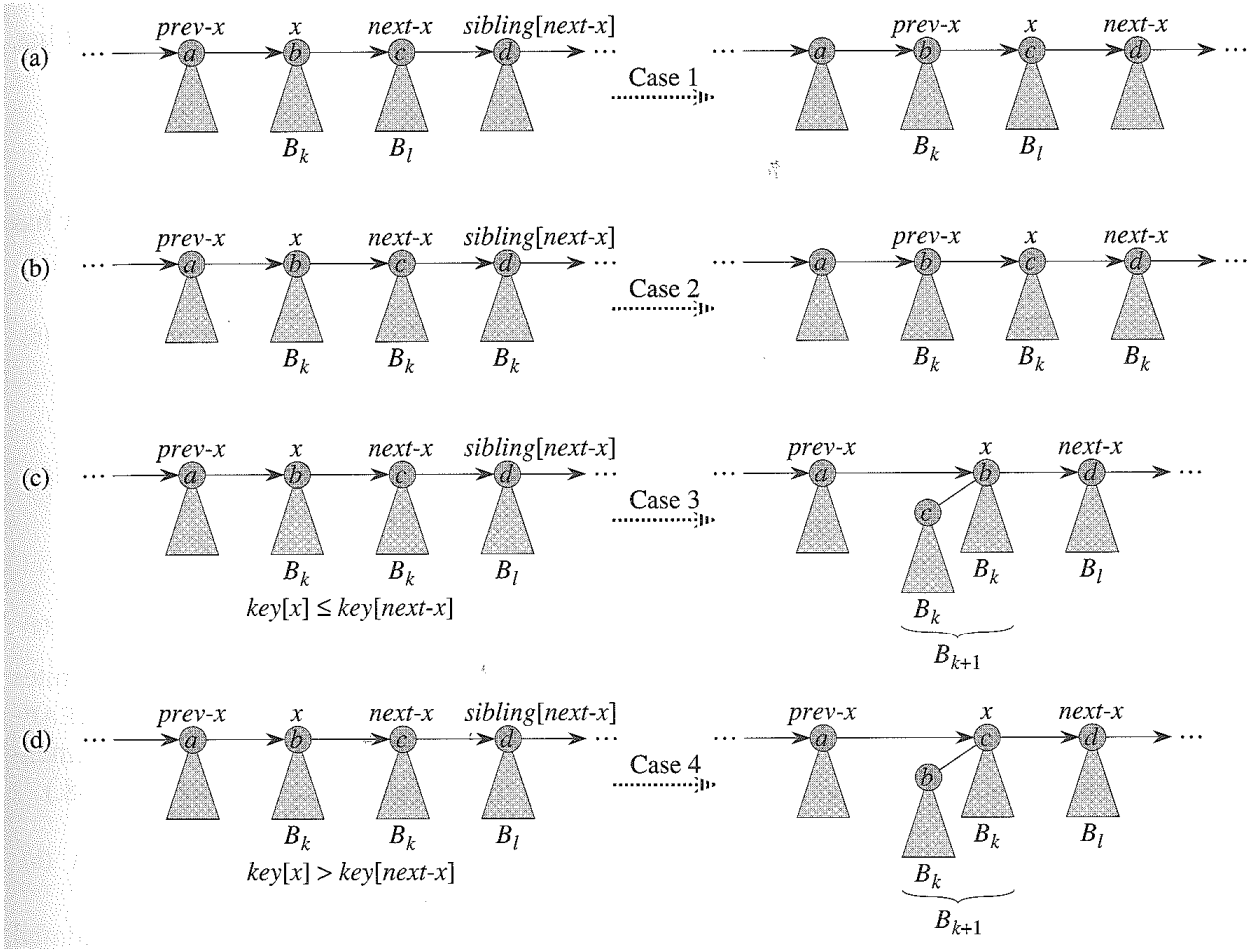
- Operaciones
  - Crear cola binomial
    - cabecera=NULL ($\Theta(1)$)
  - Encontrar el mínimo
    - Chequear todas las raíces ($\log(n) + 1 = O(\log n)$)
  - Insertar un nodo en H
    - Crear cola binomial con un solo nodo y unir con cola binomial H ($O(\log n)$)

Colas binomiales

- Operaciones
  - Unir dos colas binomiales H_1 y H_2
    1. Crear cola binomial *H*
    2. Cabecera de H = mezcla(H_1,H_2)
    3. Definir prev-x, x, next-x
    4. Mientras next-x != null
       5. Si (grado[x] != grado[next-x]) o (x, next-x, next-next-x tienen igual grado) entonces avanzar en la lista (Casos 1 y 2)
       6. Sino, enlazar x con next-x y avanzar en lista (Casos 3 y 4)
    7. Retornar *H*

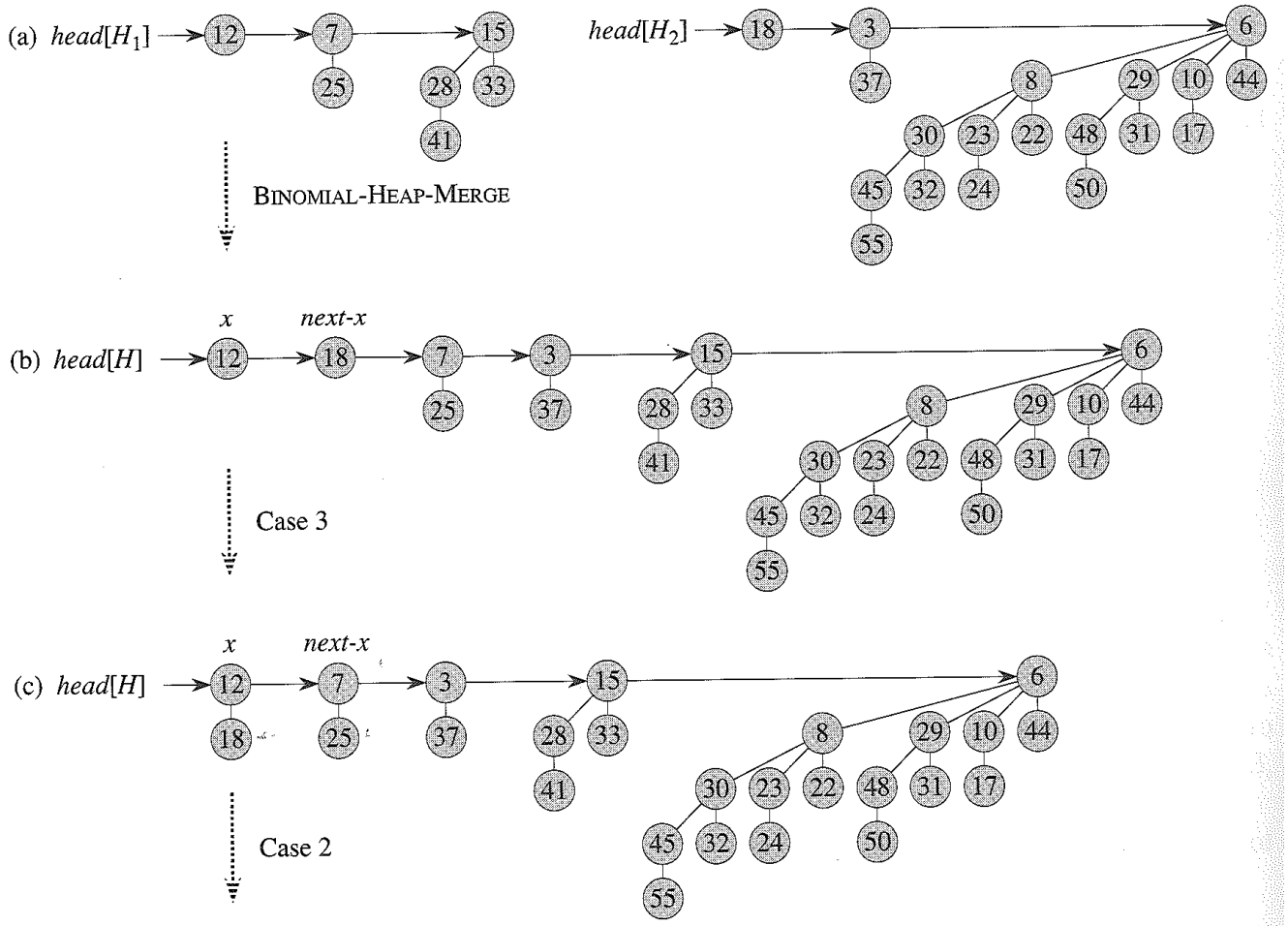
Colas binomiales

- Casos 1 a 4:



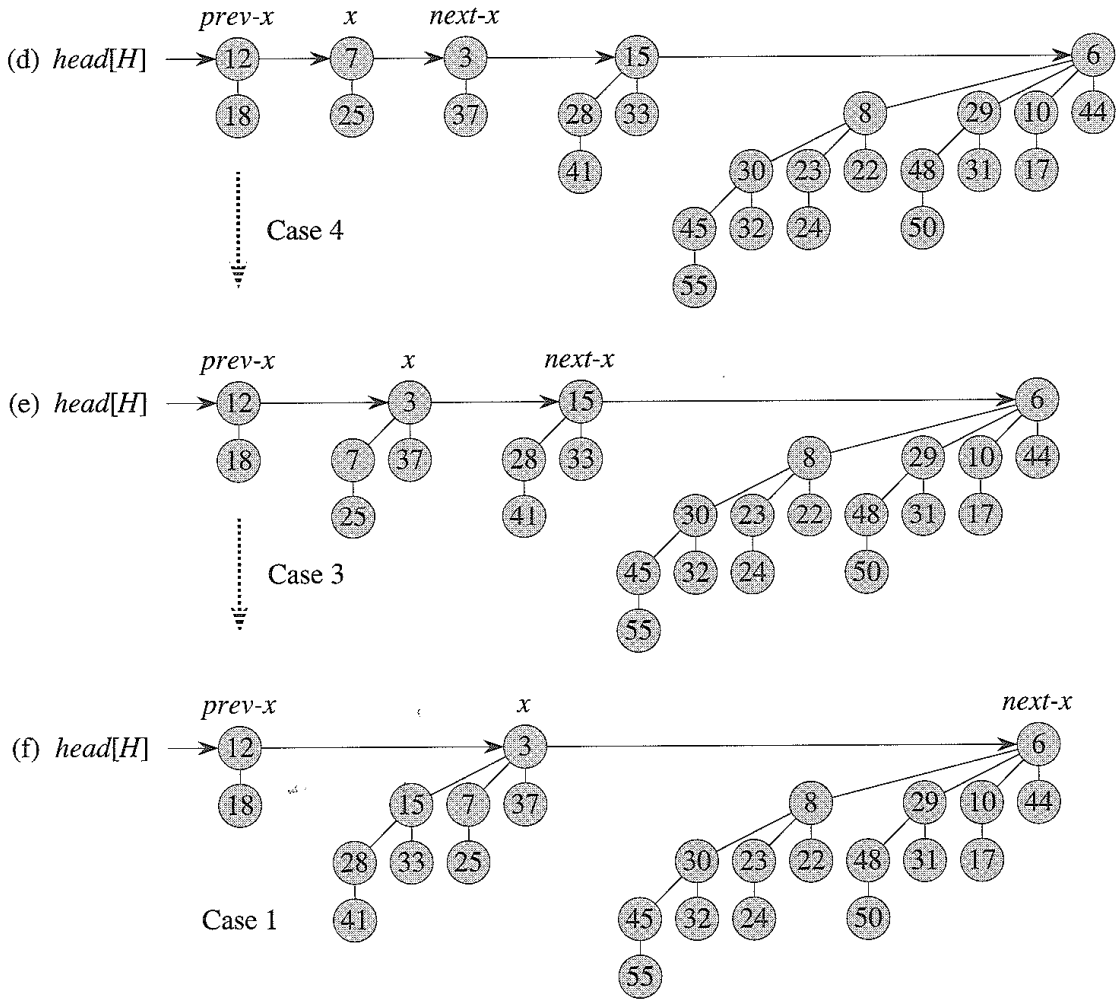
Colas binomiales

- Ejemplo:



Colas binomiales

- Ejemplo:



Colas binomiales

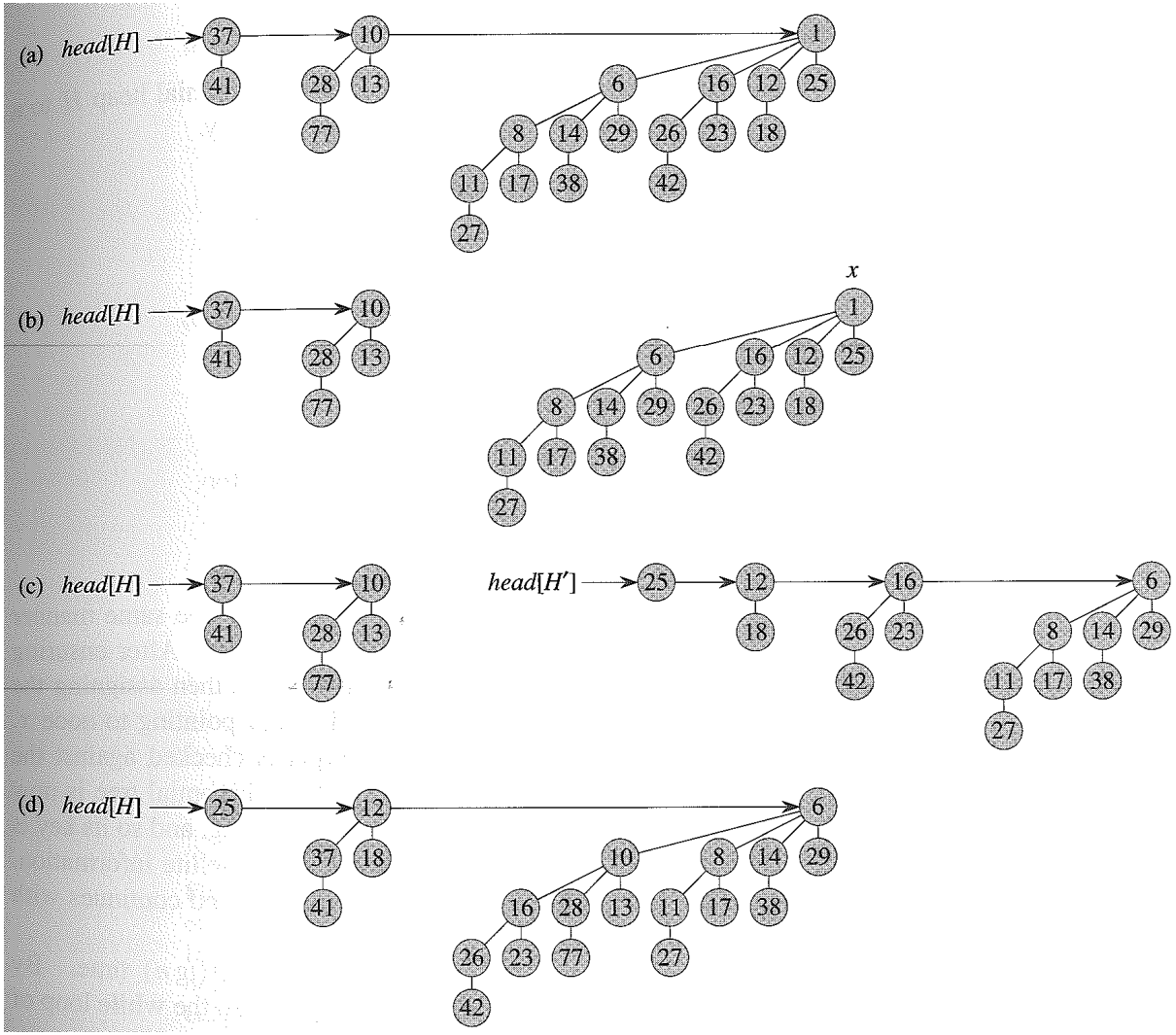
- Operaciones
  - Costo de unir dos colas binomiales:
    - Mezcla toma tiempo $O(\log n)$
      - $\log(n_1)+\log(n_2)+2 <= 2 \log(n) + 2$
    - Cada iteración toma tiempo $O(1)$ y a lo más hay $\log(n_1) + \log(n_2) + 2$ iteraciones => $O(\log n)$
    - Tiempo total: $O(\log n)$

Colas binomiales

- Operaciones
  - Extraer el mínimo
    1. Encontrar árbol *x* que contiene el mínimo, removerlo de la lista de *H*
    2. Crear cola binomial *H'*
    3. Revertir orden de los hijos de *x*, setear cabecera de *H'* como la cabecera de la lista resultante
    4. Unir *H* con *H'*
  - (1) a (4) toman tiempo $O(\log n)$ cada uno, en total toma tiempo $O(\log n)$

Colas binomiales

- Ejemplo:



Colas binomiales

- Observación: relación entre algoritmo de inserción en una cola binomial e incrementar en uno el contador binario
  - Se puede reemplazar llamada a Union
- Análisis amortizado del costo de inserción en cola binomial con nuevo algoritmo
  - Idéntico al contador binario
  - Costo amortizado por inserción: $O(1)$

Fibonacci heap

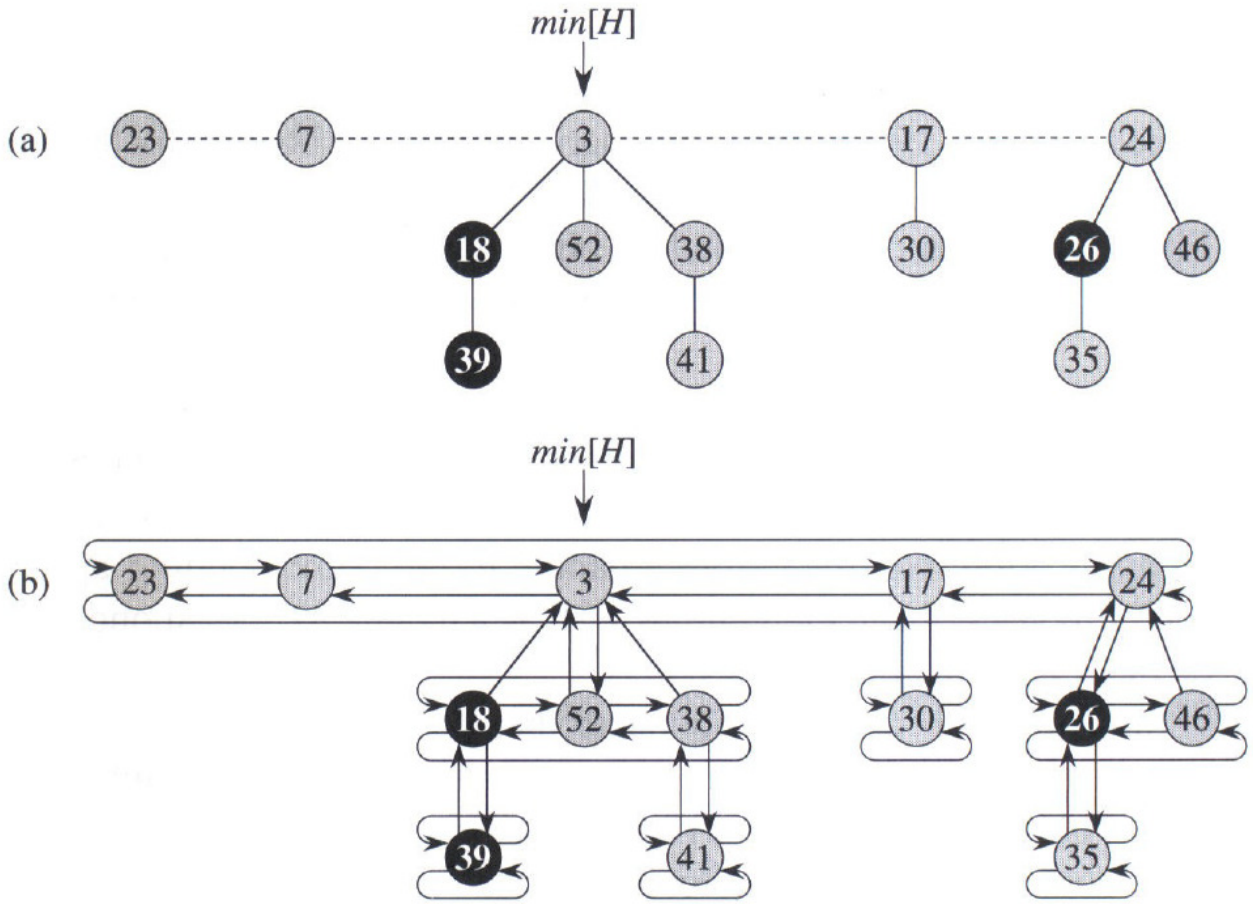
- Colección de árboles min-heap (como las colas binomiales)
- Diferencias:
  - Árboles no están restringidos a ser binomiales
  - Árboles no están ordenados

Fibonacci heap

- Estructura de datos del heap (*H*)
  - Puntero al padre
  - Puntero a uno de sus hijos (cualquiera)
  - Hijos de *x* forman una lista circular doblemente enlazada
  - Número de hijos
  - Marca de nodos (*True* o *False*)
  - *Root list*: lista de árboles en *H*, puntero a la raíz menor (*min*[*H*])
  - Número de nodos en el heap (*n*[*H*])

Fibonacci heap

- Ejemplo:



Fibonacci heap

- Función potencial
  - *t*(*H*): número de árboles en la root list
  - *m*(*H*): número de nodos marcados en H

$$\phi(H) = t(H) + 2m(H)$$

  - Potencial del ejemplo: 5+2*3=11
  - Potencial de heap vacío = 0
  - Potencial siempre positivo

Fibonacci heap

- Grado máximo de un nodo: $O(\log n)$
- Si Fibonacci heap sólo implementa funciones Make-Heap, Insert, Minimum, Extract-Min, Union
  - Fibonacci heap es una colección de árboles binomiales “desordenados”
  - Propiedades en *slide* 45 se cumplen (con pequeña variación en Propiedad 4)

Fibonacci heap

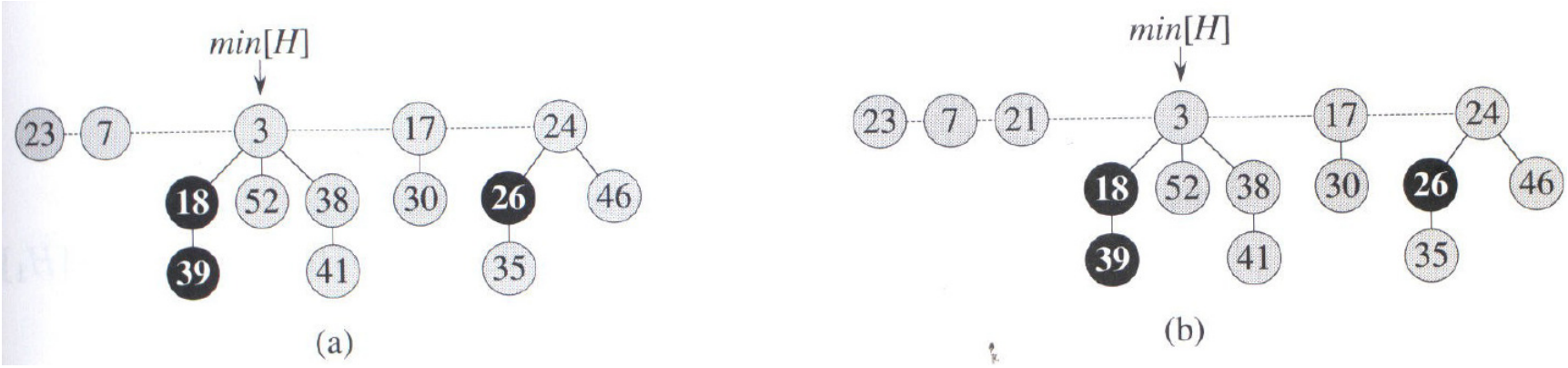
- Idea clave de operaciones de mezcla de heaps en Fibonacci heaps: retardar el trabajo lo máximo posible
- Make-Fib-Heap
  - Crea heap vacío
  - $n[H]=0$
  - $min[H]=$NULL
  - Costo: $O(1)$

Fibonacci heap

- Fib-Heap-Insert
  - Agrega *x* al root list
    - Hermano izquierdo de la raíz
    - Padre NULL
    - Hijos NULL
    - Marca de *x* = *False*
    - *n*[*H*]=*n*[*H*]+1
    - Si *min*[*H*]==NULL o key(*x*)<*min*[*H*] => *min*[*H*]=key(*x*)
  - Cambio potencial = 1 => Costo *O*(1) amortizado

Fibonacci heap

- Fib-Heap-Insert
  - Ejemplo: insertar *x*, key(*x*)=21



  - Si se insertan *k* nodos consecutivamente, se agregan *k* árboles de un solo nodo al heap

Fibonacci heap

- Fib-Heap-Minimum
  - Puntero a *min*[*H*]
  - Costo: *O*(1)
- Fib-Heap-Union
  - Concatena los *root lists* de ambos heaps y determina el nuevo mínimo entre los dos candidatos
  - Costo: *O*(1)

Fibonacci heap

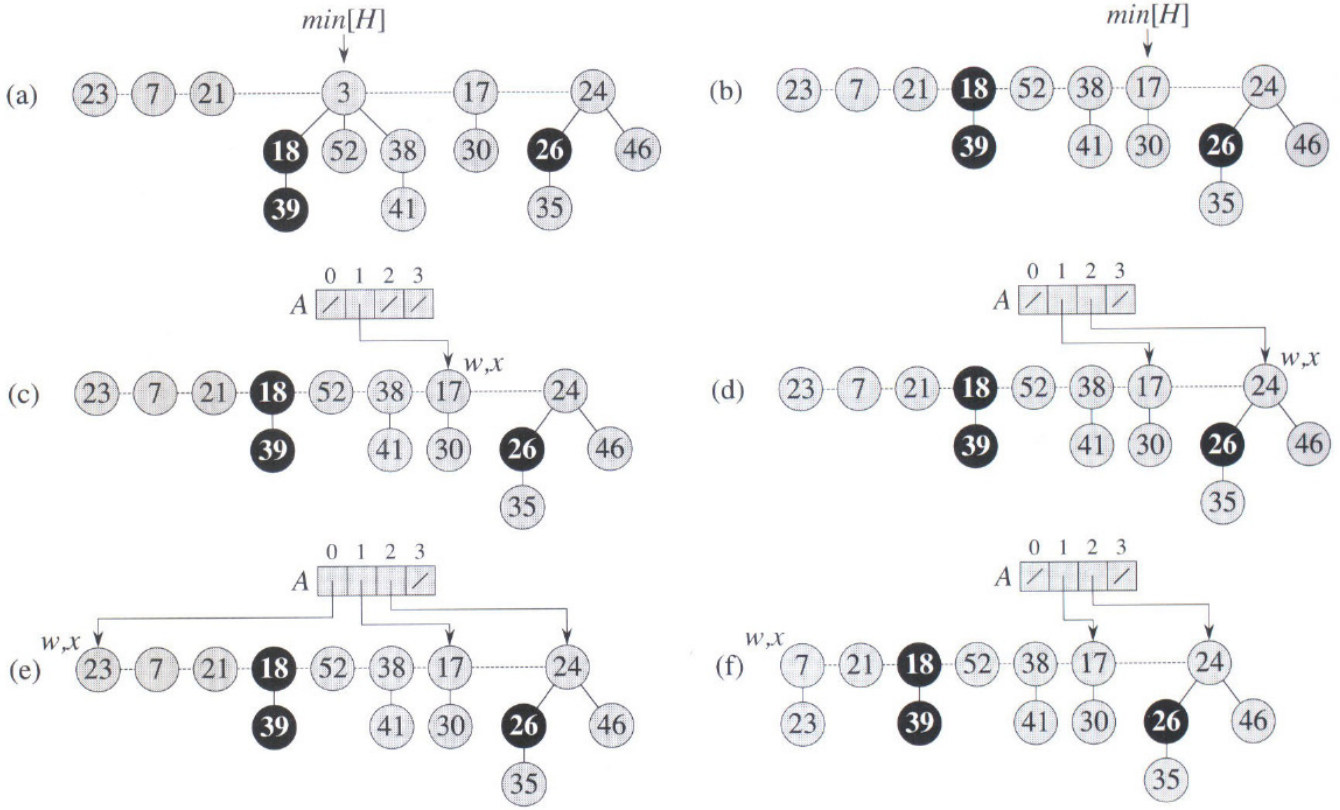
- Fib-Heap-Extract-Minimum
  - Aquí se realiza el trabajo postergado en las operaciones anteriores
  - Algoritmo:
    - Guardar puntero a *min*[*H*] en *z*
    - Si *z*!=NULL
      - Agregar cada hijo de *z* a la *root list*
      - Remover *z* de *H*
      - Invocar a Consolidar(*H*)
      - *n*[*H*]=*n*[*H*]-1
    - Retornar *z*

Fibonacci heap

- Fib-Heap-Extract-Minimum
  - Consolidar(*H*)
    - Reduce el número de árboles en *H*
    - Deja todas las raíces de la *root list* de grado distinto
  - Algoritmo
    - Encontrar dos raíces *x* e *y* de la *root list* de mismo grado, con key(*x*)<=key(*y*)
    - Enlazar *y* con *x*: y pasa a ser hijo de *x*
      - Remover *y* de root list
      - Enlazar *y* con *x*, incrementar grado de *x*
      - Marca de *y* = *False*

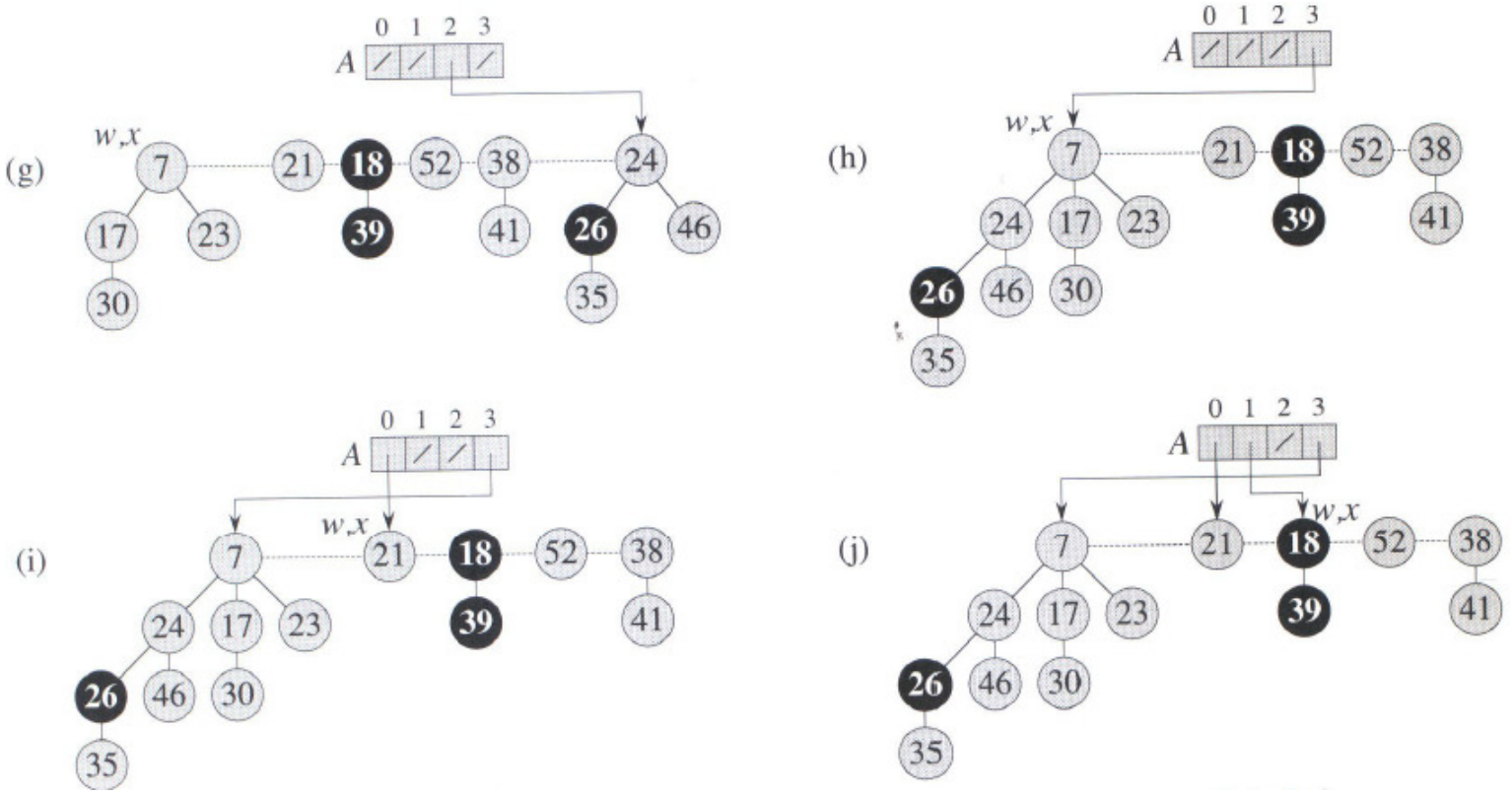
Fibonacci heap

- Ejemplo de Fib-Heap-Extract-Minimum



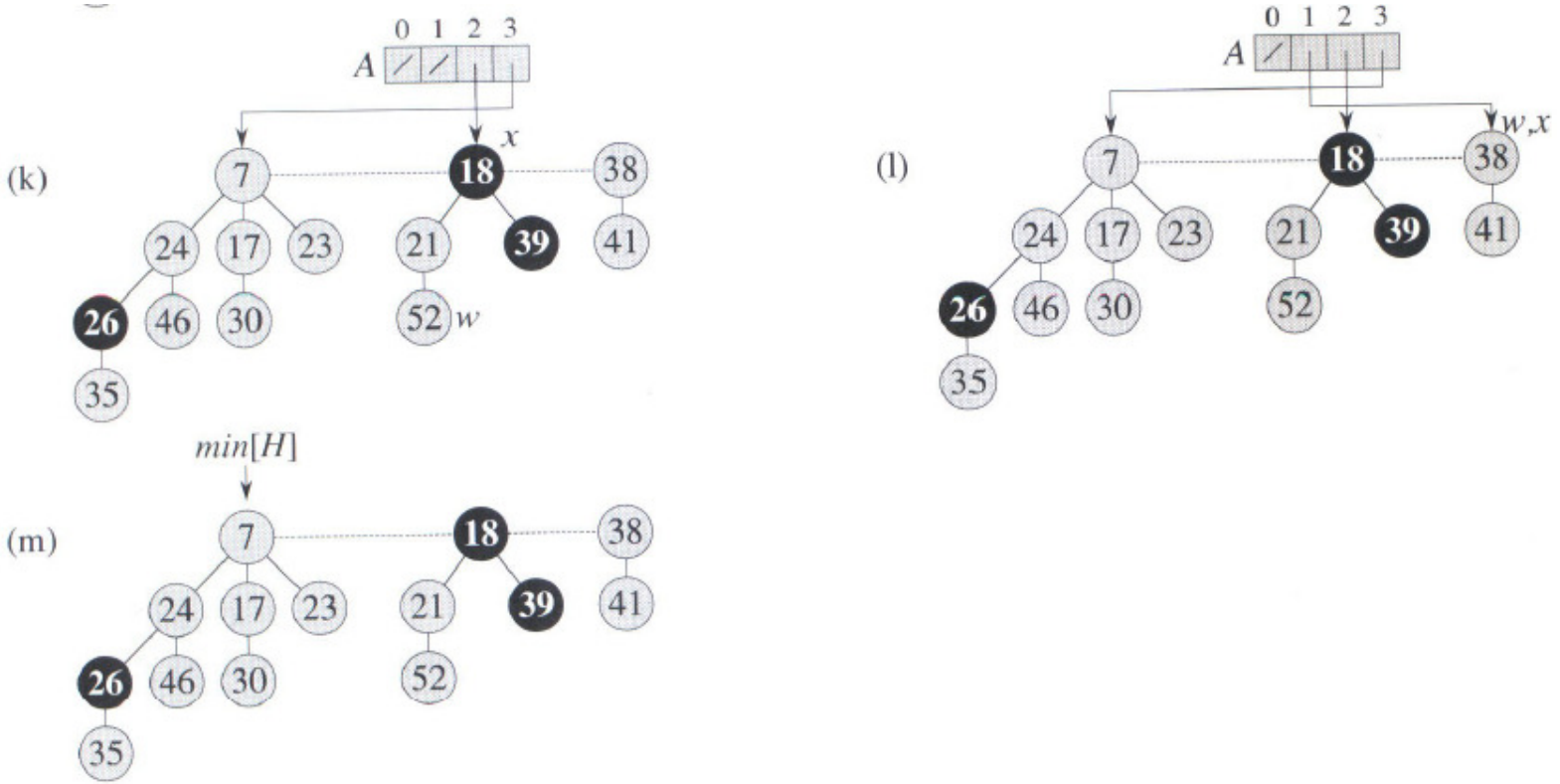
Fibonacci heap

- Ejemplo de Fib-Heap-Extract-Minimum



Fibonacci heap

- Ejemplo de Fib-Heap-Extract-Minimum



Fibonacci heap

- Costo de Fib-Heap-Extract-Minimum
  - Diferencia potencial = (log n + 1) – $t(H)$
    - Potencial antes de extraer mínimo: $t(H)+2m(H)$
    - Potencial después de extraer mínimo: acotado superiormente por (log n + 1)+$2m(H)$
  - Costo real de inserción = O(log n) + t(H)
    - A lo más log n hijos de $min[H]$ pasan a la root list
    - Consolidar(H) procesa a lo más log n + $t(H)$ -1 árboles
      - Cada iteración es de O(1)
  - => Costo amortizado: O(log n)

Fibonacci heap

- Otras operaciones posibles:
  - Fib-Heap-Decrease-Key y Fib-Heap-Delete
  - Marcan/desmarcan nodos
  - Se pierde garantía que todos los árboles en *H* son árboles binomiales desordenados
  - Se puede demostrar que tienen costo amortizado O(log *n*), log base ϕ = sección aurea

USO DE DOMINIOS DISCRETOS Y FINITOS

Dominios discretos

- En el Capítulo 1 del curso se estudió una cota $\Omega(n \log n)$ para algoritmos de ordenamiento basados en comparaciones
- Algoritmos que no se basan en comparaciones pueden romper dicha cota
- Se verán tres ejemplos
  - Utilizan el hecho de trabajar sobre dominios discretos

Counting sort

- Sea un arreglo A con n enteros
- Los enteros se encuentran en el rango [0,k]
- Sea k = O(n)
  - El arreglo A se puede ordenar en tiempo $\Theta(n)$
- Idea básica: determinar para cada valor x en A cuántos valores son menores que x
  - Esta información puede usarse para encontrar la posición de x en el arreglo ordenado

Counting sort

- Es necesario considerar los valores repetidos
- El algoritmo requiere:
  - A[1,n] : arreglo de entrada
  - B[1,n] : arreglo de salida
  - k : rango de los números
  - C[1,k] : arreglo auxiliar

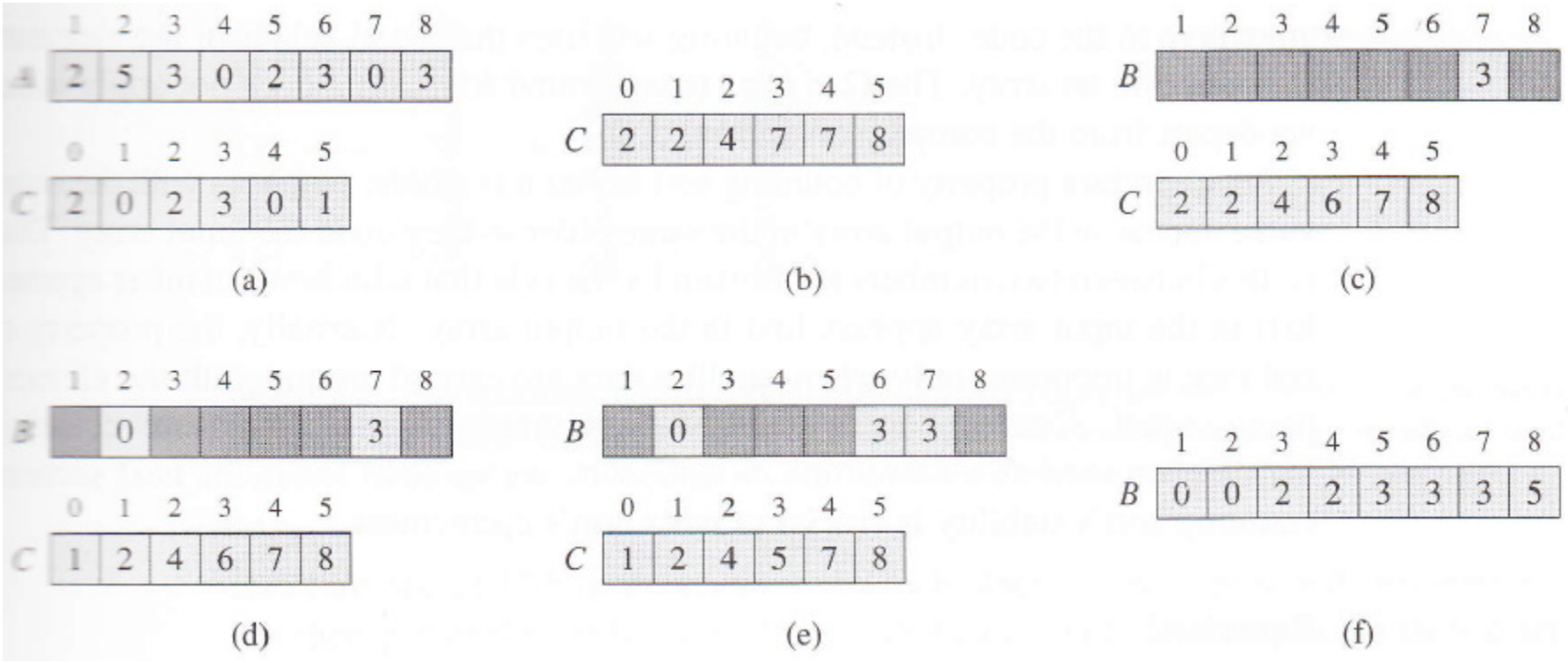
Counting sort

- Seudocódigo de Counting sort:

```
Counting-Sort(A,B,k)
1. for i <- 0 to k
2.     C[i] <- 0
3. for j <- 1 to n
4.     C[A[j]] <- C[A[j]] + 1
5. // C[i] contiene el numero de valores == i
6. for i <- 1 to k
7.     C[i] <- C[i] + C[i-1]
8. // C[i] contiene # valores <= que i
9. for j <- n downto 1
10.    B[C[A[j]]] <- A[j]
11.    C[A[j]] <- C[A[j]] - 1
```

Counting sort

- Ejemplo:



Counting sort

- Análisis de Counting sort
  - Ciclo for de lineas 1-2: $\Theta(k)$
  - Ciclo for de líneas 3-4: $\Theta(n)$
  - Ciclo for de líneas 6-7: $\Theta(k)$
  - Ciclo for de líneas 9-11: $\Theta(n)$
  - Tiempo total: $\Theta(n+k)$
  - Dado que $k = O(n)$
    - Tiempo total = $\Theta(n)$

Counting sort

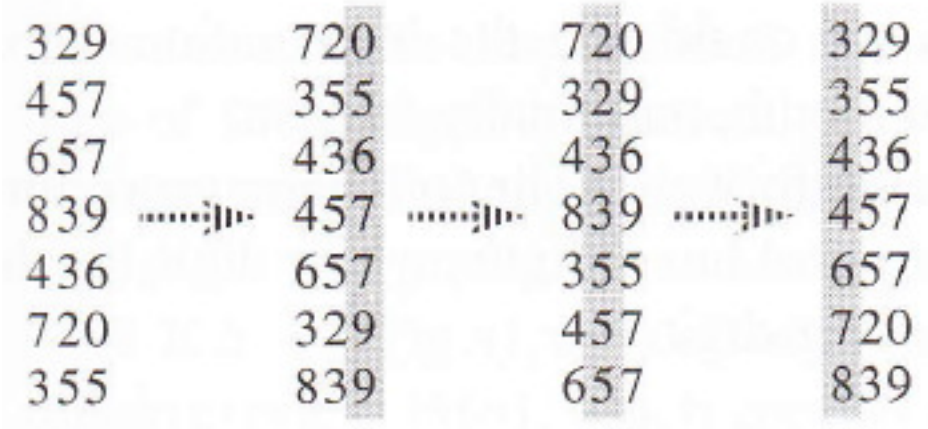
- Notas
  - Counting sort no realiza ninguna comparación entre elementos
  - Usa los valores de los elementos como índices en un arreglo
  - Counting sort es estable: si un valor en el arreglo A se repite, las ocurrencias aparecen en el mismo orden en el arreglo ordenado B
  - Counting sort no es in-place (requiere espacio extra)

Radix sort

- Sean números con d dígitos
- Problema: ordenar n números de d dígitos
- Idea de radix sort (contraintuitiva):
  - Ordenar los números primero por el dígito menos significativo
  - Combinar los números en orden (primero aquellos con digito 0, luego los con digito 1, etc.)
  - Repetir proceso con el siguiente digito significativo

Radix sort

- Cuando se ordena por el último dígito, los valores están ordenados
  - Se requieren d “pasadas” para ordenar el conjunto
- Ejemplo:



Radix sort

- Es esencial que al ordenar por cada dígito el ordenamiento sea estable
  - Se puede usar Counting sort para ordenar por dígito
- Seudocódigo:

```
Radix-Sort(A,d)
1. for i <- 1 to d
2.     ordenar A con algoritmo estable usando digito i
```

Radix sort

- Análisis
  - Radix-Sort toma tiempo $\Theta(d(n+k))$
    - Depende del algoritmo usado en cada iteración
    - Si el dígito está en el rango [0, k] y k no es muy grande, se utiliza Counting sort
    - Cada pasada sobre n números de d dígitos toma $\Theta(n+k)$
    - Se realizan d pasadas, tiempo total = $\Theta(d(n+k))$
  - Si d es constante y k = O(n), radix sort se ejecuta en tiempo lineal

Bucket sort

- Bucket sort toma tiempo lineal en promedio si la distribución de valores es uniforme
  - Supone que la distribución es uniforme en el intervalo [0,1)
- Idea:
  - Dividir el intervalo [0,1) en n intervalos equi-espaciados (buckets)
  - Distribuir los n números en los buckets
  - Ordenar cada bucket y recorrerlos en orden

Bucket sort

- Para ordenar cada bucket se utiliza ordenación por inserción
  - Ordenación por inserción es $O(n^2)$
  - Deberían haber pocos valores por bucket, por lo que no debiera costar mucho ordenarlos
- Supuestos
  - Arreglo A de tamaño n
  - 0 <= A[i] < 1
  - Arreglo auxiliar B[0,n-1] para los buckets (listas)

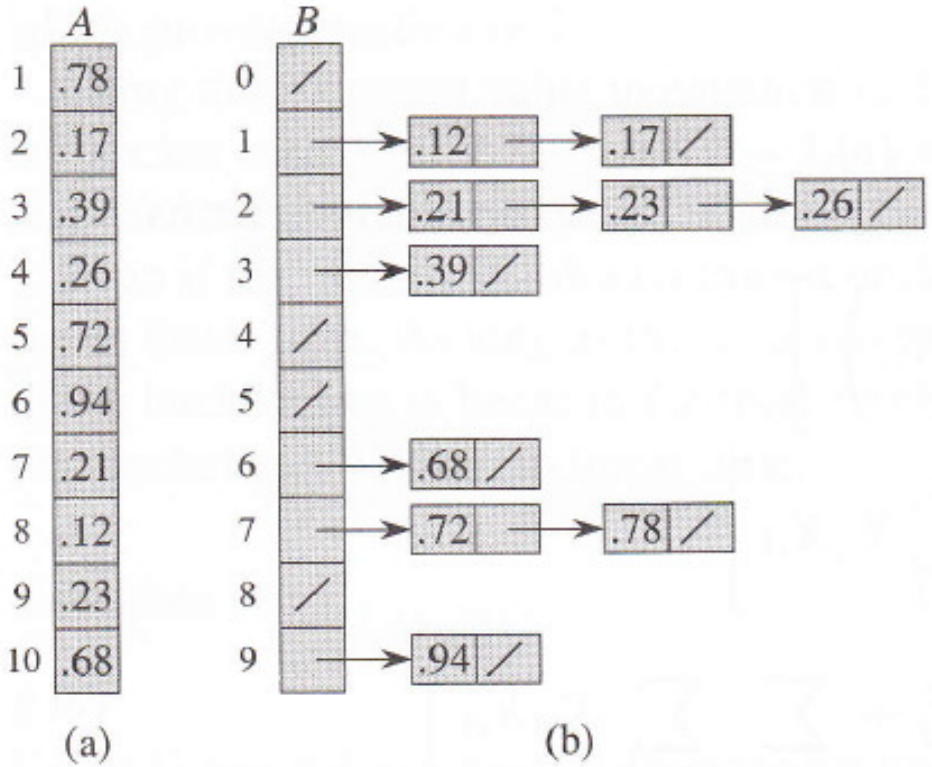
Bucket sort

- Seudocódigo:

```
Bucket-Sort(A)
1. n <- length(A)
2. for i <- 1 to n
3.     insert A[i] into list B[floor(n*A[i])]
4. for i <- 0 to n-1
5.     sort list B[i] with insertion sort
6. concatenate lists B[0],…,B[n-1] in order
```

Bucket sort

- Ejemplo:



Bucket sort

- Análisis
  - Todas las líneas del seudocódigo se ejecutan en tiempo O(n), excepto la línea 5
  - Sea n_i el número de valores que caen en el bucket i
  - Dado que inserción toma tiempo cuádrático:

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Bucket sort

- Análisis
  - Tiempo promedio (valor esperado)

$$
\begin{aligned}
\mathrm{E}[T(n)] &= \mathrm{E}\left[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] \\
&= \Theta(n) + \sum_{i=0}^{n-1} \mathrm{E}[O(n_i^2)] \\
&= \Theta(n) + \sum_{i=0}^{n-1} O\left(\mathrm{E}[n_i^2]\right)
\end{aligned}
$$

Bucket sort

- Análisis
  - Sea la variable indicadora X_{ij} = I{A[j] cae en el bucket i}
    - i = [0, n-1]
    - j = [1, n]
  - Se tiene que

$$n_i = \sum_{j=1}^{n} X_{ij}$$

Bucket sort

- Análisis
  - Expandiendo y reagrupando términos:

$$
\begin{aligned}
\mathrm{E}[n_i^2] &= \mathrm{E}\left[\left(\sum_{j=1}^{n} X_{ij}\right)^2\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}\right] \\
&= \mathrm{E}\left[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^{n}\mathrm{E}[X_{ij}^2] + \sum_{1\le j\le n}\ \sum_{\substack{1\le k\le n \\ k\ne j}} \mathrm{E}[X_{ij}X_{ik}]
\end{aligned}
$$

Bucket sort

- Análisis
  - Esperanza de una variable indicadora es su probabilidad:

$$\begin{aligned} \mathrm{E}[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\ &= \frac{1}{n} . \end{aligned}$$

$$\begin{aligned} \mathrm{E}[X_{ij} X_{ik}] &= \mathrm{E}[X_{ij}]\mathrm{E}[X_{ik}] \\ &= \frac{1}{n} \cdot \frac{1}{n} \\ &= \frac{1}{n^2} . \end{aligned}$$

Bucket sort

- Análisis
  - Substituyendo y calculando la esperanza:

$$
\begin{aligned}
\mathrm{E}[n_i^2] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{\substack{1 \le k \le n \\ k \ne j}} \frac{1}{n^2} \\
&= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n},
\end{aligned}
$$

Bucket sort

- Análisis
  - Finalmente, el tiempo total es:

$$\Theta(n) + n \cdot O(2 - 1/n) = \Theta(n)$$

  - Bucket sort toma en promedio tiempo lineal
    - Esto es válido incluso si los valores no provienen de una distribución uniforme, siempre que se cumpla que la suma de los cuadrados de los tamaños de los buckets sea lineal

Tries

- Suponga que los elementos de un conjunto se pueden representar como una secuencia de bits (o una secuencia de caracteres)
  - $X = b_0 b_1 b_2 ... b_k$
- Se cumple que ninguna representación de un elemento particular es prefijo de otra
  - Esto es cierto si todas las representaciones son del mismo largo, o si terminan con una marca especial (por ejemplo, “$”)

Tries

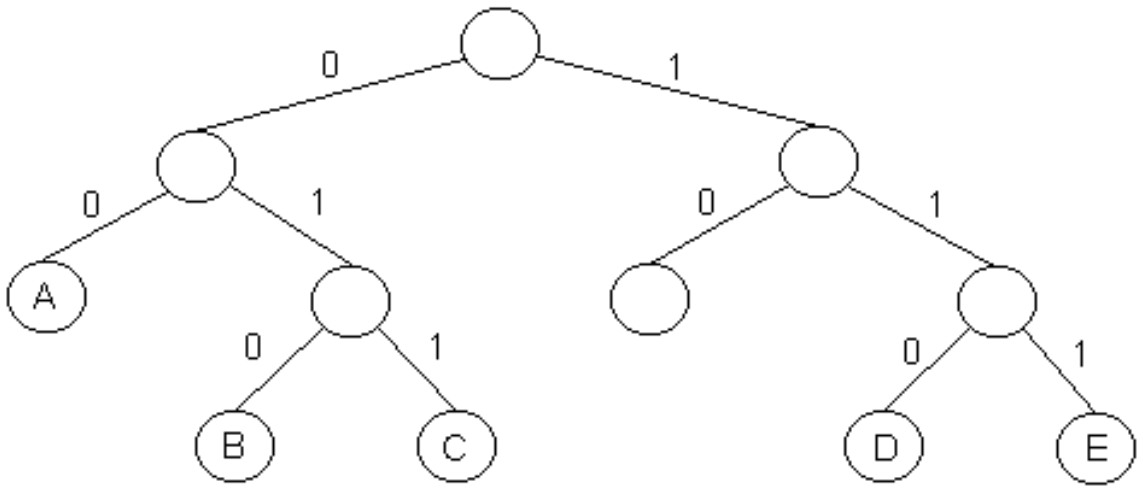
- Un trie es un árbol binario en donde:
  - La posición de inserción de un elemento ya no depende de su valor, sino de su representación binaria
  - Los elementos en un trie se almacenan sólo en sus hojas, pero no necesariamente todas las hojas contienen elementos
- Trie viene de “retrieval”, se pronuncia de forma que rime con “pie”

Tries

- Ejemplo:

Codificación:

A = 00100
B = 01000
C = 01111
D = 11000
E = 11101



Tries

- Un trie provee la siguientes operaciones
  - Insert
  - Delete
  - Print (muestra los miembros del conjunto)
- En un trie, un camino de la raíz a una hoja corresponde a una palabra del conjunto
  - Los nodos del trie corresponden a los prefijos de las palabras en el conjunto

Tries

- Algoritmo de búsqueda
  - Se examinan los bits b_i del elemento X, partiendo desde b_0 en adelante
  - Si $b_i = 0$ se avanza por la rama izquierda y se examina el siguiente bit, b_{i+1}
  - Si $b_i = 1$ se avanza por la rama derecha y se examina el siguiente bit
  - El proceso termina cuando se llega a una hoja, único lugar posible en donde puede estar insertado X (es necesario verificarlo)

Tries

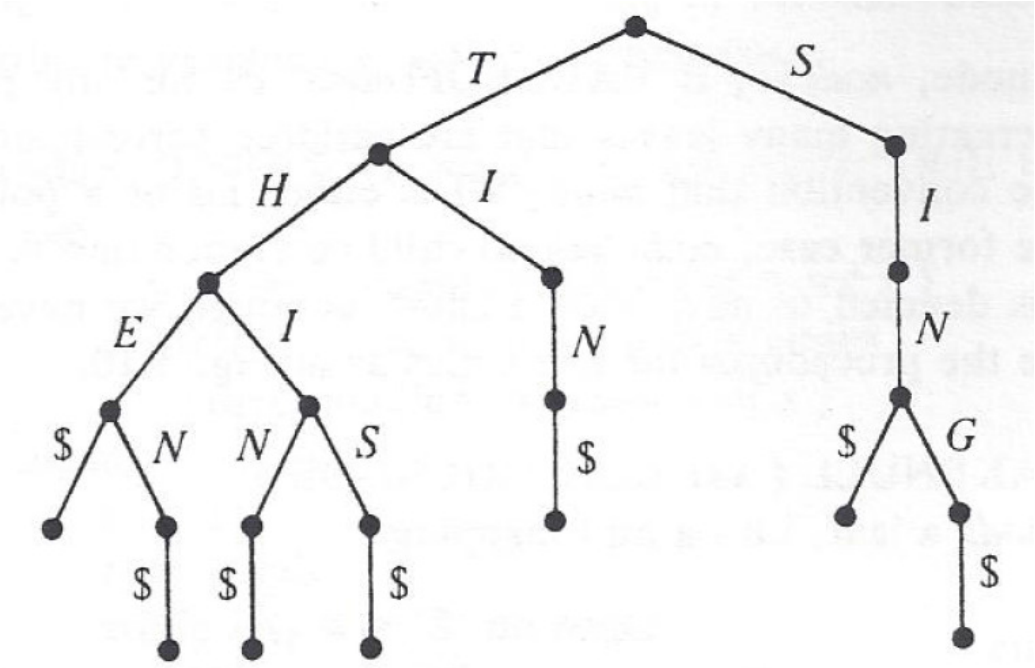
- Algoritmo de inserción
  - Suponga que se desea insertar el elemento X
    - Se realiza una búsqueda infructuosa de X hasta llegar a una hoja, suponga que el último bit utilizado en la búsqueda fue b_k
    - Si la hoja esta vacía, se almacena X en dicha hoja
    - En caso contrario, se divide la hoja utilizando el siguiente bit del elemento, b_{k+1}, y se repite el procedimiento, si es necesario, hasta que quede sólo un elemento por hoja

Tries

- Algoritmo de inserción
  - Note que con este proceso de inserción, la forma que obtiene el árbol digital es insensible al orden de inserción de los elementos
- Algoritmo de eliminación
  - Suponga que el elemento a eliminar es X, se elimina el elemento de la hoja y ésta queda vacía
  - Mientras el nodo padre tenga un como hijos una hoja vacía y una hoja con datos, se borra el padre y el nodo “abuelo” es el nuevo padre del elemento

Tries

- Otro ejemplo: el conjunto es {THE, THEN, THIN, THIS, TIN, SIN, SING}



Tries

- Observaciones sobre el ejemplo (suponer alfabeto para el idioma inglés):
  - Cada nodo del trie tiene a lo más 27 hijos, uno por cada letra más el símbolo $
  - La mayoría de los nodos tiene menos de 27 hijos
  - Una hoja con el símbolo $ no puede tener hijos
- Pregunta: Diseñe un algoritmo para retornar todas las palabras que comienzan con el mismo prefijo

Tries

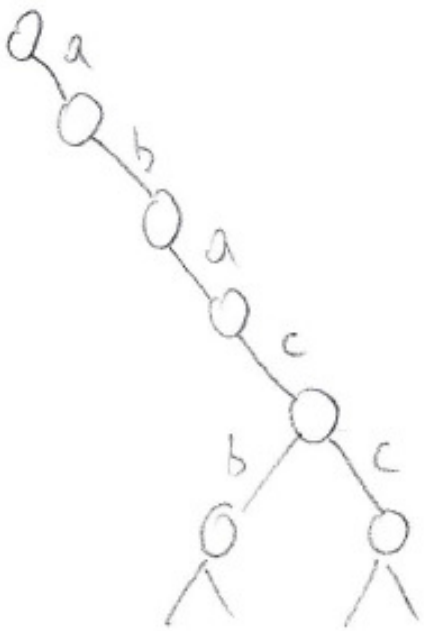
- Problema de los tries: caminos unarios
  - Cómo sería el trie para los strings:
    - abacb
    - abacc

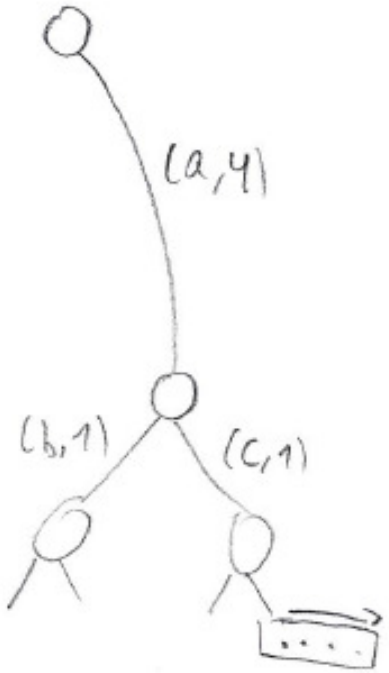
Patricia trees

- PATRICIA: Practical Algorithm To Retrieve Information Coded in Alphanumerics
  - Donald Morrison, 1968
- Un Patricia tree es un trie que
  - Garantiza que la cantidad de nodos no-hojas es menor que la cantidad de hojas
  - Elimina los caminos unarios
    - El arco indica (rótulo) cuántos caracteres se avanza al ir al nodo hijo

Patricia trees

- Trie vs. Patricia tree





Obs.: Padre de nodo hoja

Patricia trees

- Trie vs. Patricia tree
  - Trie:

  - Patricia tree:

Patricia trees

- Búsqueda en un Patricia tree
  - Tiempo proporcional al largo del patrón

```
Search-Patricia-tree(root,P) // P[1,m]
1.  i <- 1
2.  v <- root
3.  while (v no sea hoja) and (i <= m)
4.    if v no tiene hijo con rótulo P[i]
5.      return NULL
6.    u <- hijo de v con rótulo (P[i],t)
7.    i <- i + t
8.    v <- u
9.  if v no es hoja return NULL
10. if contenido de v == P return v
11. else return NULL
```

Patricia trees

- Búsqueda de prefijos en un Patricia tree
  - Linea 9: revisar un hijo de v, si cumple con el prefijo todo el subárbol con raíz v cumple

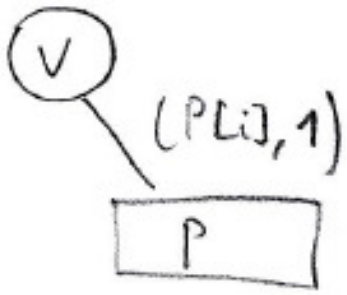
```
Prefix-Patricia-tree(root,P) // P[1,m]
1.  i <- 1
2.  v <- root
3.  while (v no sea hoja) and (i <= m)
4.    if v no tiene hijo con rótulo P[i]
5.      return NULL
6.    u <- hijo de v con rótulo (P[i],t)
7.    i <- i + t
8.    v <- u
9.  if v no es hoja, revisar un hijo de v, return correspondiente
10. if contenido de v == P return v
11. else return NULL
```

Patricia trees

- Inserción en un Patricia tree
  - Crear nodo P
  - Buscar P y no encontrarlo
  - Determinar P[i] (punto) del camino donde falla la búsqueda

Patricia trees

- Inserción en un Patricia tree
  - Volver a bajar consumiendo i-1 caracteres
    - Se llegó a nodo interno v (implica que no tiene hijo P[i])
      - Colgar P de v con el rótulo (P[i],1)

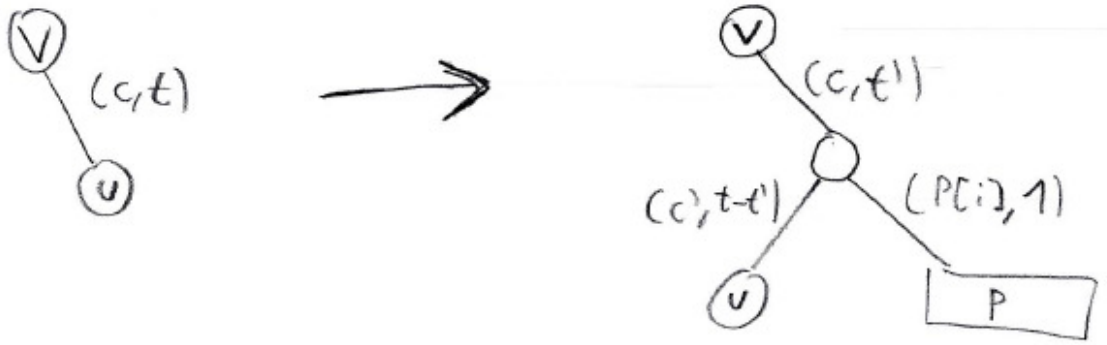


Patricia trees

- Inserción en un Patricia tree
  - Volver a bajar consumiendo i-1 caracteres
    - Se llegó al medio de una arista v -> u (v tiene rótulo (c,l)), habiendo consumido 1 <= t' <= t caracteres de la arista
      - Crear nuevo nodo x, hijo de v y padre de u y de P
      - Rótulo de v -> x es (c,t')
      - Rótulo de x -> u es (c', t – t'), x -> P es (P[i],1)

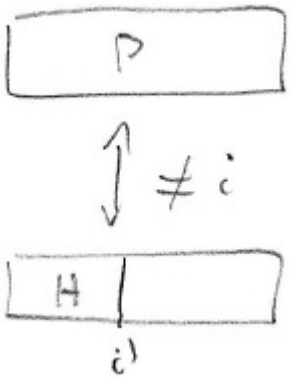
Patricia trees

- Inserción en un Patricia tree
  - Volver a bajar consumiendo i-1 caracteres
    - Se llegó al medio de una arista v -> u (v tiene rótulo (c,l)), habiendo consumido 1 <= t' <= t caracteres de la arista



Patricia trees

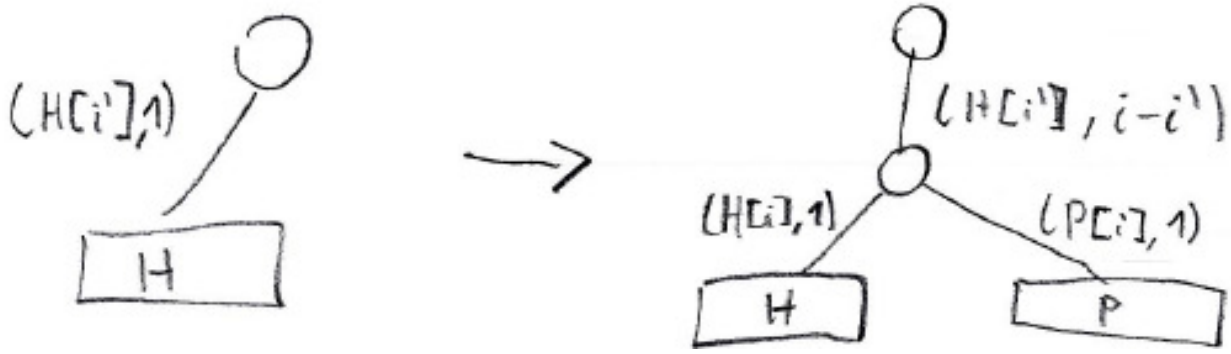
- Inserción en un Patricia tree
  - Volver a bajar consumiendo i-1 caracteres
    - Se llegó a la hoja H



    - Sea i' < i el largo del camino (sumando saltos) hasta la hoja

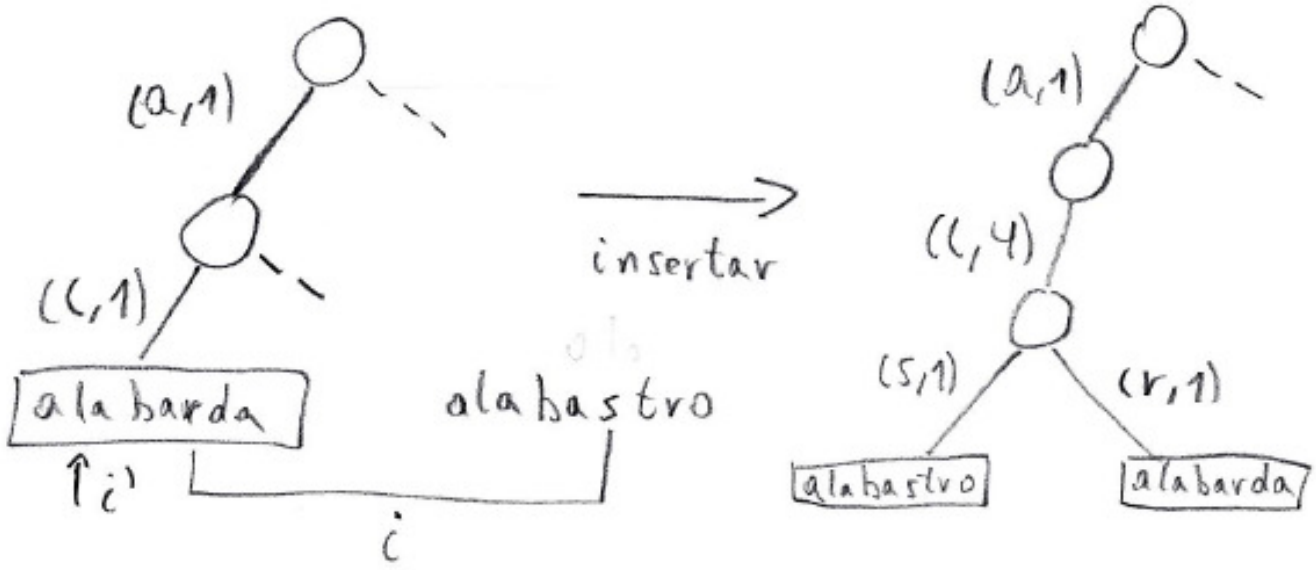
Patricia trees

- Inserción en un Patricia tree
  - Volver a bajar consumiendo i-1 caracteres
    - Se llegó a la hoja H



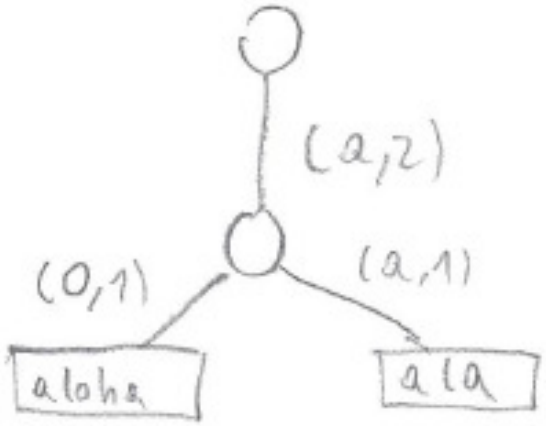
Patricia trees

- Inserción en un Patricia tree
  - Volver a bajar consumiendo i-1 caracteres
    - Se llegó a la hoja H, ejemplo



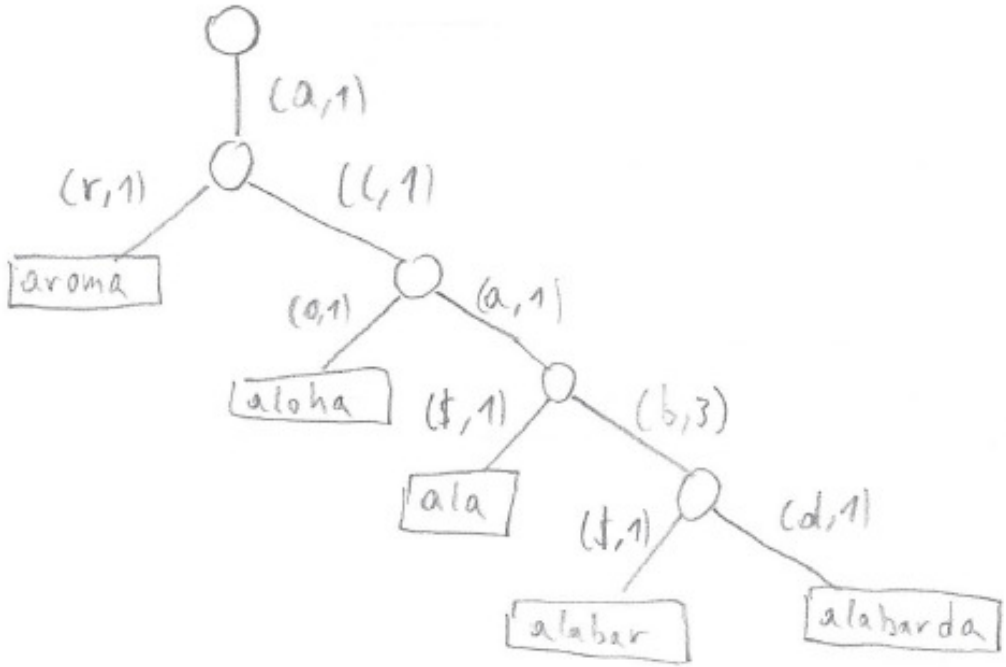
Patricia trees

- Inserción en un Patricia tree
  - Ejercicio: insertar en el trie las palabras
    - aroma
    - alabar
    - alabarda



Patricia trees

- Inserción en un Patricia tree
  - Solución



Diccionarios con Predecesor/Sucesor

- Universo: enteros en el rango [1, N]
- Estructura que permite las operaciones Insert, Delete, Lookup, FindNext y FindPrevious en tiempo óptimo O(log log N)
- Arbol de van Emde Boas (estudiados en clase auxiliar)

ALGORITMOS EN LÍNEA

Motivación

- Problema del “ski rental”
  - Ud. va de viaje a un centro de ski por un número indeterminado de días (n)
  - Arrendar un par de skies por día cuesta $a
  - Comprar un par de skies cuesta $b
  - ¿Cómo minimizar el costo por el uso de skies?
    - Algoritmo que minimice la razón de lo que se paga utilizando dicho algoritmo con lo que se haría si se conociera el valor de n (el óptimo)

Motivación

- Problema del “ski rental”
  - Por ejemplo: a=1, b=10
  - Algoritmo (con costo ALG):
    - Arrendar los primeros 9 días
    - Comprar el día 10
  - Observaciones
    - Si se está menos de 9 días, el gasto es el mínimo
    - Si se está más de 10 días, a lo más se paga 19, esto es, un 90% extra comparado con el costo óptimo (OPT)
    - Notar que: costo ALG < 2 OPT (para cualquier n)

Motivación

- Problema del “ski rental”
  - En general, el costo óptimo es:
    - OPT = min(an, b)
  - Algoritmo online (no conoce n): arrendar k días y luego comprar
    - ALG =
      - an si n<=k,
      - ak+b si n>k

Motivación

- Problema del “ski rental”
  - Competitividad: razón de los costos (ALG/OPT)

$$\frac{\text{ALG}}{\text{OPT}} = \begin{cases} \text{si } n \leq k, \ \frac{an}{\min(an,b)} \begin{cases} \text{si } an \leq b, \ \frac{an}{an} = 1 \\ \text{si } an > b, \ 1 < \frac{an}{b} < \frac{ak}{b} \end{cases} \\ \text{si } n > k, \ \frac{ak+b}{\min(an,b)} \begin{cases} \text{si } an \leq b, \ \frac{ak+b}{an} < \frac{ak+b}{ak} = 1 + \frac{b}{ak} \\ \text{si } an > b, \ \frac{ak+b}{b} = 1 + \frac{ak}{b} \end{cases} \end{cases}$$

  - Valor de k óptimo: b/a

$$\frac{\text{ALG}}{\text{OPT}} = 2$$

Definiciones

- Problema de optimización
  - Problema de optimización P (minimización de costo) consiste de:
    - Conjunto U de entradas (inputs)
    - Función de costo C
  - Asociado a cada entrada posible (I en U) hay un conjunto posible de salidas F(I) (outputs)
    - A cada salida O en F(I) se le asocia un real positivo, C(I,O), que representa el costo de la salida O con respecto a la entrada I

Definiciones

- Problema de optimización
  - Dada una entrada I válida, un algoritmo ALG para un problema de optimización P calcula una solución (salida) ALG[I] en F(I)
    - Costo asociado a dicha solución es ALG(I) = C(I,ALG[I])
    - Un algoritmo óptimo OPT cumple que

$$\mathrm{OPT}(I) = \min_{O \in F(I)} C(I, O)$$

    - ALG es una c-aproximación asintótica de P si existe constante α>=0 tal que

$$\mathrm{ALG}(I) - c \cdot \mathrm{OPT}(I) \leq \alpha, \ \alpha \geq 0$$

Definiciones

- Problema de optimización
  - El factor de aproximación c es >= 1
    - Mientras más cercano a 1, mejor es la aproximación
  - ALG es una c-aproximación de P si α=0
  - Si un problema de optimización recibe el input en forma online, y la salida debe producirse también en forma online, se denomina “problema en línea” (online problem)
  - Notar que c no necesariamente es constante, puede depender de los parámetros (pero no de I)

Definiciones

- Competitividad
  - Un algoritmo online ALG es c-competitivo si existe una constante α tal que para toda entrada finita I

$$\mathrm{ALG}(I) \leq c \cdot \mathrm{OPT}(I) + \alpha$$

  - Cuando α <= 0 se dice que ALG es estrictamente c-competitivo

Definiciones

- Competitividad
  - Un ALG (estrictamente) c-competitivo es una c-aproximación donde ALG debe calcular en forma online
    - Para cada entrada I está garantizado que ALG incurre en un costo con un factor c del costo óptimo offline
    - La razón de competitividad c es al menos 1, y mientras más pequeña sea, mejor es la eficiencia de ALG con respecto a OPT

Problema: accesos a una lista

- Datos: lista de n objetos
- Acceso a los objetos de la lista
  - El acceso a la lista es secuencial (lista desordenada)
  - Acceder al i-ésimo objeto de la lista tiene costo i
  - Si objeto no está, costo es n
  - Se puede reubicar el objeto buscado en la lista
    - Por ejemplo, ponerlo más cerca del principio

Problema: accesos a una lista

- Cualquier algoritmo que administre la lista puede reorganizarla en cualquier momento
  - El trabajo realizado en la reorganización se mide en términos del número mínimo de intercambios entre elementos consecutivos necesarios para la reorganización
  - Inmediatamente después de acceder un objeto, éste se puede mover **sin costo** a cualquier posición anterior en la lista
    - Intercambios “gratis” vs. “pagados” (cuestan 1)

Problema: accesos a una lista

- Problema de accesos a una lista es un problema de optimización
  - Objetivo: desarrollar un algoritmo para reorganizar la lista (con intercambios gratis o pagados) que minimice los costos de búsqueda y reorganización

Problema: accesos a una lista

- Ejemplos de algoritmos
  - Move-To-Front (MTF)
    - Mueve objeto accedido al principio de la lista
  - Transpose (TRANS)
    - Intercambia objeto accedido con su antecesor
  - Frequency Count (FC)
    - Mantiene un contador con la frecuencia de acceso a cada objeto. Luego de acceder o insertar un objeto, se incrementa su contador en 1. Inmediatamente después, se reorganiza la lista ordenándola por frecuencia (nonincreasing order)

Problema: accesos a una lista

- MTF, TRANS, y FC sólo usan intercambios gratis
  - Un algoritmo óptimo ocupa intercambios pagados
  - Ejercicio:
    - Muestre que para la secuencia <x1,x2,x3> el acceder a los objetos {x3,x2,x3,x2} en forma óptima (offline) cuesta 8
    - Muestre que un algoritmo (offline) óptimo que sólo utiliza intercambios gratis, el acceder a los mismos objetos cuesta 9

Problema: accesos a una lista

- Teorema de Sleator-Tarjan
  - MTF es 2-competitivo
  - Sean:
    - $ALG_p(\sigma)$: número de intercambios pagados hechos por ALG al procesar la secuencia σ de operaciones
    - $ALG_f(\sigma)$: número de intercambios gratis
    - $ALG_c(\sigma)$: costo de las operaciones que no son intercambios pagados
    - Notar que $MTF(\sigma) = ALG_c(\sigma)$

Problema: accesos a una lista

- Teorema de Sleator-Tarjan
  - Teorema: Sea σ una secuencia de q operaciones en la lista. Suponga que MTF y OPT parten con la misma configuración inicial. Se cumple que

$$\mathrm{MTF}(\sigma) \leq 2 \cdot \mathrm{OPT}_c(\sigma) + \mathrm{OPT}_p(\sigma) - \mathrm{OPT}_f(\sigma) - q$$

Problema: accesos a una lista

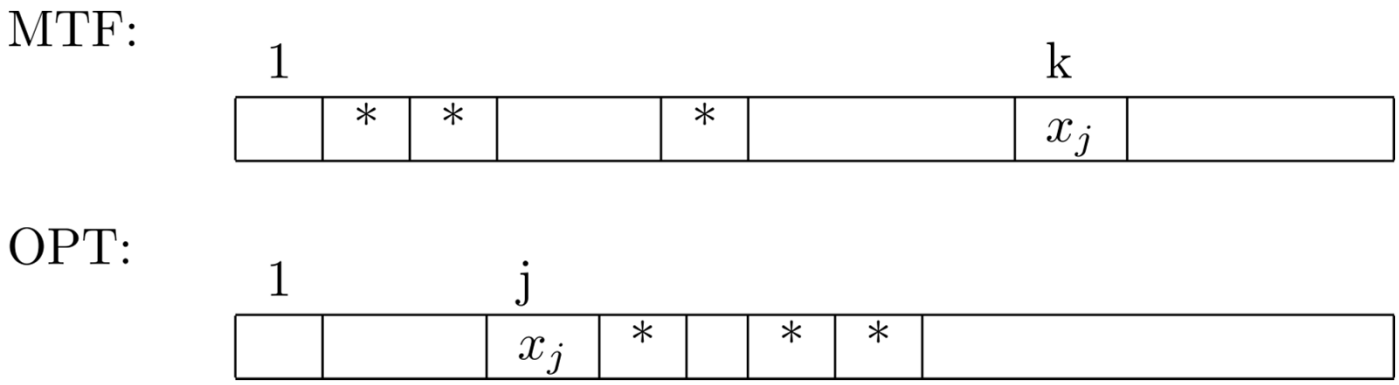
- Teorema de Sleator-Tarjan
  - Demostración:
    - Análisis amortizado, función potencial es número de inversiones en MTF con respecto a OPT
    - Se mostrará que el costo amortizado por operación es

$$(2s - 1) + P - F$$

      - s: costo de búsqueda (sin incluir intercambios)
      - P: número de intercambios pagados
      - F: número de intercambios gratis

Problema: accesos a una lista

- Teorema de Sleator-Tarjan
  - Demostración:
    - x_j se posiciona en j en OPT, en k en MTF
    - Los asteriscos muestran las v inversiones con respecto a x_j (podrían haber inversiones extra, objetos antes de x_j en OPT pero después de x_j en MTF)



Problema: accesos a una lista

- Teorema de Sleator-Tarjan
  - Demostración:
    - Entonces, k-1-v objetos preceden a x_j en ambas listas (al menos este número de objetos precede a x_j en OPT)
    - Dado que x_j está en la j-ésima posición en OPT

$$k - 1 - v \leq j - 1$$

    - Por lo tanto

$$k - v \leq j$$

Problema: accesos a una lista

- Teorema de Sleator-Tarjan
  - Demostración:
    - Cuando MTF mueve x_j al comienzo de su lista, se crean k-1-v nuevas inversiones con respecto a OPT antes que OPT procese el requerimiento
    - También, MTF elimina v inversiones
    - Por lo tanto, la contribución al costo amortizado es

$$k + (k - 1 - v) - v = 2(k - v) - 1 \leq 2j - 1 = 2s - 1$$

Problema: accesos a una lista

- Teorema de Sleator-Tarjan
  - Demostración:
    - El costo de búsqueda de OPT (que es j) no contribuye al cambio de potencial
    - Sin embargo, cada intercambio pagado contribuye a lo más en 1 al costo amortizado y cada intercambio gratis contribuye en -1
      - OPT realiza P intercambios pagados y F intercambios gratis

Problema: accesos a una lista

- Cota inferior para TRANS
  - Teorema: TRANS no es competitivo en el caso de listas dinámicas (se permite insertar objetos)
    - Estrategia del adversario para la demostración
      - Lista con n objetos
      - Se acceden sólo dos objetos: los dos últimos de la lista
      - TRANS intercambiará estos objetos en cada acceso
      - Siempre se busca el último
      - Costo por cada par de búsquedas: 2n
    - OPT mueve ambos objetos al principio de la lista
      - Costo de 3 por cada par de búsquedas

Problema: accesos a una lista

- Cota inferior para TRANS
  - Teorema: TRANS no es competitivo en el caso de listas dinámicas (permite insertar objetos)
    - Secuencia de q operaciones (suponga q par)
    - Razón del costo entre TRANS y OPT

$$\frac{\text{TRANS}}{\text{OPT}} = \frac{qn}{(q/2-1)3+2n} \approx \frac{2n}{3} \text{ (para } q \text{ grande)}$$

      - Competitivo en caso estático (en función de n)
      - Caso dinámico: no hay cota superior para el número de inserciones iniciales => no es competitivo

Problema: accesos a una lista

- Cota inferior para FC
  - Se puede demostrar que

$$\frac{\mathrm{FC}}{\mathrm{OPT}} \geq \frac{n+1}{2}$$

  - Por lo tanto, FC tampoco es competitivo en listas dinámicas

Paginamiento en disco

- Considere el siguiente sistema de memoria
  - Tiene dos niveles que pueden almacenar un número de celdas de memoria de tamaño fijo
    - Páginas
  - El primer nivel es la memoria lenta, y puede almacenar N páginas
  - El segundo nivel es la memoria rápida, y puede almacenar cualquier subconjunto de k páginas (k<N)

Paginamiento en disco

- Considere el siguiente sistema de memoria
  - Cuando se solicita la página p_i, el sistema debe dejarla disponible en la memoria rápida
    - Si p_i ya estaba en la memoria rápida (un “hit”), el sistema no necesita hacer nada
    - Sino (un “miss”), el sistema incurre en un “page fault” y debe copiar p_i de la memoria lenta a la rápida
      - Al hacer esto, el sistema debe decidir en que página de la memoria rápida colocarlo (implica sacar de la memoria rápida alguna página previamente leida)
      - Problema: definir estrategia para minimizar el número de page faults

Paginamiento en disco

- Ejemplos
  - Típico:
    - Memoria rápida: RAM
    - Memoria lenta: disco
  - Otro caso (caching):
    - Memoria rápida: cache
    - Memoria lenta: RAM
- Típicamente N se considera muy grande
  - No se considera como un parámetro relevante del problema

Paginamiento en disco

- Modelo de costo abstracto (page fault model)
  - Se cobra 1 por llevar una página de la memoria lenta a la rápida
  - No se cobra por leer o escribir una página en la memoria rápida
- Modelo de costo completo (parámetros s y k)
  - Se cobra 1 por llevar una página de la memoria lenta a la rápida
  - Se cobra 1/s por leer o escribir una página en la memoria rápida

Paginamiento en disco

- Ejemplos de algoritmos de paginamiento
  - LRU (Least-Recently-Used): reemplaza la página cuyo último acceso fue hace más tiempo
  - CLOCK: aproximación de LRU, usa un único bit que reemplaza el tiempo guardado por LRU
    - Lista circular, si se accede a la página, se marca su bit
    - Si hay page fault, busca página desmarcada en la lista, desmarcando las páginas marcadas
  - FIFO (First-In/First-Out): reemplaza la página que ha estado más tiempo en la memoria rápida

Paginamiento en disco

- Ejemplos de algoritmos de paginamiento
  - LIFO (Last-In/First-Out): reemplaza la página más recientemente movida a la memoria rápida
  - LFU (Least-Frequently-Used): reemplaza la página que ha sido menos accedida desde que se movió a la memoria rápida
  - LFD (Longest-Forward-Distace): reemplaza la página cuyo próximo requerimiento ocurrirá más tarde que las otras (algoritmo offline)

Paginamiento en disco

- Algoritmos “demand paging”
  - A no ser que ocurra un page fault, nunca sacan una página de la memoria rápida
- Todos los ejemplos anteriores cumplen con ser “demand paging”

Problema de (h,k)-paging

- Generalización del problema de paginamiento
  - Sean k, h enteros positivos, h <= k
  - Mide el rendimiento de un algoritmo de paginamiento online con un cache de tamaño k con respecto a un algoritmo óptimo offline de tamaño h <= k
    - Si h < k algoritmo offline dispone de menos recursos
    - Puede parecer injusto, pero el algoritmo offline tiene “poderes irrealistas”

LFD es óptimo

- Teorema: LFD es un algoritmo offline óptimo de paginamiento
- Demostración:
  - Se mostrará que cualquier algoritmo offline de paginamiento se puede modificar para actuar como LFD sin degradar su eficiencia

LFD es óptimo

- Sea ALG un algoritmo de paginamiento
- Sea σ una secuencia de accesos a páginas
- Para todo i = 1, 2, … |σ| se puede construir un ALG_i que cumple
  - 1) ALG_i procesa los primeros i-1 accesos como ALG
  - 2) si el i-ésimo acceso produce un page fault, ALG_i funciona como LFD
  - 3) $ALG_i(\sigma) <= ALG(\sigma)$

LFD es óptimo

- Dado ALG, se construye ALG_i:
  - Sea X un conjunto de páginas, p una página, X+p la unión de X con {p}
  - Suponga que justo después de procesar la i-ésima página, las memorias rápidas contienen
    - ALG: X+v
    - ALG_i : X+u
    - |X| = k-1, v != u (i-ésimo acceso produjo un page fault)

LFD es óptimo

- Dado ALG, se construye ALG_i:
  - Hasta que v sea accedido, ambos algoritmos realizan las mismas acciones
    - Excepción: ALG_i reemplaza u si ALG reemplaza v
    - Esto es cierto dado que tienen k-1 páginas en común
    - Más aún, si en algún momento el número de páginas en común es k (es decir, ALG reemplaza v), ambos algoritmos funcionarán igual de ahí en adelante

LFD es óptimo

- Dado ALG, se construye ALG_i:
  - ¿Qué pasa si v es accedido?
    - Esto producirá un page fault en ALG_i pero no en ALG
    - Sin embargo, u tiene que haber sido accedido antes dado el reemplazo LFD
      - Esto incurrió en un page fault en ALG pero no en ALG_i
    - Por lo tanto, el número de total de page faults de ALG_i después del acceso a v es igual al de ALG
    - Finalmente, ALG_i elimina u para poner v en la memoria principal, y ambos algoritmos quedan con k páginas comunes

Algoritmos de marca/conservativos

- Algoritmos de marca
  - Sea una secuencia σ de accesos
  - Se divide la secuencia en fases:
    - La fase 0 es la secuencia vacía
    - Para todo i >= 1, la fase i es la máxima secuencia que siga a la fase i-1 que contenga a lo más k accesos a páginas distintas
      - La fase i+1 comienza con el acceso que corresponde al (k+1)-ésimo acceso distinto
  - Esto se denomina “k-phase partition” y es independiente de cómo un ALG procesa σ

Algoritmos de marca/conservativos

- Algoritmos de marca
  - Considere σ y su k-phase partition
  - Implícitamente se asocia un bit a cada página de la memoria lenta (la marca)
    - Al comienzo de cada k-phase, todas las páginas de la memoria rápida están desmarcadas
    - Durante la k-phase, se marca una página cuando se accesa por primera vez durante la k-phase
  - Un “algoritmo de marca” nunca reemplaza una página marcada de la memoria rápida

Algoritmos de marca/conservativos

- Algoritmos de marca
  - Teorema:
    - Sea ALG un algoritmo de marca con un cache de tamaño k
    - Sea OPT un algoritmo offline óptimo con un cache de tamaño h <= k
    - ALG es k/(k-h+1)-competitivo

Algoritmos de marca/conservativos

- Algoritmos de marca
  - Demostración (ALG es k/(k-h+1)-competitivo):
    - Sea σ una secuencia de accesos y considere su k-phase partition
    - Para cualquier fase i >=1, ALG incurre a lo más en k page faults
      - Hay k páginas accedidas en cada fase (excepto, quizás, en la última, donde podría haber menos)
      - Luego de acceder a una página, ésta se marca y por lo tanto no puede ser reemplazada hasta la siguiente fase
      - Por lo tanto, ALG no puede hacer page fault dos veces en la misma fase con la misma página

Algoritmos de marca/conservativos

- Algoritmos de marca
  - Demostración (ALG es k/(k-h+1)-competitivo):
    - Para cada i >= 1, sea q el primer acceso en la fase i y considere la secuencia que comienza con el segundo acceso de la fase i hasta el primer acceso de la fase i+1
      - OPT tiene h-1 páginas sin incluir a q
      - Hay k accesos en esta secuencia
      - OPT incurre en al menos k-(h-1) = k-h+1 page faults
        - Si la fase i es la última, OPT incurre en al menos k'-h+1 page faults, con k' el número de accesos en la última fase

Algoritmos de marca/conservativos

- Algoritmos de marca
  - Demostración (ALG es k/(k-h+1)-competitivo):
    - Resumiendo:
      - En cada fase, ALG incurre en page fault a lo más k veces
      - Por cada fase, exceptuando la última, OPT incurre en al menos k-h+1 page faults
      - Por lo tanto,

$$\mathrm{ALG}(\sigma) \leq \frac{k}{k-h+1} \cdot \mathrm{OPT} + \alpha$$

      - a <= k es el máximo número de page faults que incurre ALG durante la última fase

Algoritmos de marca/conservativos

- Algoritmos de marca
  - Lema: LRU es un algoritmo de marca
    - Sea σ y su k-phase partition
    - Suponga que LRU no es algoritmo de marca
      - Entonces, reemplaza una página marcada x en alguna fase
      - Sea el primer acceso a x en alguna fase, x se marca y es la página más recientemente accedida
      - Para que x deje el cache, LRU tiene que incurrir en page fault cuando x es la página menos recientemente accedida
      - Para que esto pase, hay que incurrir en al menos k+1 (incluyendo el de la página x y el de la página que lo saca) page faults en una k-phase, contradicción!

Algoritmos de marca/conservativos

- Algoritmos de marca
  - Lema: CLOCK es un algoritmo de marca
  - Corolario: LRU y CLOCK son k/(k-h+1)-competitivos
  - Si h=k, se sigue que LRU y CLOCK son k-competitivos

$$\mathrm{LRU}(\sigma) \leq k \cdot \mathrm{OPT} + \alpha$$

  - FIFO no es algoritmo de marca

Algoritmos de marca/conservativos

- Algoritmos conservativos
  - Satisfacen la condición:
    - En cualquier subsecuencia consecutiva que contenga k o menos accesos a páginas distintas, incurren en k o menos page faults
  - LRU, CLOCK y FIFO son conservativos
  - Un algoritmo ALG conservativo cumple que

$$\mathrm{ALG}(\sigma) \leq \frac{k}{k-h+1} \cdot \mathrm{OPT} + \alpha$$

Cota inferior

- Teorema: todo algoritmo ALG de paginamiento es a lo más k-competitivo

$$\frac{\mathrm{ALG}(\sigma)}{\mathrm{OPT}(\sigma)} \geq k$$

- Lema: Por cada secuencia finita s de accesos escogidos de entre k+1 páginas, se cumple que

$$\mathrm{LFD}(\sigma) \leq \frac{|\sigma|}{k}$$

Cota inferior

- Demostración lema:
  - Suponga que en el i-ésimo acceso, página r_i, LFD reemplaza la página p
  - Por definición de LFD y dado que hay k+1 páginas en total, todas las páginas en el cache (exceptuando quizás r_i) deben ser accedidas antes del siguiente acceso a p
  - Por lo tanto, LFD incurre en page fault a lo más una vez cada k accesos

Cota inferior

- Demostración teorema:
  - Hay en total k+1 páginas, el cache guarda k
  - Estrategia del adversario: dado ALG, definir secuencia σ de forma que siempre se accede a la página que está fuera del cache
  - Por lo tanto,

$$\mathrm{ALG}(\sigma) = |\sigma|$$

- Ningún algoritmo de paginamiento es mejor que k-competitivo

LIFO y LFU no son competitivos

- LIFO: estrategia del adversario define secuencia de accesos

$$\sigma = p_1, p_2, \ldots, p_k, p_{k+1}, p_k, p_{k+1}, \ldots$$

  - LIFO incurre en page fault en todo acceso, OPT incurre en total en k+1 page faults

- LFU: sea t un entero positivo

$$\sigma = p_1^t, p_2^t, \ldots, p_{k-1}^t, (p_k, p_{k+1})^{t-1}$$

  - LFU incurre en page fault siempre después de (k-1)t accesos, OPT sólo incurre en un page fault