

Técnicas Avanzadas de Programación

Escuela de Ingeniería y Arquitectura
Curso 2012/2013

Javier Campos Laclaustra



Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

Técnicas Avanzadas de Programación



Técnicas Avanzadas de Programación

- ¿Qué es?
 - Es un curso sobre estructuras de datos avanzadas y su análisis
- ¿Dónde encontrar información sobre el curso (incluidas estas transparencias)?
 - En su página web:
<http://webdiis.unizar.es/asignaturas/TAP/>



Técnicas Avanzadas de Programación

- Índice
 - Análisis en el caso peor
 - Repaso de conceptos
 - Montículos y el problema de ordenación
 - Árboles rojinegros para implementar diccionarios
 - Análisis del caso promedio
 - Probabilidad
 - Análisis probabilista
 - Árboles binarios de búsqueda contruidos aleatoriamente
 - Tries, árboles digitales de búsqueda y Patricia
 - Listas “skip”
 - Árboles aleatorizados



Técnicas Avanzadas de Programación

- Índice (continuación)
 - Análisis amortizado
 - Conceptos básicos. Método agregado. Método contable. Método potencial
 - Primer ejemplo: análisis de tablas *hash* dinámicas
 - Montículos agregables (binomiales y de Fibonacci)
 - Estructuras de conjuntos disjuntos
 - Listas lineales auto-organizativas
 - Árboles auto-organizativos
 - Introducción a los algoritmos de biología computacional.
 - Algoritmos de reconocimiento de patrones (Knuth-Morris-Pratt y Boyer-Moore)
 - Árboles de sufijos
 - Primeras aplicaciones de los árboles de sufijos



Técnicas Avanzadas de Programación

- Bibliografía fundamental:
 - Cormen, T.H.; Leiserson, C.E.; Rivest, R.L. y Stein, C.: *Introduction to Algorithms* (3rd edition), The MIT Press, 2009.
 - Mehta, D.P. y Sahni, S.: *Handbook of Data Structures and Applications*, Chapman & Hall/CRC, 2005.
- Bibliografía:
 - Aho, A.V.; Hopcroft, J.E. y Ullman, J.D.: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
 - Brassard, G. y Bratley, P.: *Fundamentals of Algorithmics*, Prentice Hall, 1996.
 - Gonnet, G. H. y Baeza-Yates, R.: *Handbook of Algorithms and Data Structures. In Pascal and C*, Addison-Wesley, 1991.
 - Gusfield, D.: *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*, Cambridge University Press, 1997.
 - ...



Técnicas Avanzadas de Programación

- Bibliografía (continuación)

- Kingston, J. H.: *Algorithms and Data Structures. Design, Correctness, Analysis*, Addison-Wesley, 1990.
- Knuth, D.E.: *The Art of Computer Programming: Sorting and Searching* (vol. 3), Addison-Wesley, 1973.
- Kozen, D.C.: *The Design and Analysis of Algorithms*, Springer-Verlag, 1992.
- Moret, B. M. E. y Shapiro, H.D.: *Algorithms: From P to NP. Volume I: Design & Efficiency*, The Benjamin/Cummings Pub. Co., 1991.
- Sedgewick, R.: *Algorithms in C++ (third edition). Parts 1-4 (Fundamentals, Data Structures, Sorting, Searching)*, Addison-Wesley, 1998.
- Sedgewick, R.: *Algorithms in C++ (third edition). Part 5 (Graph Algorithms)*, Addison-Wesley, 2002.
- Tarjan, R.E.: *Data Structures and Network Algorithms*, SIAM, 1983.
- Weiss, M.A.: *Data Structures and Algorithm Analysis in Java*, Addison-Wesley, 1999



Técnicas Avanzadas de Programación

- Más bibliografía (artículos)
 - Martínez, C. y Roura, S.: Randomized binary search trees. *Journal of the ACM*, vol. 45, no. 2, pp. 288-323, 1998.
 - Messeguer, X.: Skip trees, an alternative data structure to skip lists in a concurrent approach. *Informatique Theorique et Applications*, vol. 31, no. 3, pp. 251-269, 1997.
 - Pugh, W.: Skip Lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, vol. 33, no. 6, pp.668-676, June 1990.
 - Pugh, W.: A skip list cookbook. Technical Report CS-TR-2286.1, University of Maryland, USA, June 1990.
 - Seidel, R. Y Aragon, C.R.: Randomized search trees. *Algorithmica*, vol. 16, pp. 464-497, 1996.



Análisis en el caso peor



Análisis en el caso peor

- El plan:
 - **Repaso de conceptos**
 - Montículos y el problema de ordenación
 - Árboles rojinegros para implementar diccionarios



Repaso de conceptos

- *Problema de cálculo:*

especificación de una relación existente entre unos valores de *entrada* (datos del problema) y otros de *salida* (resultados)

Ejemplo: *problema de ordenación*

entrada: una secuencia de números (a_1, a_2, \dots, a_n)

salida: una permutación $(a_1', a_2', \dots, a_n')$ de la secuencia de entrada tal que $a_1' \leq a_2' \leq \dots \leq a_n'$

- *Algoritmo:*

herramienta para resolver un problema de cálculo, es decir, método de cálculo bien definido que a partir de un valor o valores de entrada produce un valor o conjunto de valores de salida



Repaso de conceptos

- *Ejemplar* de un problema:
valores concretos necesarios para construir la entrada para un problema (satisfaciendo las restricciones impuestas por el problema)

Ejemplo:

$(4,2,6,5,1) \rightarrow$ algoritmo de ordenación $\rightarrow (1,2,4,5,6)$



Repaso de conceptos

- Algoritmo correcto:

para todo ejemplar de problema (todos los valores posibles de los datos de entrada) produce una salida correcta (i.e., los valores de salida verifican la relación especificada en el problema con los de entrada)

Ejemplo: algoritmo correcto de ordenación

```
algoritmo ordenaciónPorInserción(A)
principio
  para j:=2 hasta long(A) hacer
    dato:=A[j]
    i:=j-1;
    mq i>0 and A[i]>dato hacer
      A[i+1]:=A[i];
      i:=i-1
    fmq;
    A[i+1]:=dato
  fpara
fin
```



Repaso de conceptos

- *Eficiencia* de un algoritmo:
 - cantidad de recursos que usa un algoritmo para su ejecución
 - normalmente se considera el tiempo de ejecución del algoritmo, la utilización de espacio de memoria o el número de procesadores (en programación paralela)
 - en este curso nos concentraremos en el tiempo de ejecución en función del tamaño de la entrada, se denomina también *coste*, *rendimiento* o *complejidad* del algoritmo



Repaso de conceptos

- Técnicas de cálculo de la eficiencia:
 - Pruebas (en inglés *benchmarking*):
 - elaborar una muestra significativa o típica de los posibles datos de entrada y tomar medidas de los tiempos de ejecución para cada elemento de la muestra
 - a partir de los datos anteriores aplicar técnicas estadísticas para inferir el rendimiento del programa para datos de entrada no presentes en la muestra (por tanto, las conclusiones pueden no ser muy fiables)
 - Análisis (es en lo que vamos a concentrarnos)
 - procedimientos matemáticos para determinar el coste (conclusiones fiables, pero realización difícil o imposible a efectos prácticos)



Repaso de conceptos

- Tamaño de los datos:
 - El coste de un algoritmo es función del tamaño de los datos de entrada
 - Sugerencias para definir el tamaño de los datos:
 - datos naturales o enteros:
 - su propio valor
 - el tamaño de su descripción binaria
 - ¡Ojo! ¡Hay que aclararlo, se diferencian en una exponencial!



Repaso de conceptos

- Sugerencias para definir el tamaño de los datos (continuación):
 - pares o ternas de enteros:
 - en función de todos ellos
 - » es lo más deseable pero muchas veces el análisis es difícil de realizar
 - en función de uno de ellos (suponiendo los demás constantes)
 - » es una decisión irreal (utilidad discutible) pero en la práctica mucho más sencilla de realizar
 - » a efectos de comparar distintos algoritmos para resolver un mismo problema puede ser suficiente



Repaso de conceptos

- Sugerencias para definir el tamaño de los datos (continuación):
 - estructuras homogéneas (vectores, ficheros,...):
 - número de componentes (longitud del vector)
 - » normalmente es lo más adecuado
 - » puede ser inadecuado ocasionalmente si los elementos son a su vez estructuras complejas, o enteros de magnitudes muy elevadas para los que no basta una palabra de máquina
 - otros casos:
 - como se pueda, *ad hoc*, con sentido común,...



Repaso de conceptos

- Tiempo de ejecución
 - función $T(n)$ que representa el número de unidades de tiempo (segundos, milisegundos, ...) que un algoritmo tarda en ejecutarse al suministrarle unos datos de entrada de tamaño n
 - problema: variaciones sustanciales según la marca y modelo de computador utilizado
 - solución: es preferible que $T(n)$ sea el *número de instrucciones simples* (asignaciones, comparaciones, operaciones aritméticas, etc.) que se ejecutan
 - sería como el tiempo de ejecución en un computador idealizado, donde cada asignación, comparación, lectura, etc., consume 1 unidad de tiempo
 - si $T(n)$ es el tiempo de ejecución en ese computador idealizado, el tiempo de ejecución en un computador real X será $c_x T(n)$, donde c_x es una constante que depende del computador (y que puede determinarse usando técnicas de *benchmarking* y estadísticas convencionales)



Repaso de conceptos

- Caso peor, caso mejor, caso promedio:

Con frecuencia, el coste de un algoritmo depende de los datos de entrada particulares y no sólo del tamaño de los mismos

- *Coste en el caso peor:*

- coste máx. del algoritmo para datos de entrada de tamaño n
 - es el más comunmente usado, lo denotaremos $T(n)$
 - es excesivamente pesimista

- *Coste en el caso mejor:*

- coste mín. del algoritmo para datos de entrada de tamaño n
 - demasiado optimista, demasiado irreal para ser de utilidad práctica



Repaso de conceptos

- Caso peor, caso mejor, caso promedio (cont.)
 - *Coste promedio:*
 - cálculo de una media del uso de recursos para todos los datos de entrada posibles de tamaño n
 - lo denotaremos $\overline{T(n)}$
 - la media se hace ponderando por la frecuencia de aparición de cada caso, i.e., requiere el cálculo de una esperanza matemática a partir de una distribución de probabilidad
 - da más información
 - precisa más datos (la distribución de probabilidad de los datos de entrada de tamaño n), que son muy difíciles de calcular
 - normalmente se basa en la hipótesis de equiprobabilidad de las posibles entradas (hipótesis que a menudo no es cierta)



Repaso de conceptos

- El ejemplo:

```
algoritmo ordenaciónPorInserción(A)
principio
  para j:=2 hasta long(A) hacer
    dato:=A[j]
    i:=j-1;
    mq i>0 and A[i]>dato hacer

      A[i+1]:=A[i];

      i:=i-1
    fmq;
    A[i+1]:=dato
  fpara
fin
```

coste	veces
c_1	n
c_2	$n-1$
c_3	$n-1$
c_4	$\sum_{j=2}^n t_j$
c_5	$\sum_{j=2}^n (t_j - 1)$
c_6	$\sum_{j=2}^n (t_j - 1)$
c_7	$n-1$

t_j es el número de veces que se ejecuta el **mq** para cada valor de j

$$T(n) = c_1 n + c_2 (n-1) + c_3 (n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7 (n-1)$$



Repaso de conceptos

- Un ejemplo (cont.):

$$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

Si el vector ya estaba ordenado (**caso mejor**):

$$A[i] \leq \text{dato para } i=j-1 \Rightarrow t_j = 1 \text{ para } j = 2, 3, \dots, n$$

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) \\ &= an + b \end{aligned}$$

↘ **función lineal en n**



Repaso de conceptos

- Un ejemplo (cont.):

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

Si el vector estaba ordenado en sentido decreciente (**caso peor**):

$$t_j = j \text{ para } j = 2, 3, \dots, n$$

$$\sum_{j=2}^n j = \frac{n(n+1)}{2} - 1 \qquad \sum_{j=2}^n (j-1) = \frac{n(n-1)}{2}$$

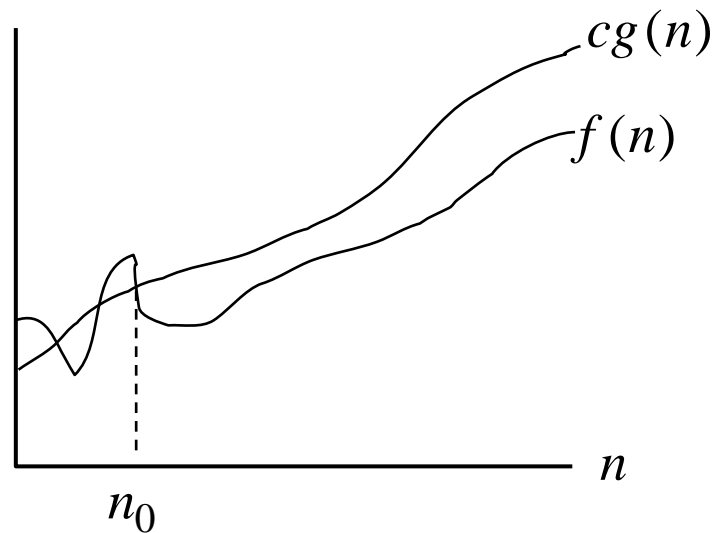
$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left(\frac{n(n+1)}{2} - 1 \right) + c_5 \left(\frac{n(n-1)}{2} \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7(n-1) \\ &= \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2} \right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7 \right) n - (c_2 + c_3 + c_4 + c_7) \\ &= an^2 + bn + c \end{aligned}$$

→ **función cuadrática en n**



Repaso de conceptos

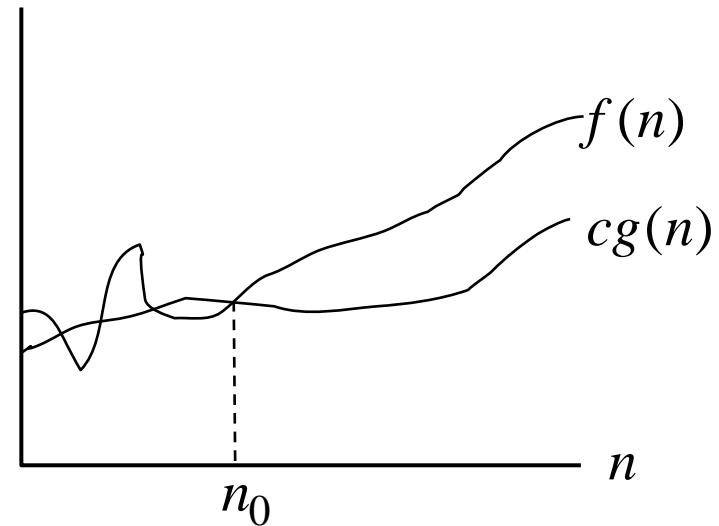
- Órdenes de crecimiento



$g(n)$ es cota superior asintótica de $f(n)$
(i.e., $f(n)$ es del mismo orden o de
orden inferior a $g(n)$)

$$f(n) \text{ es } O(g(n))$$

$$(c > 0)$$



$g(n)$ es cota inferior asintótica de $f(n)$
(i.e., $f(n)$ es del mismo orden o de
orden superior a $g(n)$)

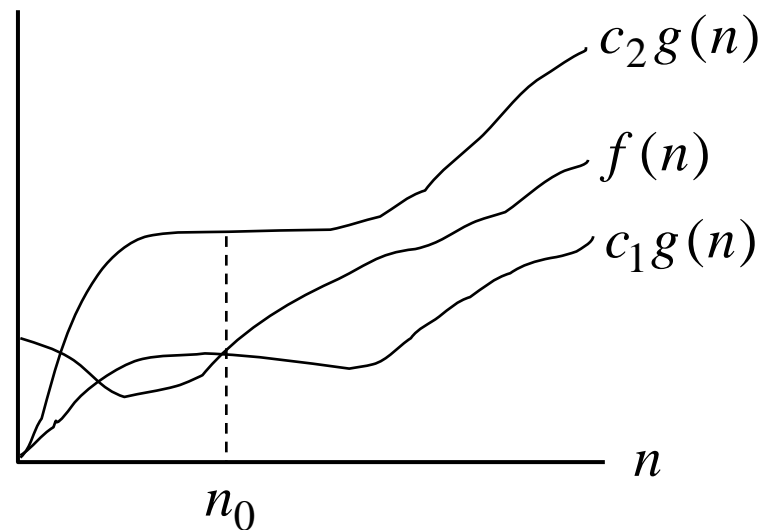
$$f(n) \text{ es } \Omega(g(n))$$

$$(c > 0)$$



Repaso de conceptos

- Órdenes de crecimiento



$f(n)$ es del orden de magnitud de $g(n)$

$$f(n) \text{ es } \Theta(g(n))$$

$$(c_1, c_2 > 0)$$



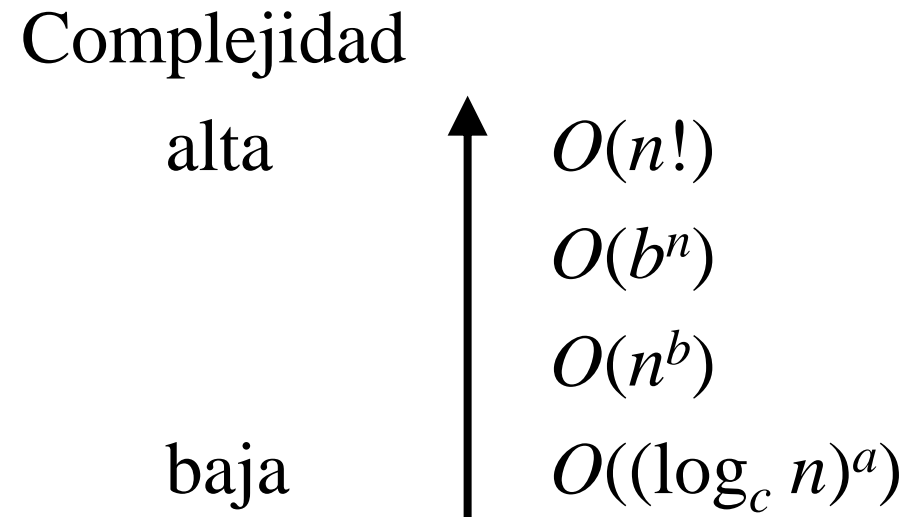
Repaso de conceptos

- Propiedades del orden de crecimiento O :
 - Propiedad 1.
 $\forall 0 \leq a \leq b: n^a$ es $O(n^b)$.
Además n^a es de orden inferior a n^b si $a < b$ y son del mismo orden si $a = b$.
 - Propiedad 2.
 $\forall a \geq 0, b > 0, c > 1: (\log_c n)^a$ es $O(n^b)$ y es de orden inferior.
 - Propiedad 3.
 $\forall a \geq 0, b > 1: n^a$ es $O(b^n)$ y es de orden inferior.
 - Propiedad 4.
 $\forall b > 1: b^n$ es $O(n!)$ y es de orden inferior.



Repaso de conceptos

- Recapitulando... principios generales de O :



Repaso de conceptos

- Recapitulando... principios generales de O :
 - Los factores constantes no importan: si $T(n)$ es $O(f(n))$ entonces $d_1 T(n)$ es $O(d_2 f(n))$, para todas $d_1, d_2 > 0$.
 - Los términos de orden inferior no importan. Por ejemplo, si $T(n) = a_k n^k + a_{k-1} n^{k-1} + \dots + a_0$, con $a_k > 0$, entonces $T(n)$ es $O(n^k)$. De hecho, si $T(n)$ es una suma de funciones $f_i(n)$ positivas entonces $T(n)$ es $O(f_k(n))$, donde $f_k(n)$ es la función de mayor orden entre las distintas $f_i(n)$.
 - La base de logaritmos no importa, porque $\log_b n = \log_c n \cdot \log_b c$, para cualesquiera $b, c > 1$.
 - Las expresiones usando la notación O son transitivas. Si $f(n)$ es $O(g(n))$ y $g(n)$ es $O(h(n))$ entonces $f(n)$ es $O(h(n))$.



Repaso de conceptos

- Coste de algoritmos y notación O :
 - Si el coste de un algoritmo o de una de sus partes es independiente del tamaño de la entrada, se dice que el coste es $O(1)$.
 - Cualquier asignación, lectura de variable, escritura, operación aritmética o comparación tiene coste $O(1)$.
 - Si en alguna expresión interviene una función, se añade el coste de la función.



Repaso de conceptos

- Coste de algoritmos y notación O (cont.):
 - Composición secuencial: regla de las sumas

Si un algoritmo consta de k partes en secuencia, cada una con coste $T_i(n)$ de $O(f_i(n))$, entonces

$$T(n) = T_1(n) + \dots + T_k(n) = O(\max\{f_1(n), \dots, f_k(n)\})$$
 - Composición condicional: también se aplica la regla de las sumas (caso peor...)
 - Composición iterativa: regla de los productos

Si el coste del cuerpo de una iteración es $O(f(n))$ y el número de iteraciones es $O(g(n))$ entonces el coste total de la iteración es $O(f(n) \cdot g(n))$



Repaso de conceptos

- Coste de algoritmos recursivos:

El coste $T(n)$ de una función para datos de tamaño n se define en función del coste $T(m)$ de las llamadas recursivas para otros tamaños m (menores que n)

Teorema (*Master Theorem*):

$$T_1(n) = \begin{cases} f(n) & \text{si } 0 \leq n < c \\ a \cdot T_1(n - c) + b \cdot n^k & \text{si } c \leq n \end{cases} \Rightarrow T_1(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n/c}) & \text{si } a > 1 \end{cases}$$

$$T_2(n) = \begin{cases} g(n) & \text{si } 0 \leq n < c \\ a \cdot T_2(n/c) + b \cdot n^k & \text{si } c \leq n \end{cases} \Rightarrow T_2(n) \in \begin{cases} \Theta(n^k) & \text{si } a < c^k \\ \Theta(n^k \cdot \log n) & \text{si } a = c^k \\ \Theta(n^{\log_c a}) & \text{si } a > c^k \end{cases}$$



Análisis en el caso peor

- El plan:
 - Repaso de conceptos
 - **Montículos y el problema de ordenación**
 - Árboles rojinegros



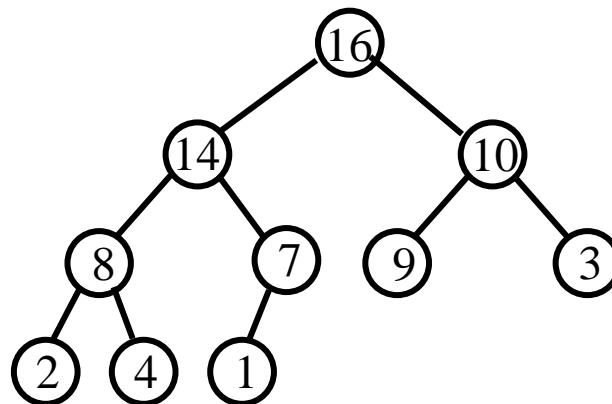
Montículos y el problema de ordenación

- Estructura de datos “montículo”

<i>dato</i>	16	14	10	8	7	9	3	2	4	1	?	?	?
<i>num</i>	11												

$$dato(i) \geq dato(2i), \text{ si } 2i \leq num$$

$$dato(i) \geq dato(2i + 1), \text{ si } 2i + 1 \leq num$$



$$padre(i) = \lfloor i/2 \rfloor$$

$$hijo \begin{cases} izquierdo(i) = 2i \\ derecho(i) = 2i + 1 \end{cases}$$



Montículos y el problema de ordenación

- Mantenimiento de un montículo:

```
algoritmo monticulea(M:montículo; pri,ult:entero)
{Pre: M.dato[pri+1]..M.dato[ult] verifican la
      propiedad del montículo}
```

principio

```
  i:=2*pri; d:=2*pri+1;
```

```
  si i≤ult and M.dato[i]>M.dato[pri]
```

```
    entonces max:=i sino max:=pri
```

```
  fsi;
```

```
  si d≤ult and M.dato[d]>M.dato[max]
```

```
    entonces max:=d
```

```
  fsi;
```

```
  si max≠pri entonces
```

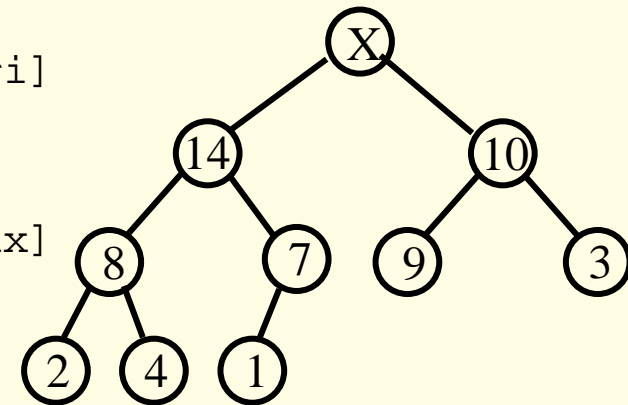
```
    intercambia(M.dato[pri],M.dato[max]);
```

```
    monticulea(M,max,ult)
```

```
  fsi
```

```
fin
```

```
{Post: M.dato[pri]..M.dato[ult] se han permutado y
      verifican la propiedad del montículo.}
```



Montículos y el problema de ordenación

- Coste del mantenimiento de un montículo:
 - El coste de “monticulea” para un subárbol de tamaño n con raíz en el nodo pri es $\Theta(1)$ más...
el coste de “monticulea” para un subárbol cuya raíz es uno de los hijos del nodo pri .
 - El tamaño de los subárboles hijos de pri es, como máximo, $2n/3$ ^(*), por tanto:

$$T(n) \leq T(2n/3) + \Theta(1)$$

- Por el Teorema (*Master*): $T(n)$ es $O(\log n)$

^(*) caso peor: la última fila del árbol está exactamente medio llena



Montículos y el problema de ordenación

- Construcción de un montículo:

```
algoritmo construye_montículo(M:montículo)
principio
    para pri:=M.num div 2 descendiendo hasta 1 hacer
        monticulea(M,pri,M.num)
    fpara
fin
```

Cota superior del coste fácil de calcular:

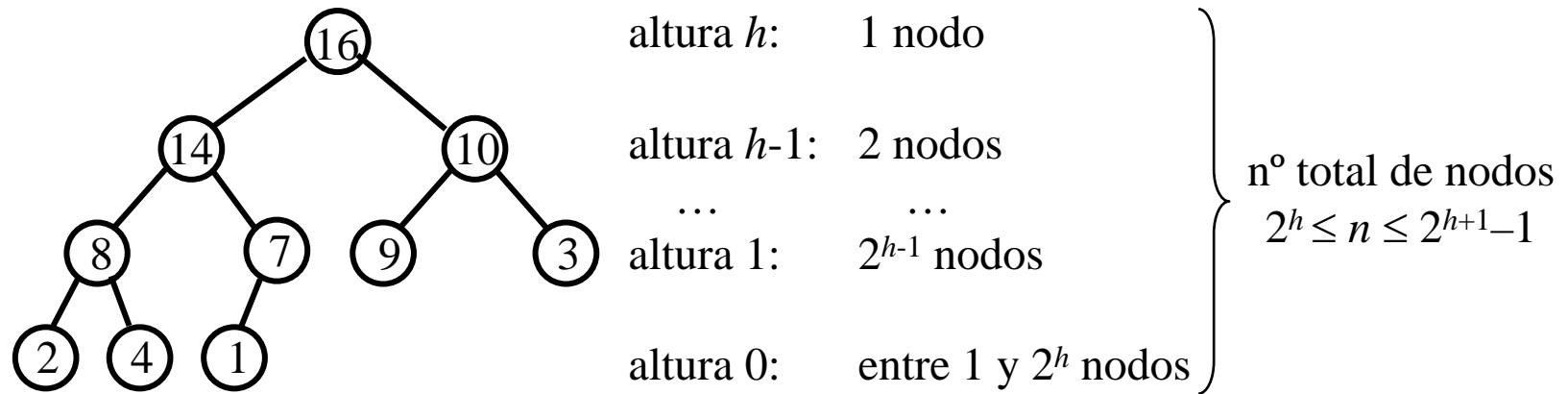
hay $O(n)$ llamadas a “monticulea” y cada una tiene un coste como máximo $O(\log n)$, por tanto el coste está acotado por $O(n \log n)$.

(Esta es una cota correcta pero no muy ajustada...)



Montículos y el problema de ordenación

- Coste de la construcción de un montículo:



(“altura” de un nodo = longitud del camino más largo hasta una hoja)

Cota superior para el coste de “construye_montículo”:

suma de las alturas de todos los nodos del árbol (completo)

$$\begin{aligned}
 S &= \sum_{i=0}^h 2^i (h-i) = h \sum_{i=0}^h 2^i - \sum_{i=0}^h i 2^i \quad \underset{\substack{\uparrow \\ \text{chuleta}}}{=} h(2^{h+1}-1) - (h2^{h+2} - (h+1)2^{h+1} + 2) \\
 &= (2^{h+1}-1) - (h+1) = n - (h+1) \in O(n)
 \end{aligned}$$



Montículos y el problema de ordenación

- Ordenación por el método del montículo:

```
algoritmo ordena (M:montículo)
principio
    construye_montículo (M) ;
    para ult:=M.num descendiendo hasta 2 hacer
        intercambia (M.dato[1] , m.dato[ult]) ;
        monticulea (M,1,ult-1)
    fpara
fin
```

Coste: $O(n \log n)$, porque “construye_montículo” es $O(n)$ y cada una de las $n-1$ llamadas a “monticulea” es $O(\log n)$.



Análisis en el caso peor

- El plan:
 - Repaso de conceptos
 - Montículos y el problema de ordenación
 - **Árboles rojinegros**



TAD Diccionario

- Tipo Abstracto de Datos (TAD):
 - Definición de un tipo de datos independientemente de implementaciones concretas
 - Especificar el conjunto de valores posibles y el conjunto de operaciones que se utilizan para manipular esos valores
 - Definición formal (especificación algebraica)
- TAD Diccionario:
 - Un TAD que almacena elementos, o valores. A un valor se accede por su clave. Operaciones básicas: crear, insertar, buscar y borrar.



Árboles rojinegros

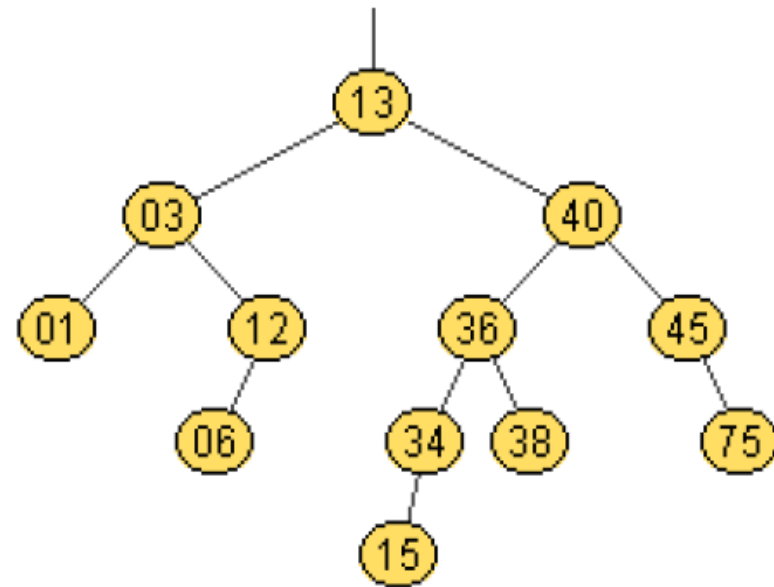
- ¿Para qué sirven?

Son una clase de árboles binarios de búsqueda
“equilibrados”

(altura máxima logarítmica en el número de nodos)
para garantizar coste $O(\log n)$ de las operaciones
básicas del TAD diccionario.

Recordar:

- Árboles binarios de búsqueda (abb)
- AVL (abb's equilibrados)



Árboles rojinegros

- Definición:
 - Árbol binario de búsqueda con un bit adicional en cada **nodo**, su *color*, que puede ser **rojo** o **negro**.
 - Ciertas **condiciones sobre los colores** de los nodos garantizan que la profundidad de ninguna hoja es más del doble que la de ninguna otra (el árbol está “algo equilibrado”).
 - Cada nodo es un registro con: *color* (rojo o negro), *clave* (la clave de búsqueda), y tres *punteros* a los *hijos* (i, d) y al *padre* (p).
Si los punteros son NIL, imaginaremos que son punteros a nodos “externos” (*hojas*).



Árboles rojinegros

- Condiciones rojinegras:

RN_1 - Cada nodo es rojo o negro.

RN_2 - Toda hoja (NIL) es negra.

RN_3 - Si un nodo es rojo, sus dos hijos son negros.

(no puede haber dos rojos consecutivos en un camino)

RN_4 - Todo camino desde un nodo a cualquier hoja descendente contiene el mismo número de nodos negros.

(cada nodo "real" tiene dos hijos)

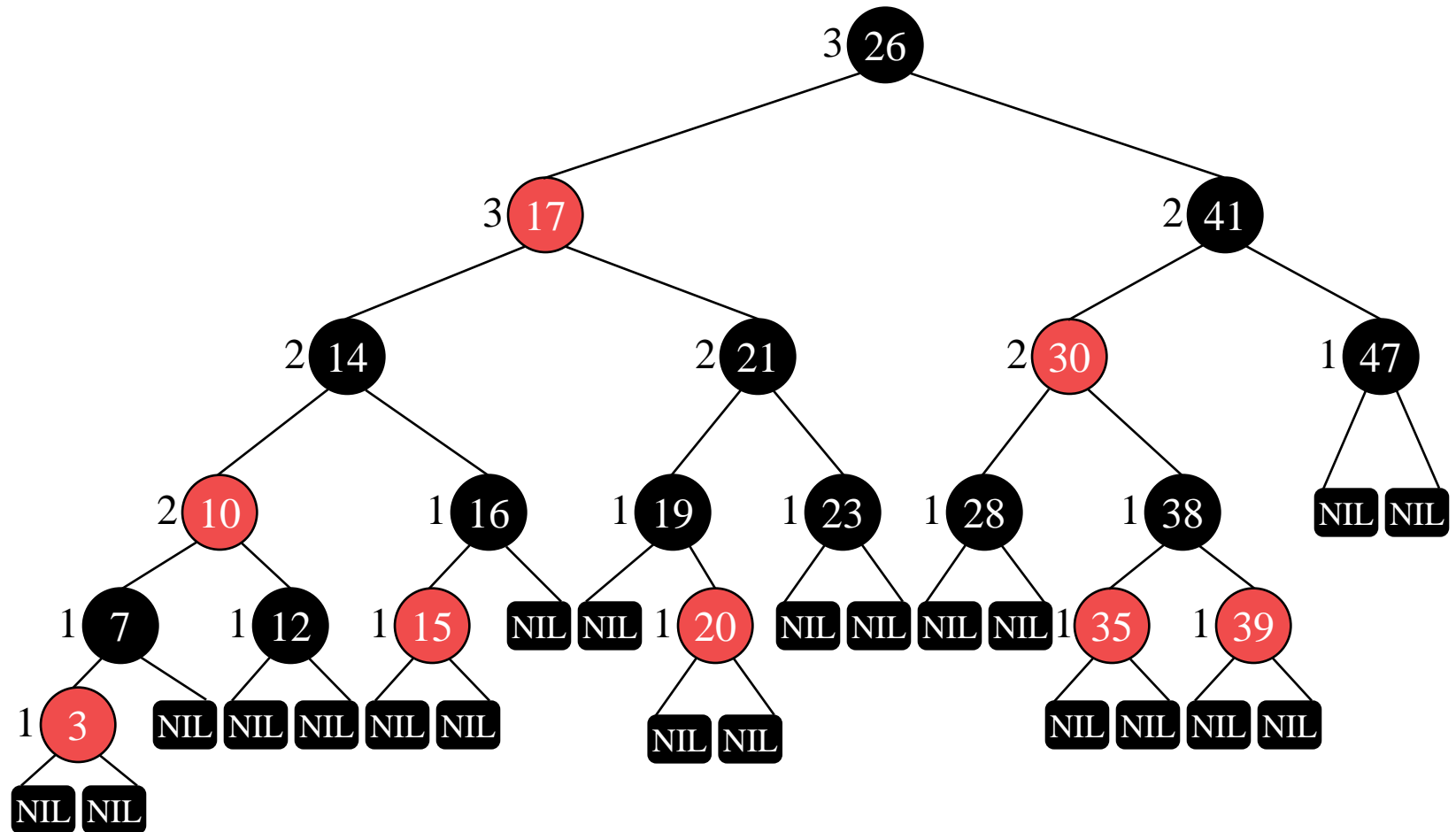
- Terminología:

- *Altura negra de un nodo x , $an(x)$* : es el número de nodos negros desde x (sin incluir x) hasta cualquier hoja descendente de x .

- *Altura negra de un árbol*: la altura negra de su raíz.



Árboles rojinegros



Árboles rojinegros

- Lema: Todo árbol rojinegro con n nodos internos tiene una altura menor o igual que $2 \log(n+1)$.

Demostración:

- El subárbol con raíz x contiene al menos $2^{an(x)}-1$ nodos internos. Por inducción en la altura de x :
 - Caso $altura(x)=0$: x debe ser una hoja (NIL), y el subárbol con raíz x contiene en efecto $2^{an(x)}-1 = 2^0-1 = 0$ nodos internos
 - Paso de inducción: considerar un nodo interno x con dos hijos; sus hijos tienen altura negra $an(x)$ ó $an(x)-1$, dependiendo de si su color es rojo o negro, respectivamente. Por hipótesis de inducción, los dos hijos de x tienen al menos $2^{an(x)-1}-1$ nodos internos. Por tanto el subárbol con raíz x tiene al menos $(2^{an(x)-1}-1)+(2^{an(x)-1}-1)+1 = 2^{an(x)}-1$ nodos internos.



Árboles rojinegros

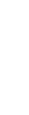
Demostración (cont.):

- [El subárbol con raíz x contiene al menos $2^{an(x)}-1$ nodos internos (ya demostrado).]
- Sea h la altura del árbol.

Por definición (RN_3 : si un nodo es rojo, sus dos hijos son negros), al menos la mitad de los nodos de cualquier camino de la raíz a una hoja (sin incluir la raíz) deben ser negros.

Por tanto, la altura negra de la raíz debe ser al menos $h/2$, luego:

$$n \geq 2^{h/2}-1 \Rightarrow \log(n+1) \geq h/2 \Rightarrow h \leq 2 \log(n+1).$$



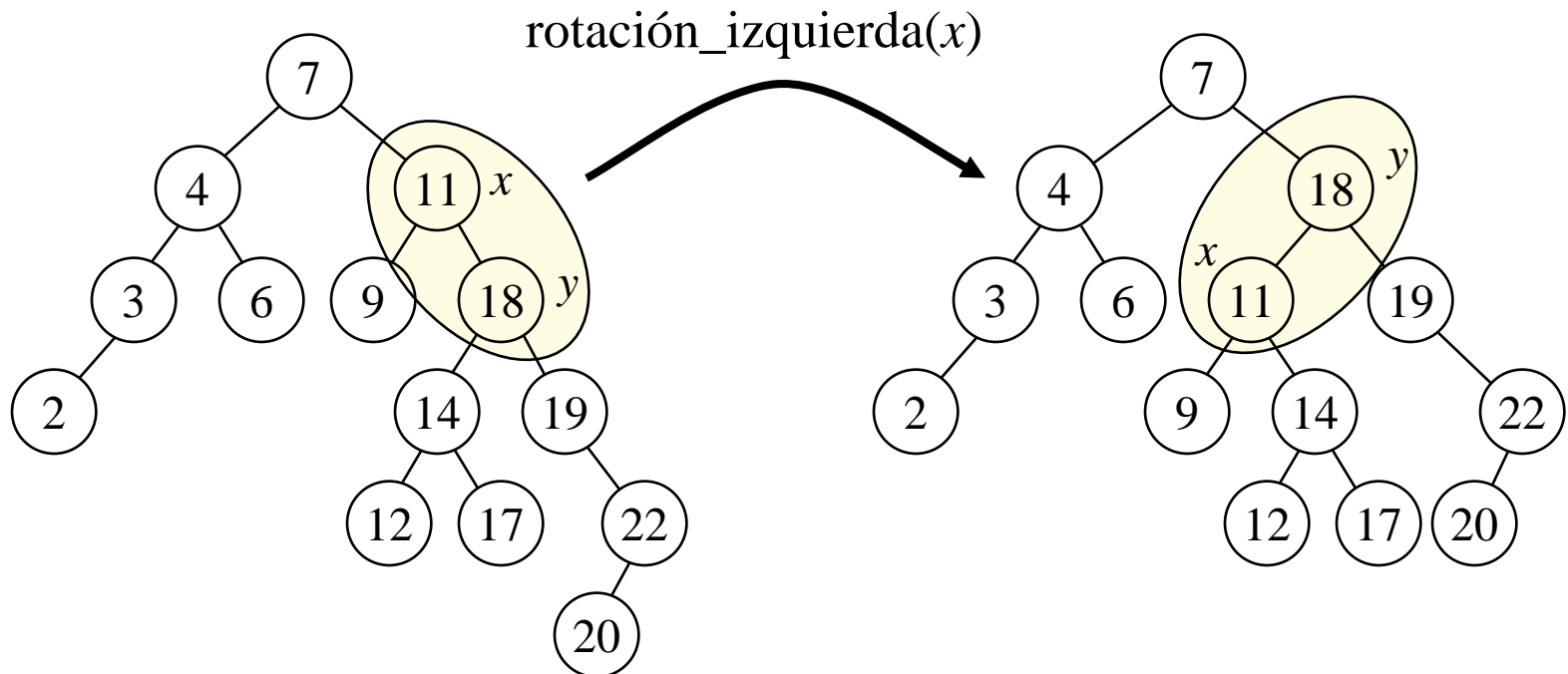
Árboles rojinegros

- Consecuencias del lema:
 - Las operaciones de búsqueda pueden implementarse en tiempo $O(\log n)$ para árboles rojinegros con n nodos.
- ¿Y la inserción y el borrado?
 - Veremos a continuación que también pueden realizarse en tiempo $O(\log n)$ sin que el árbol deje de ser rojinegro.



Árboles rojinegros

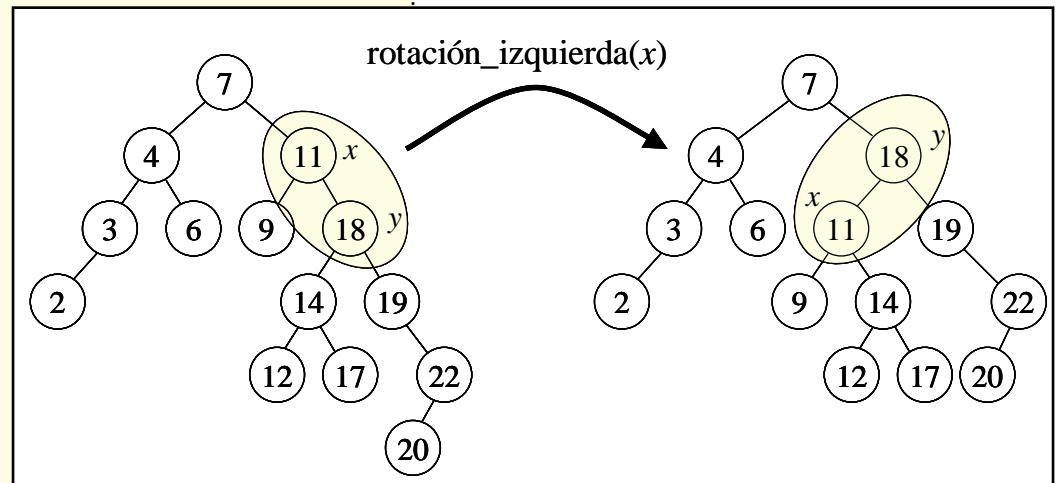
- Rotaciones:
 - Cambios locales de la estructura de un árbol de búsqueda que lo mantienen con los mismos datos y preservando las características de árbol de búsqueda.



Árboles rojinegros

```
algoritmo rota_i(A,x)
principio
  y:=d(x);
  d(x):=i(y);
  si i(y)≠NIL ent p(i(y)):=x fsi;
  p(y):=p(x);
  si p(x)=NIL ent
    raíz(A):=y
  sino
    si x=i(p(x)) ent
      i(p(x)):=y
    sino
      d(p(x)):=y
    fsi;
  i(y):=x;
  p(x):=y
fin
```

La rotación derecha
es simétrica.



Árboles rojinegros

- Inserción:
 - En primer lugar se realiza una inserción en el árbol de búsqueda (sin preocuparse del color de los nodos).
 - Si tras la inserción se viola alguna de las condiciones RN_{1-4} , hay que modificar la estructura, usando rotaciones como la anterior y cambiando los colores de los nodos.
 - Hipótesis que haremos: la raíz es siempre negra (la implementación que sigue a continuación garantiza que la raíz sigue siendo negra tras la inserción).



Inserción

1º) ¿en qué se rompe la def. de árbol r-n en las líneas 1-2?

2º) ¿cuál es el objeto del bucle 3-19?

3º) ¿qué se hace en cada uno de los casos?

algoritmo inserta(A,x)

principio

1 inserta_abb(A,x);

2 color(x):=rojo;

3 **mq** x≠raíz(A) and color(p(x))=rojo **hacer**

4 **si** p(x)=i(p(p(x))) **ent**

5 y:=d(p(p(x)));

6 **si** color(y)=rojo **ent**

7 color(p(x)):=negro;

8 color(y):=negro;

9 color(p(p(x))):=rojo;

10 x:=p(p(x))

Caso 1

11 **sino**

Caso 2

12 **si** x=d(p(x)) **ent** x:=p(x); rota_i(A,x) **fsi**;

13 color(p(x)):=negro;

14 color(p(p(x))):=rojo;

15 rota_d(A,p(p(x)))

Caso 3

16 **fsi**

17 **sino** [igual que arriba, intercambiando i/d]

18 **fsi**

Casos 4, 5 y 6

19 **fmq**;

20 color(raíz(A)):=negro

fin

Inserción

1º) ¿en qué se rompe la def. de árbol r-n en las líneas 1-2?

✓ RN_1 : Cada nodo es rojo o negro.

✓ RN_2 : Toda hoja (NIL) es negra.

✗ RN_3 : Si un nodo es rojo, sus dos hijos son negros.

✓ RN_4 : Todo camino desde un nodo a cualquier hoja descendente contiene el mismo número de nodos negros.

```
algoritmo inserta(A,x)
principio
1  inserta_abb(A,x);
2  color(x):=rojo;
3  mq x≠raíz(A) and color(p(x))=rojo hacer
4      si p(x)=i(p(p(x))) ent
5          y:=d(p(p(x)));
6          si color(y)=rojo ent
7              color(p(x)):=negro;
8              color(y):=negro;
9              color(p(p(x))):=rojo;
10             x:=p(p(x))
11         sino
12             si x=d(p(x)) ent x:=p(x); rota_i(A,x) fsi;
13             color(p(x)):=negro;
14             color(p(p(x))):=rojo;
15             rota_d(A,p(p(x)))
16         fsi
17     sino [igual que arriba, intercambiando i/d]
18     fsi
19 fmq;
20 color(raíz(A)):=negro
fin
```

```
algoritmo inserta(A,x)
```

```
principio
```

```
1  inserta_abb(A,x);
```

```
2  color(x):=rojo;
```

```
3  mq x≠raíz(A) and color(p(x))=rojo hacer
```

```
4      si p(x)=i(p(p(x))) ent
```

```
5          y:=d(p(p(x)));
```

```
6          si color(y)=rojo ent
```

```
7              color(p(x)):=negro;
```

```
8              color(y):=negro;
```

```
9              color(p(p(x))):=rojo;
```

```
10             x:=p(p(x))
```

```
11         sino
```

```
12             si x=d(p(x)) ent x:=p(x); rota_i(A,x) fsi;
```

```
13             color(p(x)):=negro;
```

```
14             color(p(p(x))):=rojo;
```

```
15             rota_d(A,p(p(x)))
```

```
16         fsi
```

```
17     sino [igual que arriba, intercambiando i/d]
```

```
18     fsi
```

```
19 fmq;
```

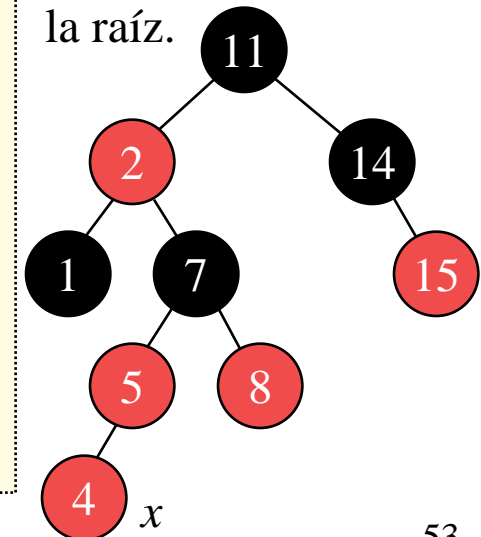
```
20 color(raíz(A)):=negro
```

```
fin
```

Inserción

2º) ¿cuál es el objeto del bucle 3-19?

Resolver el problema anterior, como en el caso de la figura (inserción de x rojo, como hijo de otro rojo), haciendo que el color rojo suba hacia la raíz.



```
algoritmo inserta(A,x)
```

```
principio
```

```
1  inserta_abb(A,x);
```

```
2  color(x):=rojo;
```

```
3  mq x≠raíz(A) and color(p(x))=rojo hacer
```

```
4      si p(x)=i(p(p(x))) ent
```

```
5          y:=d(p(p(x)));
```

```
6          si color(y)=rojo ent
```

```
7              color(p(x)):=negro;
```

```
8              color(y):=negro;
```

```
9              color(p(p(x))):=rojo;
```

```
10             x:=p(p(x))
```

Caso 1

```
11         sino
```

```
12             si x=d(p(x)) ent x:=p(x); rota_i(A,x) fsi;
```

```
13             color(p(x)):=negro;
```

```
14             color(p(p(x))):=rojo;
```

```
15             rota_d(A,p(p(x)))
```

```
16         fsi
```

```
17         sino [igual que arriba, intercambiando i/d]
```

```
18         fsi
```

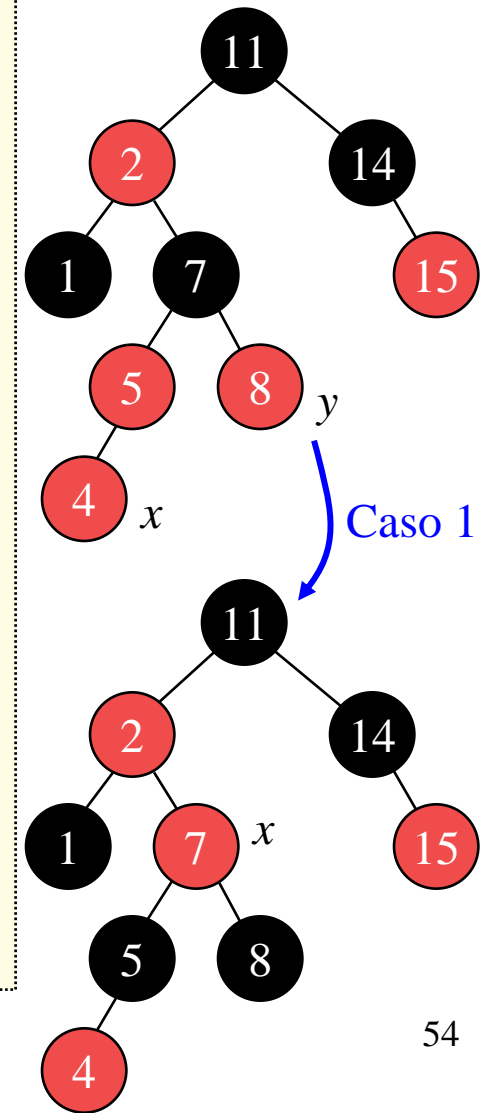
```
19     fmq;
```

```
20     color(raíz(A)):=negro
```

```
fin
```

Inserción

3º) ¿qué se hace en cada uno de los casos?



```
algoritmo inserta(A,x)
```

```
principio
```

```
1  inserta_abb(A,x);
```

```
2  color(x):=rojo;
```

```
3  mq x≠raíz(A) and color(p(x))=rojo hacer
```

```
4      si p(x)=i(p(p(x))) ent
```

```
5          y:=d(p(p(x)));
```

```
6          si color(y)=rojo ent
```

```
7              color(p(x)):=negro;
```

```
8              color(y):=negro;
```

```
9              color(p(p(x))):=rojo;
```

```
10             x:=p(p(x))
```

```
11         sino
```

```
12             si x=d(p(x)) ent x:=p(x); rota_i(A,x) fsi;
```

```
13             color(p(x)):=negro;
```

```
14             color(p(p(x))):=rojo;
```

```
15             rota_d(A,p(p(x)))
```

```
16         fsi
```

```
17     sino [igual que arriba, intercambiando i/d]
```

```
18     fsi
```

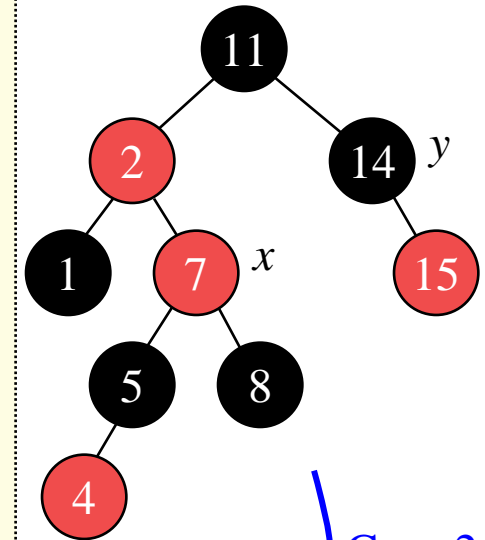
```
19 fmq;
```

```
20 color(raíz(A)):=negro
```

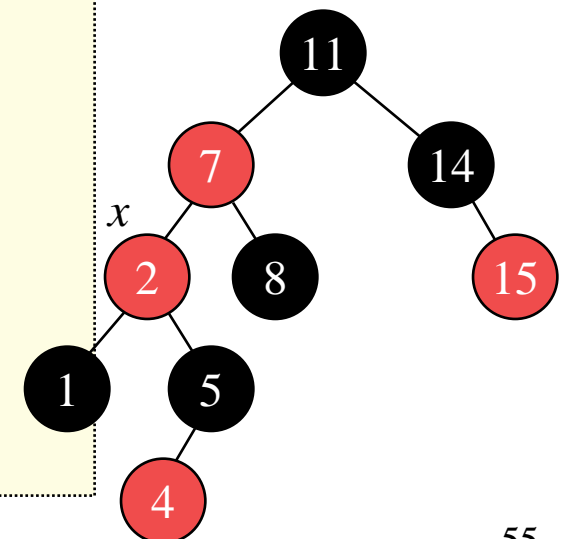
```
fin
```

Caso 2

Inserción



Caso 2




```
algoritmo inserta(A,x)
```

```
principio
```

```
1  inserta_abb(A,x);
```

```
2  color(x):=rojo;
```

```
3  mq x≠raíz(A) and color(p(x))=rojo hacer
```

```
4      si p(x)=i(p(p(x))) ent
```

```
5          y:=d(p(p(x)));
```

```
6          si color(y)=rojo ent
```

```
7              color(p(x)):=negro;
```

```
8              color(y):=negro;
```

```
9              color(p(p(x))):=rojo;
```

```
10             x:=p(p(x))
```

```
11         sino
```

```
12             si x=d(p(x)) ent x:=p(x); rota_i(A,x) fsi;
```

```
13             color(p(x)):=negro;
```

```
14             color(p(p(x))):=rojo;
```

```
15             rota_d(A,p(p(x)))
```

```
16         fsi
```

```
17     sino [igual que arriba, intercambiando i/d]
```

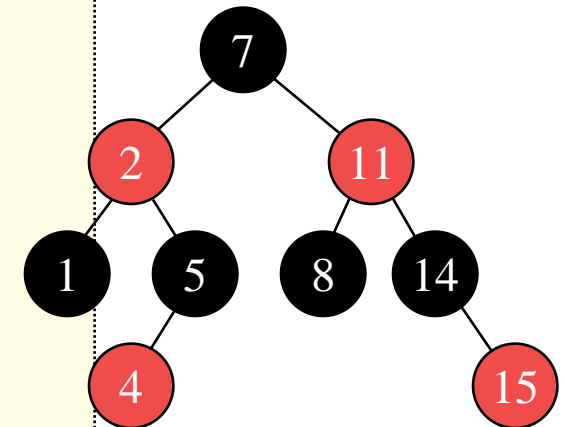
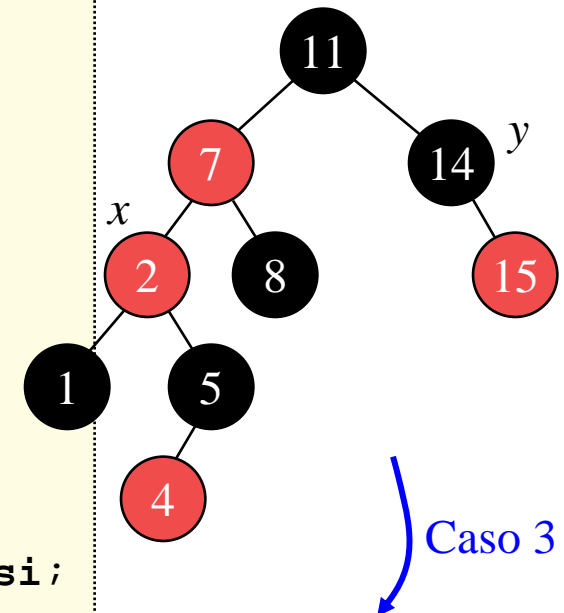
```
18     fsi
```

```
19 fmq;
```

```
20 color(raíz(A)):=negro
```

```
fin
```

Inserción



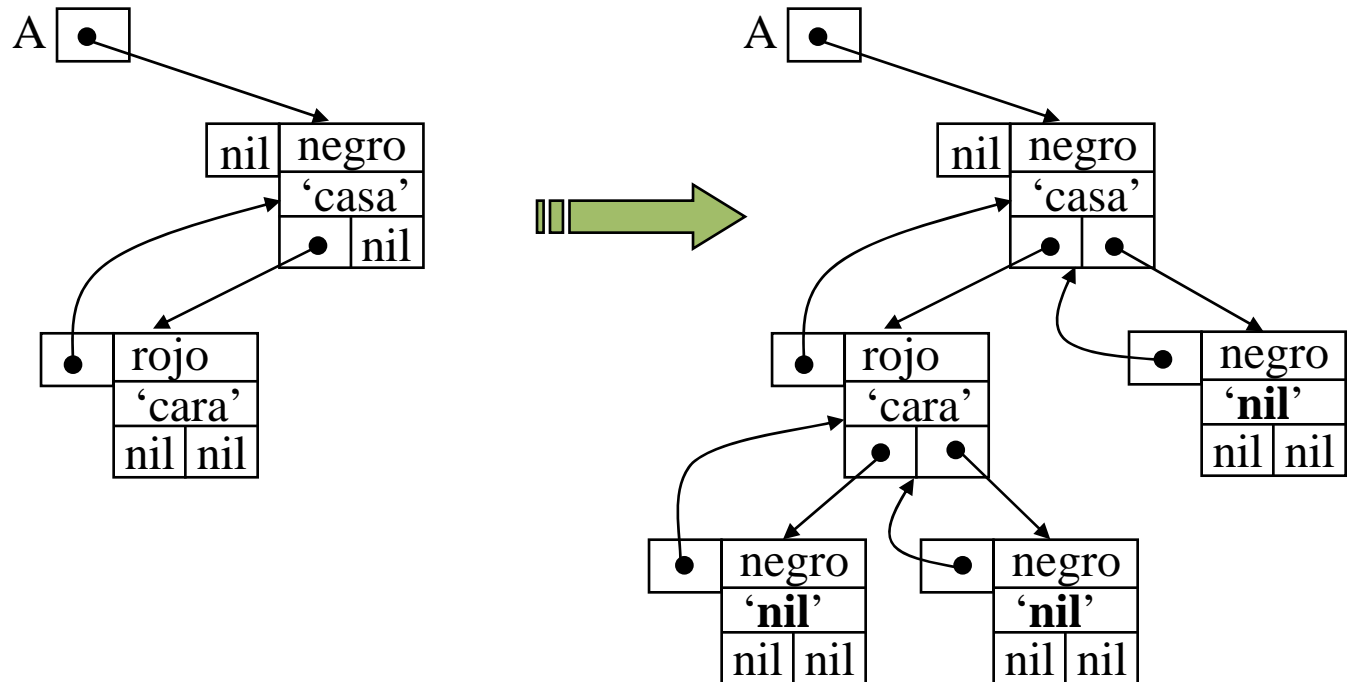
Árboles rojinegros

- Coste de la inserción:
 - La altura de un árbol r-n con n nodos es $O(\log n)$, luego la “inserción_abb” es $O(\log n)$.
 - El bucle sólo se repite si se cumple el caso 1, y en ese caso el puntero x sube en el árbol.
Por tanto el número máximo de repeticiones del bucle es $O(\log n)$.
 - Notar, además, que nunca se hacen más de dos rotaciones
(el bucle termina si se ejecuta el caso 2 o el 3).



Árboles rojinegros

- Borrado:
 - Veremos que también puede hacerse con coste en caso peor en $O(\log n)$.
 - Para simplificar el algoritmo, usaremos representación de *centinela* para el NIL:



Árboles rojinegros

- Borrado: similar al borrado de un “abb”

```
algoritmo borra(A,z)  --z es el puntero al nodo a borrar
principio
1  si clave(i(z))='nil' or clave(d(z))='nil' ent
2    y:=z
3  sino
4    y:=sucesor_abb(z)  --el sucesor en inorden de z
5  fsi;
6  si clave(i(y))≠'nil' ent x:=i(y) sino x:=d(y) fsi;
7  p(x):=p(y);
8  si p(y)='nil' ent
9    raíz(A):=x
10 sino
11   si y=i(p(y)) ent i(p(y)):=x sino d(p(y)):=x fsi
12 fsi;
13 si y≠z ent clave(z):=clave(y) fsi;
14 si color(y)=negro ent fija_borrado(A,x) fsi
fin
```



Árboles rojinegros

- Borrado: descripción del algoritmo paso a paso

```
algoritmo borra(A,z)  --z es el puntero al nodo a borrar
principio
1  si clave(i(z))='nil' or clave(d(z))='nil' ent
2      y:=z
3  sino
4      y:=sucesor_abb(z)  --el sucesor en inorden de z
5  fsi;
    . . .
```

Líneas 1-5: selección del nodo y a colocar en lugar de z.

El nodo y es:

- el mismo nodo z (si z tiene como máximo un hijo), o
- el sucesor de z (si z tiene dos hijos)



Árboles rojinegros

- Borrado: descripción del algoritmo paso a paso

```

. . .
6  si clave(i(y))≠'nil' ent x:=i(y) sino x:=d(y) fsi;
7  p(x):=p(y);
8  si p(y)='nil' ent
9    raíz(A):=x
10 sino
11   si y=i(p(y)) ent i(p(y)):=x sino d(p(y)):=x fsi
12 fsi;
13 si y≠z ent clave(z):=clave(y) fsi;
. . .
```

Línea 6: x es el hijo de y , o es NIL si y no tiene hijos.

Líneas 7-12: se realiza el empalme con y , modificando punteros en $p(y)$ y en x .

Línea 13: si el sucesor de z ha sido el nodo empalmado, se mueve el contenido de y a z , borrando el contenido anterior (en el algoritmo sólo se copia la clave, pero si y tuviese otros campos de información también se copiarían).



Árboles rojinegros

- Borrado: descripción del algoritmo paso a paso

```
    . . .  
14  si color(y)=negro ent fija_borrado(A,x) fsi  
    fin
```

Línea 14: si y es negro, se llama a “fija_borrado”, que cambia los colores y realiza rotaciones para restaurar las propiedades RN_{1-4} de árbol rojinegro .

Si y es rojo las propiedades RN_{1-4} se mantienen (no se han cambiado las alturas negras del árbol y no se han hecho adyacentes dos nodos rojos).

El nodo x pasado como argumento era el hijo único de y antes de que y fuese empalmado o el centinela del NIL si y no tenía hijos.



Árboles rojinegros

- Borrado: fijado de propiedades RN_{1-4}
 - **Problema:** Si el nodo y del algoritmo anterior era negro, al borrarlo, todo camino que pase por él tiene un nodo negro menos, por tanto no se cumple RN_4
“todo camino desde un nodo a cualquier hoja descendente contiene el mismo número de nodos negros”
para ningún antecesor de y
 - **Solución:** interpretar que el nodo x tiene un color negro “extra”, así “se cumple” RN_4 , es decir, al borrar el nodo y “empujamos su negrura hacia su hijo”
 - **Nuevo problema:** ahora el nodo x puede ser “dos veces negro”, y por tanto no verifica RN_1
 - **Solución:** aplicar el algoritmo “fija_borrado” para restaurar RN_1



Árboles rojinegros

- Borrado: fijado de propiedades rojinegras

```
algoritmo fija_borrado(A,x)
principio
1  mq x≠raíz(A) and color(x)=negro hacer
2    . . .
.    . . .
28  fmq;
29  color(x) := negro
fin
```

Objetivo del **bucle**: mover el “negro extra” hacia arriba hasta que

- 1) x es rojo, y se acaba el problema, o
- 2) x es la raíz, y el “negro extra” se “borra” (él solito)

En el interior del **bucle**, x es siempre un nodo negro distinto de la raíz y que tiene un “negro extra”.



Árboles rojinegros

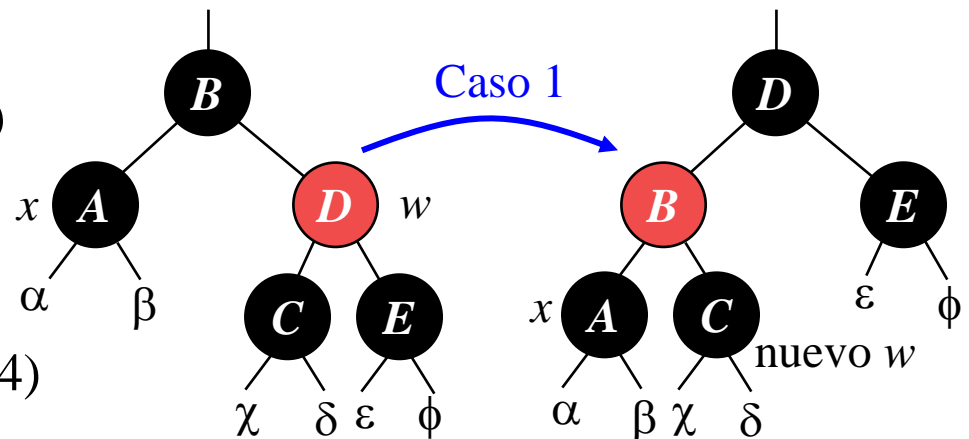
• Interior del bucle:

```

. . .
2   si x=i(p(x)) ent
3       w:=d(p(x));
4       si color(w)=rojo ent
5           color(w):=negro;
6           color(p(x)):=rojo;
7           rota_i(A,p(x));
8           w:=d(p(x))
9       fsi; {objetivo: conseguir
. . .         un hermano negro para x}
    
```

- x es hijo izquierdo
- w es el hermano de x
- x es “negro doble” \Rightarrow
 \Rightarrow clave(w) \neq nil
 (si no, el n° de negros desde $p(x)$ a la hoja w sería menor que desde $p(x)$ a x)

- w debe tener un hijo negro
- se cambia el color de w y $p(x)$ y se hace “rota_i” con $p(x)$
- el nuevo hermano de x , uno de los hijos de w , es negro, y se pasa a los otros casos (2-4)



Árboles rojinegros

- Interior del bucle (cont.):

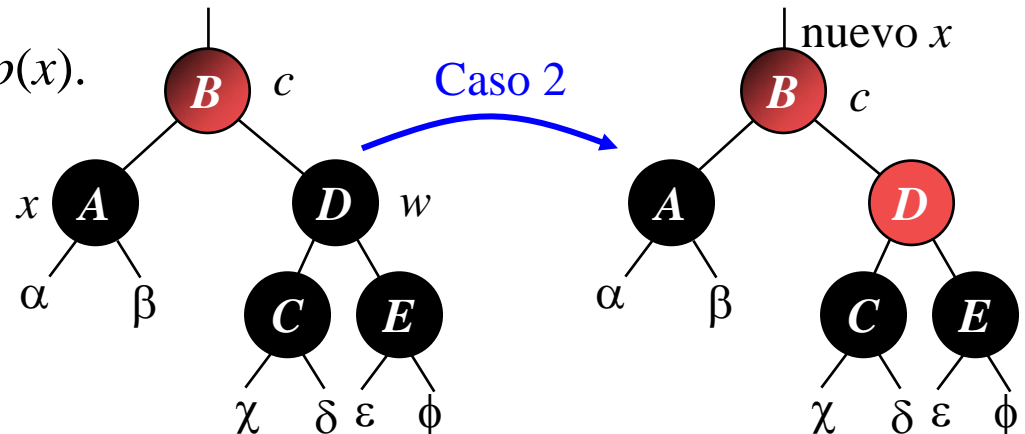
```

    . . .
10     si color(i(w))=negro and color(d(w))=negro ent
11         color(w):=rojo;
12         x:=p(x)
13     sino
    . . .

```

} Caso 2

- El nodo w (hermano de x) y los dos hijos de w son negros.
- Se quita un negro de x y “otro” de w (x queda con un solo negro y w rojo).
- Se añade un negro a $p(x)$.
- Se repite el bucle con $x:=p(x)$.
- Si se había entrado a este caso desde el caso 1, el color del nuevo x es rojo y el bucle termina.



Árboles rojinegros

- Interior del bucle (cont.):

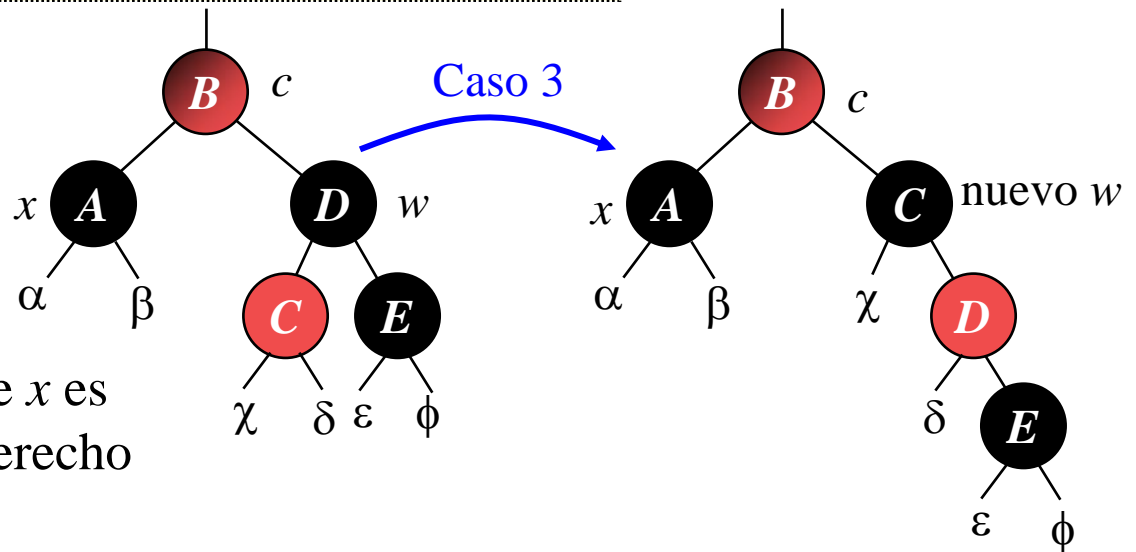
```

. . .
14      si color(d(w))=negro ent
15          color(i(w)):=negro;
16          color(w):=rojo;
17          rota_d(A,w);
18          w:=d(p(x))
19      fsi;
. . .

```

} Caso 3

- w y $d(w)$ son negros
- $i(w)$ es rojo
- Se cambia el color de w y de $i(w)$ y se hace “rota_d” sobre w .
- El nuevo hermano w de x es ahora negro con hijo derecho rojo, y ese es el caso 4.

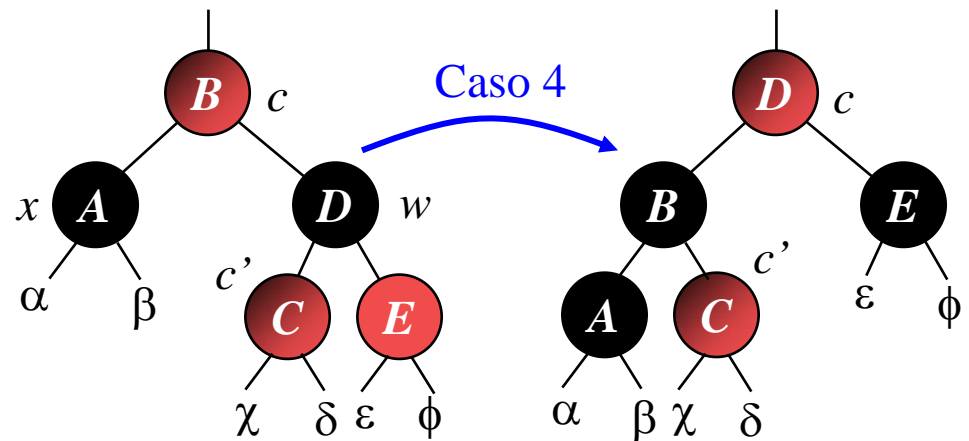


Árboles rojinegros

- Interior del bucle (cont.):

```
    . . .  
20     color(w):=color(p(x));  
21     color(p(x)):=negro;  
22     color(d(w)):=negro;  
23     rota_i(A,p(x));  
24     x:=raíz(A)  
25     fsi  
26     sino [igual que 3-25, intercambiando i/d]  
27     fsi  
    . . .
```

- w es negro y $d(w)$ es rojo
- Se cambian algunos colores y se hace “rota_i” sobre $p(x)$ y así se borra el negro “extra” de x .



Árboles rojinegros

- Coste del borrado:
 - El coste del algoritmo a excepción de la llamada a “fija_borrado” es $O(\log n)$, porque esa es la altura del árbol.
 - En “fija_borrado”, los casos 1, 3 y 4 terminan tras un nº constante de cambios de color y como máximo tres rotaciones.
 - El caso 2 es el único que puede hacer repetir el bucle mq, y en ese caso el nodo x se mueve hacia arriba en el árbol como máximo $O(\log n)$ veces y sin hacer rotaciones.
 - Por tanto, “fija_borrado”, y también el borrado completo, son $O(\log n)$ y hacen como mucho tres rotaciones.



Análisis del caso promedio



Análisis del caso promedio

- El plan:
 - **Probabilidad**
 - Análisis probabilista
 - Árboles binarios de búsqueda contruidos aleatoriamente
 - Tries, árboles digitales de búsqueda y Patricia
 - Listas “skip”
 - Árboles aleatorizados



Probabilidad

- Espacio probabilizable: (S, \mathcal{F})
 - S : conjunto de *sucesos elementales*
 - \mathcal{F} : *tribu* de *sucesos* de S (familia de subconjuntos de S que verifica *ciertas* propiedades)
- Espacio de probabilidad: $(S, \mathcal{F}, \text{Pr})$
 - (S, \mathcal{F}) : espacio probabilizable
 - Pr : *probabilidad* es una *función de medida* no negativa sobre (S, \mathcal{F}) tal que $\text{Pr}\{S\} = 1$
(S se llama *succeso seguro*)



Probabilidad

- Caso particular que nos interesa:

S es un conjunto numerable (finito o infinito)

$\mathcal{F} = \{A \mid A \subseteq S\}$ (conjunto de las *partes* de S)

\Pr es una aplicación de \mathcal{F} en los reales tal que:

- $\Pr\{A\} \geq 0$, para todo $A \subseteq S$

- $\Pr\{S\} = 1$

- $\Pr\left\{\bigcup_{i=1}^{\infty} A_i\right\} = \sum_{i=1}^{\infty} \Pr\{A_i\},$

para toda familia $\{A_i\}_{i \in \mathbf{N}}$,

$$A_i \cap A_j = \emptyset \text{ si } i \neq j$$

Axiomas de Kolmogorov



Probabilidad

- Consecuencias:
 - El *suceso vacío* tiene probabilidad 0, $\Pr\{\emptyset\} = 0$
 - Si $A \subseteq B$ entonces $\Pr\{A\} \leq \Pr\{B\}$
 - Si $\bar{A} = S - A$ entonces $\Pr\{\bar{A}\} = 1 - \Pr\{A\}$
 - $\Pr\{A \cup B\} = \Pr\{A\} + \Pr\{B\} - \Pr\{A \cap B\} \leq \Pr\{A\} + \Pr\{B\}$



Probabilidad

- Ejemplo:
 - Experimento: lanzar dos monedas simultáneamente
 - Conjunto de sucesos elementales:
 $S = \{CC, CX, XC, XX\}$
 - Ejemplo de suceso ($A \in \mathcal{F}$):
“obtener una cara y una cruz” es $A = \{CX, XC\}$
 - Si $\Pr\{CC\} = \Pr\{CX\} = \Pr\{XC\} = \Pr\{XX\} = 1/4$,
entonces $\Pr\{\text{“obtener al menos una cara”}\} =$
 $\Pr\{CC, CX, XC\} = \Pr\{CC\} + \Pr\{CX\} + \Pr\{XC\} = 3/4$
O también: $\Pr\{\text{“obtener al menos una cara”}\} =$
 $1 - \Pr\{\text{“obtener menos de una cara”}\} = 1 - \Pr\{XX\} = 3/4$



Probabilidad

- Función de probabilidad *discreta*:
 - Está definida sobre un conjunto S numerable
 - Para todo $A \subseteq S$, $\Pr\{A\} = \sum_{s \in A} \Pr\{s\}$
 - Probabilidad discreta *uniforme*:
 - Si S es finito y $\Pr\{s\} = 1/|S|$, para todo $s \in S$
(el experimento se llama “elegir un elemento de S al azar”,
y el espacio de probabilidad se llama “clásico”)
 - Ejemplo: lanzar una moneda (“buena”) n veces
Sucesos elementales: secuencias de C ó X de longitud n
Cada suceso elemental (hay 2^n) tiene probabilidad $1/2^n$
Si $A = \{k \text{ caras y } n-k \text{ cruces}\}$ entonces

$$|A| = \binom{n}{k}, \text{ y por tanto } \Pr\{A\} = \binom{n}{k} / 2^n$$



Probabilidad

- Probabilidad continua uniforme:
 - Definida sobre $S = [a, b]$, intervalo de los reales, $a < b$
 - Intuitivamente: desearíamos que cada punto del intervalo tuviese igual probabilidad

Como el número de puntos es no numerable, si damos a cada punto una probabilidad positiva es **imposible** que se satisfaga simultáneamente

$$\Pr\{S\} = 1, \quad \text{y}$$

$$\Pr\left\{\bigcup_{i=1}^{\infty} A_i\right\} = \sum_{i=1}^{\infty} \Pr\{A_i\}$$



Probabilidad

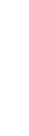
- Probabilidad continua uniforme (cont.):
 - Definimos, para cada intervalo cerrado $[c,d]$, $a \leq c \leq d \leq b$:

$$\Pr\{[c,d]\} = \frac{d-c}{b-a}$$

En particular, para cada punto $x = [x,x]$, $\Pr\{x\} = 0$.

Luego, $\Pr\{(c,d)\} = \Pr\{[c,d]\}$, porque $[c,d] = [c,c] \cup (c,d) \cup [d,d]$.

Por tanto, la tribu de sucesos (para los que está definida \Pr) es cualquier subconjunto de $[a,b]$ que puede obtenerse como unión finita o infinita numerable de intervalos abiertos y cerrados.



Probabilidad

- Probabilidad condicionada:
 - Probabilidad de un suceso teniendo una información parcial a priori de lo sucedido.
 - Ejemplo: se han tirado dos monedas (“buenas”) y sabemos que al menos una de ellas salió cara.
¿Cuál es la probab. de que las dos hayan salido cara?
La información a priori elimina la posibilidad de que hayan sido dos cruces. Los tres sucesos restantes deben ser equiprobables, cada uno con probab. $1/3$.
Como sólo uno de esos sucesos corresponde a dos caras, la respuesta es $1/3$.



Probabilidad

- Probabilidad condicionada (cont.):
 - Definición: probabilidad condicionada de A dado B

$$\Pr\{A \mid B\} = \frac{\Pr\{A \cap B\}}{\Pr\{B\}}, \text{ siempre que } \Pr\{B\} \neq 0$$

En el ejemplo:

A = “las dos son caras”

B = “al menos una es cara”

por tanto, $\Pr\{A \mid B\} = (1/4)/(3/4) = 1/3$



Probabilidad

- Independencia:

- Dos sucesos A y B son *independientes* si

$$\Pr\{A \cap B\} = \Pr\{A\}\Pr\{B\}$$

Si $\Pr\{B\} \neq 0$, lo anterior es equivalente a

$$\Pr\{A \mid B\} = \Pr\{A\}$$

- Los sucesos A_1, A_2, \dots, A_n son:

- *Independientes dos a dos* si

$$\Pr\{A_i \cap A_j\} = \Pr\{A_i\}\Pr\{A_j\} \text{ para todo } i \neq j.$$

- *Mutuamente independientes* si cada subconjunto

$$A_{i_1}, A_{i_2}, \dots, A_{i_k}, \text{ con } 2 \leq k \leq n \text{ y}$$

$$1 \leq i_1 < i_2 < \dots < i_k \leq n, \text{ satisface}$$

$$\Pr\{A_{i_1} \cap A_{i_2} \cap \dots \cap A_{i_k}\} = \Pr\{A_{i_1}\}\Pr\{A_{i_2}\} \dots \Pr\{A_{i_k}\}$$



Probabilidad

- Ejemplo de independencia dos a dos y mutua:

- Lanzamos dos monedas (“buenas”)

- A_1 = “la primera sale cara”

- A_2 = “la segunda sale cara”

- A_3 = “las dos salen distintas”

Entonces: $\Pr\{A_1\} = \Pr\{A_2\} = \Pr\{A_3\} = 1/2$

$$\Pr\{A_1 \cap A_2\} = \Pr\{A_1 \cap A_3\} = \Pr\{A_2 \cap A_3\} = 1/4$$
$$\Pr\{A_1 \cap A_2 \cap A_3\} = 0$$

Como para $1 \leq i < j \leq 3$, $\Pr\{A_i \cap A_j\} = \Pr\{A_i\} \Pr\{A_j\} = 1/4$,
los sucesos A_1 , A_2 y A_3 son independientes dos a dos.

Sin embargo, no son mutuamente independientes,

porque $\Pr\{A_1 \cap A_2 \cap A_3\} = 0$ pero

$$\Pr\{A_1\} \Pr\{A_2\} \Pr\{A_3\} = 1/8 \neq 0$$


Probabilidad

- Teorema de Bayes. Si los sucesos A y B tienen probabilidad no nula, entonces

$$\Pr\{A \mid B\} = \frac{\Pr\{A\} \Pr\{B \mid A\}}{\Pr\{B\}}$$

Dem: por definición de probabilidad condicionada

$\Pr\{A \cap B\} = \Pr\{B\} \Pr\{A \mid B\} = \Pr\{A\} \Pr\{B \mid A\}$, y despejando $\Pr\{A \mid B\}$ se sigue.

Equivalentemente:

$$\Pr\{A \mid B\} = \frac{\Pr\{A\} \Pr\{B \mid A\}}{\Pr\{A\} \Pr\{B \mid A\} + \Pr\{\bar{A}\} \Pr\{B \mid \bar{A}\}}$$

porque:

$$\Pr\{B\} = \Pr\{B \cap A\} + \Pr\{B \cap \bar{A}\} = \Pr\{A\} \Pr\{B \mid A\} + \Pr\{\bar{A}\} \Pr\{B \mid \bar{A}\}$$



Probabilidad

- Ejemplo de aplicación del teorema de Bayes:

Tenemos una moneda “buena” y otra sesgada que siempre cae de cara. Experimento:

- Elegimos una al azar
- La lanzamos una vez
- La lanzamos otra vez

Si sale cara las dos veces, ¿cuál es la probabilidad de que la elegida haya sido la sesgada?

A = “la elegida es la sesgada”,

B = “sale cara las dos veces”,

$\Pr\{A\} = 1/2$; $\Pr\{B|A\} = 1$; $\Pr\{\bar{A}\} = 1/2$; $\Pr\{B|\bar{A}\} = 1/4$

luego:

$$\Pr\{A | B\} = \frac{(1/2) \cdot 1}{(1/2) \cdot 1 + (1/2) \cdot (1/4)} = 4/5$$



Probabilidad

- Variable aleatoria discreta (v.a.d.):
 - Es una función X de un conjunto numerable (finito o infinito) de sucesos elementales S en los reales.
 - Para una v.a.d. X y un real x se define el suceso $X = x$ como $\{s \in S \mid X(s) = x\}$, por tanto:

$$\Pr\{X = x\} = \sum_{\{s \in S \mid X(s)=x\}} \Pr\{s\}$$

- *Función de densidad de probabilidad* de la v.a.d. X :

$$f(x) = \Pr\{X = x\}$$

Por los axiomas de Kolmogorov, $f(x) \geq 0$ y $\sum_x f(x) = 1$



Probabilidad

- Ejemplo: experimento de lanzar dos dados
 - 36 sucesos elementales
 - Si los dados son “legales”, la probabilidad es uniforme y $\Pr\{s\} = 1/36$, para todo $s \in S$
 - Ejemplo de v.a.d.:
 $X = \text{máximo de los valores obtenidos en los dos dados}$
 $f(3) = \Pr\{X = 3\} = 5/36$, porque X asigna el valor 3 a 5 de los 36 sucesos elementales posibles: (1,3), (2,3), (3,3), (3,2) y (3,1)



Probabilidad

- Densidad conjunta:
 - Si las v.a.d. X e Y están definidas sobre el mismo espacio de probabilidad, la *función de densidad conjunta* es:

$$f(x,y) = \Pr\{(X = x) \wedge (Y = y)\}$$

Y se tiene que:

$$\Pr\{Y = y\} = \sum_x \Pr\{(X = x) \wedge (Y = y)\}$$

$$\Pr\{X = x\} = \sum_y \Pr\{(X = x) \wedge (Y = y)\}$$

$$\Pr\{X = x | Y = y\} = \frac{\Pr\{(X = x) \wedge (Y = y)\}}{\Pr\{Y = y\}}$$

- Variables independientes:
 - X e Y son *independientes* si
$$\Pr\{(X = x) \wedge (Y = y)\} = \Pr\{X = x\} \Pr\{Y = y\}, \forall x, y$$



Probabilidad

- Momentos:
 - *Esperanza matemática* (o media):

$$E[X] = \sum_x x \Pr\{X = x\} = \sum_x x f(x)$$

Ejemplo: se lanzan dos monedas, se ganan 3 Euros por cada cara y se pierden 2 por cada cruz, si X representa la ganancia obtenida en una partida, entonces

$$\begin{aligned} E[X] &= 6 \Pr\{2 \text{ caras}\} + 1 \Pr\{1 \text{ cara y } 1 \text{ cruz}\} - 4 \Pr\{2 \text{ cruces}\} = \\ &= 6(1/4) + 1(1/2) - 4(1/4) = 1 \end{aligned}$$



Probabilidad

- Momentos (cont.):
 - Propiedades de la esperanza matemática:
 - $E[X+Y] = E[X] + E[Y]$
 - Dada X , cualquier función $g(x)$ define una nueva v.a. $g(X)$, y
$$E[g(X)] = \sum_x g(x) \Pr\{X = x\}$$
 - En particular, si $g(x)=ax$, con a una constante, $E[ax] = aE[X]$
 - Si X e Y son independientes entonces $E[XY] = E[X] E[Y]$
(lo mismo para una familia mutuamente independiente)
 - Si X toma valores en los naturales, entonces

$$\begin{aligned} E[X] &= \sum_{i=0}^{\infty} i \Pr\{X = i\} = \sum_{i=0}^{\infty} i (\Pr\{X \geq i\} - \Pr\{X \geq i+1\}) \\ &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \end{aligned}$$



Probabilidad

- Momentos (cont.):

- *Varianza:*

$$\begin{aligned}\text{Var}[X] &= E[(X-E[X])^2] = E[X^2 - 2XE[X] + E^2[X]] = \\ &= E[X^2] - 2E[XE[X]] + E^2[X] = \\ &= E[X^2] - 2E^2[X] + E^2[X] = E[X^2] - E^2[X]\end{aligned}$$

- Propiedades de la varianza:

- $\text{Var}[aX] = a^2\text{Var}[X]$, con a una constante
 - Si X e Y son independientes, $\text{Var}[X+Y] = \text{Var}[X] + \text{Var}[Y]$
(lo mismo para una familia de independientes dos a dos)

- *Desviación estándar:*

$$\sigma_X = (\text{Var}[X])^{1/2} \quad \text{es decir} \quad \text{Var}[X] = \sigma_X^2$$



Probabilidad

- Distribuciones de probabilidad importantes:

- *Experimento de Bernoulli* de parámetro p :

- X vale 1 cuando ocurre el suceso A (“éxito”), y eso ocurre con probabilidad p , y vale 0 en caso contrario (“fracaso”), con probabilidad $q = 1-p$.

$$E[X] = p$$

$$\text{Var}[X] = pq$$

- Distribución *geométrica* de parámetro p :

- X es el número de experimentos independientes de Bernoulli hasta que ocurre el suceso A (incluido este último)

$$\Pr\{X = k\} = q^{k-1}p, \text{ para todo } k \geq 1$$

$$E[X] = 1/p$$

$$\text{Var}[X] = q/p^2$$



Probabilidad

- Ejemplo de distribución geométrica:
 - Tiramos dos dados repetidamente hasta que la suma de ambos es un 7 o un 11
 - De los 36 resultados posibles, 6 dan un siete y 2 dan un 11, por tanto la probabilidad de éxito es
- Luego el número esperado de repeticiones hasta obtener un 7 o un 11 (esperanza matemática de X) es

$$p = 8/36 = 2/9$$

$$E[X] = 1/p = 9/2 = 4,5$$



Probabilidad

- Distribución *binomial* de parámetros n y p :
 - X es el número de éxitos en n experimentos independientes de Bernoulli de parámetro p (y $q = 1-p$)

$$\Pr\{X = k\} = \binom{n}{k} p^k q^{n-k}, \text{ para todo } k = 0, \dots, n$$

$$E[X] = np \qquad \text{Var}[X] = npq$$

- Las colas de la distribución binomial:

$$\Pr\{X \geq k\} \leq \binom{n}{k} p^k, \text{ para } 0 \leq k \leq n$$

$$\Pr\{X \leq k\} \leq \binom{n}{k} q^{n-k}, \text{ para } 0 \leq k \leq n$$

$$\Pr\{X < k\} < \frac{kq}{np - k} \binom{n}{k} p^k q^{n-k}, \text{ para } 0 < k < np$$

$$\Pr\{X > k\} < \frac{(n-k)p}{k - np} \binom{n}{k} p^k q^{n-k}, \text{ para } np < k < n$$



Probabilidad

- Distribución *binomial negativa* de parámetros n y p :
 - X es el número de fallos ocurridos antes del n -ésimo éxito de una serie de experimentos independientes de Bernoulli de parámetro p

$$E[X] = n(1-p)/p \quad \text{Var}[X] = n(1-p)/p^2$$



Análisis del caso promedio

- El plan:
 - Probabilidad
 - **Análisis probabilista**
 - Árboles binarios de búsqueda contruidos aleatoriamente
 - Tries, árboles digitales de búsqueda y Patricia
 - Listas “skip”
 - Árboles aleatorizados



Análisis probabilista

- Primer ejemplo: la paradoja del cumpleaños

¿cuánta gente es necesaria en una habitación para que haya una ‘probabilidad alta’ de que dos cumplan años el mismo día?

- suponer que hay k personas: $1, 2, \dots, k$
- b_i es el día ($1 \leq b_i \leq n=365$) del cumpleaños de i ($i=1, 2, \dots, k$)
- los cumpleaños se distribuyen uniformemente en el año:
 $\Pr\{b_i=r\} = 1/n$ para $i=1, 2, \dots, k$, y $r=1, 2, \dots, n$
- si asumimos que los cumpleaños son independientes:

$$\Pr\{(b_i=r) \wedge (b_j=r)\} = \Pr\{b_i=r\}\Pr\{b_j=r\} = 1/n^2$$

$$\Rightarrow \Pr\{b_i = b_j\} = \sum_{r=1}^n \Pr\{(b_i = r) \wedge (b_j = r)\} = \sum_{r=1}^n \left(1/n^2\right) = 1/n$$



Análisis probabilista

- Suceso: “al menos dos personas cumplen el mismo día” → complementario: “todos los cumpleaños son diferentes”
- $A_i =$ “el cumpleaños de $i+1$ es distinto del de j , para todo $j \leq i$ ”

$$A_i = \{b_{i+1} \neq b_j \mid j = 1, 2, \dots, i\}$$

- $B_k =$ “los cumpleaños de las k personas son en días distintos”

$$B_k = \bigcap_{i=1}^{k-1} A_i = A_{k-1} \cap B_{k-1}$$

\Downarrow

$$\begin{aligned} \Pr\{B_k\} &= \Pr\{B_{k-1}\} \Pr\{A_{k-1} \mid B_{k-1}\} = \\ &= \Pr\{B_1\} \Pr\{A_1 \mid B_1\} \Pr\{A_2 \mid B_2\} \cdots \Pr\{A_{k-1} \mid B_{k-1}\} \end{aligned}$$

$$\text{y } \Pr\{B_1\} = 1$$



Análisis probabilista

- Si b_1, b_2, \dots, b_{k-1} son distintos, la probabilidad condicionada de $b_k \neq b_i$, para $i = 1, 2, \dots, k-1$ es $(n-k+1)/n$, puesto que de los n días hay $n-(k-1)$ sin “coger”, es decir

$$\Pr\{A_{k-1} \mid B_{k-1}\} = \frac{n-k+1}{n}$$

por tanto:

$$\begin{aligned}\Pr\{B_k\} &= 1 \cdot \left(\frac{n-1}{n}\right) \left(\frac{n-2}{n}\right) \dots \left(\frac{n-k+1}{n}\right) = \\ &= 1 \cdot \left(1 - \frac{1}{n}\right) \left(1 - \frac{2}{n}\right) \dots \left(1 - \frac{k-1}{n}\right)\end{aligned}$$

y como

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots \Rightarrow e^x \geq 1 + x, \quad \forall x \in \mathbf{R}$$

entonces:

$$\begin{aligned}\Pr\{B_k\} &\leq e^{-1/n} e^{-2/n} \dots e^{-(k-1)/n} = e^{-\sum_{i=1}^{k-1} i/n} = e^{-k(k-1)/2n} \stackrel{?}{\leq} 1/2\end{aligned}$$

(Note: An arrow points from the inequality $e^{-1/n} \geq 1 - \frac{1}{n}$ in the circled box to the first term $e^{-1/n}$ in the product above.)



Análisis probabilista

- Por tanto, $\Pr\{B_k\} \leq 1/2$ si $-k(k-1)/2n \leq \ln(1/2)$
- Es decir, la probabilidad de que los k cumpleaños sean distintos es como mucho $1/2$ si $k \geq (1 + \sqrt{1 + (8 \ln 2)n})/2$
- Para $n = 365$, resulta $k \geq 23$
Es decir, si hay 23 personas en una habitación, la probabilidad de que al menos dos cumplan años el mismo día es mayor o igual que 0,5.
- En Marte, un año dura 669 días, así que tienen que juntarse 31 marcianos para lograr lo mismo...



Análisis probabilista

- Otro método para el mismo problema
 - Para cada par de personas i, j de las k de la habitación se define la v.a. X_{ij} , $1 \leq i < j \leq k$,

$$X_{ij} = \begin{cases} 1 & \text{si } i \text{ y } j \text{ cumplen años el mismo día} \\ 0 & \text{en otro caso} \end{cases}$$

- Como la probabilidad de que dos cumplan años el mismo día es $1/n$, se tiene

$$E[X_{ij}] = 1 \cdot (1/n) + 0 \cdot (1 - 1/n) = 1/n$$

- Y el número medio de pares de personas cumpliendo años el mismo día es

$$\sum_{j=2}^k \sum_{i=1}^{j-1} E[X_{ij}] = \binom{k}{2} \frac{1}{n} = \frac{k(k-1)}{2n} \quad ? \geq 1$$



Análisis probabilista

- Por tanto, podemos esperar que como mínimo haya un par de personas cumpliendo años el mismo día si:

$$k \geq \sqrt{2n}$$

Por ejemplo, si $k = 28$, el número esperado de parejas con el mismo cumpleaños es $(28 \cdot 27) / (2 \cdot 365) \approx 1,0356$

(Marcianos harían falta como mínimo 38...)



Análisis probabilista

- Segundo ejemplo: urnas y bolas
 - Se meten aleatoriamente una colección de bolas idénticas en b urnas numeradas $1, 2, \dots, b$
 - Las introducciones son independientes, y cada una tiene igual probabilidad $1/b$ de ser metida en cada urna
 - Pueden plantearse varias preguntas interesantes...
 - ¿Cuántas bolas caen en una urna dada?
 - Cae un número que sigue una distribución binomial(*) de parámetros n y $p = 1/b$
 - Por tanto, si se meten n bolas, el número medio de bolas que caen en cada urna es n/b

(*) Recordar: binomial = n° éxitos de probabilidad p en n experimentos independientes



Análisis probabilista

- ¿Cuántas bolas hay que meter, en media, hasta que una urna dada contenga una bola?
 - El número necesario sigue una distribución geométrica(*) de parámetro $1/b$, luego la media hasta que hay un “éxito” es b
 - (*) Recordar: geom = n° experim. indep. hasta ocurre un éxito
- ¿Cuántas bolas hay que meter, en media, para que todas las urnas tengan al menos una bola?
 - “acierto”: un lanzamiento en el que la bola cae en una urna vacía
 - queremos saber el número medio n de lanzamientos para obtener b aciertos
 - partición de los lanzamientos en “fases”:
 - fase i -ésima: lanzamientos desde el acierto $i-1$ hasta el acierto i
 - en la fase 1 sólo hay un lanzamiento (el primero)



Análisis probabilista

- En cada lanzamiento de la fase i hay $i-1$ urnas con bolas y $b-i+1$ urnas vacías, por tanto, para cada lanzamiento de la fase i , la probabilidad de acierto es $(b-i+1)/b$
- Sea n_i el número de lanzamientos de la fase i .
- El número de lanzamientos para obtener b aciertos es

$$n = \sum_{i=1}^b n_i$$

- Cada variable aleatoria n_i tiene una distribución geométrica de parámetro $(b-i+1)/b$, por tanto

$$E[n_i] = \frac{b}{b-i+1}$$

- Luego:

$$E[n] = E\left[\sum_{i=1}^b n_i\right] = \sum_{i=1}^b E[n_i] = \sum_{i=1}^b \frac{b}{b-i+1} = b \sum_{i=1}^b \frac{1}{i} \leq b(\ln b + O(1))$$

serie armónica

$$H_n = \sum_{k=1}^n \frac{1}{k} = \ln n + O(1)$$



Análisis probabilista

- Tercer ejemplo: rachas

- Se lanza una moneda (“buena”) n veces,
¿cuál es la longitud L de la secuencia más larga de
caras consecutivas que puede esperarse?

- Definimos el suceso A_{ik} = los k lanzamientos
sucesivos $i, i+1, \dots, i+k-1$ son todos cara ($1 \leq k \leq n$,
 $1 \leq i \leq n-k+1$)

- La probabilidad del suceso sigue una distribución geométrica
con $p = 1/2$, es decir: $\Pr\{A_{ik}\} = 1/2^k$

1ª parte) $L \in O(\log n)$ • Para $k = 2 \lceil \log n \rceil$, $\Pr\{A_{i, 2 \lceil \log n \rceil}\} = 1/2^{2 \lceil \log n \rceil} \leq 1/2^{2 \log n} = 1/n^2$

- Y la probabilidad de una racha de caras de longitud mayor o
igual que $2 \lceil \log n \rceil$ empezando en cualquier posición es:

$$\Pr\left\{\bigcup_{i=1}^{n-2 \lceil \log n \rceil + 1} A_{i, 2 \lceil \log n \rceil}\right\} \leq \sum_{i=1}^n 1/n^2 = 1/n$$



Análisis probabilista

- Es decir, la probabilidad de una racha de longitud mayor o igual que $2 \lceil \log n \rceil$ es como mucho $1/n$, luego la probabilidad de que la máxima racha sea de longitud menor que $2 \lceil \log n \rceil$ es como mínimo $1 - 1/n$.
- Como la longitud máxima de una racha es n , *la longitud media de la racha más larga está acotada superiormente por* $(2 \lceil \log n \rceil)(1 - 1/n) + n(1/n) = O(\log n)$

Análogamente...

- Para $r \geq 1$, la probabilidad de que una racha de $r \lceil \log n \rceil$ caras empiece en la posición i es

$$\Pr\{A_{i, r \lceil \log n \rceil}\} = 1/2^{r \lceil \log n \rceil} \leq 1/n^r$$

Por tanto, la probabilidad de que la máxima racha sea de longitud menor que $r \lceil \log n \rceil$ es como mínimo $1 - 1/n^{r-1}$.

- Por ejemplo, para $n = 1000$ lanzamientos: la probabilidad de una racha de $2 \lceil \log n \rceil = 20$ caras, o más, es menor o igual que $1/n = 1/1000$, y la probabilidad de una racha de $3 \lceil \log n \rceil = 30$, o más, es menor o igual que $1/n^2 = 1/1.000.000$



Análisis probabilista

2ª parte) $L \in \Omega(\log n)$ • Veamos ahora que la longitud media de la racha más larga de caras en n lanzamientos es $\Omega(\log n)$, y como ya hemos visto que es $O(\log n)$, tendremos que es $\Theta(\log n)$

$$\Pr\{A_{ik}\} = 1/2^k \Rightarrow \Pr\{A_{i, \lfloor \log n \rfloor / 2}\} = 1/2^{\lfloor \log n \rfloor / 2} \geq 1/\sqrt{n}$$

y por tanto la probabilidad de que una secuencia de al menos $\lfloor \log n \rfloor / 2$ caras no empiece en la posición i es como mucho $1 - 1/\sqrt{n}$

Partimos los n lanzamientos en $\lfloor 2n / \lfloor \log n \rfloor \rfloor$ grupos de $\lfloor \log n \rfloor / 2$ lanzamientos consecutivos, por tanto la probabilidad de que todos y cada uno de estos grupos fallen en ser rachas de $\lfloor \log n \rfloor / 2$ caras consecutivas es

$$(1 - 1/\sqrt{n})^{\lfloor 2n / \lfloor \log n \rfloor \rfloor} \leq (1 - 1/\sqrt{n})^{2n/\log n - 1} \leq$$

$$\leq e^{-(2n/\log n - 1)/\sqrt{n}} \leq e^{-\log n} \leq 1/n$$

$$1+x \leq e^x$$

$$(2n/\log n - 1)/\sqrt{n} \geq \log n$$

Análisis probabilista

- Luego la probabilidad de que la racha más larga de caras exceda $\lfloor \log n \rfloor / 2$ caras es como mínimo $1 - 1/n$
- Como la longitud mínima para la racha más larga de caras es 0 (si todo son cruces), entonces la longitud media de la racha más larga de caras es como mínimo:
$$(\lfloor \log n \rfloor / 2)(1 - 1/n) + 0 \cdot (1/n) = \Omega(\log n)$$



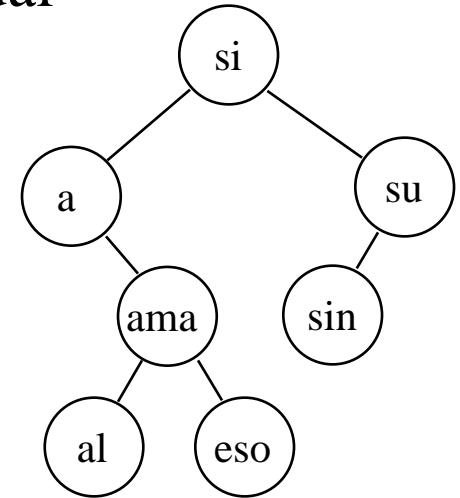
Análisis del caso promedio

- El plan:
 - Probabilidad
 - Análisis probabilista
 - **Árboles binarios de búsqueda contruidos aleatoriamente**
 - Tries, árboles digitales de búsqueda y Patricia
 - Listas “skip”
 - Árboles aleatorizados



Abb's contruidos aleatoriamente

- Recordar... árbol binario de búsqueda (*abb*):
 - La *clave* de todo nodo es mayor o igual que las de sus descendientes izquierdos y menor que las de sus descendientes derechos.
 - Las operaciones básicas (búsqueda, inserción, borrado...) tienen un coste $O(h)$, donde h es la altura del árbol.
 - La altura de un *abb* varía con las inserciones y borrados y, en el **peor caso**, puede llegar a ser igual al número de nodos, $O(n)$.



¿Qué ocurrirá en un caso “promedio”?



Abb's contruidos aleatoriamente

- *Abb* de n nodos construido aleatoriamente:
 - Es el resultante tras la inserción consecutiva de n claves en orden aleatorio en un árbol vacío, suponiendo que las $n!$ permutaciones posibles de las claves son equiprobables.
 - La pregunta: ¿cuál es el coste promedio de las operaciones con estos árboles?, es decir, ¿cuál es la altura media de estos árboles?
 - La respuesta no es trivial... ¡iremos por partes!



Abb's contruidos aleatoriamente

- *Longitud de los caminos internos y externos de un árbol binario*

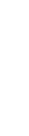
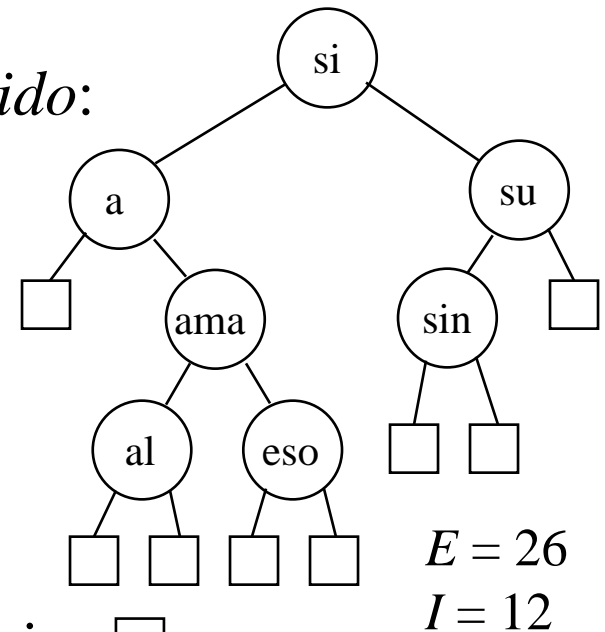
- Definición. *Árbol binario extendido*:

- el resultante de añadir un nodo especial en el lugar de cada subárbol vacío.

- Definición. *Longitud de*

- caminos externos, E* : es la suma de las longitudes de todos los caminos desde la raíz hasta las hojas, \square .

- Definición. *Longitud de caminos internos, I* : es la suma de las longitudes de todos los caminos desde la raíz hasta todos los nodos internos, \bigcirc .



Abb's construidos aleatoriamente

- Teorema: $E = I + 2n$, donde n es el número de nodos internos.
 - Demostración: supongamos que se borra un nodo interno V a distancia k de la raíz y tal que sus dos hijos son hojas.
 - En tal caso, E disminuye en $2(k+1)$, porque los hijos de V desaparecen, y aumenta en k , porque V pasa a ser hoja, luego E se convierte en $E-k-2$.
 - Por otra parte, I disminuye en k .
 - Por tanto, $E-I$ disminuye en 2 por cada nodo que se borra.
 - Borrando los n nodos se llega a un árbol vacío, es decir con $E = I = E-I = 0$, por tanto, $E-I = 2n$.



Abb's contruidos aleatoriamente

- Notación y relación fundamental:
 - C_n : número medio de comparaciones en una búsqueda con éxito en un árbol de n nodos construido aleatoriamente, suponiendo que la búsqueda de toda clave es equiprobable.
 - C_n' : número medio de comparaciones en una búsqueda sin éxito, suponiendo que cada uno de los $n+1$ intervalos entre las claves y fuera de sus valores extremos son equiprobables.
 - Entonces: $C_n = 1 + I/n$ y $C_n' = E/(n+1)$
 - Por tanto ($E = I + 2n$): $C_n = \left(1 + \frac{1}{n}\right)C_n' - 1$



Abb's contruidos aleatoriamente

- Finalmente...
 - Supongamos que cada una de las $n!$ ordenaciones posibles de las n claves es una secuencia de inserción igualmente probable para construir el árbol
 - El número de comparaciones necesarias para encontrar una clave es exactamente uno más que el número de comparaciones que se necesitaban cuando esta clave no estaba en el árbol, por tanto:

$$C_n = 1 + \frac{C_0' + C_1' + \dots + C_{n-1}'}{n}$$

que unido a $C_n = \left(1 + \frac{1}{n}\right)C_n' - 1$

nos da la recurrencia:

$$(n+1)C_n' = 2n + C_0' + C_1' + \dots + C_{n-1}'$$



Abb's construidos aleatoriamente

- Para resolver $(n+1)C_n' = 2n + C_0' + C_1' + \dots + C_{n-1}'$
se resta $nC_{n-1}' = 2(n-1) + C_0' + C_1' + \dots + C_{n-2}'$
y se obtiene $(n+1)C_n' - nC_{n-1}' = 2 + C_{n-1}'$
es decir $C_n' = C_{n-1}' + 2/(n+1)$
y como $C_0' = 0$, entonces $C_n' = 2H_{n+1} - 2$
donde H_n es la serie armónica

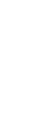
$$H_n = \sum_{i=1}^n \frac{1}{i} = \ln n + O(1)$$

Aplicando

$$C_n = \left(1 + \frac{1}{n}\right)C_n' - 1$$

y simplificando:

$$C_n = 2\left(1 + \frac{1}{n}\right)H_n - 3 = O(\log n)$$



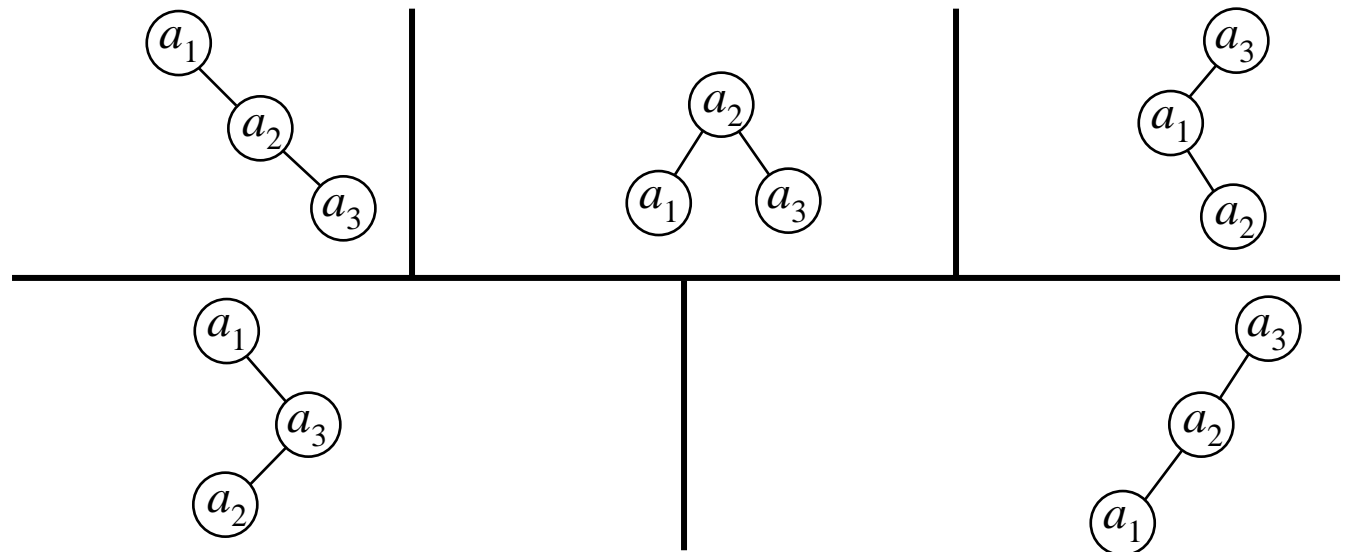
Abb's contruidos aleatoriamente

- El anterior es un buen resultado, pero...
 - Si un *abb* se construye aleatoriamente y se “somete” después a una secuencia de inserciones aleatorias y de **borrados** aleatorios, el árbol resultante pierde la aleatoriedad, con lo que se carece de resultados teóricos sobre el coste promedio de las operaciones.
 - No obstante, la *evidencia empírica* sugiere que tras una secuencia de borrados e inserciones aleatorios, la altura del árbol tiende a disminuir, si bien no he encontrado en la literatura la explicación teórica de este comportamiento.



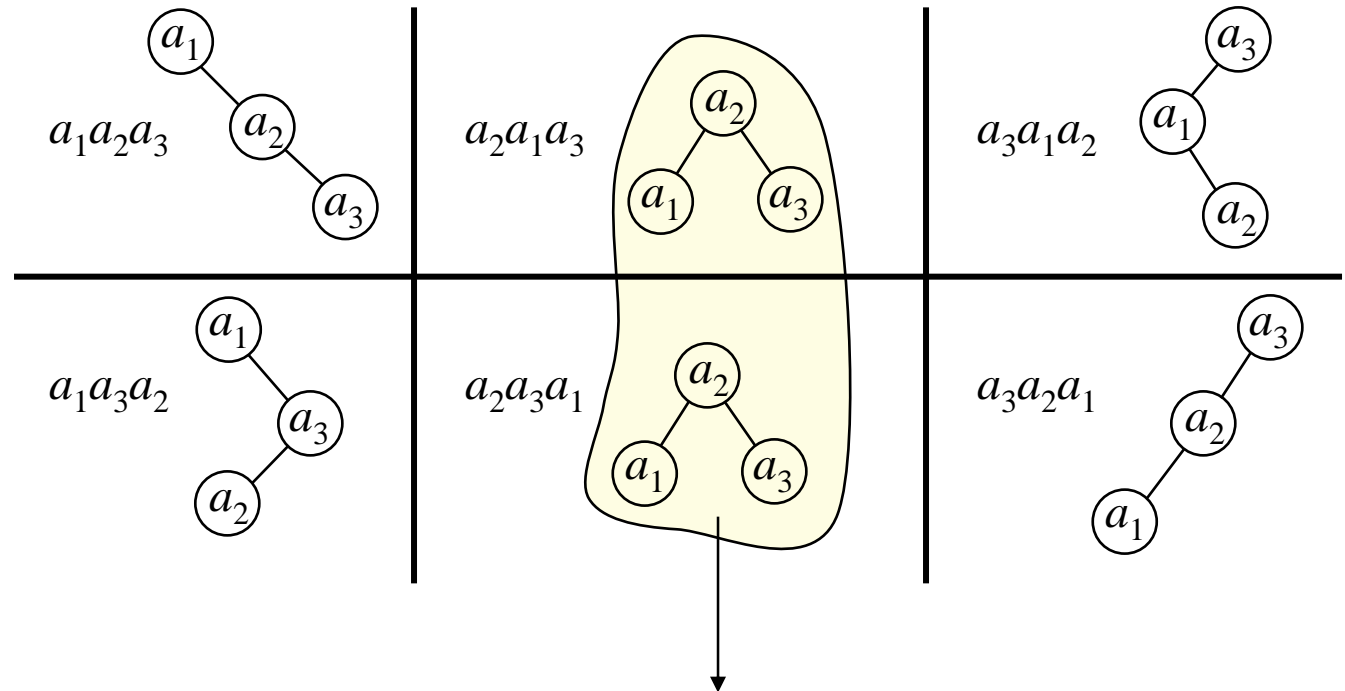
Abb's contruidos aleatoriamente

- Otra definición de “*abb*’s elegidos al azar”:
 - Hipótesis de equiprobabilidad de todos los *abb*’s posibles de n nodos.
 - Primera cuestión: ver que ésta es una definición de “*abb* aleatorio” distinta a la anterior.
 - Hay 5 *abb*’s distintos de 3 nodos (suponer $a_1 < a_2 < a_3$), por tanto cada uno tiene probabilidad $1/5$:



Abb's construidos aleatoriamente

- Sin embargo, construyéndolos a partir de las permutaciones posibles de a_1, a_2, a_3 salen 6, cada uno con probabilidad $1/6$:



Es decir, ¡éste árbol tiene el doble de probabilidad que los otros! ($1/3$)

Abb's construidos aleatoriamente

- Cálculo de la profundidad media de un nodo en un *abb* de n nodos elegido al azar:
 - Recordar: *longitud de caminos internos*, I_n , de un árbol de n nodos (suma de las longitudes de todos los caminos desde la raíz hasta todos los nodos internos).
 - $I_1 = 0$
 - Un árbol de n nodos tiene un subárbol izquierdo con i nodos (para algún $0 \leq i < n$) y un subárbol derecho con $n-i-1$ nodos, por tanto:

$$I_n = I_i + I_{n-i-1} + n - 1$$



Abb's contruidos aleatoriamente

- Hipótesis adicional: el número de nodos de un subárbol (izquierdo o derecho) de un *abb* de n nodos elegido al azar es equiprobable para todos los valores posibles (de 0 a $n-1$), por tanto:

$$E[I_i] = E[I_{n-i-1}] = \frac{1}{n} \sum_{j=0}^{n-1} I_j$$

↙ promedio de la longitud interna del subárbol izquierdo

- Y aplicándolo a la recurrencia anterior (se omite el operador *esperanza* por comodidad), se obtiene la recurrencia:

$$I_n = \frac{2}{n} \left[\sum_{j=0}^{n-1} I_j \right] + n - 1$$



Abb's construidos aleatoriamente

– Resolución de:

$$I_n = \frac{2}{n} \left[\sum_{j=0}^{n-1} I_j \right] + n - 1$$

- Multiplicando por n :

$$nI_n = 2 \left[\sum_{j=0}^{n-1} I_j \right] + n^2 - n$$

- Y por tanto:

$$(n-1)I_{n-1} = 2 \left[\sum_{j=0}^{n-2} I_j \right] + (n-1)^2 - (n-1)$$

- Restando ambas:

$$nI_n - (n-1)I_{n-1} = 2I_{n-1} + 2n - 2$$

- Es decir:

$$nI_n = (n+1)I_{n-1} + 2n - 2$$



Abb's contruidos aleatoriamente

- Y para resolver esta nueva recurrencia:

$$nI_n = (n+1)I_{n-1} + 2n - 2$$

- Dividiendo por $n(n+1)$ y despreciando la constante:

- Luego:

$$\left. \begin{array}{l} \frac{I_n}{n+1} = \frac{I_{n-1}}{n} + \frac{2}{n+1} \\ \frac{I_{n-1}}{n} = \frac{I_{n-2}}{n-1} + \frac{2}{n} \\ \frac{I_{n-2}}{n-1} = \frac{I_{n-3}}{n-2} + \frac{2}{n-1} \\ \vdots \\ \frac{I_2}{3} = \frac{I_1}{2} + \frac{2}{3} \end{array} \right\} \text{sumando: } \frac{I_n}{n+1} = \frac{I_1}{2} + 2 \sum_{i=3}^{n+1} \frac{1}{i}$$



Abb's contruidos aleatoriamente

- Finalmente, como

$$2 \sum_{i=3}^{n+1} \frac{1}{i} = O(\log n) \quad (\text{serie armónica})$$

- Se sigue que

$$I_n = O(n \log n)$$

- Y como I_n es la suma de las longitudes de todos los caminos desde la raíz hasta todos los nodos internos, entonces la profundidad media de cada nodo es $O(\log n)$.



Abb's construidos aleatoriamente

- De nuevo, al anterior es un buen resultado pero...
 - El efecto de las operaciones de borrado destruye la aleatoriedad.
 - No he encontrado en la literatura resultados teóricos sobre el coste promedio resultante considerando secuencias de operaciones de inserción y borrado.

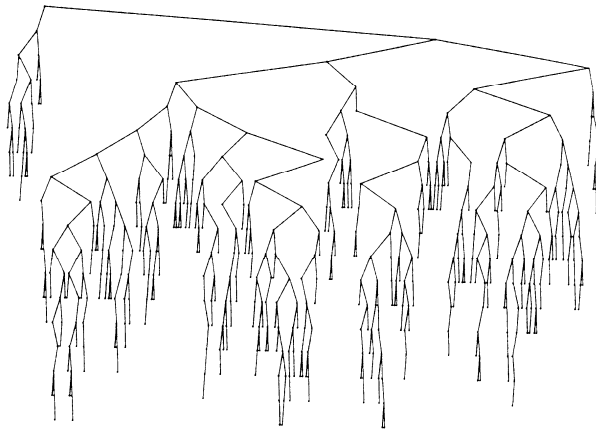
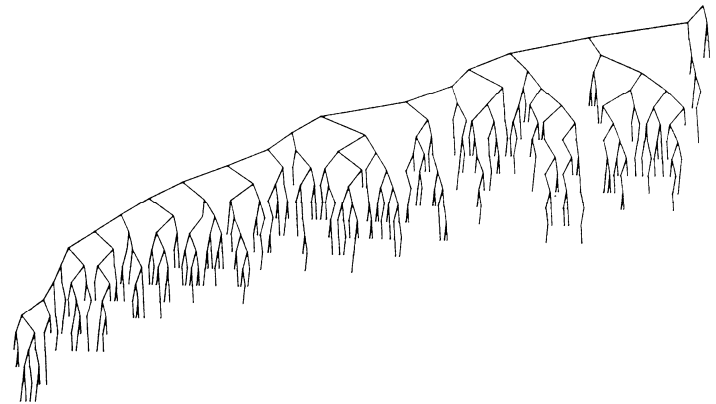


Abb de 500 nodos
Profundidad media: 9'98



Tras 250.000 pares de operaciones de inserción y borrado. Profundidad media: 12'51

Abb's contruidos aleatoriamente

- Para evitar el crecimiento de la altura del árbol hay dos tipos de soluciones:
 - Realizar alguna operación de “equilibrado” del árbol tras cada inserción/borrado (árboles AVL, árboles 2-3, árboles B, árboles rojinegros).
 - Renunciar a un equilibrado tan estricto y aplicar alguna *regla de re-estructuración* que ayude a que las futuras secuencias de operaciones sean más eficientes (*análisis amortizado*, lo veremos más adelante).
 - Este tipo de estructuras se llaman *auto-organizativas* (listas auto-organizativas, árboles auto-organizativos o “splay trees”).



Análisis del caso promedio

- El plan:
 - Probabilidad
 - Análisis probabilista
 - Árboles binarios de búsqueda contruidos aleatoriamente
 - **Tries, árboles digitales de búsqueda y Patricia**
 - Listas “skip”
 - Árboles aleatorizados



Tries, árboles digitales de búsqueda y Patricia

- *Tries*: motivación...
 - Letras centrales de la palabra “retrieval”, recuperación (de información).
 - Diccionario de Unix: 23.000 palabras y 194.000 caracteres ➔ una media de 8 caracteres por palabra...

Hay información redundante:

bestial bestir bestowal bestseller bestselling

- Para ahorrar espacio:
agrupar los prefijos comunes
- Para ahorrar tiempo:
si las palabras son más cortas
es más rápida la búsqueda...

```
best
---- i
---- - al
---- - r
---- owal
---- sell
---- ---- er
---- ---- ing
```



Tries, árboles digitales de búsqueda y Patricia

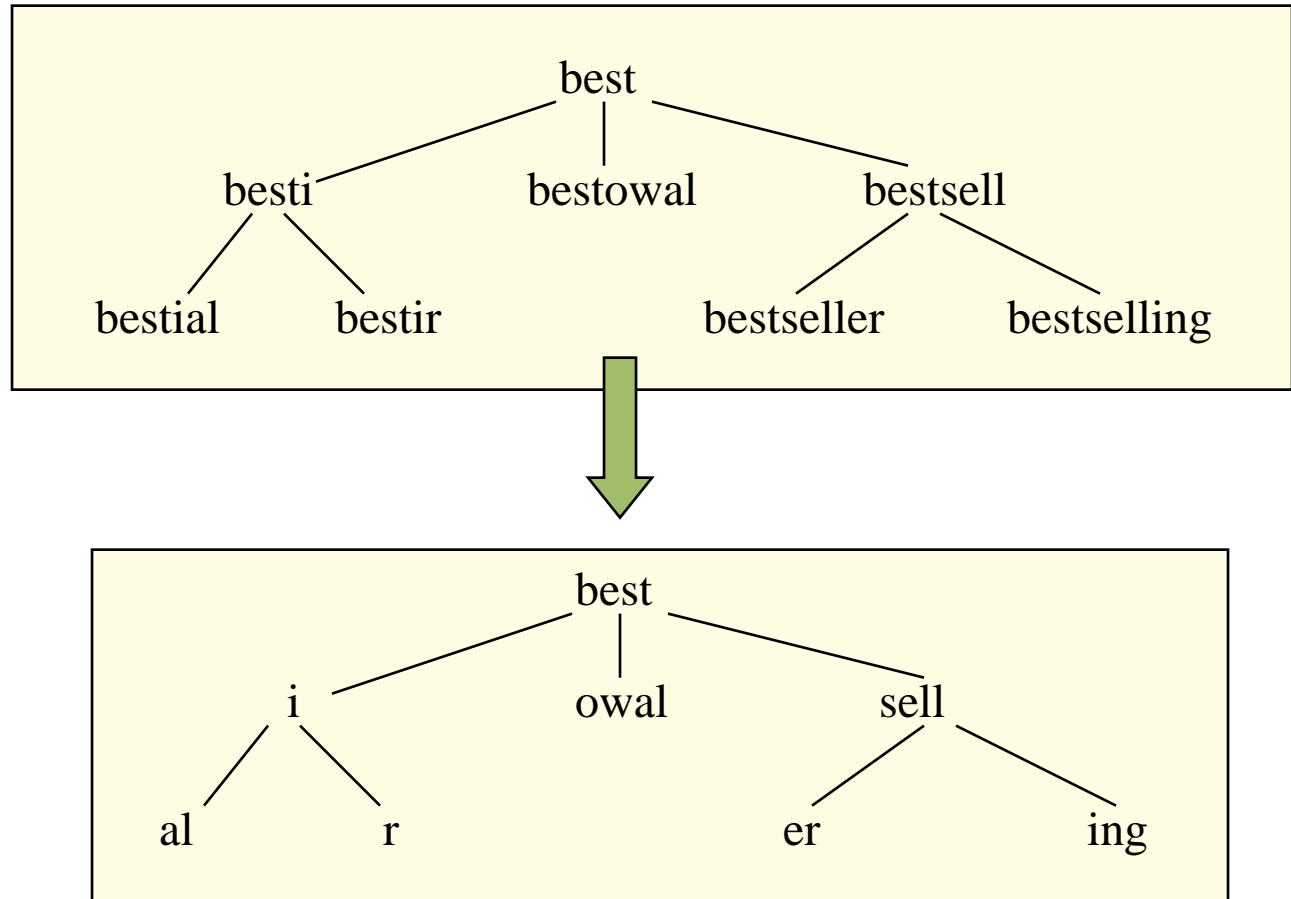
- Trie: definición formal
 - Sea $\Sigma = \{\sigma_1, \dots, \sigma_m\}$ un **alfabeto** finito ($m > 1$).
 - Sea Σ^* el conjunto de las **palabras** (o secuencias) de símbolos de Σ , y X un subconjunto de Σ^* (es decir un conjunto de palabras).
 - El **trie** asociado a X es:
 - $\text{trie}(X) = \emptyset$, si $X = \emptyset$
 - $\text{trie}(X) = \langle x \rangle$, si $X = \{x\}$
 - $\text{trie}(X) = \langle \text{trie}(X \setminus \sigma_1), \dots, \text{trie}(X \setminus \sigma_m) \rangle$, si $|X| > 1$, donde $X \setminus \sigma$ representa el subconjunto de todas las palabras de X que empiezan por σ quitándoles la primera letra.
 - Si el alfabeto tiene definida una relación de orden (caso habitual), el trie se llama árbol lexicográfico.



Tries, árboles digitales de búsqueda y Patricia

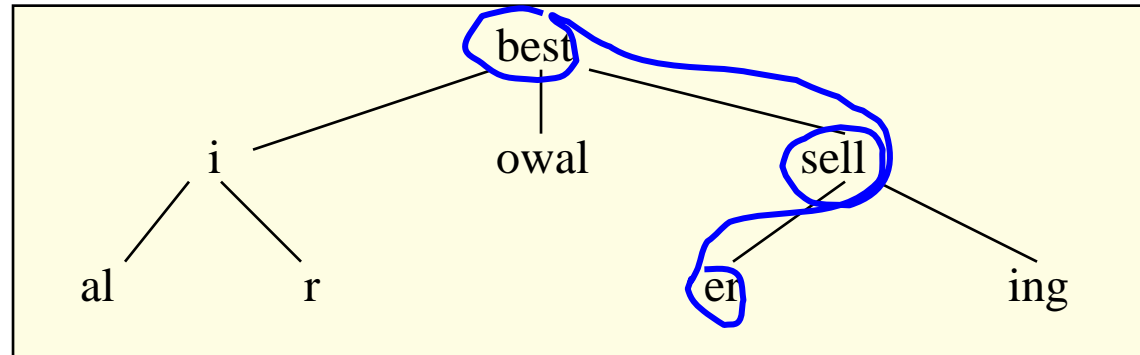
- Es decir, un trie es un *árbol de prefijos*:

bestial bestir bestowal bestseller bestselling

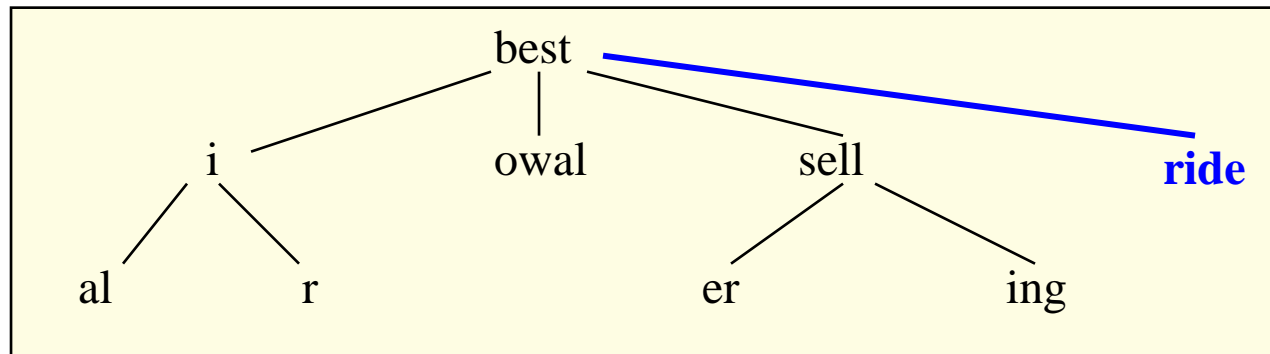


Tries, árboles digitales de búsqueda y Patricia

- Utilidad del trie:
 - Soporta operaciones de búsqueda de palabras:



- También se pueden implementar inserciones y borrados ➔ TAD *diccionario*



Tries, árboles digitales de búsqueda y Patricia

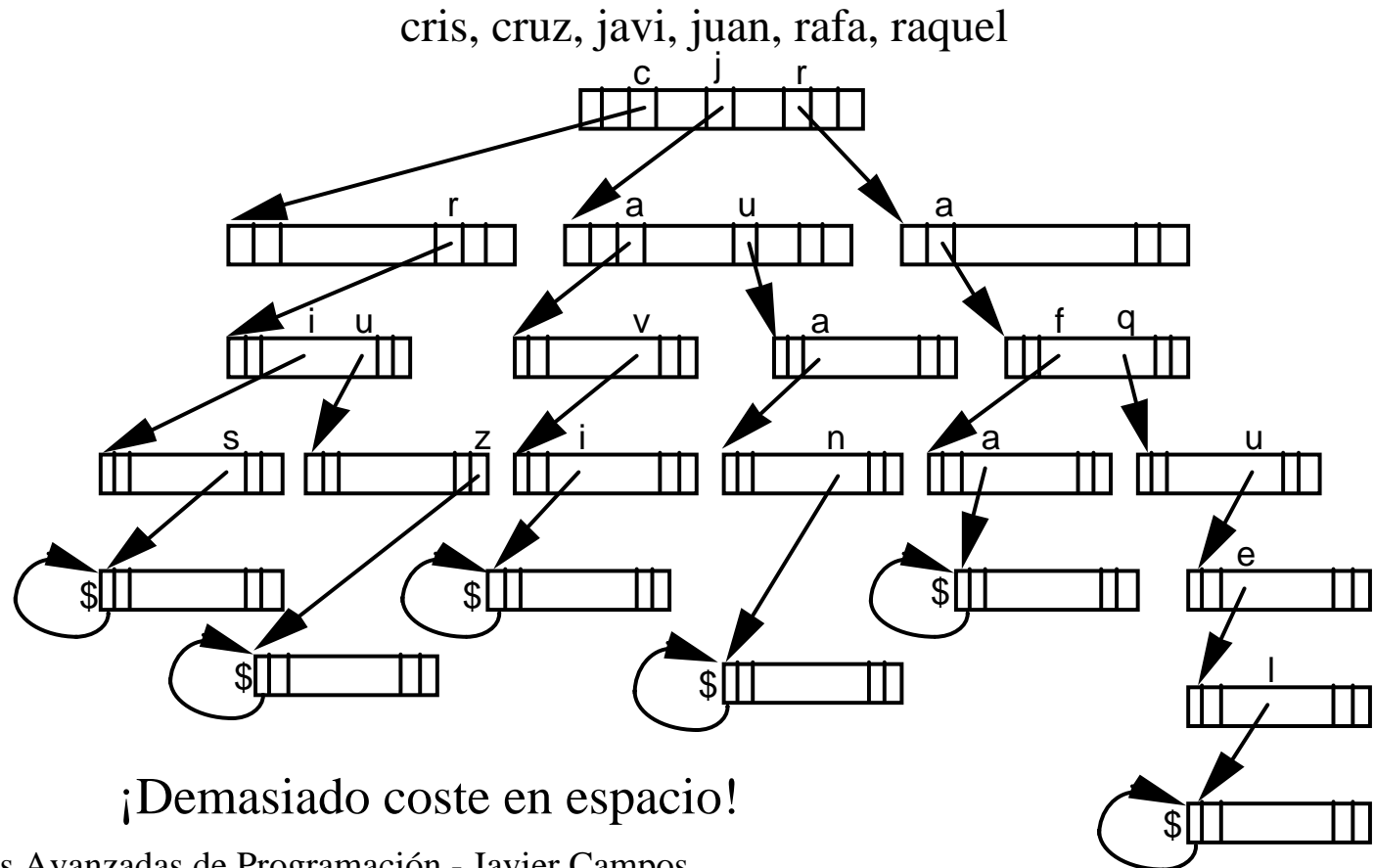
- Y también uniones e intersecciones → TAD *conjunto*
- Y comparaciones de subcadenas → procesamiento de textos, biología computacional...
- No lo hemos dicho, pero no pueden almacenarse palabras que sean prefijos de otras... uso de un carácter terminador si eso fuese preciso.

Los tries son una de las estructuras de datos de propósito general más importantes en informática



Tries, árboles digitales de búsqueda y Patricia

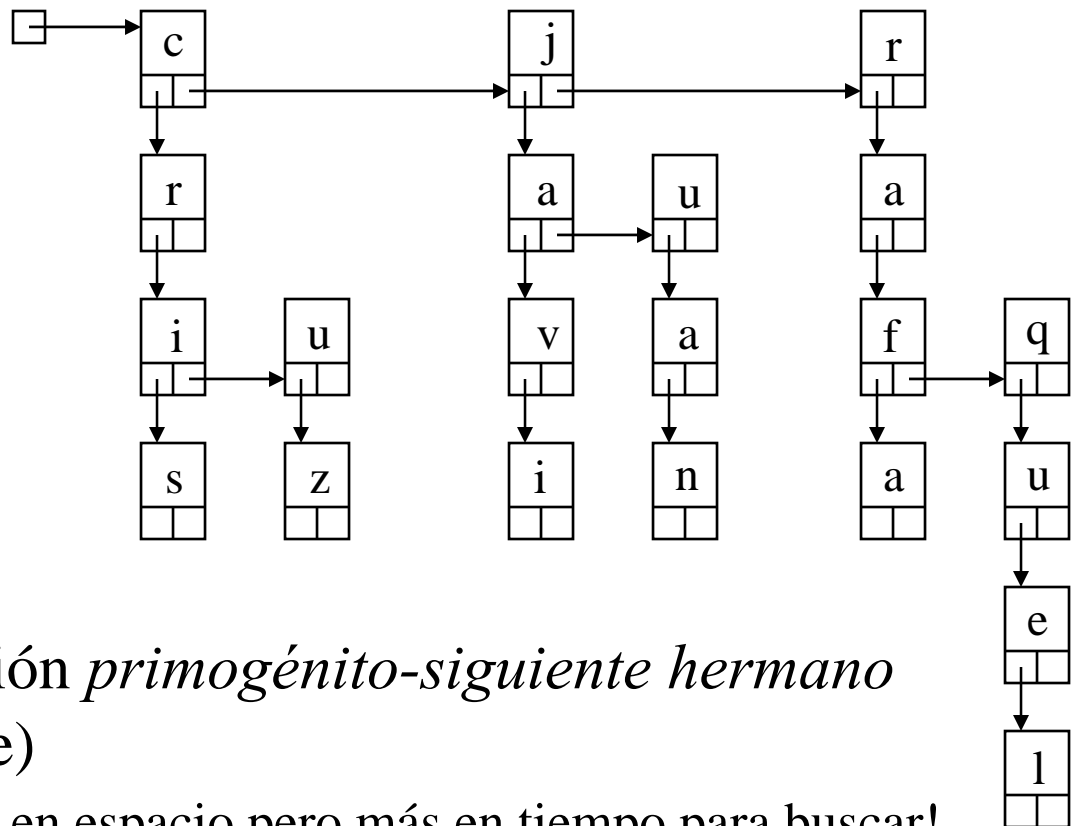
- Implementaciones de tries:
 - *Nodo-vector*: cada nodo es un vector de punteros para acceder a los subárboles directamente



Tries, árboles digitales de búsqueda y Patricia

- *Nodo-lista*: cada nodo es una lista enlazada por punteros que contiene las raíces de los subárboles

cris, cruz, javi, juan, rafa, raquel



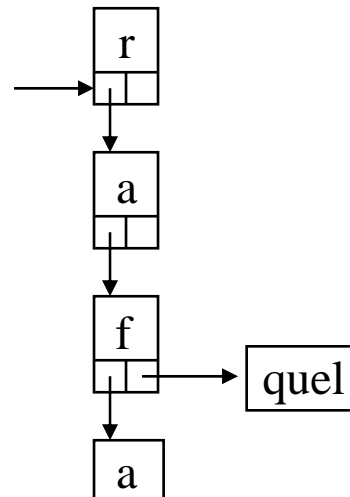
(representación *primogénito-siguiente hermano* de un bosque)

¡Menos coste en espacio pero más en tiempo para buscar!

Tries, árboles digitales de búsqueda y Patricia

- Precisión sobre las implementaciones anteriores:

En realidad, cuando un cierto nodo es la raíz de un subtrie que ya sólo contiene una palabra, se puede almacenar esa palabra (sufijo) directamente en un nodo externo (eso ahorra espacio, aunque obliga a manejar punteros a tipos distintos...)



Tries, árboles digitales de búsqueda y Patricia

- *Nodo-abb*: la estructura se llama también *árbol ternario de búsqueda*.

Cada nodo contiene:

- Dos punteros al hijo izquierdo y derecho (como en un árbol binario de búsqueda).
- Un puntero, central, a la raíz del trie al que da acceso el nodo.

Objetivo: combinar la eficiencia en tiempo de los tries con la eficiencia en espacio de los *abb*'s.

- Una búsqueda compara el carácter actual en la cadena buscada con el carácter del nodo.
- Si el carácter buscado es menor, la búsqueda de ese carácter sigue en el hijo izquierdo.
- Si el carácter buscado es mayor, se sigue en el hijo derecho.
- Si el carácter es igual, se va al hijo central, y se pasa a buscar el siguiente carácter de la cadena buscada.

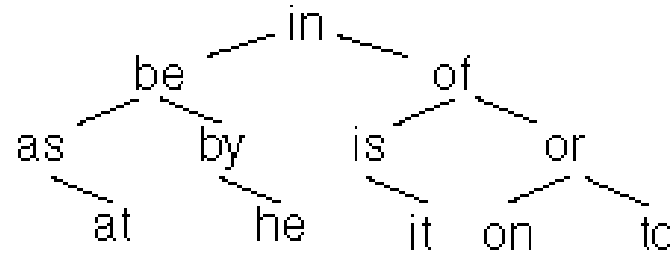


Tries, árboles digitales de búsqueda y Patricia

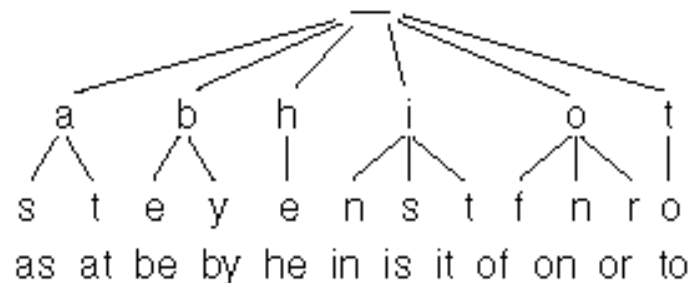
Árbol ternario de búsqueda para las palabras...

as at be by he in is it of on or to

En un *abb*:

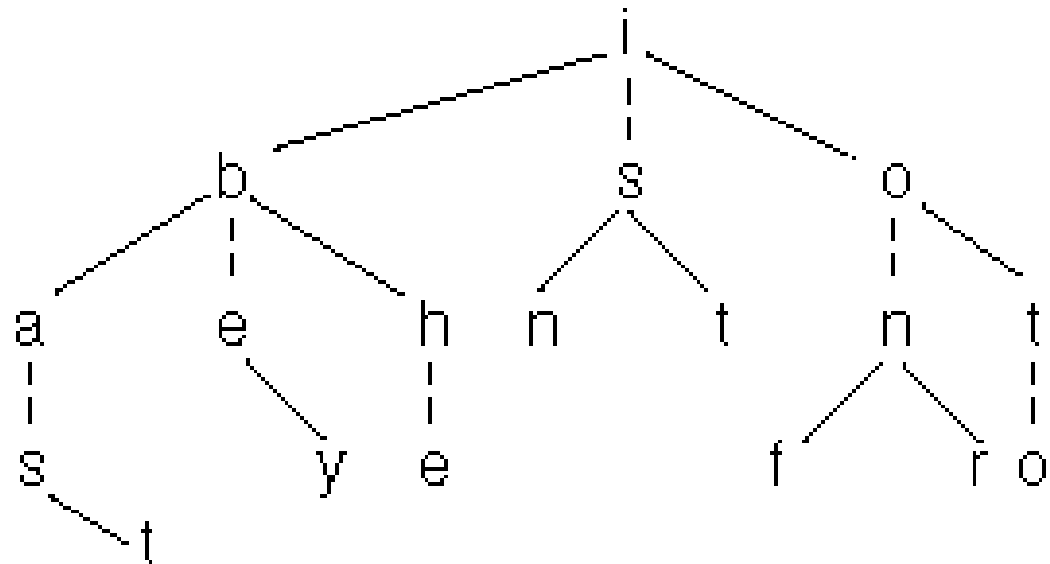


En un trie (representación nodo-vector o nodo-lista):



Tries, árboles digitales de búsqueda y Patricia

Árbol ternario de búsqueda:



as at be by he in is it of on or to



Tries, árboles digitales de búsqueda y Patricia

- *Árboles digitales de búsqueda:*
 - Caso binario ($m = 2$, es decir, sólo dos símbolos):
 - Almacenar claves completas en los nodos, pero usando los bits del argumento para decidir si se sigue por el subárbol izquierdo o por el derecho.

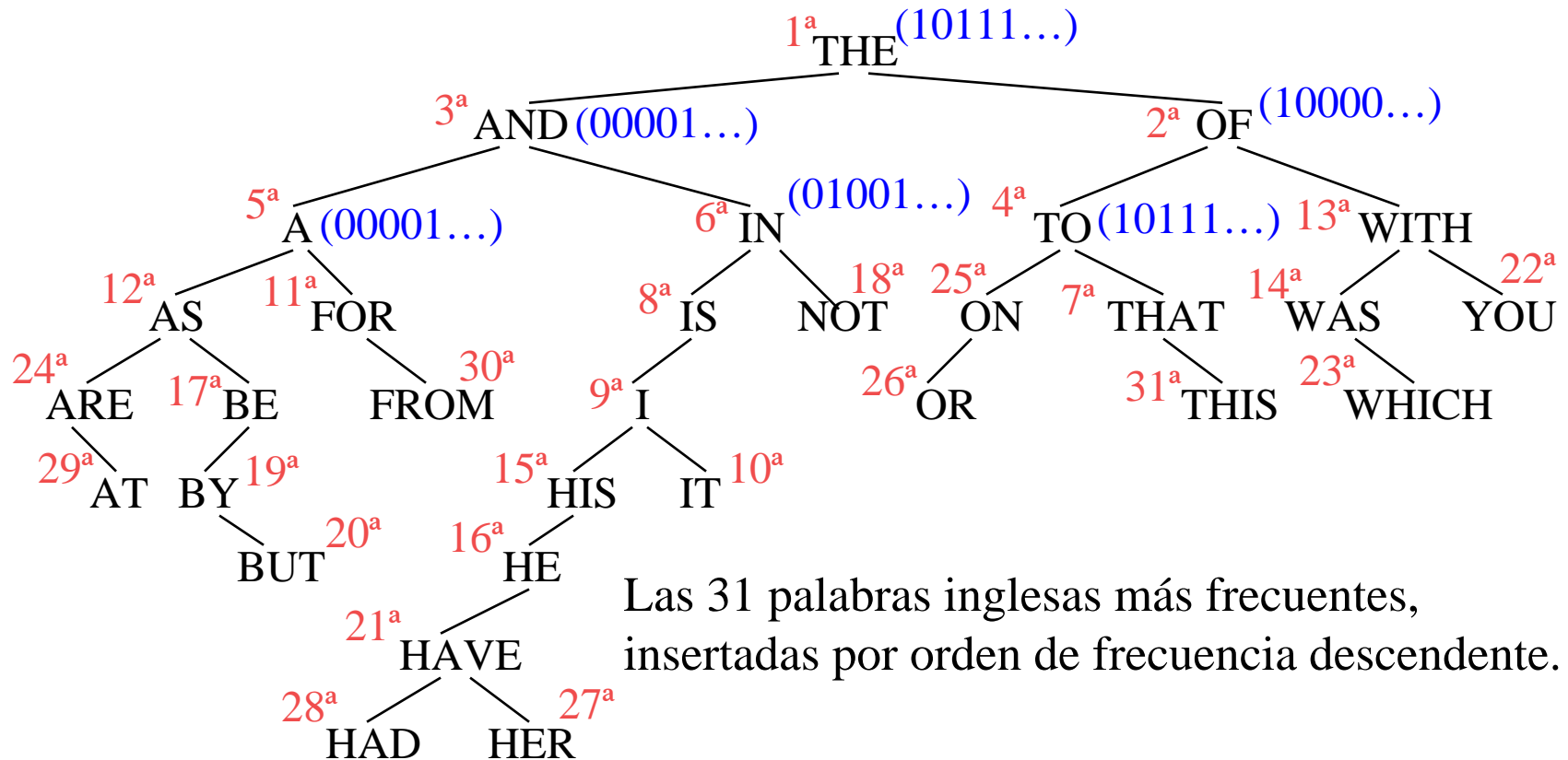
- Ejemplo,
usando el
código MIX
(D.E. Knuth)

	0	00000	I	9	01001	R	19	10011
A	1	00001	J	11	01011	S	22	10110
B	2	00010	K	12	01100	T	23	10111
C	3	00011	L	13	01101	U	24	11000
D	4	00100	M	14	01110	V	25	11001
E	5	00101	N	15	01111	W	26	11010
F	6	00110	O	16	10000	X	27	11011
G	7	00111	P	17	10001	Y	28	11100
H	8	01000	Q	18	10010	Z	29	11101



Tries, árboles digitales de búsqueda y Patricia

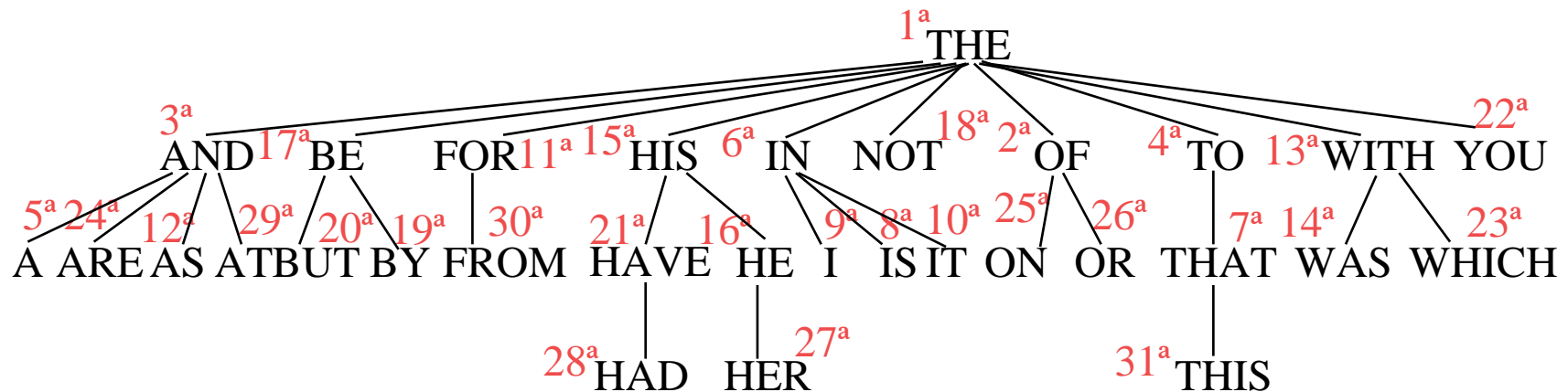
- Árboles digitales de búsqueda (caso binario):



¡Ojo! Es un árbol de búsqueda pero considerando la codificación binaria de las claves (código MIX).

Tries, árboles digitales de búsqueda y Patricia

- La búsqueda en el árbol anterior es binaria pero puede ampliarse fácilmente a m -aria ($m > 2$), para un alfabeto con m símbolos.



Los mismos datos de antes, insertados en igual orden, pero en un árbol digital de búsqueda de orden 27.



Tries, árboles digitales de búsqueda y Patricia

- *Patricia* (Practical Algorithm To Retrieve Information Coded In Alphanumeric)
 - Problema del trie: si $|\{\text{claves}\}| \ll |\{\text{claves potenciales}\}|$, la mayoría de los nodos internos tienen un solo hijo \rightarrow aumenta el coste en espacio
 - Idea: árbol binario, pero evitando las bifurcaciones de una sola dirección.
 - Patricia: representación compacta de un trie en la que todos los nodos con un solo hijo “se mezclan” con sus padres.
 - Ejemplo de utilización: tablas de encaminamiento en los *router*, asignatura “Sistemas de transporte de datos” (búsqueda de direcciones = *longest prefix matching*)

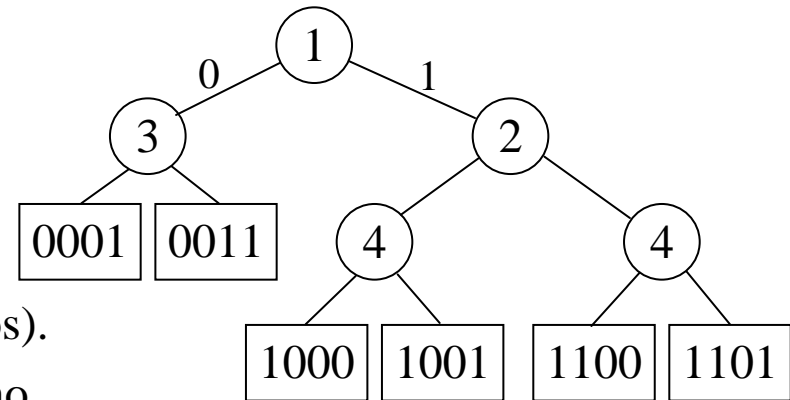


Tries, árboles digitales de búsqueda y Patricia

– Ejemplo de Patricia:

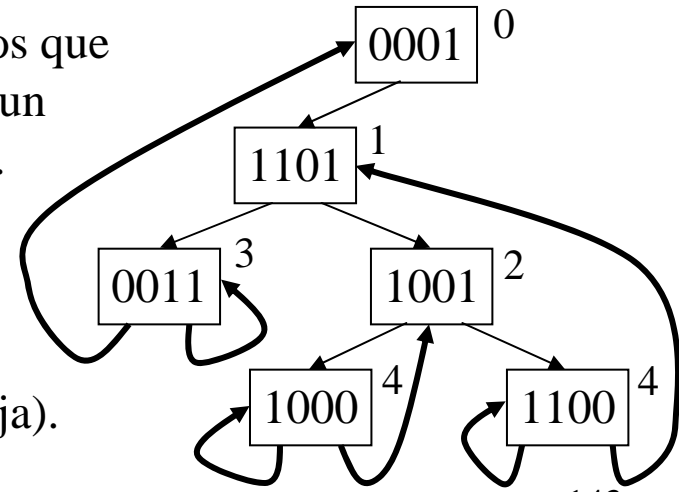
- Partimos de:

- Un trie binario con las claves almacenadas en las hojas y compactado (de manera que cada nodo interno tiene dos hijos).
- La etiqueta del nodo interno indica el bit usado para bifurcar.



- En un Patricia, las claves se almacenan en los nodos internos.

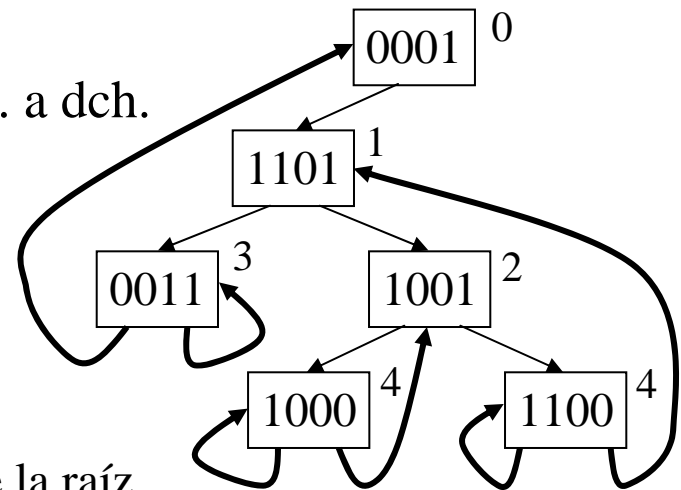
- Como hay un nodo interno menos que n° de elementos, hay que añadir un nuevo nodo (se pone como raíz).
- Cada nodo sigue guardando el n° de bit usado para bifurcar.
- Ese n° distingue si el puntero sube o baja (si $>$ el del padre, baja).



Tries, árboles digitales de búsqueda y Patricia

– Búsqueda de clave:

- Se usan los bits de la clave de izq. a dch. Bajando en el árbol.
- Cuando el puntero que se ha seguido va hacia arriba se compara la clave con la del nodo.
- Ejemplo, búsqueda de 1101:
 - Se empieza yendo al hijo izq. de la raíz.
 - El puntero que se ha seguido es hacia abajo (se sabe porque el bit que etiqueta el nodo 1101, 1, es mayor que el del padre, 0).
 - Se busca según el valor del bit 1 de la clave **buscada**, como es un 1, vamos al hijo derecho (el 1001).
 - El n° de bit del nodo alcanzado es 2, se sigue hacia abajo según ese bit, como el 2° bit de la clave es 1, vamos al hijo derecho.
 - Ahora se usa el 4° bit de la clave buscada, como es 1 seguimos el puntero hijo dch. y llegamos a la clave 1101.
 - Como el n° de bit es 1 (<4) comparamos la clave actual con la buscada, como coincide, terminamos con éxito.

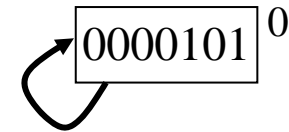


Tries, árboles digitales de búsqueda y Patricia

– Inserción de claves:

- Partimos de árbol vacío (ninguna clave).

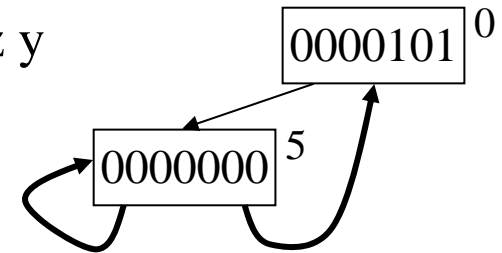
- Insertamos la clave 0000101.



- Ahora insertamos la clave 00000000.

Buscando esa clave llegamos a la raíz y vemos que es distinta. Vemos que el primer bit en que difieren es el 5°.

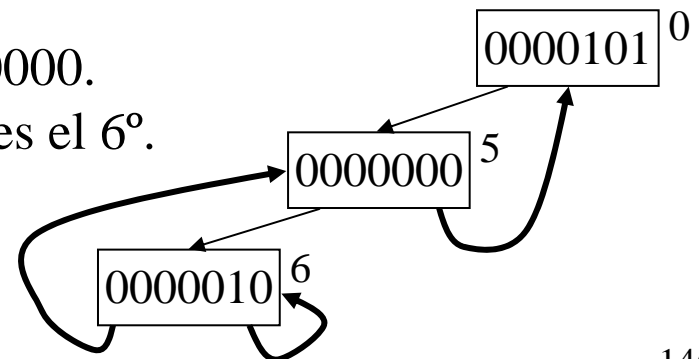
Creamos el hijo izq. etiquetado con el bit 5 y guardamos la clave en él. Como el 5° bit de la clave insertada es 0, el puntero izq. de ese nodo apunta al mismo nodo. El puntero dch. apunta al nodo raíz.



- Insertamos 0000010.

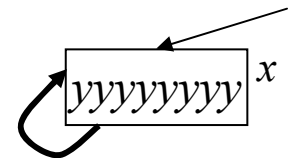
La búsqueda termina en 0000000.

El primer bit en que difieren es el 6°.



Tries, árboles digitales de búsqueda y Patricia

- Estrategia general de inserción (a partir de la 2ª clave):
 - Se busca la clave C a insertar, la búsqueda termina en un nodo con clave C' .
 - Se calcula el nº b de bit más a la izq. en que C y C' difieren.
 - Se crea un nuevo nodo con la nueva clave, etiquetado con el nº de bit anterior y se inserta en el camino desde la raíz al nodo de C' de forma que las etiquetas de nº de bit sean crecientes en el camino.
 - Esa inserción ha roto un puntero del nodo p al nodo q .
 - Ahora el puntero va de p al nuevo nodo.
 - Si el bit nº b de C es 1, el hijo dch. del nuevo nodo apuntará al mismo nodo, si no, el hijo izq. será el “auto-puntero”.
 - El hijo restante apuntará a q .



Tries, árboles digitales de búsqueda y Patricia

– Inserción de claves (cont.):

- Insertamos la clave 0001000.

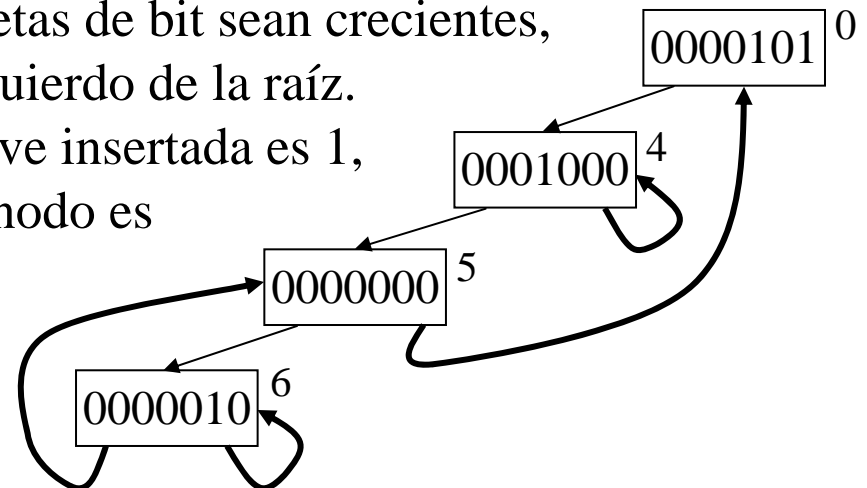
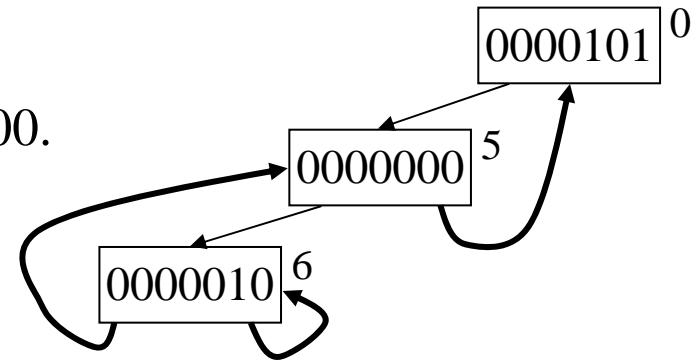
La búsqueda termina en 0000000.

El primer bit en que difieren es el 4.

Creamos un nuevo nodo con etiqueta 4 y ponemos en él la nueva clave.

Se inserta ese nodo en el camino de la raíz a 0000000 de forma que las etiquetas de bit sean crecientes, es decir, como hijo izquierdo de la raíz.

Como el bit 4 de la clave insertada es 1, el hijo dch. del nuevo nodo es un auto-puntero.



Tries, árboles digitales de búsqueda y Patricia

– Inserción de claves (cont.):

- Insertamos la clave 0000100.

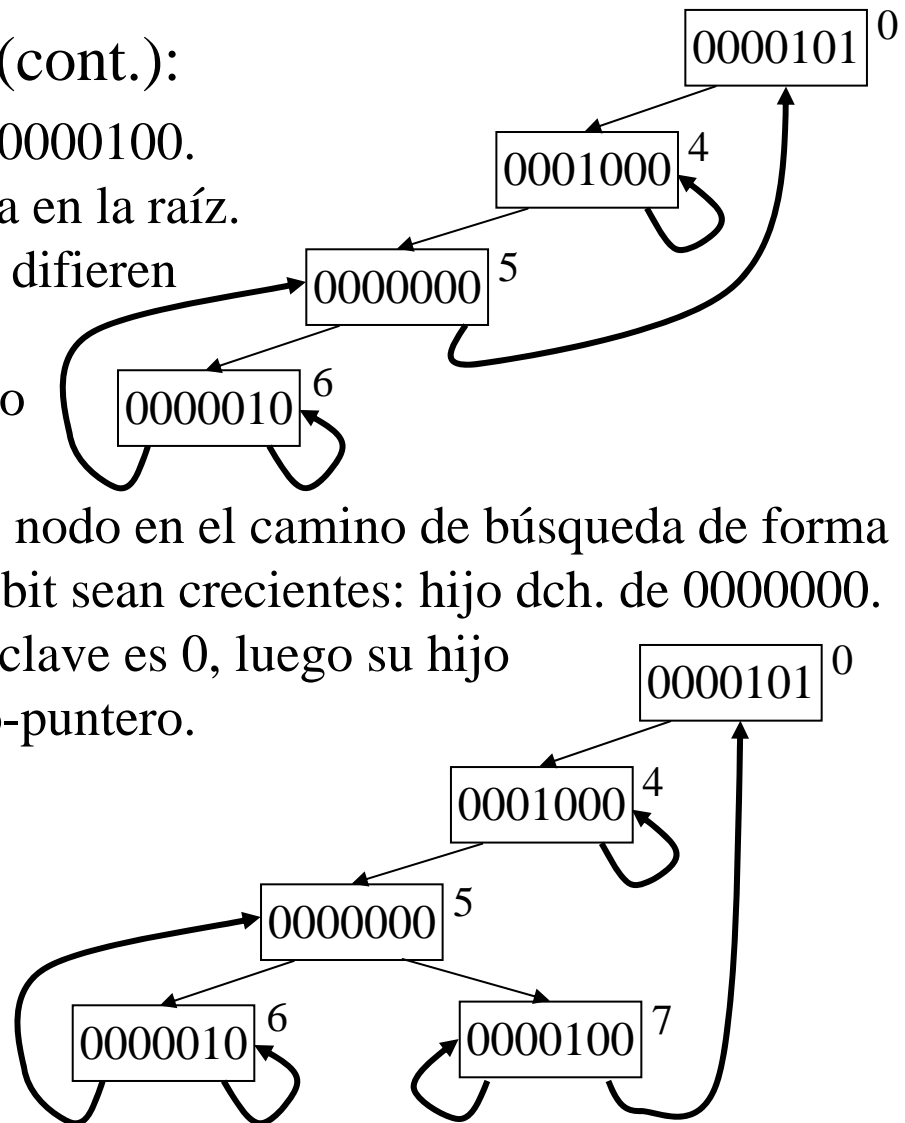
La búsqueda termina en la raíz.

El primer bit en que difieren es el 7º.

Creamos nuevo nodo con etiqueta 7.

Insertamos el nuevo nodo en el camino de búsqueda de forma que las etiquetas de bit sean crecientes: hijo dch. de 0000000.

El bit 7 de la nueva clave es 0, luego su hijo izquierdo es un auto-puntero.



Tries, árboles digitales de búsqueda y Patricia

– Inserción de claves (cont.):

- Insertamos la clave 0001010.

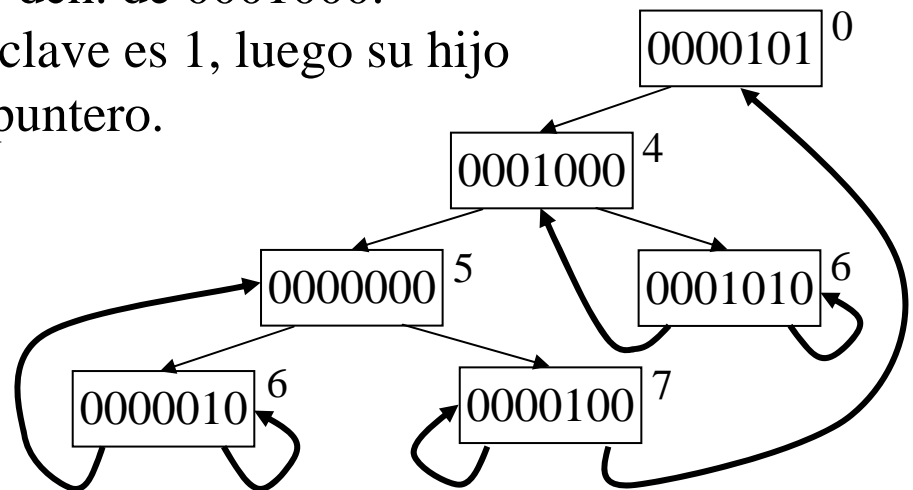
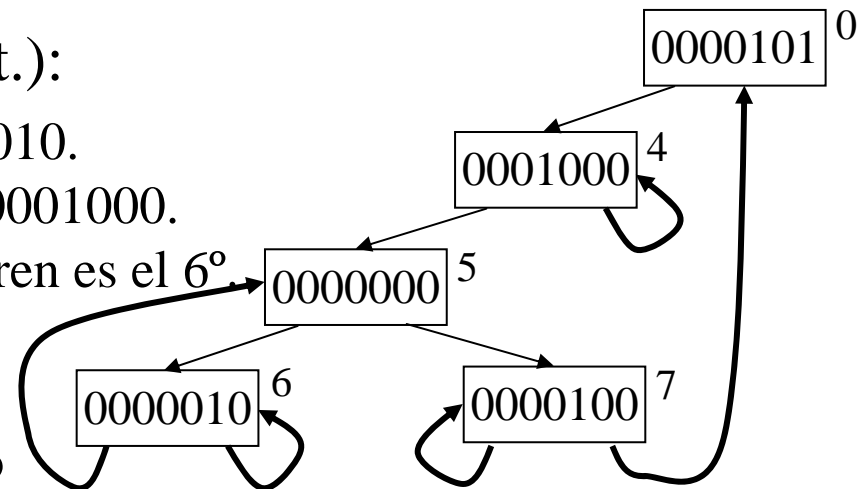
La búsqueda termina en 0001000.

El primer bit en que difieren es el 6º

Creamos nuevo nodo
con etiqueta 6.

Insertamos el nuevo nodo
en el camino de búsqueda de forma que las etiquetas de bit
sean crecientes: hijo dch. de 0001000.

El bit 6 de la nueva clave es 1, luego su hijo
derecho es un auto-puntero.



Tries, árboles digitales de búsqueda y Patricia

– Borrado de claves:

- Sea p el nodo con la clave a borrar; dos casos:
 - p tiene un auto-puntero:
 - » si p es la raíz, es el único nodo, se borra y queda vacío
 - » si p no es la raíz, hacemos que el puntero que va del padre de p a p pase a apuntar al hijo de p (que no es auto-punt.)
 - p no tiene auto-puntero:
 - » buscamos el nodo q que tiene un puntero hacia arriba a p (es el nodo desde el que llegamos a p en la búsqueda de la clave a borrar)
 - » la clave de q se mueve a p y se procede a borrar q
 - » para borrar q se busca el nodo r que tiene un puntero hacia arriba a q (se consigue buscando la clave de q)
 - » se cambia el puntero de r que va a q para que vaya a p
 - » el puntero que baja del padre de q a q se cambia al hijo de q que se usó para localizar r



Tries, árboles digitales de búsqueda y Patricia

- Análisis de los algoritmos:
 - Es evidente que el coste de las operaciones de búsqueda, inserción y borrado es de orden lineal en la altura del árbol, pero... ¿cuánto es esto?
 - Caso de tries binarios ($m = 2$, es decir, sólo dos símbolos)...
 - Hay una curiosa relación entre este tipo de árboles y un método de ordenación, el “*radix*-intercambio”, así que veamos primero un resumen sobre los métodos de ordenación *radix*.



Tries, árboles digitales de búsqueda y Patricia

- El método de la oficina de correos, llamado también ordenación por distribución (*bucket sort*)
 - Si hay que ordenar cartas por provincias, se pone una caja o cubo (*bucket*) por cada provincia y se recorren secuencialmente todas las cartas almacenándolas en la caja correspondiente → ¡muy eficiente!
 - Si hay que ordenar las cartas por códigos postales, se necesitan 100.000 cajas (y una oficina muy grande) → el método sólo es útil (y mucho) si el número de elementos posibles pertenece a un conjunto pequeño.
 - En general, si se ordenan n elementos que pueden tomar m valores distintos (número de cajas), el coste en tiempo (y en espacio) es $O(m+n)$.



Tries, árboles digitales de búsqueda y Patricia

- Primera idea para la extensión natural del método *bucket sort*:

en el caso de la ordenación de cartas por código postal...

- Primera fase: se usan 10 cajas y se ordenan las cartas por el primer dígito del código
 - cada caja incluye ahora 10.000 códigos distintos
 - el coste de esta fase es $O(n)$
- Segunda fase: se procede igual para cada caja
- Hay cinco fases...



Tries, árboles digitales de búsqueda y Patricia

- Una extensión más elaborada: *radix sort*

“Once upon a time, computer programs were written in Fortran and entered on punched cards, about 2000 to a tray. Fortran code was typed in columns 1 to 72, but columns 73-80 could be used for a card number. If you ever dropped a large deck of cards you were really in the poo, unless the cards had been numbered in columns 73-80. If they had been numbered you were saved; they could be fed into a card sorting machine and restored to their original order.

The card sorting machine was the size of three or four large filing cabinets. It had a card input hopper and ten output bins, numbered 0 to 9. It read each card and placed it in a bin according to the digit in a particular column, e.g. column 80. This gave ten stacks of cards, one stack in each bin. The ten stacks were removed and concatenated, in order: stack 0, then 1, 2, and so on up to stack 9. The whole process was then repeated on column 79, and again on column 78, 77, etc., down to column 73, at which time your deck was back in its original order!

The card sorting machine was a physical realisation of the radix sort algorithm.”

<http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Sort/Radix.html>



Tries, árboles digitales de búsqueda y Patricia

- *Radix sort*:
 - Se usa una cola de claves para implementar cada “caja” (tantas como el valor de la base de numeración utilizado, cualquier base es válida).
 - Se clasifican las claves según su dígito más a la derecha (el menos significativo), es decir, cada clave se coloca en la cola correspondiente a su dígito más a la derecha.
 - Se concatenan todas las colas (ordenadas según el dígito más a la derecha).
 - Se repite el proceso, pero clasificando según el segundo dígito por la derecha.
 - Se repite el proceso para todos los dígitos.



Tries, árboles digitales de búsqueda y Patricia

```
algoritmo radix(X,n,k)  {pre: X=vector[1..n] de claves,  
                        cada una con k dígitos;  post: X ordenado}
```

```
principio
```

```
  colocar los elmtos. de X en cola GQ {puede usarse X};
```

```
  para i:=1 hasta d hacer {d=base de numeración usada)  
    colaVacía(Q[i])
```

```
  fpara;
```

```
  para i:=k descendiendo hasta 1 hacer
```

```
    mq not esVacía(GQ) hacer
```

```
      x:=primero(GQ); eliminar(GQ);
```

```
      d:=dígito(i,x); insertar(x,Q[d])
```

```
    fmq;
```

```
    para t:=1 hasta d hacer insertarCola(Q[t],GQ) fpara
```

```
  fpara;
```

```
  para i:=1 hasta n hacer
```

```
    X[i]:=primero(GQ); eliminar(GQ)
```

```
  fpara
```

```
fin
```



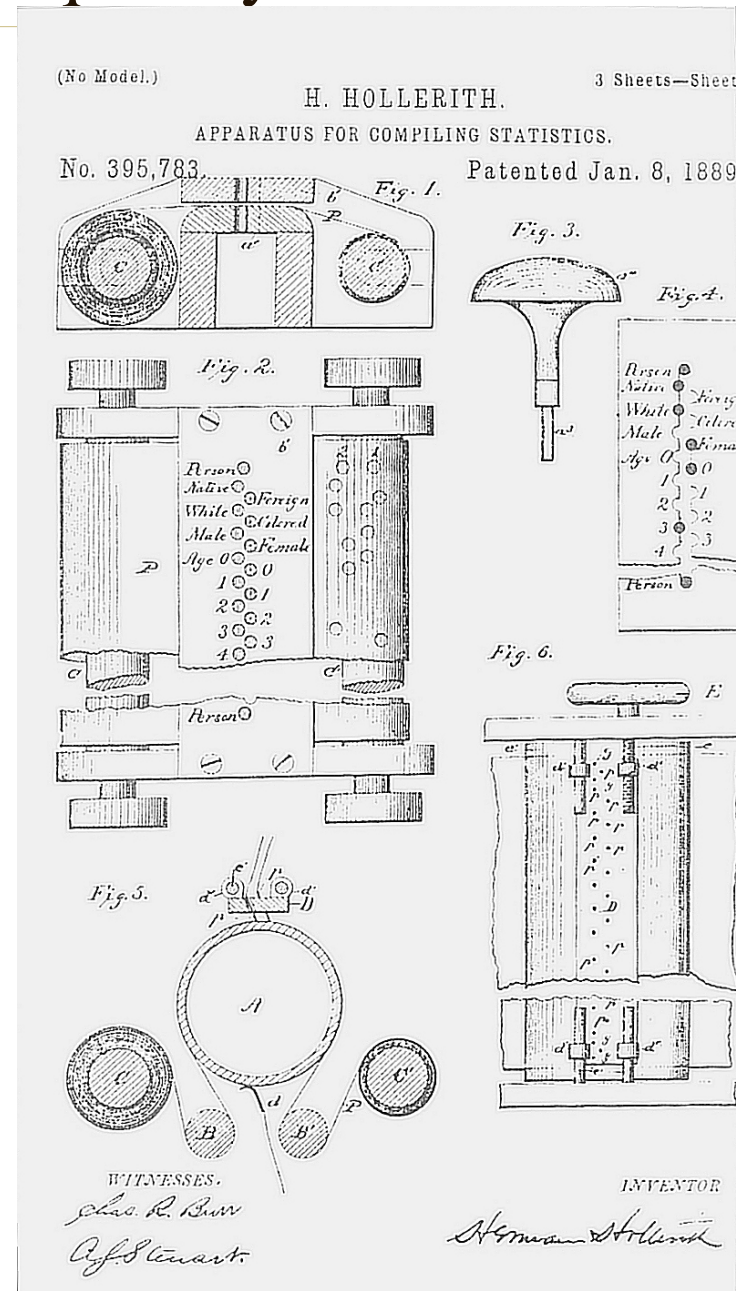
Tries, árboles digitales de búsqueda y Patricia

- Análisis del método *radix sort*:
 - En tiempo: $O(kn)$, es decir, considerando que k es una constante, es un método lineal en el tamaño del vector
 - Nótese que a mayor base de numeración menor coste
 - En espacio: $O(n)$ espacio adicional.
(Es posible hacerlo in-situ, y sólo se precisa un espacio adicional $O(\log n)$ para guardar posiciones del vector)



Tries, árboles digitales de búsqueda y Patricia

- Un poco de historia:
orígenes del *radix sort*
 - EE.UU., 1880: no se puede terminar el censo de la década anterior (en concreto, no se llega a contar el número de habitantes solteros)
 - Herman Hollerith (empleado de la oficina del censo, de 20 años de edad) inventa una máquina tabuladora eléctrica para resolver el problema; en esencia es una implementación física del *radix sort*



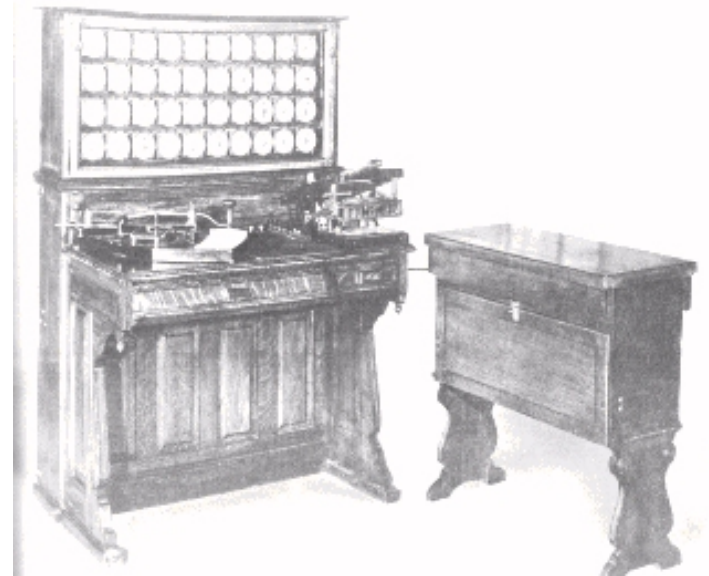
- Un poco de historia: orígenes del *radix sort* (continuación)

- Person ☐
Native ☐ Foreign ☐
White ☐ Colored ☐
Male ☐ Female ☐
Age 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10 ☐ 11 ☐ 12 ☐ 13 ☐ 14 ☐ 15 ☐ 16 ☐ 17 ☐ 18 ☐ 19 ☐ 20 ☐ 21 ☐ 22 ☐ 23 ☐ 24 ☐ 25 ☐ 26 ☐ 27 ☐ 28 ☐ 29 ☐ 30 ☐ 31 ☐ 32 ☐ 33 ☐ 34 ☐ 35 ☐ 36 ☐ 37 ☐ 38 ☐ 39 ☐ 40 ☐ 41 ☐ 42 ☐ 43 ☐ 44 ☐ 45 ☐ 46 ☐ 47 ☐ 48 ☐ 49 ☐ 50 ☐ 51 ☐ 52 ☐ 53 ☐ 54 ☐ 55 ☐ 56 ☐ 57 ☐ 58 ☐ 59 ☐ 60 ☐ 61 ☐ 62 ☐ 63 ☐ 64 ☐ 65 ☐ 66 ☐ 67 ☐ 68 ☐ 69 ☐ 70 ☐ 71 ☐ 72 ☐ 73 ☐ 74 ☐ 75 ☐ 76 ☐ 77 ☐ 78 ☐ 79 ☐ 80 ☐ 81 ☐ 82 ☐ 83 ☐ 84 ☐ 85 ☐ 86 ☐ 87 ☐ 88 ☐ 89 ☐ 90 ☐ 91 ☐ 92 ☐ 93 ☐ 94 ☐ 95 ☐ 96 ☐ 97 ☐ 98 ☐ 99 ☐ 100 ☐



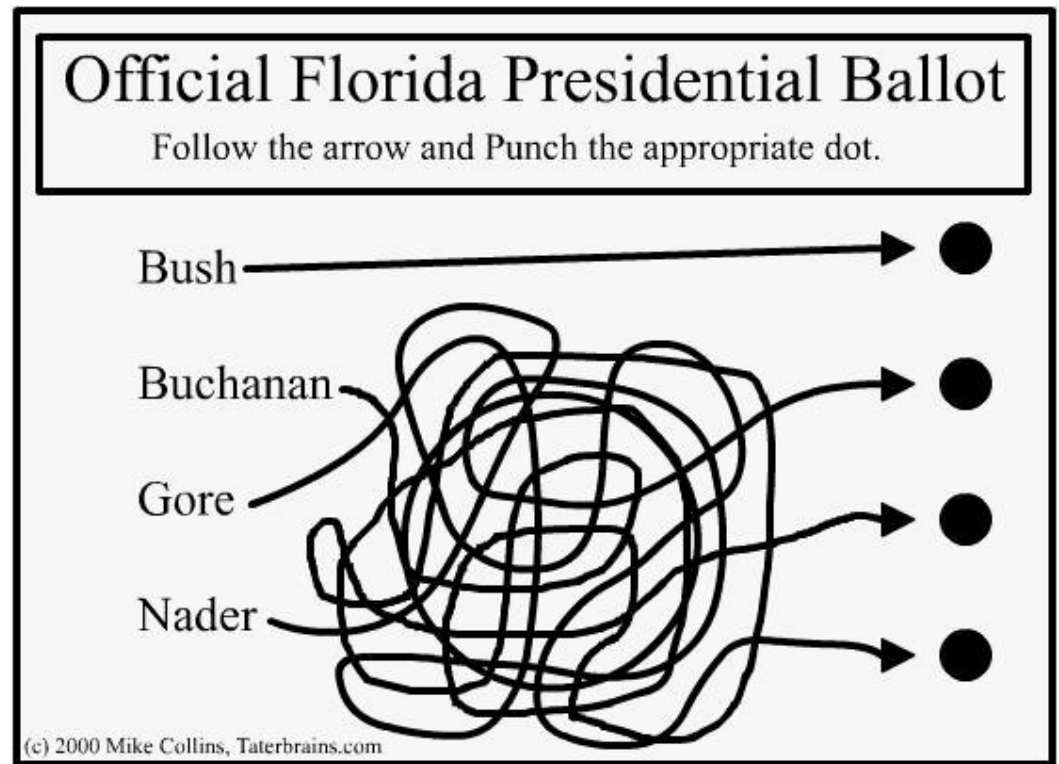
Tries, árboles digitales de búsqueda y Patricia

- Un poco de historia: orígenes del *radix sort* (continuación)
 - 1900: Hollerith resuelve otra crisis federal inventando una nueva máquina con alimentación automática de tarjetas (útil, con más o menos variaciones, hasta 1960)
 - 1911: la empresa de Hollerith se fusiona con otras dos, creando la *Calculating-Tabulating-Recording Company* (CTR)
 - 1924: Thomas Watson cambia el nombre a la CTR y la llama *International Business Machines* (IBM)



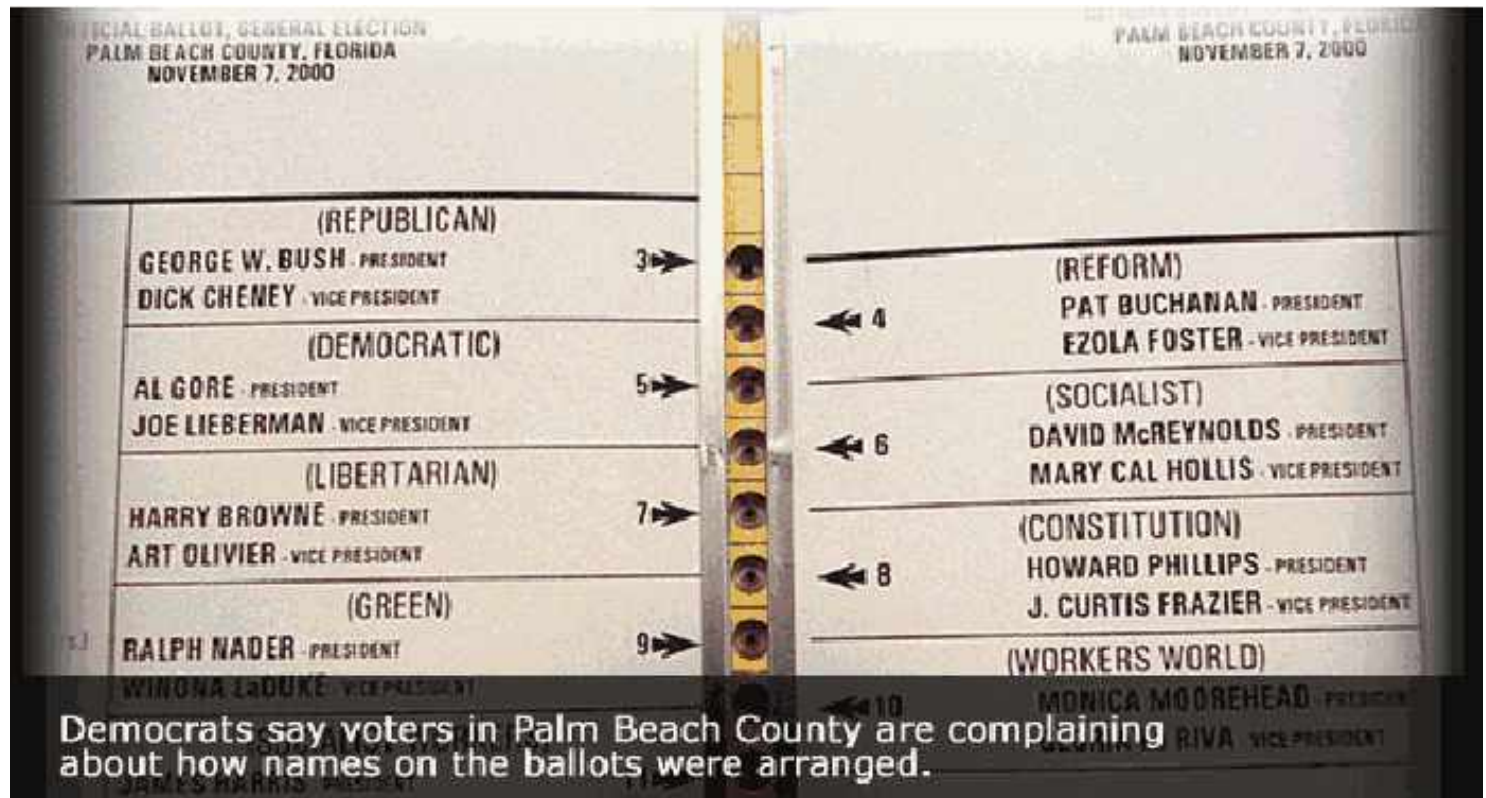
Tries, árboles digitales de búsqueda y Patricia

- Un poco de historia: orígenes del *radix sort* (continuación)
 - El resto de la historia es bien conocido... hasta:
 - 2000: crisis del recuento de votos en las Presidenciales



Tries, árboles digitales de búsqueda y Patricia

- Un poco de historia: orígenes del *radix sort* (continuación)
 - Ésta es la auténtica:



Tries, árboles digitales de búsqueda y Patricia

- Método de ordenación *radix*-intercambio:
 - (versión del libro de D. Knuth)
 - Suponer las claves en su representación binaria
 - En vez de comparar claves se comparan sus bits
 - Paso 1: se ordenan las claves según su bit más significativo
 - Se halla la clave k_i más a la izquierda que tenga un primer bit igual a 1 y la clave k_j más a la derecha que tenga un primer bit 0, se intercambian y se repite el proceso hasta que $i > j$
 - Paso 2: se parte en dos la secuencia de claves y se aplica recursivamente
 - La secuencia de claves ha quedado dividida en dos: las que empiezan por 0 y las que empiezan por 1; se aplica recursivamente el paso anterior a cada una de las dos subsecuencias de claves, pero considerando ahora el segundo bit más significativo, etcétera.



Tries, árboles digitales de búsqueda y Patricia

- El *radix*-intercambio y el *quicksort* son muy similares:
 - Ambos están basados en la idea de la *partición*.
 - Se intercambian las claves hasta que la secuencia queda dividida en dos partes:
 - La subsecuencia izquierda, en la cual todas las claves son menores o iguales que una clave K y la de la derecha en la cual todas las claves son mayores o iguales que K .
 - El *quicksort* toma como K una clave existente en la secuencia mientras que el *radix*-intercambio toma una clave artificial basada en la representación binaria de las claves.
 - Históricamente, el *radix*-intercambio fue publicado un año antes que el *quicksort* (en 1959).



Tries, árboles digitales de búsqueda y Patricia

- Análisis del *radix*-intercambio:
 - El análisis asintótico (caso promedio) del radix-intercambio es... digamos... ¡no trivial!

Según Knuth^(*),

$$U_n = \underline{n \log n} + n \left(\frac{\gamma - 1}{\ln 2} - \frac{1}{2} + f(n) \right) + O(1),$$

con $\gamma = 0,577215\dots$ la constante de Euler

y $f(n)$ una función "bastante extraña" tal que $|f(n)| < 173 \cdot 10^{-9}$

(*) Requiere manipulación de series infinitas y su aproximación, análisis matemático de variable compleja (integrales complejas, función Gamma)...



Tries, árboles digitales de búsqueda y Patricia

- Relación con el análisis de tries (caso binario):
 - El número de nodos internos de un trie binario que almacena un conjunto de claves es igual al número de particiones realizado para ordenar dichas claves con el método *radix*-intercambio.
 - El número medio de consultas de bit necesarias para encontrar una clave en un trie binario de n claves es $1/n$ veces el número de consultas de bit necesarias en la ordenación de esas n claves mediante el *radix*-intercambio.



tap

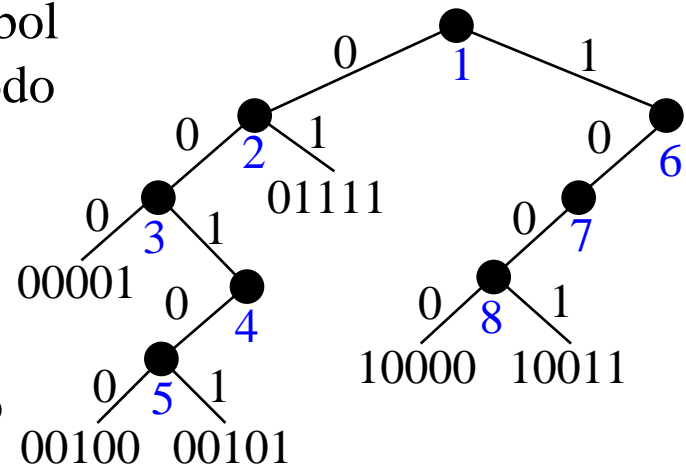
- Ejemplo: con 6 claves, las letras de ‘ORDENA’

(claves codificadas con el código MIX, pág.69)

(claves codificadas con el código MIX, pag.69)							i	d	bit
1	10000	10011	00100	00101	01111	00001	1	6	1
2	00001	01111	00100	00101	10011	10000	1	4	2
3	00001	00101	00100	01111	10011	10000	1	3	3
4	00001	00101	00100	01111	10011	10000	2	3	4
5	00001	00101	00100	01111	10011	10000	2	3	5
6	00001	00100	00101	01111	10011	10000	5	6	2
7	00001	00100	00101	01111	10011	10000	5	6	3
8	00001	00100	00101	01111	10011	10000	5	6	4
9	00001	00100	00101	01111	10000	10011			

8 particiones: los nodos internos del árbol corresponden con las particiones (el nodo k -ésimo del recorrido en pre-orden corresponde con la k -ésima partición).

El nº de consultas de bit en un nivel de partición es igual al nº de claves dentro del subárbol del nodo correspondiente.



Tries, árboles digitales de búsqueda y Patricia

- Por tanto, el coste promedio de una búsqueda en un trie binario con n claves es:

$$U_n = \log n + \frac{\gamma - 1}{\ln 2} - \frac{1}{2} + f(n) + O(n^{-1}),$$

con $\gamma = 0,577215\dots$ la constante de Euler

y $f(n)$ una función "bastante extraña" tal que $|f(n)| < 173 * 10^{-9}$

- El número medio de nodos de un trie binario de n claves es:

$$\frac{n}{\ln 2} + ng(n) + O(1),$$

con $g(n)$ otra función despreciable, como $f(n)$



Tries, árboles digitales de búsqueda y Patricia

- Análisis de tries m -arios:
 - El análisis es igual de difícil o más que el caso binario..., resulta:
 - El número de nodos necesarios para almacenar n claves al azar en un trie m -ario es aproximadamente $n/\ln m$
 - El número de dígitos o caracteres examinados en una búsqueda al azar es aproximadamente $\log_m n$
- El análisis de los árboles digitales de búsqueda y de Patricia da resultados muy parecidos
- Según Knuth, el análisis de Patricia es...

“posiblemente el hueso asintótico más duro que hemos tenido que roer...”



Análisis del caso promedio

- El plan:
 - Probabilidad
 - Análisis probabilista
 - Árboles binarios de búsqueda contruidos aleatoriamente
 - Tries, árboles digitales de búsqueda y Patricia
 - **Listas “skip”**
 - Árboles aleatorizados



Listas “skip”^(*)

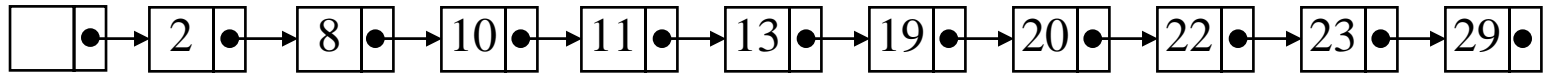
- Son “estructuras de datos probabilistas”.
- Son una alternativa a los árboles de búsqueda equilibrados (AVL, 2-3, rojinegros,...) para almacenar diccionarios de n datos con un coste promedio $O(\log n)$ de las operaciones.
- Son mucho más fáciles de implementar que, por ejemplo, los árboles AVL o los rojinegros.

(*) Listas con saltos.



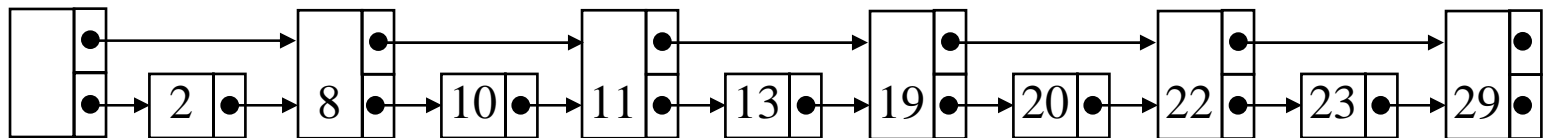
Listas “skip”

- Lista enlazada (ordenada):

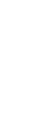


- El coste de una búsqueda en el caso peor es lineal en el número de nodos.

- Añadiendo un puntero a cada nodo par...

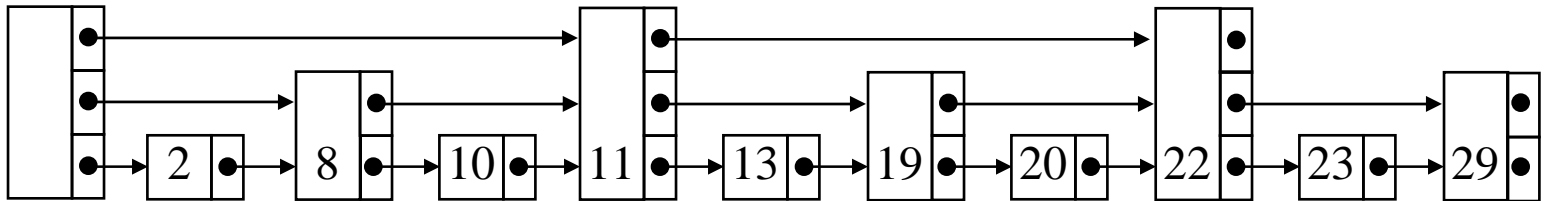


- Ahora el número de nodos examinados en una búsqueda es, como mucho, $\lceil n/2 \rceil + 1$.



Listas “skip”

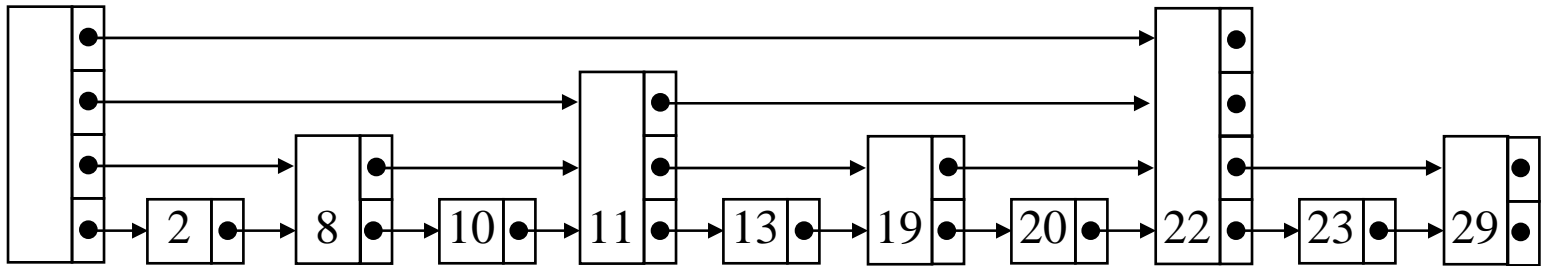
- Y añadiendo otro a cada nodo múltiplo de 4...



- Ahora el número de nodos examinados en una búsqueda es, como mucho, $\lceil n/4 \rceil + 2$.

Listas “skip”

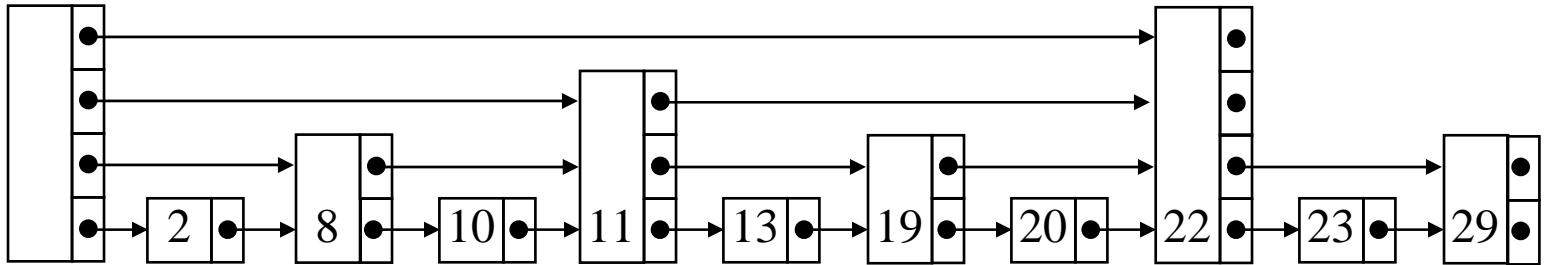
- Y por fin el **caso límite**: cada nodo múltiplo de 2^i apunta al nodo que está 2^i lugares por delante (para todo $i \geq 0$):



- El número total de punteros se ha duplicado (con respecto a la lista enlazada inicial).
- Ahora el tiempo de una búsqueda está acotado superiormente por $\lceil \log_2 n \rceil$, porque la búsqueda consiste en avanzar al siguiente nodo (por el puntero alto) o bajar al nivel siguiente de punteros y seguir avanzando...
- En esencia, se trata de una búsqueda binaria.
- Problema: la inserción es demasiado difícil.

Listas “skip”

- En particular...



- Si llamamos nodo de nivel k al que tiene k punteros, se cumple la siguiente propiedad (más débil que la definición del “caso límite”):

El puntero i -ésimo de cualquier nodo de nivel k ($k \geq i$) apunta al siguiente nodo de nivel i o superior.

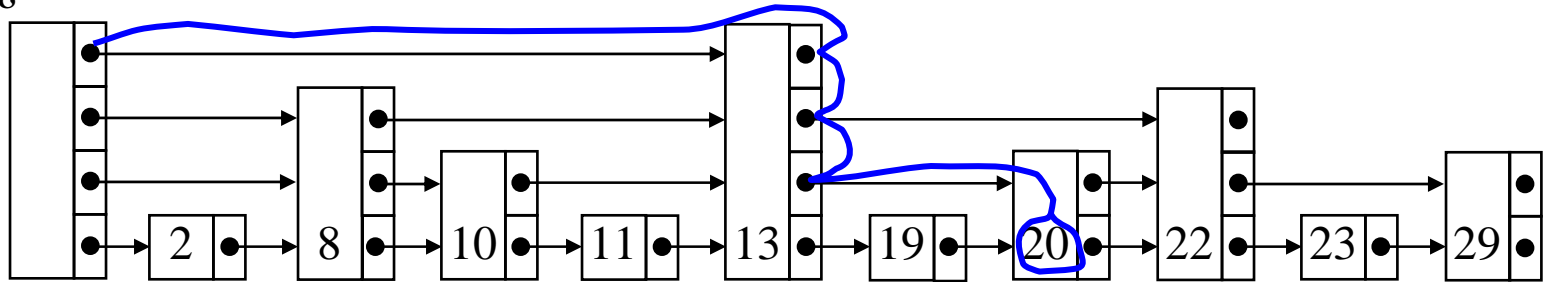
- Adoptamos ésta como definición de lista “skip” (junto con la decisión aleatoria del nivel de un nuevo nodo).



Listas “skip”

- Un ejemplo de lista “skip”:

¿20?



- ¿Cómo implementar la búsqueda?
 - Se empieza en el puntero más alto de la cabecera.
 - Se continúa por ese nivel hasta encontrar un nodo con clave mayor que la buscada (o NIL), entonces se desciende un nivel y se continúa de igual forma.
 - Cuando el avance se detiene en el nivel 1, se está frente el nodo con la clave buscada o tal clave no existe en la lista.

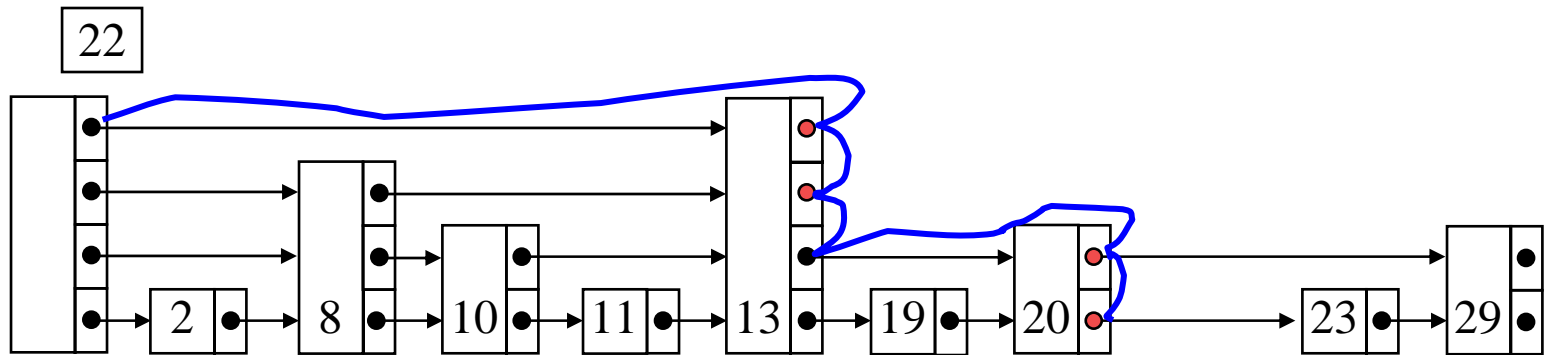
Listas “skip”

- ¿Y la inserción? (el borrado es similar)
 - Primero: para insertar un nuevo elemento hay que decidir de qué nivel debe ser.
- En una lista de las del “caso límite” la mitad de los nodos son de nivel 1, una cuarta parte de nivel 2 y, en general, $1/2^i$ nodos son de nivel i .
- Se elige el nivel de un nuevo nodo de acuerdo con esas probabilidades: se tira una moneda al aire hasta que salga cara, y se elige el número total de lanzamientos realizados como nivel del nodo.
 - Distribución geométrica de parámetro $p = 1/2$.
(En realidad se puede plantear con un parámetro arbitrario, p , y luego seleccionar qué valor de p es más adecuado)

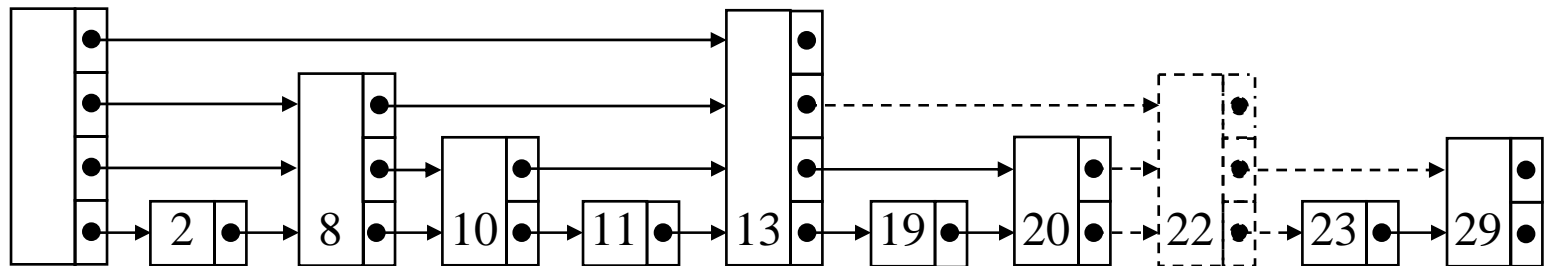


Listas “skip”

- Segundo: hay que saber dónde insertar.
 - Se hace igual que en la búsqueda, guardando traza de los nodos en los que se desciende de nivel.



- Se determina aleatoriamente el nivel del nuevo nodo y se inserta, enlazando los punteros convenientemente.



Listas “skip”

- Se requiere una estimación a priori del tamaño de la lista (igual que en tablas *hash*) para determinar el número de niveles.
- Si no se dispone de esa estimación, se puede asumir un número grande o usar una técnica similar al *rehashing* (reconstrucción).
- Resultados experimentales muestran que las listas “skip” son tan eficientes como muchas implementaciones de árboles de búsqueda equilibrados, y son más fáciles de implementar.



Listas “skip”

```
algoritmo buscar(lista:lista_skip; clave_buscada:clave)
principio
  x:=lista.cabecera;
  {invariante:  $x \uparrow .clave < clave\_buscada$ }
  para i:=lista.nivel descendiendo hasta 1 hacer
    mq  $x \uparrow .sig[i] \uparrow .clave < clave\_buscada$  hacer
      x:= $x \uparrow .sig[i]$ 
    fmq
  fpara;
  { $x \uparrow .clave < clave\_buscada \leq x \uparrow .sig[1] \uparrow .clave$ }
  x:= $x \uparrow .sig[1]$ ;
  si  $x \uparrow .clave = clave\_buscada$  ent
    devuelve  $x \uparrow .valor$ 
  sino
    fracaso en la búsqueda
  fsi
fin
```



Listas “skip”

```
algoritmo nivel_aleatorio devuelve natural
principio
  nivel:=1;
  {la función random devuelve un valor uniforme [0,1) }
  mq random<p and nivel<MaxNivel hacer
    nivel:=nivel+1
  fmq;
  devuelve nivel
fin
```

MaxNivel se elige como $\log_{1/p} N$, donde N = cota superior de la longitud de la lista.

Por ejemplo, si $p = 1/2$, MaxNivel = 16 es apropiado para listas de hasta 2^{16} (= 65.536) elementos.



Listas “skip”

```
algoritmo inserta(lista:listas_skip; c:clave; v:valor)
variable traza:vector[1..MaxNivel] de punteros
principio
  x:=lista.cabecera;
  para i:=lista.nivel descendiendo hasta 1 hacer
    mq x↑.sig[i]↑.clave<c hacer
      x:=x↑.sig[i]
    fmq;
    {x↑.clave<c≤x↑.sig[i]↑.clave}
    traza[i] :=x
  fpara;
  x:=x↑.sig[1];
  si x↑.clave=c ent
    x↑.valor:=v
  sino {inserción}
  . . .
```



Listas “skip”

```
. . .  
sino {inserción}  
    nivel:=nivel_aleatorio;  
    si nivel>lista.nivel ent  
        para i:=lista.nivel+1 hasta nivel hacer  
            traza[i]:=lista.cabecera  
        fpara;  
        lista.nivel:=nivel  
    fsi;  
    x:=nuevoDato(nivel,c,v);  
    para i:=1 hasta nivel hacer  
        x↑.sig[i]:=traza[i]↑.sig[i];  
        traza[i]↑.sig[i]:=x  
    fpara  
    fsi  
fin
```



Listas “skip”

```
algoritmo borra(lista:lista_skip; c:clave)
variable traza:vector[1..MaxNivel] de punteros
principio
  x:=lista.cabecera;
  para i:=lista.nivel descendiendo hasta 1 hacer
    mq x↑.sig[i]↑.clave<c hacer
      x:=x↑.sig[i]
    fmq;
    {x↑.clave<c≤x↑.sig[i]↑.clave}
    traza[i] :=x
  fpara;
  x:=x↑.sig[1];
  si x↑.clave=c ent {borrado}
  . . .
```



Listas “skip”

```
. . .  
si x↑.clave=c ent {borrado}  
  para i:=1 hasta lista.nivel hacer  
    si traza[i]↑.sig[i]≠x ent break fsi;  
    traza[i]↑.sig[i]:=x↑.sig[i]  
  fpara;  
  liberar(x);  
  mq lista.nivel>1 and  
    lista.cabecera↑.sig[lista.nivel]=NIL hacer  
    lista.nivel:=lista.nivel-1  
  fmq  
fsi  
fin
```



Listas “skip”

- Análisis del coste:
 - El tiempo requerido por la búsqueda, la inserción y el borrado están dominados por el tiempo de búsqueda de un elemento.
 - Para la inserción y el borrado hay un coste adicional proporcional al nivel del nodo que se inserta o borra.
 - El tiempo necesario para la búsqueda es proporcional a la longitud del camino de búsqueda.
 - La longitud del camino de búsqueda está determinada por la estructura de la lista, es decir, por el patrón de apariciones de nodos con distintos niveles.
 - La estructura de la lista está determinada por el número de nodos y por los resultados de la generación aleatoria de los niveles de los nodos.



Listas “skip”

- Análisis del coste (detalles):
 - Analizamos la longitud del camino de búsqueda pero de derecha a izquierda, es decir, empezando desde la posición inmediatamente anterior al elemento buscado
- Primero: ¿cuál es el número de punteros que hay que recorrer para ascender desde el nivel 1 (del elemento anterior al buscado) hasta el nivel $L(n) = \log_{1/p} n$?
 - Estamos asumiendo que es seguro que se alcanza ese nivel $L(n)$; esta hipótesis es como asumir que la lista se alarga infinitamente hacia la izquierda.
 - Suponer que, durante esa escalada, estamos en el puntero i -ésimo de un cierto nodo x .
 - El nivel de x debe ser al menos i , y la probabilidad de que el nivel de x sea mayor que i es p .



Listas “skip”

- Escalada desde el nivel 1 hasta el nivel $L(n)$...
 - Podemos interpretar la escalada hasta el nivel $L(n)$ como una serie de experimentos de Bernoulli independientes, llamando “éxito” a un movimiento hacia arriba y “fracaso” a un movimiento hacia la izquierda.
 - Entonces, el número de movimientos a la izquierda en la escalada hasta el nivel $L(n)$ es el número de fallos hasta el $(L(n)-1)$ -ésimo éxito de la serie de experimentos, es decir, es una binomial negativa $BN(L(n)-1, p)$.
 - El número de movimientos hacia arriba es exactamente $L(n)-1$, por tanto:
coste de escalar al nivel $L(n)$
en una lista de longitud infinita $=_{\text{prob}} (L(n)-1) + BN(L(n)-1, p)$
Nota: $X =_{\text{prob}} Y$ si $\Pr\{X > t\} = \Pr\{Y > t\}$, para todo t
además, $X \leq_{\text{prob}} Y$ si $\Pr\{X > t\} \leq \Pr\{Y > t\}$, para todo t .
 - La hipótesis de lista infinita es pesimista, es decir:
coste de escalar al nivel $L(n)$
en una lista de longitud $n \leq_{\text{prob}} (L(n)-1) + BN(L(n)-1, p)$



Listas “skip”

- Segundo: una vez en el nivel $L(n)$, ¿cuál es el número de movimientos a la izquierda hasta llegar a la cabecera?
 - Está acotado por el número de elementos de nivel $L(n)$ o superior en la lista. Este número es una variable aleatoria binomial $B(n, 1/np)$.
- Tercero: una vez en la cabecera hay que escalar hasta el nivel más alto.
 - M = variable aleatoria “máx. nivel en una lista de n elementos”
 $\Pr\{\text{nivel de un nodo} > k\} = p^k \Rightarrow \Pr\{M > k\} = 1 - (1 - p^k)^n < np^k$
 - Se tiene que: $M \leq_{\text{prob}} L(n) + BN(1, 1-p) + 1$
Dem: $\Pr\{BN(1, 1-p) + 1 > i\} = p^i \Rightarrow \Pr\{L(n) + BN(1, 1-p) + 1 > k\} = \Pr\{BN(1, 1-p) + 1 > k - L(n)\} = 1/2^{k-L(n)} = np^k$.
Luego: $\Pr\{M > k\} < \Pr\{L(n) + BN(1, 1-p) + 1 > k\}$ para todo k .



Listas “skip”

– Juntando resultados:

número de comparaciones en la búsqueda =

$$= \text{longitud del camino de búsqueda} + 1 \leq_{\text{prob}}$$

$$\leq_{\text{prob}} L(n) + BN(L(n)-1, p) + B(n, 1/np) + BN(1, 1-p) + 1$$

Su valor medio es

$$L(n)/p + 1/(1-p) + 1 = \underline{O(\log n)}$$

Elección de p ...

p	tiempo de búsqueda (normalizado para 1/2)	nº medio de punteros por nodo
1/2	1	2
1/e	0,94...	1,58...
1/4	1	1,33...
1/8	1,33...	1,14...
1/16	2	1,07...



Listas “skip”

- Comparación con otras estructuras de datos:
 - El coste de las operaciones es del mismo orden de magnitud que con los árboles equilibrados (AVL) y con los árboles auto-organizativos (los veremos...).
 - Las operaciones son más fáciles de implementar que las de árboles equilibrados y auto-organizativos.
 - La diferencia la marcan los factores constantes:
 - Estos factores son fundamentales, especialmente en algoritmos sub-lineales (como es el caso): si A y B resuelven el mismo problema en $O(\log n)$ pero B es el doble de rápido que A , entonces en el tiempo en que A resuelve un problema de tamaño n , B resuelve uno de tamaño n^2 .



Listas “skip”

- La “complejidad” (en el sentido de dificultad de implementar) inherente a un algoritmo supone una cota inferior para los factores constantes de cualquier implementación del mismo.
 - Por ejemplo, los árboles auto-organizativos se reordenan continuamente mientras se realiza una búsqueda, sin embargo el bucle más interno de la operación de borrado en listas “skip” se compila en tan solo seis instrucciones en una CPU 68020.
- Si un algoritmo es “difícil”, los programadores postponen (... o nunca llevan a cabo) las posibles optimizaciones de la implementación.
 - Por ejemplo, los algoritmos de inserción y borrado de árboles equilibrados se plantean como recursivos, con el coste adicional que eso supone (en cada llamada recursiva). Sin embargo, dada la dificultad de esos algoritmos, no suelen implementarse soluciones iterativas.



Listas “skip”

implementación	búsqueda	inserción	borrado
lista “skip”	0,051 ms (1,0)	0,065 ms (1,0)	0,059 ms (1,0)
AVL no-recursivo	0,046 ms (0,91)	0,10 ms (1,55)	0,085 ms (1,46)
árbol 2-3 recurs.	0,054 ms (1,05)	0,21 ms (3,2)	0,21 ms (3,65)
árbol auto-organ.:			
ajuste desc.	0,15 ms (3,0)	0,16 ms (2,5)	0,18 ms (3,1)
ajuste asc.	0,49 ms (9,6)	0,51 ms (7,8)	0,53 ms (9,0)

- Todas son implementaciones optimizadas.
- Se refieren a tiempos de CPU en una Sun-3/60 y con una estructura de datos de 2^{16} (= 65.536) claves enteras.
- Los valores entre paréntesis son valores relativos normalizados para las listas “skip”.
- Los tiempos de inserción y borrado NO incluyen el tiempo de manejo de memoria dinámica (“nuevoDato” y “liberar”).



Análisis del caso promedio

- El plan:
 - Probabilidad
 - Análisis probabilista
 - Árboles binarios de búsqueda contruidos aleatoriamente
 - Tries, árboles digitales de búsqueda y Patricia
 - Listas “skip”
 - **Árboles aleatorizados**



Árboles aleatorizados

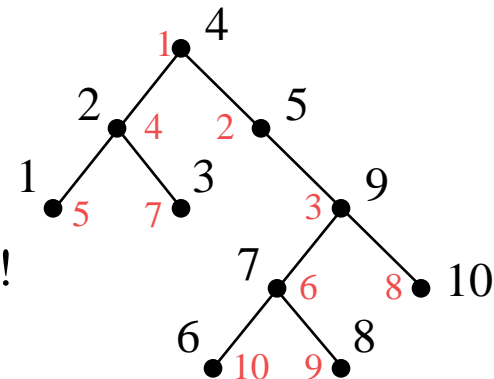
- *Treap* aleatorio (*treap* = *tree* + *heap*)
 - Otra estructura de datos probabilista para almacenar diccionarios con coste promedio logarítmico de las operaciones fundamentales
 - Definición. *Treap* es un árbol binario con dos claves:
 - con respecto a una de las claves (que es única) es un *abb*;
 - con respecto a la otra clave (que también es única), llamada prioridad, el árbol tiene estructura de montículo (*heap*).
 - Ejemplo:

Claves: 1..10

Prioridad: {4,5,9,2,1,7,3,10,8,6}

1 2 3 4 5 6 7 8 9 10

¡El *treap* es único!

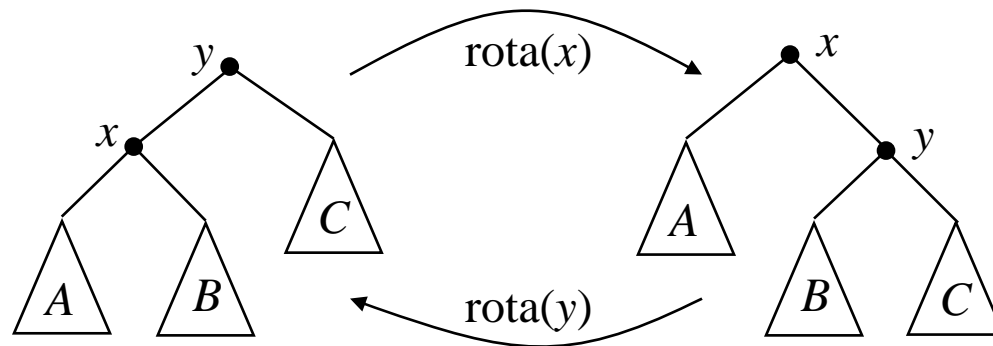


- Definición. *Treap* aleatorio es un *treap* en el que la prioridad de un nodo se asigna aleatoriamente cuando se va a insertar el nodo.



Árboles aleatorizados

- Operaciones con *treaps* aleatorios:
 - Búsqueda: igual que en un *abb*.
 - Inserción:
 - se coloca el nuevo nodo como una nueva hoja en su posición adecuada de acuerdo al *abb*,
 - se asigna una prioridad aleatoriamente (que debe ser distinta de las anteriores),
 - la hoja insertada va subiendo hacia arriba, mediante *rotaciones* hasta alcanzar la posición adecuada a su prioridad



Preserva:

- *abb*
- *heap*



Árboles aleatorizados

– Borrado:

- se busca el dato a borrar (como en un *abb*),
- se hace descender con rotaciones hasta que es una hoja, preservando el *heap*:
 - por ejemplo, si los hijos del dato a borrar, m , son j y k y la prioridad de j es mayor que la de k , se rota m hacia abajo en la dirección de j , para hacer j antecesor de k

– Comentario:

- La posición de cualquier elemento se determina desde el momento de la inserción
- El árbol sufre pocas re-estructuraciones con las inserciones y borrados (veremos que el número medio de rotaciones en una inserción o en un borrado es como máximo 2)



Árboles aleatorizados

- Análisis de los *treaps* aleatorios:
 - El promedio (entre todas las asignaciones aleatorias de prioridad) del tiempo de ejecución de una búsqueda, inserción o borrado es $O(\log n)$.
 - Hacemos el análisis para el borrado (ejercicio: ver que el coste no es mayor para una búsqueda o inserción)
 - Suponer que el *treap* guarda los datos $1..n$ (con una probabilidad asignada de forma aleatoria a cada uno) y hay que borrar el dato m .
 - Longitud media del camino de la raíz a m :
 - Sean $m_{\leq} = \{1, 2, \dots, m\}$, $m_{\geq} = \{m, m+1, \dots, n\}$, y A el conjunto de antecesores de m en el árbol (incluido él mismo).
 - Sea X la variable aleatoria “longitud del camino de la raíz a m ”

$$X = |m_{\leq} \cap A| + |m_{\geq} \cap A| - 2$$



Árboles aleatorizados

- Por tanto:

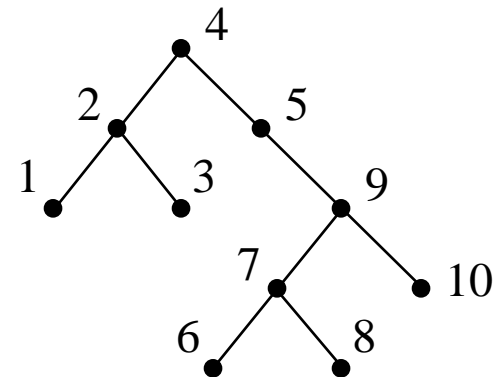
$$E[X] = E[|m_{\leq} \cap A|] + E[|m_{\geq} \cap A|] - 2$$

- Por simetría, basta calcular $E[|m_{\leq} \cap A|]$
- Nótese que un elemento de m_{\leq} está en A si es mayor que todos los elementos anteriores a él, suponiendo la ordenación por prioridad descendente

prioridad: {4,5,9,2,1,7,3,10,8,6}

borrar: $m = 8$

$m_{\leq} = \{1,2,3,4,5,6,7,8\}$, ordenados
según la prioridad: {4,5,2,1,7,3,8,6}



recorriendo de izquierda a derecha, los que son mayores que todos los anteriores son {4,5,7,8} = elementos de m_{\leq} que están en el camino de la raíz a m



Árboles aleatorizados

- Problema: calcular la media de la variable aleatoria...
 - H_m = “número de éxitos obtenidos al recorrer una permutación aleatoria de $\{1, 2, \dots, m\}$ de izquierda a derecha buscando elementos mayores que todos los de su izquierda”
 - Sea σ una permutación aleatoria de $\{1, 2, \dots, m\}$.
 - Sea σ' la permutación de $\{2, 3, \dots, m\}$ obtenida borrando el elemento 1 de σ .
 - Un elemento distinto del 1 se cuenta como éxito en σ si y sólo si se cuenta como éxito en σ' , por tanto el número esperado de éxitos, sin contar el 1, es $E[H_{m-1}]$.
 - El elemento 1 se cuenta como éxito sólo si está el primero, y eso ocurre con probabilidad $1/m$.
 - Por tanto: $E[H_m] = E[H_{m-1}] + 1/m$ y $E[H_1] = 1$
 - La solución es:

$$E[H_m] = \sum_{k=1}^m \frac{1}{k} = O(\log m)$$



Árboles aleatorizados

- Por tanto, el sitio del elemento se encuentra en promedio en $O(\log n)$.
- Falta contar el número de rotaciones necesarias para borrar el elemento m de su posición
 - Ese número es la suma de la longitud del camino más a la derecha en el subárbol izquierdo de m y de la longitud del camino más a la izquierda en el subárbol derecho de m .
 - En efecto, al rotar m hacia abajo a la izquierda (derecha) la longitud del camino más a la derecha (izquierda) en el subárbol izquierdo (derecho) disminuye en 1 y la longitud del más a la izquierda (derecha) del subárbol derecho (izquierdo) se queda igual



Árboles aleatorizados

- Problema: calcular la media de la variable aleatoria...
 - G_m = “longitud del camino más a la derecha del subárbol izquierdo de m ”
 - Por simetría, la longitud media del camino más a la izquierda del subárbol derecho de m es $E[G_{n-m+1}]$, y por tanto el número esperado de rotaciones para borrar m es $E[G_m] + E[G_{n-m+1}]$.
 - De forma parecida a $E[H_m]$, $E[G_m]$ es el número esperado de éxitos obtenidos buscando de izquierda a derecha en una permutación aleatoria de $\{1, 2, \dots, m\}$ elementos k tales que:
 - k aparece a la derecha de m
 - k es mayor que todos los elementos de $\{1, 2, \dots, m-1\}$ que aparecen a la izquierda de k y a la izquierda o a la derecha de m
 - O lo que es lo mismo:
 - $E[G_m]$ es el número esperado de éxitos obtenidos buscando de izquierda a derecha en una permutación aleatoria de $\{1, 2, \dots, m-1\}$ elementos k mayores que todos los que están a su izquierda, después se coloca aleatoriamente m en la lista y se descuentan los k que quedan a la izquierda de m



Árboles aleatorizados

– Cálculo de $E[G_m]$:

- Igual que en el caso de $E[H_m]$, el número esperado de éxitos sin contar el elemento 1 coincide con $E[G_{m-1}]$.
- La probabilidad de que haya éxito con el 1 es $1/(m(m-1))$, porque el 1 se cuenta sólo si aparece m en el extremo izquierdo, seguido inmediatamente del 1.
- Por tanto:

$$E[G_m] = E[G_{m-1}] + \frac{1}{m(m-1)}$$

- Y además: $E[G_1] = 0$
- La solución de esta recurrencia es:

$$E[G_m] = \frac{m-1}{m} < 1$$

- Y como el número medio de rotaciones en un borrado es $E[G_m] + E[G_{n-m+1}]$, se tiene que como máximo es 2.



Análisis amortizado



Análisis amortizado

- El plan:
 - **Conceptos básicos:**
 - Método agregado
 - Método contable
 - Método potencial
 - Primer ejemplo: análisis de tablas *hash* dinámicas
 - Montículos agregables (binomiales y de Fibonacci)
 - Estructuras de conjuntos disjuntos
 - Listas lineales auto-organizativas
 - Árboles auto-organizativos ("splay trees")



-
- “Amortización” (wikipedia):

Bla bla bla... %& \$~€ €¬&@#... bla bla... %\$€~€¬& %€.. bla bla... \$~€ €
%\$€.. bla bla... ~€¬& #~%¬ /%#€¬)6& &@# ... Se trata de técnicas aritméticas
para repartir un importe determinado, el valor a amortizar, en varias cuotas,
correspondientes a varios periodos. \$~€ €¬&@#... bla bla bla... %& \$~€
€¬&@#... bla bla... %\$€~€¬& %€.. bla bla... \$~€ €%\$€~€¬& ¬ &@#... %&
\$~€ €¬&@#... bla bla... %\$€~€¬& %€.. bla bla... \$~€ €%\$€~€¬& ¬ &@# ...



Conceptos básicos

- **Análisis amortizado**
 - Cálculo del coste medio de una operación obtenido dividiendo el coste en el caso peor de la ejecución de una secuencia de operaciones (no necesariamente de igual tipo) dividido por el número de operaciones
- **Utilidad:**
 - Es posible que el coste en el caso peor de la ejecución aislada de una operación sea muy alto y sin embargo si se considera una secuencia de operaciones el coste promedio disminuya
- **Nota:**
 - No es un análisis del caso promedio tal y como el que hemos visto en el tema anterior (la probabilidad ahora no interviene)



Conceptos básicos

- En realidad el coste amortizado de una operación es un “artificio contable” que no tiene ninguna relación con el coste real de la operación.
 - El coste amortizado de una operación puede definirse como **cualquier cosa** con la única condición de que considerando una secuencia de n operaciones:

$$\sum_{i=1}^n A(i) \geq \sum_{i=1}^n C(i)$$

donde $A(i)$ y $C(i)$ son el coste amortizado y el coste exacto, respectivamente, de la operación i -ésima de la secuencia.



Método agregado

- Consiste en calcular el coste en el caso peor $T(n)$ de una secuencia de n operaciones, no necesariamente del mismo tipo, y calcular el coste medio o *coste amortizado* de una operación como $T(n)/n$.
- Los otros dos métodos que veremos (contable y potencial) calculan un coste amortizado específico para cada tipo de operación.



Método agregado

- Ejemplo: pila con operación de *multiDesapilar*
 - Considerar una pila representada mediante una lista de registros encadenados con punteros y con las operaciones de *creaVacía*, *apilar*, *desapilar* y *esVacía*.
 - El coste de todas esas operaciones es $\Theta(1)$ y, por tanto, el coste de una secuencia de n operaciones de *apilar* y *desapilar* es $\Theta(n)$.
 - Añadimos la operación *multiDesapilar*(p, k), que elimina los k elementos superiores de la pila p , si los hay, o deja la pila vacía si no hay tantos elementos.

```
algoritmo multiDesapilar( $p, k$ )  
principio  
    mq not esVacía( $p$ ) and  $k \neq 0$  hacer  
        desapilar( $p$ );  $k := k - 1$   
    fmq  
fin
```



Método agregado

- El coste de *multiDesapilar* es, obviamente, $\Theta(\min(h,k))$, si h es la altura de la pila antes de la operación.
- ¿Cuál es el coste de una secuencia de n operaciones de *apilar*, *desapilar* o *multiDesapilar*?
 - La altura máxima de la pila puede ser de orden n , así que el coste máximo de una operación de *multiDesapilar* en esa secuencia puede ser $O(n)$.
 - Por tanto, el coste máximo de una secuencia de n operaciones está acotado por $O(n^2)$.
 - Esta cálculo es correcto, pero la cota $O(n^2)$, obtenida *considerando el caso peor de cada operación de la secuencia*, no es ajustada.
 - El método agregado considera el caso peor de la secuencia de forma conjunta...



Método agregado

- Análisis agregado de la secuencia de operaciones:
 - Cada elemento puede ser desapilado como máximo una sola vez en toda la secuencia de operaciones.
 - Por tanto, el máximo número de veces que la operación *desapilar* puede ser ejecutada en una secuencia de n operaciones (incluyendo las llamadas en *multiDesapilar*) es igual al máximo número de veces que se puede ejecutar la operación *apilar*, que es n .
 - Es decir, el coste total de cualquier secuencia de n operaciones de *apilar*, *desapilar* o *multiDesapilar* es $O(n)$.
 - Por tanto, el *coste amortizado* de cada operación es la media: $O(n)/n = O(1)$.



Método agregado

- Otro ejemplo: incrementar un contador binario
 - Considerar un contador binario de k bits, almacenado en un vector $A[0..k-1]$ de bits.
 - La cifra menos significativa se guarda en $A[0]$, por tanto, el número almacenado en el contador es:

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

- Inicialmente, $x = 0$, es decir, $A[i] = 0$, para todo i .
 - Para añadir 1 (módulo 2^k) al contador se ejecuta el algoritmo *incrementar...*



Método agregado

```
algoritmo incrementar(A)
principio
  i:=0;
  mq i<length(A) and A[i]=1 hacer
    A[i]:=0;
    i:=i+1
  fmq;
  si i<length(A) entonces
    A[i]:=1
  fsi
fin
```

- Es, esencialmente, el algoritmo implementado en hardware en un sumador “ripple-carry”.



Método agregado

- Una secuencia de 16 incrementos desde $x = 0$:

x	$A[7]$...				$A[0]$				coste acumulado	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	1	1	1	1	1	en relieve: los bits
2	0	0	0	0	0	0	0	1	0	0	0	0	3	que cambian para
3	0	0	0	0	0	0	0	1	1	1	1	1	4	pasar al siguiente
4	0	0	0	0	0	0	1	0	0	0	0	0	7	valor del contador
5	0	0	0	0	0	0	1	0	1	1	1	1	8	
6	0	0	0	0	0	0	1	1	0	0	0	0	10	
7	0	0	0	0	0	0	1	1	1	1	1	1	11	el coste de cada
8	0	0	0	0	1	0	0	0	0	0	0	0	15	incremento es lineal en
9	0	0	0	0	1	0	0	0	1	1	1	1	16	el número de bits que
10	0	0	0	0	1	0	1	0	0	0	0	0	18	cambian de valor
11	0	0	0	0	1	0	1	1	1	1	1	1	19	
12	0	0	0	0	1	1	0	0	0	0	0	0	22	nótese que el coste
13	0	0	0	0	1	1	0	1	1	1	1	1	23	acumulado nunca es
14	0	0	0	0	1	1	1	0	0	0	0	0	25	mayor del doble del
15	0	0	0	0	1	1	1	1	1	1	1	1	26	número de incrementos
16	0	0	0	1	0	0	0	0	0	0	0	0	31	



Método agregado

- Primera aproximación al análisis:
 - Una ejecución de *incrementar* tiene un coste $\Theta(k)$ en el peor caso (cuando el vector contiene todo 1's).
 - Por tanto, una secuencia de n incrementos empezando desde 0 tiene un coste $O(nk)$ en el caso peor.
 - Este análisis es correcto pero poco ajustado.
- Análisis agregado de la secuencia de incrementos:
 - $A[0]$ cambia de valor en cada incremento
 - $A[1]$ cambia de valor $\lfloor n/2 \rfloor$ veces en una secuencia de n incrementos
 - $A[2]$ cambia $\lfloor n/4 \rfloor$ veces en la misma secuencia
 - En general, $A[i]$ cambia $\lfloor n/2^i \rfloor$ veces, $i = 0, 1, \dots, \lfloor \log n \rfloor$



Método agregado

- Por tanto, el número total de cambios de bit en toda la secuencia es:

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

- Es decir, el coste total de toda la secuencia es en el peor caso $O(n)$ y, por tanto, el coste amortizado de cada operación es $O(n)/n = O(1)$.



Método contable

- El coste amortizado **de cada operación** es un *precio* que se le asigna y puede ser mayor o menor que el coste real de la operación.
- Cuando el precio de una operación excede su coste real, el *crédito* resultante se puede usar después para pagar operaciones cuyo precio sea menor que su coste real.
- Puede definirse una *función de potencial*, $P(i)$, para cada operación de la secuencia:

$$P(i) = A(i) - C(i) + P(i - 1), \quad i = 1, 2, \dots, n$$

donde $A(i)$ y $C(i)$ son el coste amortizado y el coste exacto, respectivamente, de la operación i -ésima.

- El potencial en cada operación es el crédito disponible para el resto de la secuencia.




Método contable

- Sumando el potencial de todas las operaciones:

$$\begin{aligned}\sum_{i=1}^n P(i) &= \sum_{i=1}^n (A(i) - C(i) + P(i-1)) \\ \Rightarrow \sum_{i=1}^n (P(i) - P(i-1)) &= \sum_{i=1}^n (A(i) - C(i)) \\ \Rightarrow P(n) - P(0) &= \sum_{i=1}^n (A(i) - C(i)) \Rightarrow \underline{P(n) - P(0) \geq 0}\end{aligned}$$

(por definición de coste amortizado)

$$\sum_{i=1}^n A(i) \geq \sum_{i=1}^n C(i)$$


- Debe asignarse un precio a cada operación que haga que el crédito disponible sea siempre no negativo.



Método contable

- Volvamos al ejemplo de la pila con la operación de *multiDesapilar*
 - Recordar el coste real:
 - $C(\text{apilar}) = 1$
 - $C(\text{desapilar}) = 1$
 - $C(\text{multiDesapilar}) = \Theta(\min(h, k))$
 - Asignamos (arbitrariamente) el coste amortizado (*precio*) de cada operación como:
 - $A(\text{apilar}) = 2$
 - $A(\text{desapilar}) = 0$
 - $A(\text{multiDesapilar}) = 0$
 - Para ver si el coste amortizado es correcto hay que demostrar que el crédito es siempre no negativo.



Método contable

- Es decir, hay que ver si $P(n) - P(0) \geq 0, \forall n$.
- Al apilar cada elemento, con precio 2, pagamos el coste real de una unidad por la operación de apilar y nos sobra otra unidad como crédito.
 - En cada instante de tiempo tenemos una unidad de crédito por cada elemento de la pila.
 - Es el *pre-pago* para cuando haya que desapilarlo.
- Al desapilar, el precio de la operación es cero y el coste real se paga con el crédito asociado al elemento desapilado.
- De esta forma, pagando un poco más por *apilar* (2 en lugar de 1) no hemos necesitado pagar por *desapilar* ni por *multiDesapilar*.



Método contable

- El otro ejemplo: el contador binario
 - Definimos como 2 unidades el coste amortizado (precio) de la operación de poner un bit a 1.
 - Cuando un bit se pone a 1, se paga su coste de una unidad y la unidad restante queda como crédito para operaciones futuras.
 - Así, cada bit que vale 1 guarda asociado un crédito de una unidad, y ese crédito se usa para pagar la operación de ponerlo a 0, con lo que el coste amortizado de esta operación se puede dejar como 0.



Método contable

- Veamos ahora el coste amortizado de *incrementar*:

```
algoritmo incrementar(A)
principio
  i:=0;
  mq i<length(A) and A[i]=1 hacer
    A[i]:=0;
    i:=i+1
  fmq;
  si i<length(A) entonces
    A[i]:=1
  fsi
fin
```

coste amortizado nulo

coste amortizado menor o igual que 2

Método potencial

- Consideramos, de nuevo, una función de potencial:

$$P(i) = A(i) - C(i) + P(i - 1), \quad i = 1, 2, \dots, n \quad (*)$$

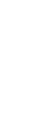
tal que:

$$P(n) - P(0) \geq 0, \quad \forall n$$

- En este método de análisis, se parte de una función de potencial “dada” (que hay que elegir) y usando la ecuación (*) se calcula $A(i)$:

$$A(i) = C(i) + P(i) - P(i - 1), \quad i = 1, 2, \dots, n$$

- El gran problema: ¿cómo elegir el potencial?



Método potencial

- Una vez más: la pila con *multiDesapilar*
 - La función de potencial puede interpretarse a menudo como una “energía potencial” asociada a la estructura de datos del problema que puede utilizarse para pagar el coste de las operaciones futuras.
 - Ejemplo: definir el potencial de la pila como su altura.
 - Se tiene: $P(0) = 0$ (pila vacía), y $P(n) \geq 0, \forall n$.
 - Por tanto la función de potencial es válida y por eso el coste amortizado total de las n operaciones es una cota superior del coste total exacto de todas ellas.



Método potencial

- Cálculo del coste amortizado a partir del potencial:
 - Suponer que la operación i -ésima es *apilar* y se realiza sobre una pila de altura h :

$$A(i) = C(i) + P(i) - P(i - 1) = 1 + (h + 1) - h = 2$$

- Suponer que la operación i -ésima es *desapilar* y se realiza sobre una pila de altura h :

$$A(i) = C(i) + P(i) - P(i - 1) = 1 + (h - 1) - h = 0$$

- Suponer que la operación i -ésima es *multiDesapilar* k elementos y se realiza sobre una pila de altura h , entonces se desapilan $k' = \min(k, h)$ elementos, y:

$$A(i) = C(i) + P(i) - P(i - 1) = k' - k' = 0$$

- El coste amortizado en los tres casos es $O(1)$, por tanto el coste amortizado total de toda la secuencia es $O(n)$.



Método potencial

- Y por último: el ejemplo del contador binario
 - Elección de la función de potencial:
 - $P(i) = b_i$, el número de 1's en el contador después del i -ésimo incremento.
 - Se tiene: $P(0) = 0$ (el contador se inicializa a 0), $P(n) \geq 0 \forall n$.
 - Cálculo del coste amortizado:

- Suponer que la i -ésima operación (incremento) pone a 0 t_i bits

- Entonces, su coste es como mucho $t_i + 1$

- $b_i \leq b_{i-1} - t_i + 1$

```
algoritmo incrementar(A)
principio
    i:=0;
    mq i<length(A) and A[i]=1 hacer
        A[i]:=0;
        i:=i+1
    fsmq;
    si i<length(A) entonces
        A[i]:=1
    fsi
fin
```



Método potencial

- Cálculo del coste amortizado (cont.):
 - $A(i) = C(i) + P(i) - P(i-1) \leq (t_i + 1) + (b_{i-1} - t_i + 1) - b_{i-1} = 2$
 - Por tanto, el coste amortizado total de n incrementos consecutivos empezando con el contador a 0 es $O(n)$.
- El método potencial permite también calcular el coste si el contador empieza en un valor no nulo:
 - Si inicialmente hay b_0 bits a 1 y tras n incrementos hay b_n 1's.
 - $A(i) \leq 2$, al igual que antes.
 - $A(i) = C(i) + P(i) - P(i-1)$, $i = 1, 2, \dots, n \Rightarrow$

$$\begin{aligned}\sum_{i=1}^n C(i) &= \sum_{i=1}^n A(i) - P(n) + P(0) \leq \sum_{i=1}^n 2 - b_n + b_0 \\ &= 2n - b_n + b_0\end{aligned}$$

- Como $b_0 \leq k$ (número de bits del contador), si se ejecutan al menos $n = \Omega(k)$ incrementos, el coste real total es $O(n)$.



Análisis amortizado

- El plan:
 - Conceptos básicos:
 - Método agregado
 - Método contable
 - Método potencial
 - **Primer ejemplo: análisis de tablas *hash* dinámicas**
 - Montículos agregables (binomiales y de Fibonacci)
 - Estructuras de conjuntos disjuntos
 - Listas lineales auto-organizativas
 - Árboles auto-organizativos ("splay trees")



Primer ejemplo: análisis de tablas dinámicas

- Considerar una tabla *hash* con operaciones de creación, búsqueda e inserción (y más adelante borrado) y con espacio limitado.
- Si la tabla está llena, la inserción obliga a *expandirla* a otra con, por ejemplo, el doble de capacidad y copiar todos los datos a la nueva.
- Se trata de analizar el coste de las inserciones, teniendo en cuenta que pueden traer consigo la expansión de la tabla.



Primer ejemplo: análisis de tablas dinámicas

```
algoritmo insertar(T,x)
principio
  si T.capacidad=0 entonces
    crearTabla(T,1);
    T.capacidad:=1
  fsi;
  si T.numdatos=T.capacidad ent
    crearTabla(nuevaT,2*T.capacidad);
    insertarTabla(T,nuevaT);
    liberar(T);
    T:=nuevaT;
    T.capacidad:=2*T.capacidad;
  fsi;
  insertarDato(T,x);
  T.numdatos:=numdatos+1
fin
```

capacidad máxima

crear con capacidad 1

número de datos
actualmente almacenados

expansión al doble
de capacidad

volcar la tabla vieja
a la nueva

Definimos la “unidad de coste” como el coste de esta operación
de *inserción elemental*



Primer ejemplo: análisis de tablas dinámicas

- Analicemos la secuencia de n inserciones en una tabla inicialmente vacía.
 - Coste de la operación i -ésima: $C(i)$
 - Si hay espacio: $C(i) = 1$
 - Si hay que expandir: $C(i) = i$
 - Si se consideran n inserciones el coste peor de una inserción es $O(n)$ y, por tanto, el coste peor para toda la secuencia está acotado por $O(n^2)$.
 - Este cálculo es correcto pero muy poco ajustado.



Primer ejemplo: análisis de tablas dinámicas

- Análisis mediante el método agregado:
 - $C(i) = i$, si $i - 1$ es una potencia exacta de 2,
 $C(i) = 1$, en otro caso.
 - Por tanto:

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

- Es decir, el coste amortizado de cada inserción es 3.



Primer ejemplo: análisis de tablas dinámicas

- Análisis mediante el método contable:
 - Nos aporta la intuición de por qué el coste amortizado de una inserción es 3.
 - Cada elemento insertado paga por tres inserciones elementales:
 - Suponer que tenemos una tabla de capacidad m justo tras una expansión, y sin ningún crédito disponible.
 - Entonces el número de elementos en la tabla es $m/2$.
 - Por cada inserción posterior pagamos 3 unidades
 - Una de ellas es el coste de la inserción elemental del dato.
 - Otra queda como crédito, asociada al dato insertado.
 - La otra queda como crédito de uno de los restantes $m/2$ datos que ya estaban en la tabla y que no tenían crédito.
 - Tras $m/2$ inserciones, la tabla está llena, con m datos, y cada dato tiene crédito de 1 unidad.
 - Con ese crédito se hace la expansión gratis.



Primer ejemplo: análisis de tablas dinámicas

- Análisis mediante el método potencial:
 - Primero: definir la función de potencial
 - $P(i)$ = potencial de la tabla T tras la i -ésima inserción
 - Que valga 0 tras cada expansión.
 - Que haya aumentado hasta igualar la capacidad de la tabla cuando ésta esté llena (para que la siguiente expansión pueda pagarse con el potencial).
 - Por ejemplo: $P(i) = 2 \cdot \text{numdatos}(i) - \text{capacidad}(i)$
 - Tras una expansión, $\text{numdatos}(i) = \text{capacidad}(i)/2$, luego $P(i) = 0$.
 - Inmediatamente antes de una expansión, $\text{numdatos}(i) = \text{capacidad}(i)$, luego $P(i) = \text{numdatos}(i)$.
 - El valor inicial es 0, y como la tabla siempre está medio llena, $\text{numdatos}(i) \geq \text{capacidad}(i)/2$, luego P es siempre no negativo.



Primer ejemplo: análisis de tablas dinámicas

– Segundo: calcular el coste amortizado

- Inicialmente: $numdatos(i) = 0$, $capacidad(i) = 0$, $P(i) = 0$.
- Si la i -ésima inserción no genera expansión, $capacidad(i) = capacidad(i-1)$ y:

$$\begin{aligned} A(i) &= C(i) + P(i) - P(i-1) \\ &= 1 + (2 \cdot numdatos(i) - capacidad(i)) - \\ &\quad - (2 \cdot numdatos(i-1) - capacidad(i-1)) \\ &= 1 + (2 \cdot numdatos(i) - capacidad(i)) - \\ &\quad - (2(numdatos(i) - 1) - capacidad(i)) \\ &= 3 \end{aligned}$$

- Si la i -ésima inserción genera expansión, $capacidad(i)/2 = capacidad(i-1) = numdatos(i-1) = numdatos(i) - 1$, y:

$$\begin{aligned} A(i) &= C(i) + P(i) - P(i-1) \\ &= numdatos(i) + (2 \cdot numdatos(i) - (2 \cdot numdatos(i) - 2)) - \\ &\quad - (2(numdatos(i) - 1) - numdatos(i) - 1)) \\ &= 3 \end{aligned}$$



Primer ejemplo: análisis de tablas dinámicas

- Tabla con operación de borrado:
 - Si al borrar un elemento la tabla queda “muy vacía”, se puede *contraer* la tabla.
 - Primera estrategia posible: “muy vacía” = la tabla tiene menos de la mitad de posiciones ocupadas.
 - Se garantiza que el factor de carga sea siempre superior a $\frac{1}{2}$.
 - El coste amortizado puede ser demasiado grande. Ejemplo:
 - Hacemos n operaciones (con n una potencia de 2).
 - Las primeras $n/2$ son inserciones, con un coste amortizado total de $O(n)$.
 - Al acabar las inserciones, $numdatos(n) = capacidad(n) = n/2$.
 - Las siguientes $n/2$ operaciones son: I, B, B, I, I, B, B, I, I, ... (con I = inserción y B = borrado).



Primer ejemplo: análisis de tablas dinámicas

- La primera inserción provoca una expansión a tamaño n .
- Los dos borrados siguientes provocan una contracción a tamaño $n/2$.
- Las dos inserciones siguientes provocan una nueva expansión a tamaño n , y así sucesivamente.
- De esta forma, el coste de toda la secuencia es $\Theta(n^2)$, y por tanto el coste amortizado de cada operación es $\Theta(n)$.
- El problema de esa estrategia: después de una expansión no se realizan suficientes borrados para justificar el pago de una contracción, y viceversa, después de una contracción no se hacen suficientes inserciones para pagar una expansión.



Primer ejemplo: análisis de tablas dinámicas

- Nueva estrategia (para los borrados):
 - Duplicar la capacidad de la tabla cuando hay que insertar en una tabla llena, pero **contraer la tabla a la mitad cuando un borrado hace que quede llena en menos de $\frac{1}{4}$ de su capacidad.**
 - De esta forma, tras una expansión el factor de carga es $\frac{1}{2}$ y por tanto la mitad de los elementos deben ser borrados para que ocurra una contracción.
 - Tras una contracción el factor de carga es $\frac{1}{2}$ y, por tanto, el número de elementos de la tabla debe duplicarse hasta provocar una expansión.



Primer ejemplo: análisis de tablas dinámicas

- Análisis de n operaciones de inserción y/o borrado mediante el método potencial:
 - Primero: definir el potencial, función P , que
 - sea 0 justo tras una expansión o contracción y
 - crezca mientras el factor de carga, $\alpha(i) = \text{numdatos}(i)/\text{capacidad}(i)$, crece hacia 1 o disminuye hacia $1/4$.
 - Como en una tabla vacía $\text{numdatos} = \text{capacidad} = 0$ y $\alpha = 1$, siempre se tiene que $\text{numdatos}(i) = \alpha(i) \cdot \text{capacidad}(i)$.
 - Por ejemplo:

$$P(i) = \begin{cases} 2\text{numdatos}(i) - \text{capacidad}(i), & \text{si } \alpha(i) \geq 1/2 \\ \text{capacidad}(i)/2 - \text{numdatos}(i), & \text{si } \alpha(i) < 1/2 \end{cases}$$

así, el potencial nunca es negativo y el de la tabla vacía es 0.



Primer ejemplo: análisis de tablas dinámicas

- Propiedades de esta función

$$P(i) = \begin{cases} 2numdatos(i) - capacidad(i), & \text{si } \alpha(i) \geq 1/2 \\ capacidad(i)/2 - numdatos(i), & \text{si } \alpha(i) < 1/2 \end{cases}$$

- Cuando $\alpha(i) = 1/2$, el potencial es 0.
- Cuando $\alpha(i) = 1$, se tiene que $numdatos(i) = capacidad(i)$, y por tanto $P(i) = numdatos(i)$, es decir, el potencial permite pagar por una expansión si se inserta un nuevo dato.
- Cuando $\alpha(i) = 1/4$, $capacidad(i) = 4 \cdot numdatos(i)$, y por tanto $P(i) = numdatos(i)$, es decir, el potencial permite pagar por una contracción si se borra un dato.



Primer ejemplo: análisis de tablas dinámicas

- Segundo: cálculo del coste amortizado
 - Inicialmente, $numdatos(0)=capacidad(0)=P(0)=0$ y $\alpha(0)=1$.
 - Si la i -ésima operación es una inserción:
 - Si $\alpha(i-1) \geq 1/2$, el análisis es idéntico al caso anterior (con sólo operaciones de inserción), y el coste amortizado de la operación es menor o igual a 3.
 - Si $\alpha(i-1) < 1/2$, la tabla no precisa expandirse, y hay dos casos:

» Si $\alpha(i) < 1/2$:

$$\begin{aligned} A(i) &= C(i) + P(i) - P(i-1) \\ &= 1 + (capacidad(i)/2 - numdatos(i)) - \\ &\quad - (capacidad(i-1)/2 - numdatos(i-1)) \\ &= 1 + (capacidad(i)/2 - numdatos(i)) - \\ &\quad - (capacidad(i)/2 - numdatos(i)-1) \\ &= 0 \end{aligned}$$

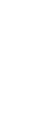


Primer ejemplo: análisis de tablas dinámicas

» Si $\alpha(i-1) < 1/2$ pero $\alpha(i) \geq 1/2$:

$$\begin{aligned} A(i) &= C(i) + P(i) - P(i-1) \\ &= 1 + (2 \cdot \text{numdatos}(i) - \text{capacidad}(i)) - \\ &\quad - (\text{capacidad}(i-1)/2 - \text{numdatos}(i-1)) \\ &= 1 + (2(\text{numdatos}(i-1) + 1) - \text{capacidad}(i-1)) - \\ &\quad - (\text{capacidad}(i-1)/2 - \text{numdatos}(i-1)) \\ &= 3 \cdot \text{numdatos}(i-1) - 3/2 \cdot \text{capacidad}(i-1) + 3 \\ &= 3 \cdot \alpha(i-1) \text{capacidad}(i-1) - 3/2 \cdot \text{capacidad}(i-1) + 3 \\ &< 3/2 \cdot \text{capacidad}(i-1) - 3/2 \cdot \text{capacidad}(i-1) + 3 \\ &= 3 \end{aligned}$$

- Por tanto, el coste amortizado de una inserción es como mucho 3.



Primer ejemplo: análisis de tablas dinámicas

- Si la i -ésima operación es un borrado:
 - En este caso $numdatos(i) = numdatos(i-1) - 1$
 - Si $\alpha(i-1) < 1/2$ y la operación no provoca una contracción, entonces $capacidad(i) = capacidad(i-1)$ y el coste es:

$$\begin{aligned} A(i) &= C(i) + P(i) - P(i-1) \\ &= 1 + (capacidad(i)/2 - numdatos(i)) - \\ &\quad - (capacidad(i-1)/2 - numdatos(i-1)) \\ &= 1 + (capacidad(i)/2 - numdatos(i)) - \\ &\quad - (capacidad(i)/2 - (numdatos(i) + 1)) \\ &= 2 \end{aligned}$$



Primer ejemplo: análisis de tablas dinámicas

- Si $\alpha(i-1) < 1/2$ y la operación provoca una contracción:

$C(i) = \text{numdatos}(i) + 1$, porque se borra un dato y se mueven $\text{numdatos}(i)$

$$\text{capacidad}(i)/2 = \text{capacidad}(i-1)/4 = \text{numdatos}(i) + 1$$

$$\begin{aligned} A(i) &= C(i) + P(i) - P(i-1) \\ &= (\text{numdatos}(i) + 1) + (\text{capacidad}(i)/2 - \text{numdatos}(i)) - \\ &\quad - (\text{capacidad}(i-1)/2 - \text{numdatos}(i-1)) \\ &= (\text{numdatos}(i) + 1) + ((\text{numdatos}(i) + 1) - \text{numdatos}(i)) - \\ &\quad - ((2 \cdot \text{numdatos}(i) + 2) - (\text{numdatos}(i) + 1)) \\ &= 1 \end{aligned}$$

- Si $\alpha(i-1) \geq 1/2$ el coste amortizado también se puede acotar con una constante (**ejercicio**).

- En resumen, el coste total de las n operaciones es $O(n)$.



Análisis amortizado

- El plan:
 - Conceptos básicos:
 - Método agregado
 - Método contable
 - Método potencial
 - Primer ejemplo: análisis de tablas *hash* dinámicas
 - **Montículos agregables (binomiales y de Fibonacci)**
 - Estructuras de conjuntos disjuntos
 - Listas lineales auto-organizativas
 - Árboles auto-organizativos ("splay trees")



Montículos agregables (binomiales y de Fibonacci)

- Montículo agregable:
 - TAD que soporta las operaciones de:
 - Crear vacío
 - Insertar dato (con una clave)
 - Consultar el dato de clave mínima
 - Borrar el dato de clave mínima
 - Unión de dos estructuras

montículo (= cola con prioridades)

→ “montículo agregable”
 - Además permitirán:
 - Modificar la clave de un dato por otra menor
 - Borrar un dato dada su clave



Montículos agregables (binomiales y de Fibonacci)

- Implementaciones:
 - Montículo binario (el que ya conocemos)
 - Montículo binomial (veremos el coste en el caso peor)
 - Montículo de Fibonacci (veremos el coste amortizado)

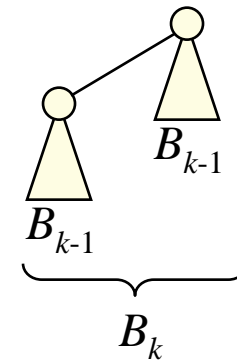
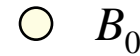
operación	mont. binario (caso peor)	mont. binomial (caso peor)	mont. Fibonacci (amortizado)
crear vacío	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insertar	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
mínimo	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
borrar mínimo	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
unión	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
{ reducir clave borrar	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

→ manteniendo un puntero a cada nodo



Montículos agregables (binomiales y de Fibonacci)

- Montículos binomiales
 - Un montículo binomial es una colección de árboles binomiales
 - Árbol binomial, B_k , se define recursivamente:
 - B_0 es un solo nodo
 - B_k consiste en dos árboles binomiales B_{k-1} *enlazados* de la siguiente forma: la raíz de uno es el hijo más a la izquierda de la raíz del otro.



Montículos binomiales

- Propiedades del árbol binomial B_k :
 - tiene 2^k nodos;
 - su altura es k ;
 - tiene $\binom{k}{i}$ nodos en el nivel i , para $i = 0, 1, \dots, k$;
 - la raíz tiene *grado* k (número de hijos) y es el nodo de máximo grado; más aún, si se numeran los hijos de la raíz de izquierda a derecha como $k - 1, k - 2, \dots, 0$, el hijo i es la raíz de un subárbol B_i .



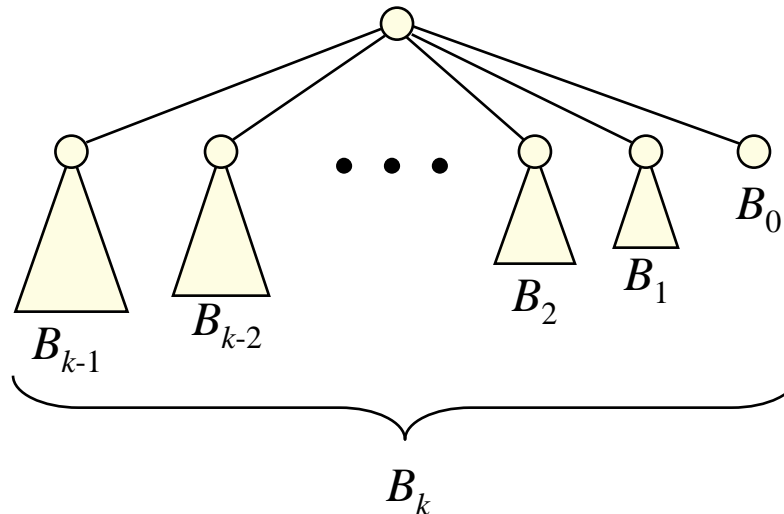
Montículos binomiales

- Demostración de las propiedades, por inducción:
 - Se cumplen para B_0 .
 - Suponer que se cumplen para B_{k-1} :
 - B_k consiste en dos copias de B_{k-1} , luego B_k tiene $2^{k-1} + 2^{k-1} = 2^k$ nodos.
 - Por construcción de B_k , su altura incrementa en 1 la de B_{k-1} .
 - Sea $D(k,i)$ el n° de nodos de B_k en el nivel i . Por construcción de B_k , $D(k,i) = D(k-1,i) + D(k-1,i-1)$
$$= \binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$$
 - El único nodo con grado mayor en B_k que en B_{k-1} es la raíz, que tiene un hijo más que en B_{k-1} . Como la raíz de B_{k-1} tiene grado $k-1$, la de B_k tiene grado k .



Montículos binomiales

- Por último, por hipótesis de inducción los hijos de la raíz de B_{k-1} son, de izquierda a derecha, las raíces de B_{k-2} , B_{k-3} , ..., B_0 . Por tanto, cuando se enlaza B_{k-1} a B_{k-1} , los hijos de la raíz resultante son las raíces de B_{k-1} , B_{k-2} , ..., B_0 .



- Corolario: el grado máximo de cualquier nodo en un árbol binomial de n nodos es $\log n$.
 - Demostración se sigue de las propiedades 1ª y 4ª.

Montículos binomiales

- Montículo binomial:
 - Es un conjunto de árboles binomiales tales que:
 - Cada árbol binomial es un *árbol parcialmente ordenado*, es decir, la clave de todo nodo es mayor o igual que la de su padre.
 - Contiene no más de un árbol binomial B_i para cada grado i .
- Propiedad (consecuencia de la definición):
 - Todo montículo binomial M de n nodos consta de, como mucho, $\lfloor \log n \rfloor + 1$ árboles binomiales.
 - Demostración: la representación binaria de n tiene $\lfloor \log n \rfloor + 1$ bits, $\langle b_{\lfloor \log n \rfloor}, b_{\lfloor \log n \rfloor - 1}, \dots, b_0 \rangle$, de forma que

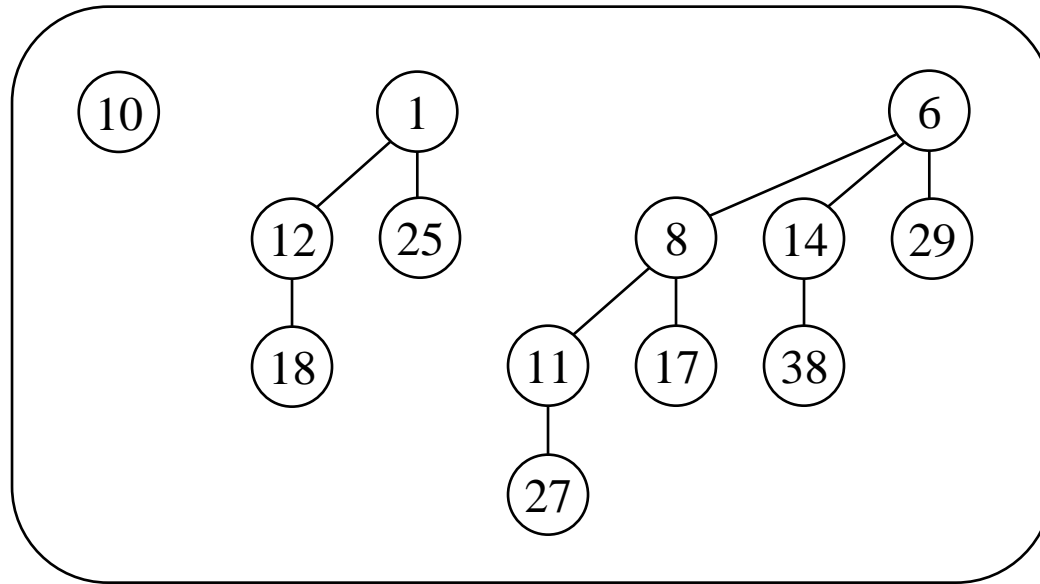
$$n = \sum_{i=0}^{\lfloor \log n \rfloor} b_i 2^i$$

Como B_i tiene 2^i nodos, B_i aparece en M si y sólo si $b_i = 1$.



Montículos binomiales

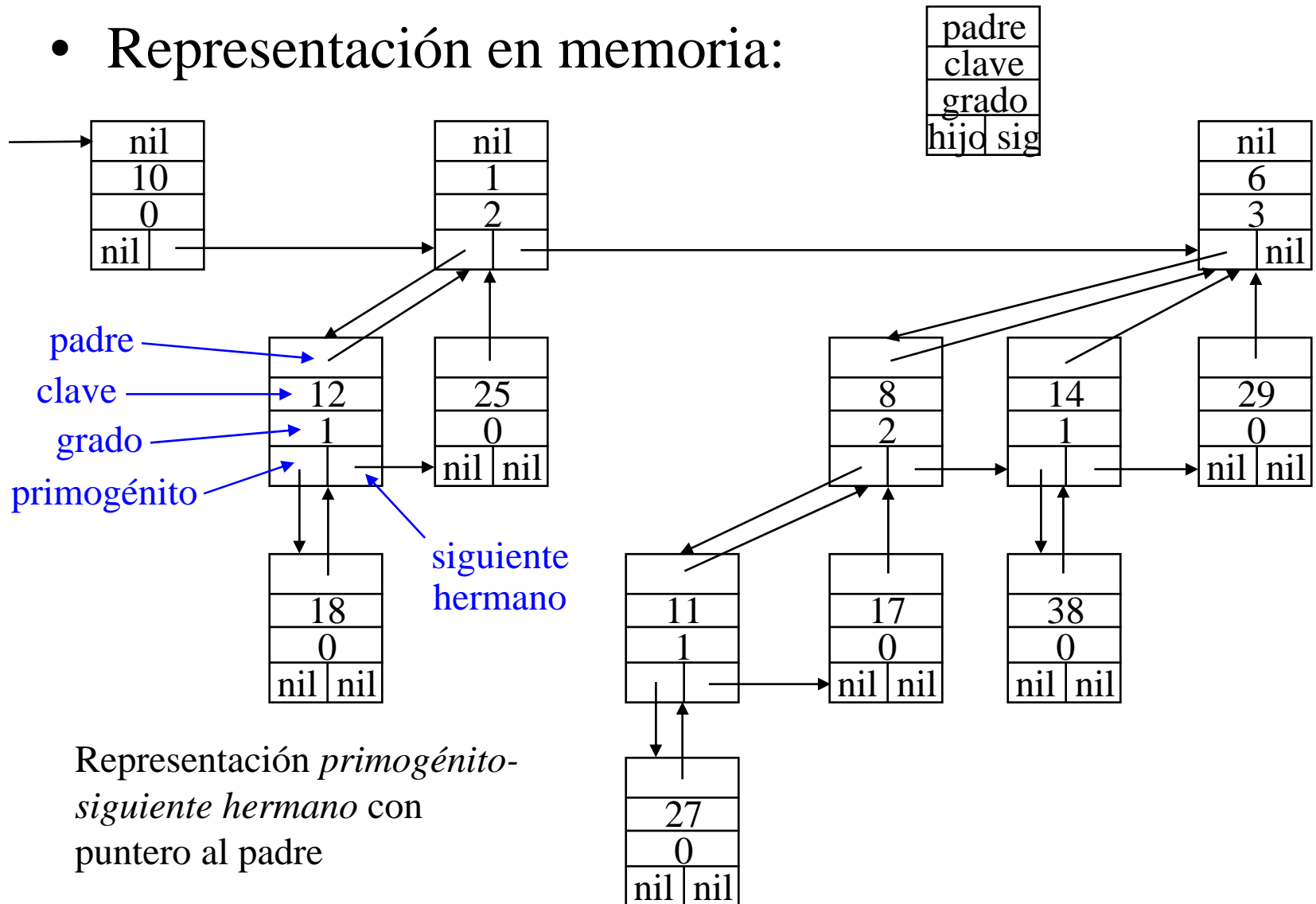
- Un ejemplo de montículo binomial de 13 nodos:



- La representación binaria de 13 es $\langle 1,1,0,1 \rangle$, por tanto M contiene los árboles binomiales B_3 , B_2 y B_0 , con 8, 4 y 1 nodos, respectivamente.

Montículos binomiales

- Representación en memoria:



Montículos binomiales

- Operaciones con montículos binomiales:
 - La creación es simplemente asignar el valor NIL a un puntero, así que tiene coste $\Theta(1)$.
 - Encontrar el elemento de clave mínima:

- Es decir, se elige la raíz con clave mínima.
- El número de árboles es como mucho $\lfloor \log n \rfloor + 1$, así que el coste es $O(\log n)$.

```
función mínimo(M) dev puntero
principio
  y:=nil; x:=M; min:=∞;
  mq x≠nil hacer
    si x↑.clave<min entonces
      min:=x↑.clave; y:=x
    fsi;
    x:=x↑.sig
  fmq;
  devuelve y
fin
```



Montículos binomiales

- Unión de dos montículos binomiales:
 - Una operación auxiliar: enlazar árboles binomiales

```
algoritmo enlazar( $y, z$ )  
{enlaza el árbol  $B_{k-1}$  de raíz  $y$  al árbol  $B_{k-1}$   
  de raíz  $z$ ;  $z$  queda como padre de  $y$ , como  
  raíz de un árbol  $B_k$ }  
principio  
   $y \uparrow$ .padre :=  $z$ ;  
   $y \uparrow$ .sig :=  $z \uparrow$ .hijo;  
   $z \uparrow$ .hijo :=  $y$ ;  
   $z \uparrow$ .grado :=  $z \uparrow$ .grado + 1  
fin
```

- Ahora veamos la unión de montículos; se usa un algoritmo “mezclar” que mezcla dos listas en una, ordenada por grado de los árboles (es el algoritmo habitual de “mezcla ordenada”).



Montículos binomiales

```
algoritmo unión ( $M_1, M_2, M$ )  
principio  
   $M := \text{mezclar}(M_1, M_2)$  ;  
  ...
```

- En primer lugar se mezclan las listas de raíces de árboles M_1 y M_2 , en una lista M , ordenada por grados crecientes.
- Podría haber hasta dos raíces de cada grado (y en tal caso aparecerán consecutivas), por tanto, después habrá que enlazar los árboles de igual grado usando “enlazar”.
- Si las listas M_1 y M_2 tienen m raíces entre las dos, el coste de “mezclar” es $O(m)$:
 - “mezclar” examina la primera raíz de M_1 y de M_2 y añade la de menor grado a M , eliminándola de M_1 o de M_2 , repitiendo el proceso hasta que M_1 y M_2 están vacías.



Montículos binomiales

```
...
si M≠nil entonces
  antx:=nil; x:=M; sigx:=x↑.sig
  mq sigx≠nil hacer
    si x↑.grado≠sigx↑.grado → Caso 1
      or (sigx↑.sig≠nil and
          sigx↑.sig↑.grado=x↑.grado) } → Caso 2
    ent
      antx:=x; x:=sigx → simplemente, se avanza en la lista
                          (el puntero sigx se actualiza luego)
  sino
    ...
```

- **Caso 1:** x es la raíz de un árbol B_k y $sigx$ de un B_l con $l > k$.
- **Caso 2:** x es la primera de las tres raíces consecutivas de igual grado (ya veremos cómo pueden aparecer tres consecutivas).



Montículos binomiales

```
...
sino {x es la 1ª de 2 raíces de igual grado}
  si  $x \uparrow .clave \leq sigx \uparrow .clave$  ent
     $x \uparrow .sig := sigx \uparrow .sig;$ 
    enlazar(sigx, x)
  sino
    si antx=nil ent
      M:=sigx
    sino
       $antx \uparrow .sig := sigx$ 
    fsi;
    enlazar(x, sigx);
    x:=sigx
  fsi
fsi;
sigx:=x $\uparrow$ .sig
fmq
fsi
fin
```

Caso 3

Caso 4

- Casos 3 y 4: se enlazan x y $sigx$.
Se distinguen en si x o $sigx$ tiene la clave menor (lo que determina cuál será la raíz tras enlazarlos).



Montículos binomiales

- En los casos 3 y 4, hemos enlazado 2 árboles B_k para formar uno B_{k+1} , al cual apunta ahora el puntero x .
- Antes había cero, uno o dos árboles B_{k+1} , por tanto x es ahora el primero de uno, dos o tres árboles B_{k+1} .
 - Si es el único, vamos al caso 1 en la siguiente iteración.
 - Si es el primero de dos, vamos al caso 3 ó 4 en la siguiente iteración.
 - Si es el primero de tres, vamos al caso 2 en la siguiente iteración.



Montículos binomiales

- Veamos que el coste de la unión es $O(\log n)$:
 - Suponer que M_1 tiene n_1 nodos y M_2 tiene n_2 nodos, con $n_1 + n_2 = n$.
 - Entonces M_1 tiene como mucho $\lfloor \log n_1 \rfloor + 1$ raíces y M_2 tiene como mucho $\lfloor \log n_2 \rfloor + 1$ raíces.
 - Luego M tiene como mucho $\lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor + 2 \leq 2\lfloor \log n \rfloor + 2 = O(\log n)$ raíces inmediatamente tras la ejecución de “mezclar”.
 - El coste de “mezclar” es, por tanto, $O(\log n)$.
 - Cada iteración del bucle “mq” es $O(1)$ y hay, como mucho, $\lfloor \log n_1 \rfloor + \lfloor \log n_2 \rfloor + 2$ iteraciones porque cada iteración avanza los punteros una posición o elimina un elemento de la lista, por tanto el coste total es $O(\log n)$.

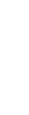


Montículos binomiales

– Inserción de un elemento:

- Crea un montículo binomial M_1 con un solo nodo (x), en tiempo $O(1)$.
- Luego lo une con el montículo M de n nodos en $O(\log n)$.

```
algoritmo insertar( $M, x$ )  
principio  
   $x \uparrow . \text{padre} := \text{nil};$   
   $x \uparrow . \text{hijo} := \text{nil};$   
   $x \uparrow . \text{sig} := \text{nil};$   
   $x \uparrow . \text{grado} := 0;$   
   $M_1 := x;$   
  unión( $M, M_1, M_2$ ) ;  
   $M := M_2$   
fin
```



Montículos binomiales

– Borrado del elemento de clave mínima:

```
algoritmo borrarMínimo(M)
principio
    encontrar la raíz  $x$  de clave mínima en  $M$ 
    y quitar ese árbol de  $M$ ;
    crear un nuevo montículo  $M1$  con los hijos
    de  $x$ , colocados en orden inverso;
    unión( $M, M1, M2$ ) ;
     $M := M2$ 
fin
```

- Si el montículo tiene n nodos, las operaciones del algoritmo toman $O(\log n)$ tiempo.



Montículos binomiales

- Operación de reducción del valor de una clave:
 - En primer lugar: se asume que se puede acceder directamente a la dirección de cualquier dato x .
 - Es decir, es necesario mantener un puntero a cada nodo del montículo.
 - Si el problema requiere realizar esta operación, el requisito anterior no supone un gran inconveniente (ni en espacio ni en tiempo).
 - El método es como en un montículo ordinario:
 - Primero se cambia la clave del dato (y cualquier otra información asociada a la clave, propia del problema).
 - El dato cambiado se compara con su padre y sube si es menor que él, y así hasta alcanzar el lugar correcto (un padre menor que el dato cambiado) o llegar a la raíz.



Montículos binomiales

```
algoritmo reducir(M, x, c)
{pre:  $c < x \uparrow . \text{clave}$ } {post: la clave de x pasa a ser c}
principio
   $x \uparrow . \text{clave} := c;$ 
   $y := x;$ 
   $z := y \uparrow . \text{padre};$ 
  mq  $z \neq \text{nil}$  and  $y \uparrow . \text{clave} < z \uparrow . \text{clave}$  hacer
    intercambiar( $y \uparrow . \text{clave}$  y  $z \uparrow . \text{clave}$  y sus otros
      campos de información adicional si los hay);
     $y := z;$ 
     $z := y \uparrow . \text{padre}$ 
  fmq
fin
```

- Coste de la operación de reducción del valor de una clave:
 - Si el montículo tiene n elementos, la máxima profundidad de x es $\lfloor \log n \rfloor$, luego el “mq” del algoritmo itera, como mucho, $\lfloor \log n \rfloor$ veces.
 - Por tanto la operación está en $O(\log n)$ en tiempo.



Montículos binomiales

- Operación de borrado de un elemento:

```
algoritmo borrar(M,x)
principio
    reducir(M,x,- $\infty$ ) ;
    borrarMínimo(M)
fin
```

- El algoritmo hace decrecer el valor de la clave del elemento a borrar hasta el valor mínimo posible y con el algoritmo “reducir” el dato sube hasta la raíz, en tiempo $O(\log n)$.
- Después, con la operación de borrado del mínimo, de coste $O(\log n)$, se borra esa raíz.
- El coste total es, por tanto, $O(\log n)$.



Análisis amortizado

- El plan:
 - Conceptos básicos:
 - Método agregado
 - Método contable
 - Método potencial
 - Primer ejemplo: análisis de tablas *hash* dinámicas
 - **Montículos agregables (binomiales y de Fibonacci)**
 - Estructuras de conjuntos disjuntos
 - Listas lineales auto-organizativas
 - Árboles auto-organizativos ("splay trees")



Montículos de Fibonacci

- ¿Qué se pretende?

operación	mont. binario (caso peor)	mont. binomial (caso peor)	mont. Fibonacci (amortizado)
crear vacío	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insertar	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
mínimo	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
borrar mínimo	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
unión	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
reducir clave	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
borrar	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

Montículos de Fibonacci

- Situaciones de interés:
 - Problemas en los que las operaciones de borrado del mínimo y de borrado de cualquier elemento son poco frecuentes en proporción con el resto.
 - Ejemplo: muchos algoritmos de grafos en los que se precisan colas con prioridades y con la ejecución frecuente de la operación de reducción de clave.
 - Algoritmo de Dijkstra para el cálculo de caminos mínimos (ver transparencias de *Esquemas Algorítmicos: Algoritmos voraces*, pp. 27-37).
 - Algoritmo de Prim para el cálculo de árboles de recubrimiento de coste mínimo (ver transparencias de *Esquemas Algorítmicos: Algoritmos voraces*, pp. 38-46).



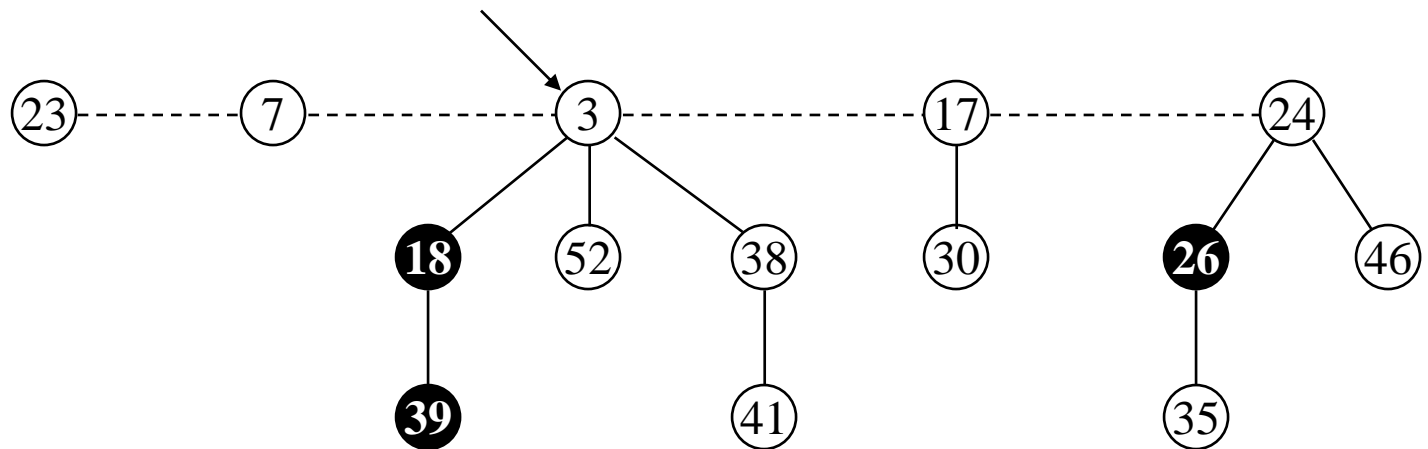
Montículos de Fibonacci

- Desde un punto de vista práctico:
 - Las constantes multiplicativas en el coste y la complejidad de su programación los hacen menos aconsejables que los montículos “ordinarios” en muchas aplicaciones.
 - Por tanto, salvo que se manejen MUCHAS claves, tienen un interés eminentemente teórico.
- ¿Qué son?
 - Podría decirse que son una *versión perezosa* de los montículos binomiales.
 - Si no se ejecutan operaciones de borrado ni de reducción de claves en un montículo de Fibonacci, entonces cada uno de sus árboles es un árbol binomial.
 - Tienen una estructura “más relajada”, permitiendo retrasar la reorganización de la estructura hasta el momento más conveniente para reducir el coste amortizado.



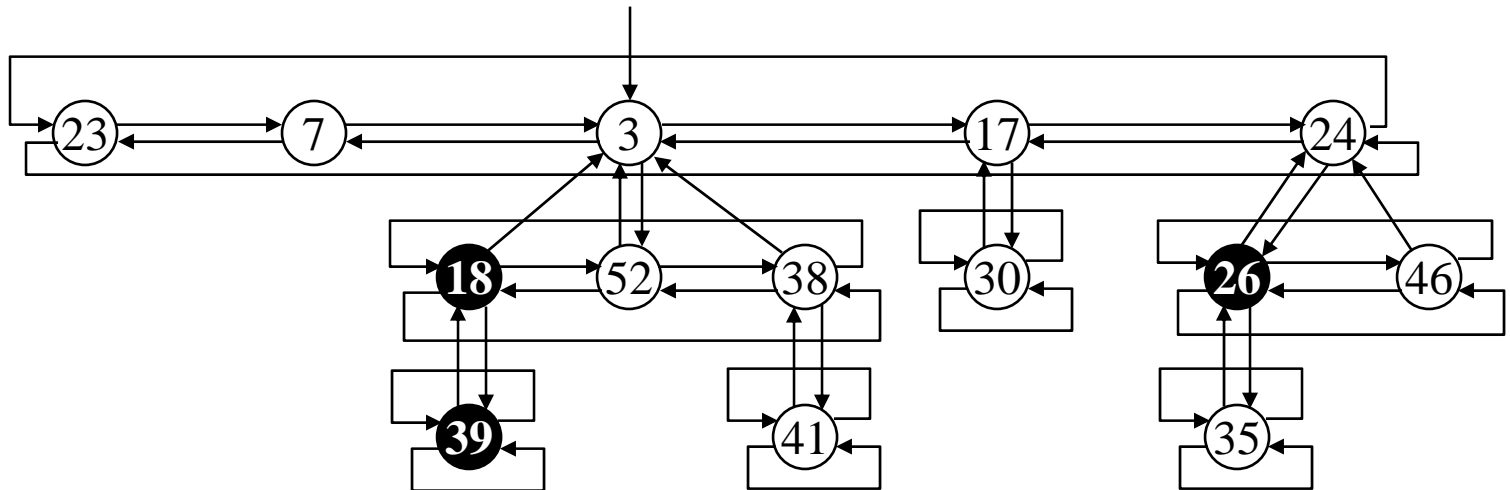
Montículos de Fibonacci

- Estructura de un montículo de Fibonacci:
 - Es un conjunto de *árboles parcialmente ordenados*, es decir, la clave de todo nodo es mayor o igual que la de su padre.
 - Los árboles no precisan ser binomiales.
 - Los árboles no están ordenados.
 - Se accede por un puntero a la raíz de clave mínima.



Montículos de Fibonacci

- Representación en memoria:



- Cada nodo tiene un puntero al padre y un puntero a alguno de sus hijos.
- Los hijos de un nodo se enlazan con una lista circular doblemente enlazada (cada nodo tiene un puntero a su hermano izquierdo y al derecho). El orden en esa lista es arbitrario.
 - Ventaja: se puede eliminar un elemento en $O(1)$ y se pueden juntar dos listas en $O(1)$.

Montículos de Fibonacci

- Representación en memoria (continuación):
 - Además, cada nodo x contiene:
 - El número de hijos de x : llamado *grado*.
 - Un booleano, *marca*, que indica si el nodo x ha perdido un hijo desde la última vez que x se puso como hijo de otro nodo.
 - Los nodos recién creados tienen la marca a falso.
 - También se pone la marca a falso cuando el nodo se convierte en hijo de otro nodo.
 - Veremos para qué sirve la marca más adelante...
 - Al montículo se accede mediante un puntero que apunta al elemento mínimo de la lista circular doblemente encadenada de raíces de árboles.
 - El orden de los árboles en la lista de raíces es arbitrario.

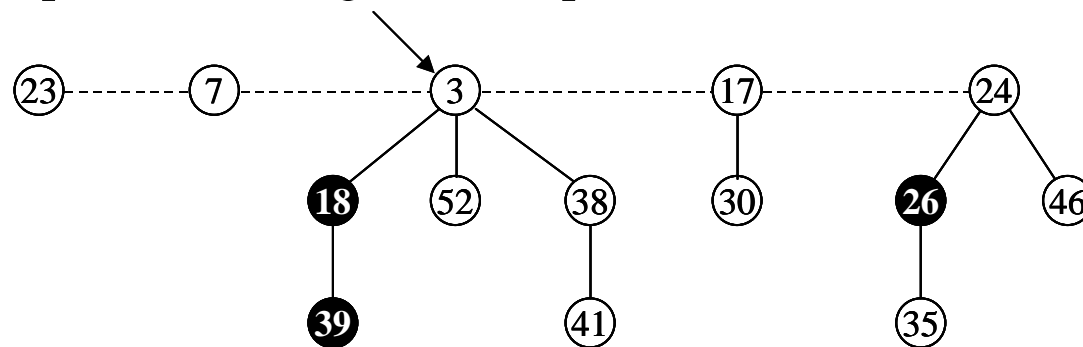


Montículos de Fibonacci

- Definición de la función de potencial:
 - Notación: dado un montículo de Fibonacci, M , sea $a(M)$ el número de árboles en la lista de raíces y $m(M)$ el número de nodos marcados (i.e., con el valor de la marca igual a verdad).
 - Definimos el potencial de M como:

$$P(M) = a(M) + 2m(M)$$

- Ejemplo: el de la figura tiene potencial $5 + 2*3 = 11$



Montículos de Fibonacci

- Hipótesis de trabajo:
 - Supondremos que el grado máximo de cualquier nodo en un montículo de Fibonacci de n nodos está acotado superiormente por $D(n)$.
 - Luego veremos que $D(n) = O(\log n)$.
- Operaciones de montículo agregable:
 - Crear, insertar, ver el mínimo, borrar el mínimo y unión.
 - Con estas operaciones un montículo de Fibonacci será un conjunto de *árboles binomiales desordenados* (a diferencia del montículo binomial).



Montículos de Fibonacci

- Definición: árbol binomial desordenado, U_k :
 - U_0 es un solo nodo
 - U_k consiste en dos árboles binomiales desordenados U_{k-1} *enlazados* de la siguiente forma: la raíz de uno es *cualquier hijo* de la raíz del otro.
- Las propiedades de los árboles binomiales se mantienen con la siguiente variación de la cuarta propiedad:
 - la raíz tiene *grado* k y es el nodo de máximo grado; más aún, los hijos de la raíz son raíces de subárboles U_0, U_1, \dots, U_{k-1} en algún orden.
- Por tanto: si un montículo de Fibonacci de n nodos es un conjunto de árboles binomiales desordenados, entonces $D(n) = \log n$.



Montículos de Fibonacci

- Creación de un montículo de Fibonacci vacío:
 - Consta de dos campos,
 - el puntero a la raíz mínima: $M.min := \text{nil}$, y
 - el número de nodos: $M.n := 0$.
 - Como $a(M) = m(M) = 0$, el potencial es también 0.
 - El coste amortizado de la operación es igual a su coste real, $O(1)$.



Montículos de Fibonacci

- Operación de inserción de un nuevo nodo:

```
algoritmo insertar(M,x)
principio
  x↑.padre:=nil;
  x↑.hijo:=nil;
  x↑.izq:=nil;
  x↑.dch:=nil;
  x↑.grado:=0;
  x↑.marca:=falso;
  añadir x a la lista de raíces de M;
  si M.min=nil or x↑.clave<M.min↑.clave ent
    M.min:=x
  fsi;
  M.n:=M.n+1
fin
```

– El coste real de la operación es $O(1)$.



Montículos de Fibonacci

- Nótese que si se insertan k nodos consecutivos, la lista de raíces se incrementa con k árboles de un solo nodo (a diferencia de los montículos binomiales, en los que se hace un esfuerzo de *compactación*).
- Coste amortizado:
 - Si M es el montículo antes de la inserción y M' es el resultante tras la inserción, $a(M') = a(M) + 1$ y $m(M') = m(M)$ por lo que el incremento del potencial es 1.
 - Por tanto, el coste amortizado de la operación es:
$$A(M') = C(M') + P(M') - P(M) = O(1) + 1 = O(1).$$



Montículos de Fibonacci

- Operación de búsqueda del elemento de clave mínima:
 - El nodo de clave mínima es precisamente el apuntado por $M.min$, por tanto el coste real de la operación es $O(1)$.
 - Como el potencial no cambia, el coste amortizado es también $O(1)$.



Montículos de Fibonacci

- Operación de unión de montículos de Fibonacci:

```
algoritmo unión(M1,M2,M)
principio
  crearVacío(M);
  M.min:=M1.min;
  concatenar la lista de raíces de M2 con la de M;
  si M.min=nil or (M2.min≠nil and M2.min<M1.min) ent
    M.min:=M2.min
  fsi;
  M.n:=M1.n+M2.n
fin
```

- Tampoco hay *compactación*.
- Coste real: $O(1)$.



Montículos de Fibonacci

- Cambio del potencial con la operación de unión:

$$\begin{aligned} P(M) - (P(M_1) + P(M_2)) \\ = (a(M) + 2m(M)) - ((a(M_1) + 2m(M_1)) + (a(M_2) + 2m(M_2))) \\ = 0 \end{aligned}$$

porque $a(M) = a(M_1) + a(M_2)$ y $m(M) = m(M_1) + m(M_2)$

- Coste amortizado:

$$A(M) = C(M) + P(M) - (P(M_1) + P(M_2)) = O(1)$$



Montículos de Fibonacci

- Operación de borrado del elemento con clave mínima (es en la que se *compactan* los árboles):

```
algoritmo borrarMínimo (M)
principio
  z := M.min;
  si z ≠ nil ent
    para todo x hijo de z hacer
      añadir x a la lista de raíces de M;
      x↑.padre := nil;
    fpara;
  borrar z de la lista de raíces de M;
  si z = z↑.dch ent M.min := nil
  sino
    M.min := z↑.dch; compactar (M)
  fsi;
  M.n := M.n - 1
fsi
fin
```



Montículos de Fibonacci

- El algoritmo “compactar” debe juntar las raíces de igual grado hasta conseguir que haya como mucho una raíz de cada grado (así se reduce el número de árboles en el montículo), y ordena las raíces por grado.
- Para compactar, repetimos los siguientes pasos hasta que todas las raíces de la lista de raíces del montículo tengan distinto grado:
 - Buscar dos raíces x e y con igual grado y con la clave de x menor o igual que la clave de y .
 - Enlazar y a x :
 - borrar y de la lista de raíces y hacer que y sea hijo de x (algoritmo “enlazar”);
 - se incrementa el grado de x y la *marca* de y se pone a falso.



Montículos de Fibonacci

- Veamos primero el algoritmo “enlazar”:

```
algoritmo enlazar(M,y,x)
{borrar y de la lista de raíces
 y hacer que sea hijo de x}
principio
    borrar y de la lista de raíces de M;
    poner y como hijo de x;
     $x^{\uparrow}.\text{grado} := x^{\uparrow}.\text{grado} + 1;$ 
     $y^{\uparrow}.\text{marca} := \text{falso}$ 
fin
```



Montículos de Fibonacci

- Y ahora el algoritmo “compactar”:
 - Se utiliza un vector auxiliar $A[0..D(n)]$, donde n es el número de datos del montículo; $A[i] = y$ significa que y es una raíz con grado i .

algoritmo compactar (M)

principio

```
para i:=0 hasta D(M.n) hacer  
    A[i] := nil  
fpara;  
para todo w en la lista de raíces de M hacer  
    x:=w;  
    d:=x↑.grado;  
    ...  
    A[d] :=x;  
fpara;  
...
```

} inicializar A

} se “procesa” toda raíz w (copiada en x);
al final, $A[x↑.grado] = x$



Montículos de Fibonacci

- Veamos el procesamiento de cada raíz w :

```
...  
para todo  $w$  en la lista de raíces de  $M$  hacer  
   $x := w$ ;  $d := x \uparrow . \text{grado}$ ;  
  mq  $A[d] \neq \text{nil}$  hacer inicialmente era nil, si no lo es  
hay otras raíces de grado  $d$   
     $y := A[d]$ ;  
    si  $x \uparrow . \text{clave} > y \uparrow . \text{clave}$  ent } hay que enlazar  $x$  e  $y$   
      intercambiar( $x, y$ )  
    fsi;  
    enlazar( $M, y, x$ ) ;  
     $A[d] := \text{nil}$ ; → el nodo  $y$  ya no es una raíz  
     $d := d + 1$  → el nodo  $x$  tiene un hijo más ( $y$ )  
  fmq; → ahora seguro que no hay otra raíz con  
igual grado que  $x$   
   $A[d] := x$ ;  
fpara;  
...
```

Montículos de Fibonacci

- Y ya sólo queda reconstruir la lista de raíces a partir de la información del vector A:

```
...  
M.min:=nil;  
para i:=0 hasta D(M.n) hacer  
  si A[i]≠nil ent  
    añadir A[i] a la lista de raíces de M;  
    si M.min=nil  
      or A[i]↑.clave<M.min↑.clave ent  
        M.min:=A[i]  
    fsi  
  fsi  
fpara  
fin
```



Montículos de Fibonacci

- Hay que verificar que la operación de borrado mantiene los árboles del montículo en la clase de los *binomiales desordenados*:
 - En la operación “borrarMínimo” todo hijo x de z se convierte en raíz de un nuevo árbol, pero esos hijos son a su vez árboles binomiales desordenados (por la propiedad cuarta).
 - En la operación de compactación se enlazan parejas de árboles binomiales desordenados de igual grado, el resultado es un nuevo árbol binomial desordenado de grado una unidad mayor.



Montículos de Fibonacci

- Coste amortizado del borrado del elemento mínimo
 - Primero hay que calcular el coste real:
Algoritmo “borrarMínimo”: $O(D(n))$
Algoritmo “compactar”:
 - los bucles primero y tercero tienen un coste $O(D(n))$
 - el bucle intermedio:
 - » Antes de ejecutar “compactar” el tamaño de la lista de raíces es como mucho $D(n) + a(M) - 1$, es decir, el tamaño antes de empezar la operación, $a(M)$, menos el nodo raíz extraído, más los hijos del nodo extraído, que son como mucho $D(n)$.
 - » En cada ejecución del “mq” interno una raíz se enlaza con otra, por tanto el coste total es como mucho proporcional a $D(n) + a(M)$.

Luego el coste real es $O(D(n) + a(M))$



Montículos de Fibonacci

- Coste amortizado del borrado del mínimo (cont.)
 - Después hay que calcular el potencial
 - Antes de extraer el mínimo el potencial es $a(M) + 2m(M)$.
 - Después, es como mucho: $(D(n) + 1) + 2m(M)$, porque quedan como mucho $D(n) + 1$ raíces y no se marcan nodos durante la operación.
 - El coste amortizado es, por tanto:
$$\begin{aligned}A(M') &= C(M') + P(M') - P(M) \\&= O(D(n) + a(M)) + ((D(n) + 1) + 2m(M) - a(M) + 2m(M)) \\&= O(D(n))\end{aligned}$$
 - La demostración de que $D(n) = O(\log n)$ vendrá más tarde...



Montículos de Fibonacci

- Operación de reducción del valor de una clave:
 - Vemos antes un subalgoritmo auxiliar:
- Corta el enlace entre un nodo y su padre, convirtiendo al hijo en raíz.

```
algoritmo cortar(M,x,y)
principio
    borrar x de la lista de hijos de y,
        reduciendo  $y \uparrow$ .grado;
    añadir x a la lista de raíces de M;
     $x \uparrow$ .padre:=nil;
     $x \uparrow$ .marca:=falso
fin
```



Montículos de Fibonacci

- El algoritmo de reducción de una clave:

```
algoritmo reducir(M,x,c)
{pre:c<x↑.clave}{post:la clave de x pasa a ser c}
principio
  x↑.clave:=c;
  y:=x↑.padre;
  si y≠nil and x↑.clave<y↑.clave ent
    cortar(M,x,y);
    cortar_arriba(M,y);
  fsi;
  si x↑.clave<M.min↑.clave ent
    M.min:=x;
  fsi
fin
```

hay que re-estructurar el árbol

x pasa a ser una raíz de M

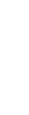
el nodo y acaba de perder un hijo, entonces...
lo vemos a continuación

Montículos de Fibonacci

– “Cortar_arriba”, una versión recursiva de “cortar”:

- Si y es una raíz entonces el algoritmo no hace nada.
- Si y no está marcado, el algoritmo lo marca y deja de subir.
- Si y está marcado, se corta, y se llama a si mismo subiendo hacia el padre.
- En definitiva: un nodo está marcado cuando ha perdido el 1^{er} hijo pero no ha perdido el 2^o.

```
algoritmo cortar_arriba(M,y)
principio
  z:=y↑.padre;
  si z≠nil ent
    si y↑.marca=falso ent
      y↑.marca:=verdad
    sino
      cortar(M,y,z);
      cortar_arriba(M,z)
    fsi
  fsi
fin
```



Montículos de Fibonacci

- ¿Para qué vale la marca de los nodos?
 - Suponer que un nodo x “sufre” la siguiente historia:
 - en algún momento, x fue raíz;
 - entonces, x se enlazó con otro nodo;
 - después, dos hijos de x se eliminaron de su lista de hijos mediante cortes.
 - Tan pronto como x pierde su 2º hijo, x se corta de su padre y pasa a ser raíz.
 - El campo *marca* es verdad si han ocurrido los dos primeros pasos pero no ha perdido aún el 2º hijo.



Montículos de Fibonacci

- Coste amortizado de la reducción de una clave:
 - Primero: calcular el coste real
 - El algoritmo “reducir” tiene un coste del mismo orden que “cortar_arriba”.
 - Suponer que “cortar_arriba” se ejecuta recursivamente c veces en una llamada desde “reducir”, como cada llamada a “cortar_arriba” solo cuesta $O(1)$ más la nueva llamada recursiva, el coste de “reducir” es $O(c)$.



Montículos de Fibonacci

- Segundo: calcular el cambio de potencial
 - Sea M el montículo antes de reducir la clave.
 - Cada llamada recursiva a “cortar_arriba”, excepto la última, “corta” un nodo marcado y lo desmarca.
 - » Después de “reducir” quedan $a(M) + c$ árboles en el montículo: los $a(M)$ originales, los $c - 1$ que se han cortado y el de raíz x
 - » Después de “reducir” quedan como mucho $m(M) - c + 2$ nodos marcados: $c - 1$ se desmarcan con la llamada y uno puede marcarse en la última llamada recursiva.
 - Por tanto, el cambio de potencial está acotado por:

$$((a(M) + c) + 2(m(M) - c + 2)) - (a(M) + 2m(M)) = 4 - c$$

- Luego el coste amortizado es, como mucho:

$$A(M') = C(M') + P(M') - P(M) = O(c) + 4 - c \stackrel{\uparrow}{=} O(1)$$

podemos escalar las unidades del potencial para
“dominar” la constante escondida en $O(c)$



Montículos de Fibonacci

- Ahora vemos el porqué de esa función de potencial...
 - Cuando un nodo marcado y se corta en una operación de “cortar_arriba”, su *marca* desaparece, luego el potencial se reduce en 2 unidades ($P(M) = a(M) + 2m(M)$).
 - Una de esas unidades paga por el corte y por quitar la marca y la otra unidad compensa el incremento de potencial en una unidad por el hecho de que y pasa a ser raíz de un nuevo árbol.



Montículos de Fibonacci

- Operación de borrado de un elemento:

```
algoritmo borrar(M,x)
principio
    reducir(M,x,  $-\infty$ ) ;
    borrarMínimo(M)
fin
```

- Es igual que para los montículos binomiales.
- El algoritmo hace decrecer el valor de la clave del elemento a borrar hasta el valor mínimo posible con el algoritmo “reducir”, en tiempo (amortizado) $O(1)$.
- Después, con la operación de borrado del mínimo, de coste (amortizado) $O(D(n))$, se borra esa raíz.
- El coste total (amortizado) es, por tanto, $O(D(n))$.



Montículos de Fibonacci

- Falta ver que el grado, $D(n)$, de cualquier nodo de un montículo de Fibonacci de n nodos está acotado por $O(\log n)$.
 - Ya vimos que si todos sus árboles son árboles binomiales desordenados, entonces $D(n) = \log n$.
 - Pero los cortes del algoritmo “reducir” pueden hacer que los árboles dejen de ser “binomiales desordenados”.
 - Veremos ahora que, dado que un nodo se corta de su padre tan pronto como pierde dos hijos, se sigue teniendo que $D(n)$ es $O(\log n)$.



Montículos de Fibonacci

- Lema 1: Sea x un nodo cualquiera de grado k de un montículo de Fibonacci, con hijos y_1, y_2, \dots, y_k (en el orden en que fueron enlazados). Se tiene:
 - $y_1 \uparrow . \text{grado} \geq 0$, y
 - $y_i \uparrow . \text{grado} \geq i - 2$, para $i = 2, 3, \dots, k$.

Demostración: La primera afirmación es obvia.

Veamos la segunda:

- Cuando y_i ($i \geq 2$) fue enlazado a x , los otros y_1, y_2, \dots, y_{i-1} ya eran hijos de x , por tanto $x \uparrow . \text{grado} \geq i - 1$.
- El nodo y_i se enlaza a x sólo si $x \uparrow . \text{grado} = y_i \uparrow . \text{grado}$, por tanto $y_i \uparrow . \text{grado} \geq i - 1$ en el momento de enlazarse.
- Desde enlazarse a x , y_i sólo ha podido perder como mucho un hijo, de lo contrario (si hubiese perdido dos hijos) se habría cortado de x , por tanto $y_i \uparrow . \text{grado} \geq i - 2$.

Montículos de Fibonacci

Llegamos a la explicación del nombre de estos montículos...

- Lema 2: Sea x un nodo cualquiera de grado k de un montículo de Fibonacci. Sea $card(x)$ el número de nodos del subárbol con raíz x (incluido x).

Entonces:

$$card(x) \geq F_{k+2} \geq \phi^k, \text{ donde:}$$

$$F_k = \begin{cases} 0, & \text{si } k = 0 \\ 1, & \text{si } k = 1 \\ F_{k-1} + F_{k-2}, & \text{si } k \geq 2 \end{cases} \text{ es el } k - \text{ésimo n}^\circ \text{ de Fibonacci, y}$$

$$\phi = (1 + \sqrt{5})/2 \text{ es la razón áurea } (=1'61803...).$$



Montículos de Fibonacci

– Demostración del Lema:

- Sea s_k el valor mínimo de $card(z)$ para todos los z con $z \uparrow . grado = k$.
- Por tanto: $s_0 = 1, s_1 = 2, s_2 = 3, s_k \leq card(x)$.
- Sean y_1, y_2, \dots, y_k los hijos de x (en el orden en que fueron enlazados).
- Por el Lema 1, se tiene:

$$card(x) \geq s_k \geq 2 + \sum_{i=2}^k s_{i-2}$$

(1 por el nodo x + 1 por y_1)



Montículos de Fibonacci

- Demostración del Lema (cont.):
 - Veamos un resultado previo para los F_k :

$$F_{k+2} = 1 + \sum_{i=0}^k F_i, \text{ para } k \geq 0$$

- Demostración: por inducción sobre k .
 - » $k = 0$: $F_2 = 1 + F_0 = 1 + 0 = 1$
 - » Suponiendo que

$$F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$$

se tiene:

$$F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) = 1 + \sum_{i=0}^k F_i$$



Montículos de Fibonacci

– Demostración del Lema (cont.):

- Ahora veamos por inducción sobre k que $s_k \geq F_{k+2}$ para todo k no negativo.

- Para $k = 0$ y $k = 1$ es trivial.

- Sea $k \geq 2$. Supóngase que $s_i \geq F_{i+2}$ para $i = 0, 1, \dots, k-1$.

Entonces:

$$s_k \geq 2 + \sum_{i=2}^k s_{i-2} \geq 2 + \sum_{i=2}^k F_i = 1 + \sum_{i=0}^k F_i = F_{k+2}$$

- Sólo falta ver que $F_{k+2} \geq \phi^k$.

- Se puede demostrar (ejercicio) a partir del resultado:

$$F_i = \frac{\phi^i - \hat{\phi}^i}{\sqrt{5}}, \text{ donde } \hat{\phi} = \frac{1 - \sqrt{5}}{2}$$

que, a su vez, puede demostrarse por inducción a partir de la definición.



Montículos de Fibonacci

- Corolario (del Lema 2): El grado máximo, $D(n)$ de cualquier nodo de un montículo de Fibonacci de n nodos es $O(\log n)$.

Demostración:

- Sea x un nodo cualquiera y $k = x \uparrow . \text{grado}$.
- Por el Lema 2, $n \geq \text{card}(x) \geq \phi^k$. Por tanto: $k \leq \log_{\phi} n$, y se sigue el resultado.



Análisis amortizado

- El plan:
 - Conceptos básicos:
 - Método agregado
 - Método contable
 - Método potencial
 - Primer ejemplo: análisis de tablas *hash* dinámicas
 - Montículos agregables (binomiales y de Fibonacci)
 - **Estructuras de conjuntos disjuntos**
 - Listas lineales auto-organizativas
 - Árboles auto-organizativos ("splay trees")



Estructuras de conjuntos disjuntos

- Motivación:
 - Algoritmo de Kruskal para el cálculo del árbol de recubrimiento de peso mínimo de un grafo

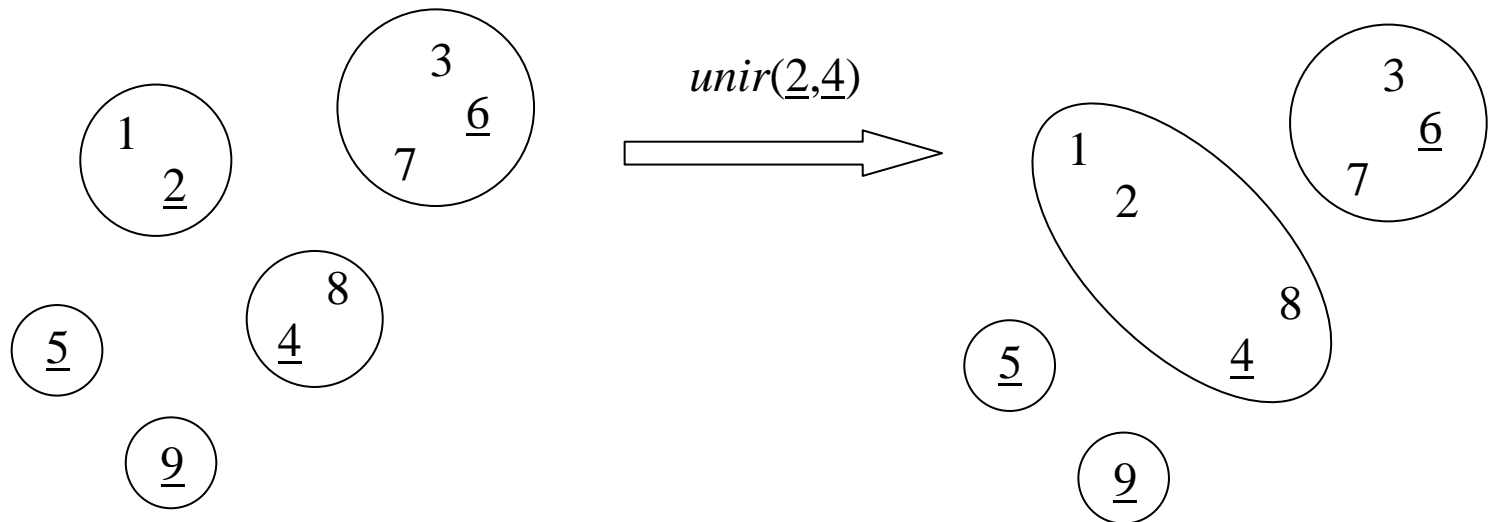
```
var g: grafo no dirig. conexo etiquetado no negat.  
    T: montículo de aristas; gsol: grafo; u,v: vért; x: etiq;  
    C: estructura de conjuntos disjuntos; ucomp,vcomp: nat  
creaECD(C); {cada vért. forma un conjunto}  
creaVacío(gsol); creaVacía(T);  
para todo v en vért, para todo <u,x> en adyac(g,v) hacer  
    inserta(T,v,u,x)  
fpara;  
mq numConjuntos(C) > 1 hacer  
    <u,v,x> := primero(T); borra(T);  
    ucomp := indConjunto(C,u); vcomp := indConjunto(C,v);  
    si ucomp ≠ vcomp entonces  
        fusiona(C,ucomp,vcomp); añade(gsol,u,v,x)  
    fsi  
fmq;  
devuelve gsol
```

(extraído de “Esquemas Algorítmicos”)



Estructuras de conjuntos disjuntos

- Mantener una partición de $S=\{1,2,\dots,n\}$ con las operaciones de



$encontrar(3)=\underline{6}$

Estructuras de conjuntos disjuntos

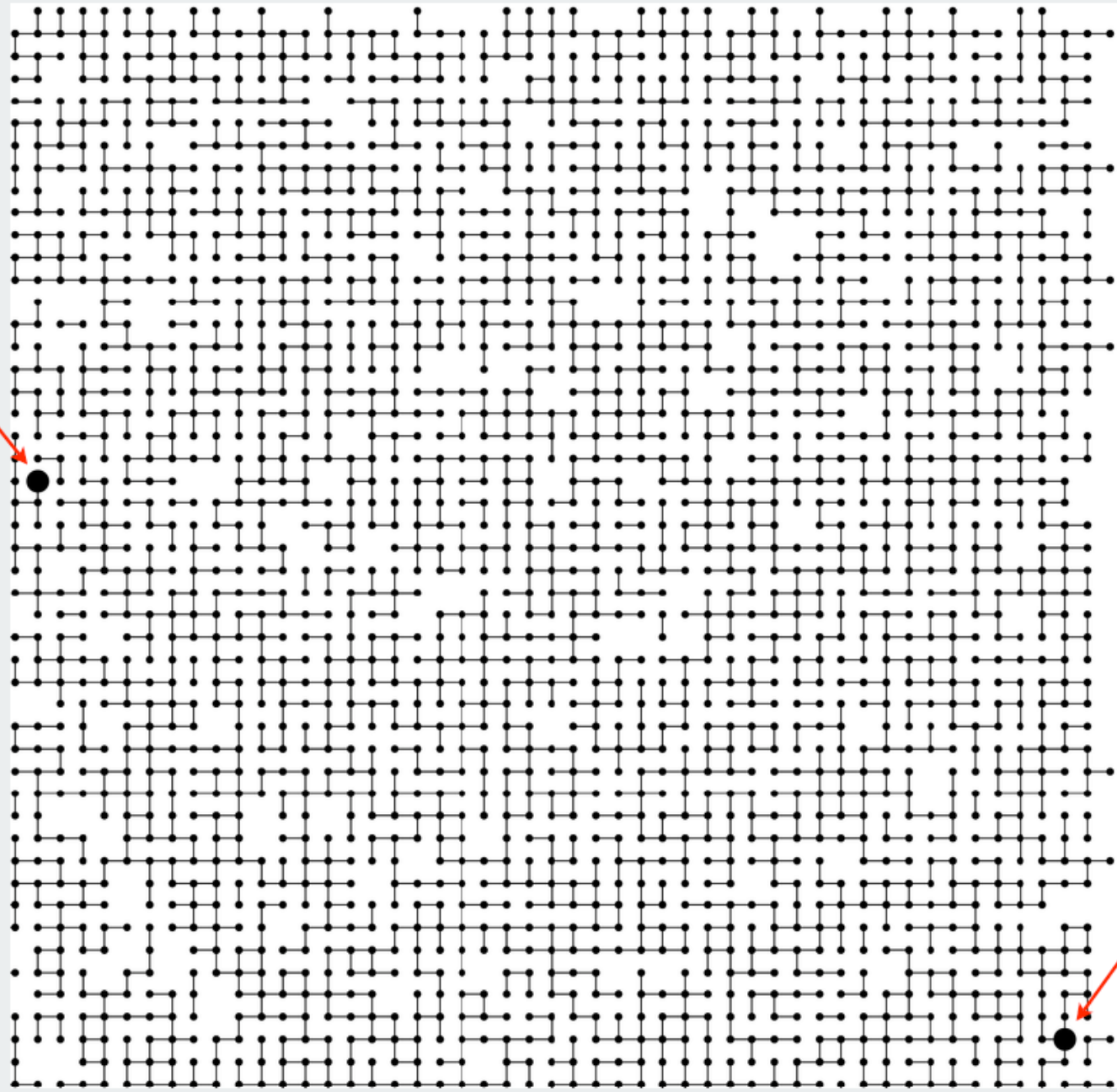
- Definición:

- Una estructura de conjuntos disjuntos (ECD) (en muchos libros en inglés, “union-find structure”) mantiene una colección de conjuntos disjuntos

$$S = \{S_1, S_2, \dots, S_k\}$$

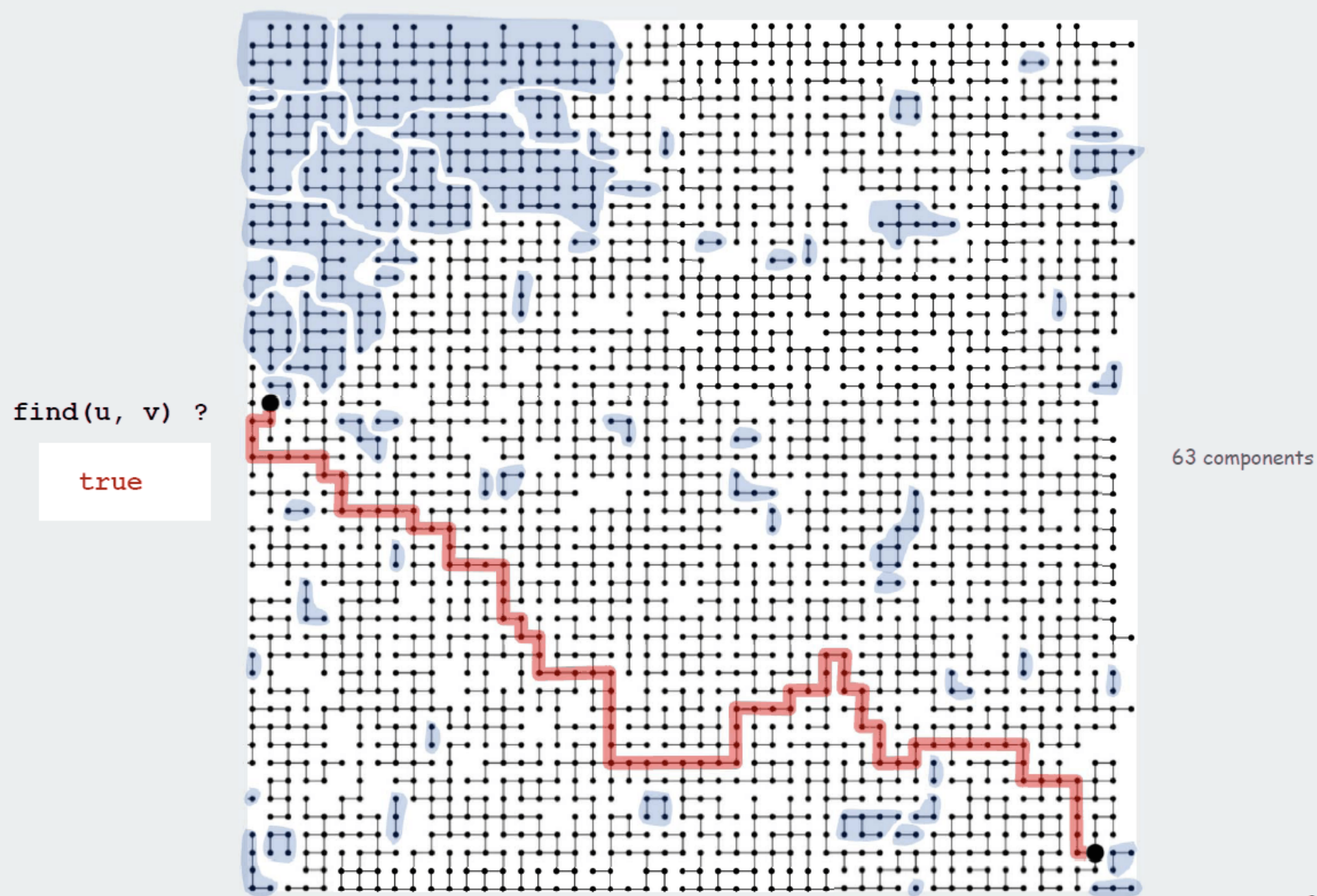
- Cada conjunto se identifica por un representante (un miembro del conjunto)
- Se precisan (como mínimo) las operaciones de:
 - *crear*(x): crea un nuevo conjunto con un solo elemento (x , que no debe pertenecer a ningún otro conjunto)
 - *unir*(x, y): une los conjuntos que contienen a x y a y en un nuevo conjunto, y destruye los viejos conjuntos de x e y
 - *encontrar*(x): devuelve el representante del (único) conjunto que contiene a x





`find(u, v) ?`





Estructuras de conjuntos disjuntos

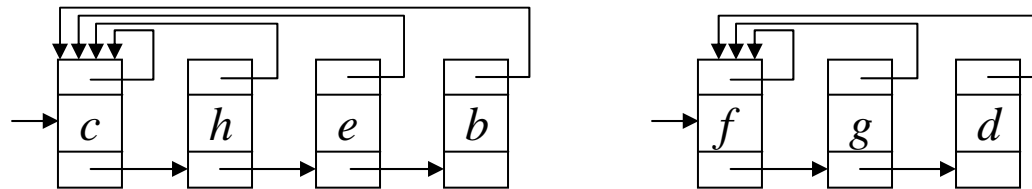
- Ejemplo sencillo de aplicación: calcular las componentes conexas de un grafo no dirigido

```
algoritmo componentes_conexas(g)
principio
  para todo v vértice de g hacer
    crear(v)
  fpara;
  para toda (u,v) arista de g hacer
    si encontrar(u) ≠ encontrar(v) entonces
      unir(u,v)
    fsi
  fpara
fin
```



Estructuras de conjuntos disjuntos

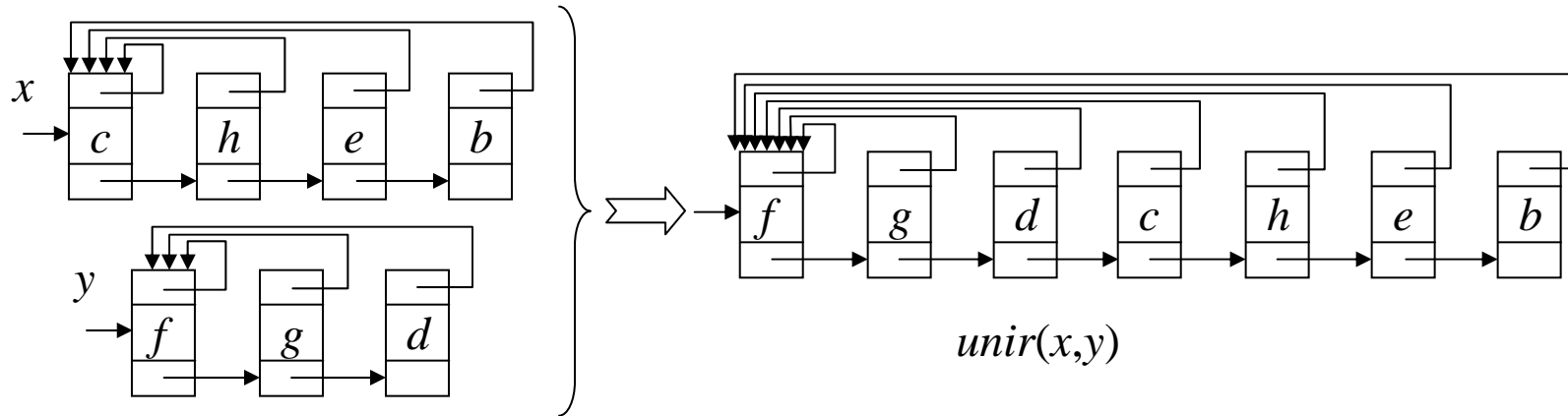
- Primera implementación: listas encadenadas
 - Cada conjunto se almacena en una lista encadenada
 - Cada registro de la lista contiene:
 - Un elemento del conjunto
 - Un puntero al siguiente registro de la lista
 - Un puntero al primer registro de la lista
 - El primer elemento de cada lista es el representante



- Las implementaciones de *crear* y *encontrar* son triviales y requieren $O(1)$ en tiempo

Estructuras de conjuntos disjuntos

– Implementación de *unir*:



- Concatenar la primera lista tras la segunda y modificar los punteros al primero en la primera lista (coste en tiempo lineal en la longitud de esa lista)
- El coste amortizado **con esta implementación** no puede reducirse (a algo menor que lineal)...

Estructuras de conjuntos disjuntos

- Ejemplo en el que el coste amortizado de *unir* es lineal:

<u>operación</u>	<u>nº de objetos modificados</u>
<i>crear</i> (x_1)	1
<i>crear</i> (x_2)	1
...	...
<i>crear</i> (x_n)	1
<i>unir</i> (x_1, x_2)	1
<i>unir</i> (x_2, x_3)	2
<i>unir</i> (x_3, x_4)	3
...	...
<i>unir</i> (x_{q-1}, x_q)	$q-1$

- Es una secuencia de $m = n + q - 1$ operaciones y requiere un tiempo $\Theta(n + q^2) = \Theta(m^2)$ (porque $n = \Theta(m)$ y $q = \Theta(m)$).
- Por tanto, cada operación requiere, en media (coste amortizado), $\Theta(m)$.



Estructuras de conjuntos disjuntos

- Mejora de la implementación de *unir*:
 - *Heurística de la unión*: si cada lista almacena explícitamente su longitud, optar por añadir siempre la lista más corta al final de la más larga.
 - Con esta heurística sencilla el coste de una única operación sigue siendo el mismo, pero el coste amortizado se reduce:
Una secuencia de m operaciones de *crear*, *unir* y *encontrar*, n de las cuales sean de *crear*, o lo que es lo mismo, una secuencia de m operaciones con una ECD con n elementos distintos cuesta $O(m + n \log n)$ en tiempo.



Estructuras de conjuntos disjuntos

- Demostración:

- 1º) Calcular para cada elemento x de un conjunto de tamaño n una cota superior del nº de veces que se cambia su puntero al representante:

La 1ª vez que se cambia, el conjunto resultante tiene al menos 2 elementos.

La 2ª vez, tiene el menos 4 elementos (x siempre debe estar en el conjunto más pequeño). Con igual argumento, para todo $k \leq n$, después de cambiar $\lceil \log k \rceil$ veces el puntero al representante de x , el conjunto resultante debe tener como mínimo k elementos.

Como el conjunto más grande tiene no más de n elementos, el puntero al representante de cada elemento se ha cambiado no más de $\lceil \log n \rceil$ veces en todas las operaciones de *unir*.

Por tanto, el coste total debido a los cambios del puntero al representante es $O(n \log n)$.

- 2º) Cada operación *crear* y *encontrar* está en $O(1)$, y hay $O(m)$ de esas operaciones.

Por tanto, el coste total de toda la secuencia es $O(m + n \log n)$.



Estructuras de conjuntos disjuntos

- Ejercicio:

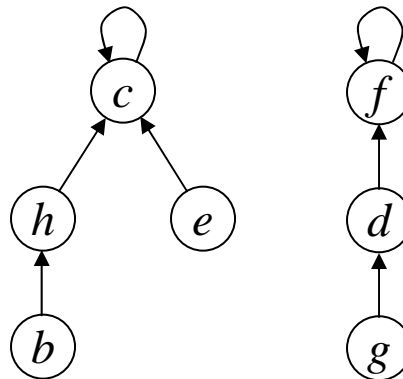
Teniendo en cuenta que el coste de una secuencia de m operaciones con una ECD con n elementos distintos es $O(m + n \log n)$, se puede demostrar que:

- El coste amortizado de una operación de *crear* o de *encontrar* es $O(1)$.
- El coste amortizado de una operación de *unir* es $O(\log n)$.



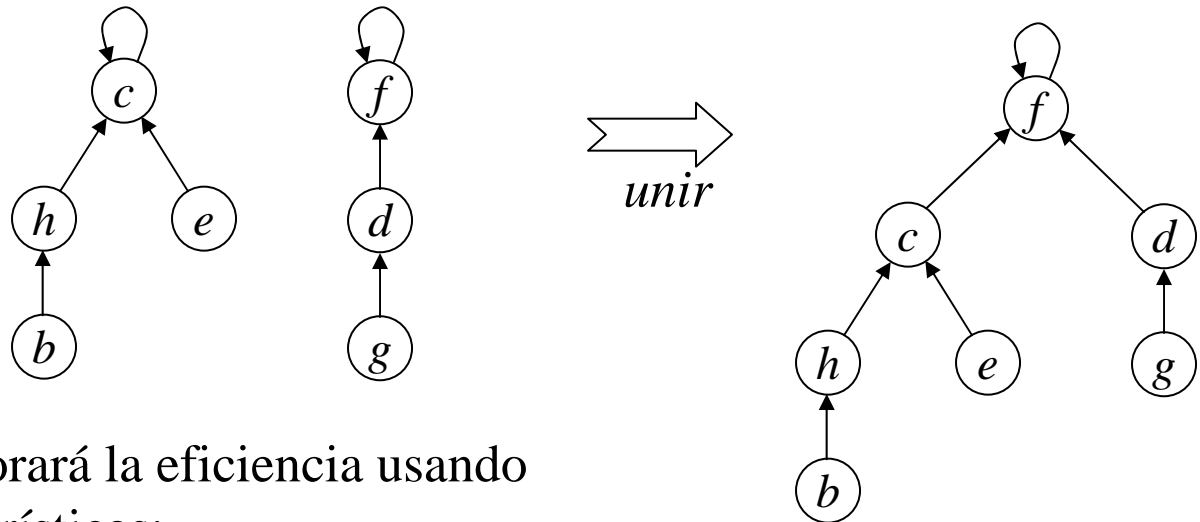
Estructuras de conjuntos disjuntos

- Segunda implementación: bosque
 - Conjunto de árboles, cada uno representando un conjunto disjunto de elementos.
 - Representación con puntero al padre.
 - La raíz de cada árbol es el representante y apunta a si misma.



Estructuras de conjuntos disjuntos

- La implementación “trivial” de las operaciones no mejora la eficiencia de la implementación con listas:
 - *Crear* crea un árbol con un solo nodo.
 - *Encontrar* sigue el puntero al padre hasta llegar a la raíz del árbol (los nodos recorridos se llaman *camino de búsqueda*).
 - *Unir* hace que la raíz de un árbol apunte a la raíz del otro.



- Se mejorará la eficiencia usando dos heurísticas:
 - *Unión por rango*
 - *Compresión de caminos*

Estructuras de conjuntos disjuntos

- Conectividad en redes.
- Percolación (flujo de un líquido a través de un medio poroso).
- Procesamiento de imágenes.
- Antecesor común más próximo (en un árbol).
- Equivalencia de autómatas de estados finitos.
- Inferencia de tipos polimórficos o tipado implícito (algoritmo de Hinley-Milner).
- Árbol de recubrimiento mínimo (algoritmo de Kruskal).
- Juego (*Go*, *Hex*)
- Compilación de la instrucción EQUIVALENCE en Fortran.
- Mantenimiento de listas de copias duplicadas de páginas web.



Estructuras de conjuntos disjuntos

- Implementación en el bosque de la heurística “unión por rango”
 - Similar a la heurística de la unión usada con listas.
 - Hacer que la raíz que va a apuntar a la otra sea la del árbol con menos nodos.
 - En lugar de mantener el tamaño de los árboles, mantenemos siempre el *rango* de los árboles, que es una cota superior de su altura.
 - Al *crear* un árbol, su rango es 0.
 - Con la operación de *encontrar* el rango no cambia.
 - Al *unir* dos árboles se coloca como raíz la del árbol de mayor rango, y éste no cambia; en caso de empate, se elige uno cualquiera y se incrementa su rango en una unidad.



Estructuras de conjuntos disjuntos

```
algoritmo crear(x)
principio
    nuevóArbol(x);
    x.padre:=x;
    x.rango:=0
fin
```

```
algoritmo unir(x,y)
principio
    enlazar(encontrar(x),encontrar(y))
fin
```

```
algoritmo enlazar(x,y)
principio
    si x.rango>y.rango ent
        y.padre:=x
    sino
        x.padre:=y;
        si x.rango=y.rango ent y.rango:=y.rango+1 fsi
    fsi
fin
```



Estructuras de conjuntos disjuntos

- Implementación en el bosque de la heurística de “compresión de caminos”
 - Muy simple y muy efectiva.
 - Se usa en la operación de *encontrar*.
 - Hace que los nodos recorridos en el *camino de búsqueda* pasen a apuntar directamente a la raíz.
 - No se modifica la información sobre el rango.

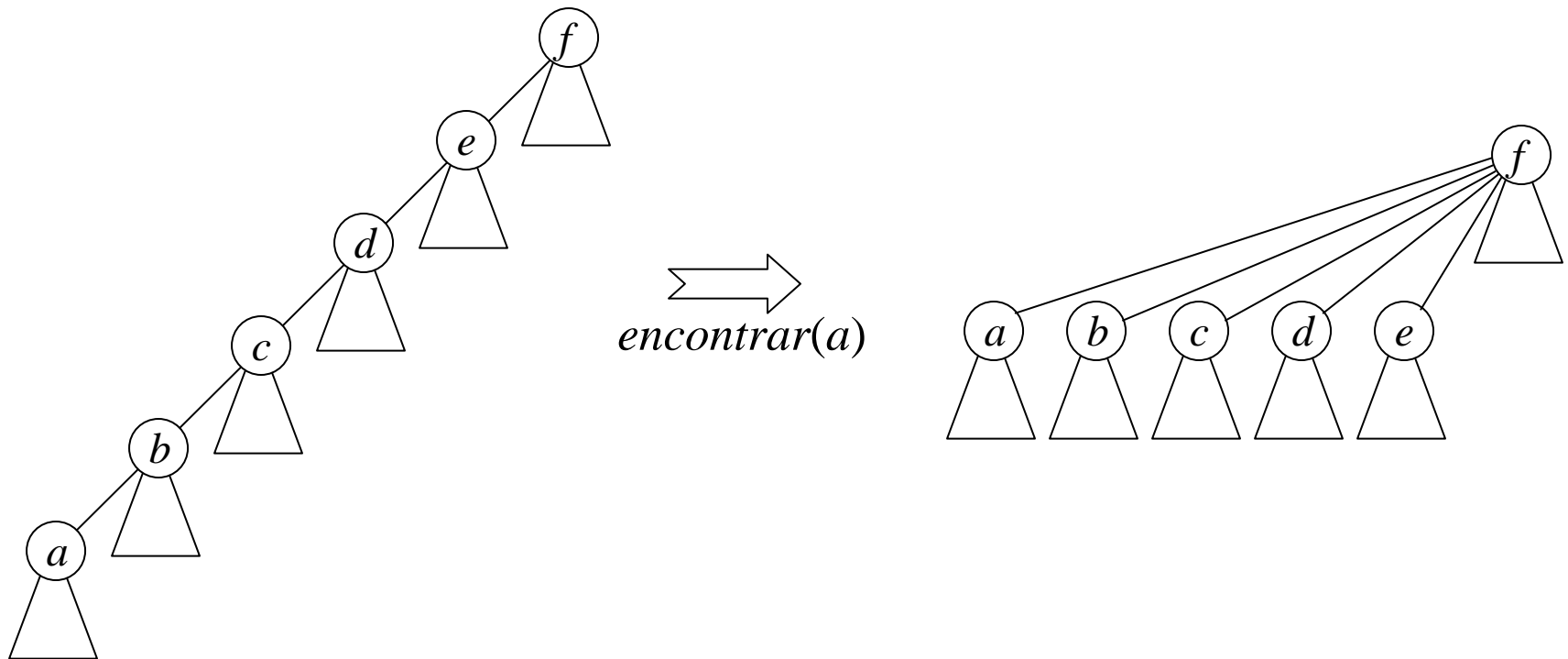
```
algoritmo encontrar(x) devuelve puntero a nodo  
principio  
    si x≠x.padre ent x.padre:=encontrar(x.padre) fsi;  
    devuelve x.padre  
fin
```

Ojo! Modifica x



Estructuras de conjuntos disjuntos

- Efecto de la compresión de caminos en una operación de *encontrar*



Estructuras de conjuntos disjuntos

- Coste de las operaciones usando las heurísticas:
 - El uso separado de las heurísticas de unión por rango y de compresión de caminos mejora la eficiencia de las operaciones:
 - La unión por rango (sin usar compresión de caminos) da un coste de $O(m \log n)$ para una secuencia de m operaciones con una ECD con n elementos distintos.
 - La compresión de caminos (sin usar unión por rango) da un coste de $\Theta(f \log_{(1+f/n)} n)$ para una secuencia de n operaciones de *crear* (y por tanto hasta un máximo de $n-1$ de *unir*) y f operaciones de *encontrar*, si $f \geq n$, y $\Theta(n + f \log n)$ si $f < n$.
 - El uso conjunto de las dos heurísticas mejora aún más la eficiencia de las operaciones...



Estructuras de conjuntos disjuntos

- Coste utilizando ambas heurísticas:
 - Si se usan la unión por rango y la compresión de caminos, el coste en el caso peor para una secuencia de m operaciones con una ECD con n elementos distintos es $O(m \alpha(m,n))$, donde $\alpha(m,n)$ es una función (parecida a la inversa de la función de Ackerman) que crece **muy** despacio.

Tan despacio que en cualquier aplicación práctica que podamos imaginar se tiene que $\alpha(m,n) \leq 4$, por tanto puede interpretarse el coste como lineal en m , en la práctica.



Estructuras de conjuntos disjuntos

- ¿Cómo es la función $\alpha(m,n)$?

- Funciones previas:

- Para todo entero $i \geq 0$, abreviamos $2^{2^{\cdot^{\cdot^2}}}$ $\left. \vphantom{2^{2^{\cdot^{\cdot^2}}}} \right\} i \text{ veces}$ con $g(i)$:

$$g(i) = \begin{cases} 2^1, & \text{si } i = 0 \\ 2^2, & \text{si } i = 1 \\ 2^{g(i-1)}, & \text{si } i > 1 \end{cases}$$

- La función “logaritmo estrella”, $\log^* n$, es la inversa de g :

$$\log^* n = \min\{i \geq 0 \mid \log^{(i)} n \leq 1\}$$

$$\log^{(i)} n = \begin{cases} n, & \text{si } i = 0 \\ \log(\log^{(i-1)} n), & \text{si } i > 0 \text{ y } \log^{(i-1)} n > 0 \\ \text{no definido,} & \text{si } i > 0 \text{ y } \log^{(i-1)} n \leq 0 \text{ ó } \log^{(i-1)} n \text{ no está definido} \end{cases}$$

$$\text{Es decir, } \log^* g(n) = \log^* 2^{2^{\cdot^{\cdot^2}}} \left. \vphantom{2^{2^{\cdot^{\cdot^2}}}} \right\} n \text{ veces} = n + 1.$$



Estructuras de conjuntos disjuntos

- La función de Ackerman, definida para enteros $i, j \geq 1$:

$$A(1, j) = 2^j, \text{ para } j \geq 1$$

$$A(i, 1) = A(i-1, 2), \text{ para } i \geq 2$$

$$A(i, j) = A(i-1, A(i, j-1)), \text{ para } i, j \geq 2$$

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
$i = 1$	2^1	2^2	2^3	2^4
$i = 2$	2^2	2^{2^2}	$2^{2^{2^2}}$	$2^{2^{2^{2^2}}}$
$i = 3$	2^{2^2}	$2^{2^{2^2}} \} 16$	$2^{2^{2^{2^2}}} \} 16$	$2^{2^{2^{2^{2^2}}}} \} 16$



Estructuras de conjuntos disjuntos

- Por fin, la función $\alpha(m,n)$:
 - Es “una especie de inversa” de la función de Ackerman (no en sentido matemático estricto, sino en cuanto a que crece tan despacio como deprisa lo hace la de Ackerman).
 - $\alpha(m,n) = \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) > \log n\}$
 - ¿Por qué podemos suponer que siempre (en la práctica) $\alpha(m,n) \leq 4$?
 - Nótese que $\lfloor m/n \rfloor \geq 1$, porque $m \geq n$.
 - La función de Ackerman es estrictamente creciente con los dos argumentos, luego $\lfloor m/n \rfloor \geq 1 \Rightarrow A(i, \lfloor m/n \rfloor) \geq A(i, 1)$, para $i \geq 1$.
 - En particular, $A(4, \lfloor m/n \rfloor) \geq A(4, 1) = A(3, 2) = 2^{2^{\cdot^{\cdot^2}}}$ ¹⁶
 - Luego sólo ocurre para n 's “enormes” que $A(4, 1) \leq \log n$, y por tanto $\alpha(m,n) \leq 4$ para todos los valores “razonables” de m y n .



Estructuras de conjuntos disjuntos

- La demostración^(*) [Tar83, pp.23-31] no es fácil... veremos una cota **ligeramente** peor: $O(m \log^* n)$.
 - El “logaritmo estrella” crece **casi tan despacio** como la función $\alpha(m,n)$:

$$\log^* 2 = 1$$

$$\log^* 4 = \log^* 2^2 = 2$$

$$\log^* 16 = \log^* 2^{2^2} = 3$$

$$\log^* 65536 = \log^* 2^{2^{2^2}} = 4$$

$$\log^* (2^{65536}) = \log^* 2^{2^{2^{2^2}}} = 5$$

el número estimado de átomos en el Universo observable es de unos 10^{80} , que es **mucho** menor que 2^{65536} ($\approx 2 \cdot 10^{19728}$), así que... ¡¡¡ difícilmente nos encontraremos (en este Universo) un n tal que $\log^* n > 5$!!!

(*) de que el coste en el caso peor para una secuencia de m operaciones con una ECD con n elementos distintos es $O(m \alpha(m,n))$



Estructuras de conjuntos disjuntos

- Algunas propiedades del rango de los árboles.
- [P_1] – Para todo nodo x , $rango(x) \leq rango(padre(x))$, y la desigualdad es estricta si $x \neq padre(x)$.
- [P_2] – El valor de $rango(x)$ es inicialmente 0 y crece con el tiempo hasta que $x \neq padre(x)$, desde entonces el valor de $rango(x)$ ya no cambia más.
- [P_3] – El valor de $rango(padre(x))$ es una función monótona creciente con respecto al tiempo.

Todas estas propiedades se deducen fácilmente de la implementación.



Estructuras de conjuntos disjuntos

- [P_4] – Si se define el *cardinal*(x) como el nº de nodos del árbol con raíz x (incluido x), entonces

$$\text{cardinal}(x) \geq 2^{\text{rango}(x)}$$

Demostración: por inducción en el número de operaciones de *enlazar* (la operación *encontrar* no modifica el cardinal ni el rango de la raíz del árbol).

- Base de inducción:

Antes de la primera operación de *enlazar* los rangos son 0 y cada árbol contiene al menos un nodo, luego

$$\text{cardinal}(x) \geq 1 = 2^0 = 2^{\text{rango}(x)}$$



Estructuras de conjuntos disjuntos

- Paso de inducción:

Suponer que es cierto antes de *enlazar* x e y que
 $\text{cardinal}(x) \geq 2^{\text{rango}(x)}$ y $\text{cardinal}(y) \geq 2^{\text{rango}(y)}$

- si $\text{rango}(x) \neq \text{rango}(y)$, asumir que $\text{rango}(x) < \text{rango}(y)$, entonces y es la raíz del árbol resultante y se tiene que:

$$\begin{aligned}\text{cardinal}'(y) &= \text{cardinal}(x) + \text{cardinal}(y) \geq \\ &\geq 2^{\text{rango}(x)} + 2^{\text{rango}(y)} \geq 2^{\text{rango}(y)} = 2^{\text{rango}'(y)},\end{aligned}$$

y no cambian más rangos ni cardinales

- si $\text{rango}(x) = \text{rango}(y)$, el nodo y es de nuevo la raíz del nuevo árbol y se tiene que:

$$\begin{aligned}\text{cardinal}'(y) &= \text{cardinal}(x) + \text{cardinal}(y) \geq \\ &\geq 2^{\text{rango}(x)} + 2^{\text{rango}(y)} = 2^{\text{rango}(y)+1} = 2^{\text{rango}'(y)}\end{aligned}$$



Estructuras de conjuntos disjuntos

- $[P_5]$ – Para todo entero $r \geq 0$, hay como máximo $n/2^r$ nodos de rango r .

Demostración:

- suponer que cuando se asigna un rango r a un nodo x (en los algoritmos *crear* o *enlazar*) se añade una etiqueta x a cada nodo del subárbol de raíz x
- por la propiedad $[P_4]$, se etiquetan como mínimo 2^r nodos cada vez
- si cambia la raíz del árbol que contiene el nodo x , por la propiedad $[P_1]$, el rango del nuevo árbol será, como mínimo, $r + 1$
- como asignamos etiquetas sólo cuando una raíz recibe el rango r , ningún nodo del nuevo árbol se etiquetará más veces, es decir, cada nodo se etiqueta como mucho una sola vez, cuando su raíz recibe por vez primera el rango r



Estructuras de conjuntos disjuntos

- como hay n nodos, hay como mucho n nodos etiquetados, con como mínimo 2^r etiquetas asignadas para cada nodo de rango r
 - si hubiese más de $n/2^r$ nodos de rango r , habría más de $2^r(n/2^r) = n$ nodos etiquetados por un nodo de rango r , lo cual es contradictorio
 - por tanto, hay como mucho $n/2^r$ nodos a los que se asigna un rango r
- Corolario: el rango de todo nodo es como máximo $\lfloor \log n \rfloor$

Demostración:

- si $r > \log n$, hay como mucho $n/2^r < 1$ nodos de rango r
- como el rango es un natural, se sigue el corolario



Estructuras de conjuntos disjuntos

- Demostraremos ahora con el método agregado que el coste amortizado para una secuencia de m operaciones con una ECD con n elementos distintos es $O(m \log^* n)$.
 - Primero veamos que se pueden considerar m operaciones de *crear*, ***enlazar*** y *encontrar* en lugar de m de *crear*, ***unir*** y *encontrar*:
 S' = sec. de m oper. de *crear*, *unir* y *encontrar*
 S = sec. obtenida de S' sustituyendo cada oper. de *unión* por dos de *encontrar* y una de *enlazar*
Si S está en $O(m \log^* n)$, entonces S' también esté en $O(m \log^* n)$.
Demostración:
 - cada *unión* en S' se convierte en 3 operaciones en S , luego $m' \leq m \leq 3m'$
 - como $m = O(m')$, una cota $O(m \log^* n)$ para la secuencia S implica una cota $O(m' \log^* n)$ para la secuencia original S' .



Estructuras de conjuntos disjuntos

- Veamos ahora el coste amortizado:
 - El coste de cada operación de *crear* y de *enlazar* es claramente $O(1)$.

```
algoritmo crear(x)
principio
    nuevóArbol(x) ;
    x.padre:=x;
    x.rango:=0
fin
```

```
algoritmo enlazar(x,y)
principio
    si x.rango>y.rango ent
        y.padre:=x
    sino
        x.padre:=y;
        si x.rango=y.rango ent y.rango:=y.rango+1 fsi
    fsi
fin
```



Estructuras de conjuntos disjuntos

- Veamos la operación *encontrar*, antes... algo de notación:

Bloque 0:

{0,1}

Bloque 1:

{2}

Bloque 2:

{3,4}

Bloque 3:

{5,...,16}

Bloque 4:

{17,...,65536}

Bloque 5:

{65537,...,BIG}

- Agrupamos los rangos de los nodos en *bloques*:
el rango r está en el bloque que numeramos con $\log^* r$, para $r = 0, 1, \dots, \lfloor \log n \rfloor$ (recordar que $\lfloor \log n \rfloor$ es el máximo rango).
- El bloque con mayor número es por tanto el numerado con $\log^* (\log n) = \log^* n - 1$.
- Notación: para todo entero $j \geq -1$,

$$B(j) = \begin{cases} -1, & \text{si } j = -1 \\ 1, & \text{si } j = 0 \\ 2, & \text{si } j = 1 \\ 2^{2^{\cdot^{\cdot^2}}}, & \text{si } j \geq 2 \end{cases}$$

$$\log^* 2 = 1$$

$$\log^* 4 = 2$$

$$\log^* 16 = 3$$

$$\log^* 65536 = 4$$

$$\log^* (2^{65536}) = 5$$

- Entonces, para $j = 0, 1, \dots, \log^* n - 1$, el bloque j -ésimo consiste en el conjunto de rangos $\{B(j-1)+1, B(j-1)+2, \dots, B(j)\}$.



Estructuras de conjuntos disjuntos

- Definimos dos tipos de “unidades de coste” asociadas a la operación *encontrar*: unidad de coste de bloque y unidad de coste de camino.
- Supongamos que *encontrar* empieza en el nodo x_0 y que el camino de búsqueda es x_0, x_1, \dots, x_l , con $x_i = \text{padre}(x_{i+1})$, $i = 0, 1, \dots, l-1$, y $x_l = \text{padre}(x_l)$, es decir x_l es una raíz.
- Asignamos una unidad de coste de bloque a:
 - el último^(*) nodo del camino de búsqueda cuyo rango está en el bloque j , para todo $j = 0, 1, \dots, \log^* n - 1$; es decir, a todo x_i tal que $\log^* \text{rango}(x_i) < \log^* \text{rango}(x_{i+1})$; y a
 - el hijo de la raíz, es decir, $x_i \neq x_l$ tal que $\text{padre}(x_i) = x_l$.
- Asignamos una unidad de coste de camino a:
 - cada nodo del camino de búsqueda al que no le asignamos unidad de coste de bloque.

(*) Nótese que por la propiedad P_1 , los nodos con rango en un bloque dado aparecen consecutivos en el camino de búsqueda.



Estructuras de conjuntos disjuntos

- Una vez que un nodo (distinto de la raíz o sus hijos) ha recibido en alguna operación de *encontrar* una unidad de coste de bloque, ya nunca (en sucesivas operaciones de *encontrar*) recibirá unidades de coste de camino.

Demostración:

- en cada compresión de caminos que afecta a un nodo x_i tal que $x_i \neq \text{padre}(x_i)$, el rango de ese nodo se mantiene, pero el nuevo padre de x_i tiene un rango estrictamente mayor que el que tenía el anterior padre de x_i ;
- por tanto, la diferencia entre los rangos de x_i y su padre es una función monótona creciente con respecto al tiempo;
- por tanto, también lo es la diferencia entre $\log^* \text{rango}(\text{padre}(x_i))$ y $\log^* \text{rango}(x_i)$;
- luego, una vez que x_i y su padre tienen rangos en distintos bloques, siempre tendrán rangos en distintos bloques, y por tanto nunca más se le volverá a asignar a x_i una unidad de coste de camino.



Estructuras de conjuntos disjuntos

- Como hemos asignado una unidad de coste para cada nodo visitado en cada operación de *encontrar*, el nº total de unidades de coste asignadas coincide con el nº total de nodos visitados; queremos demostrar que ese total es $O(m \log^* n)$.
- El nº total de unidades de coste de bloque es fácil de acotar: en cada ejecución de *encontrar* se asigna una unidad por cada bloque, más otra al hijo de la raíz;
como los bloques varían en $0, 1, \dots, \log^* n - 1$, entonces hay como mucho $\log^* n + 1$ unidades de coste de bloque asignadas en cada operación de *encontrar*, y por tanto no más de $m(\log^* n + 1)$ en toda la secuencia de m operaciones.
- Falta acotar el nº total de unidades de coste de camino...



Estructuras de conjuntos disjuntos

- Acotación del n° total de unidades de coste de camino:
 - Observar que si a un nodo x_i se le asigna una unidad de coste de camino entonces $\text{padre}(x_i) \neq x_i$ antes de la compresión, luego a x_i se le asignará un nuevo padre en la compresión, y el nuevo padre tendrá un rango mayor que el anterior padre.
 - Supongamos que el rango de x_i está en el bloque j .
 - ¿Cuántas veces se le puede asignar a x_i un nuevo padre (y por tanto asignarle una unidad de coste de camino), antes de asignarle un padre cuyo rango esté en un bloque distinto (después de lo cual ya nunca se le asignarán más unidades de coste de camino)?
 - » Ese n° es máximo si x_i tiene el rango más pequeño de su bloque, $B(j-1)+1$, y el rango de sus padres va tomando los valores $B(j-1)+2, B(j-1)+3, \dots, B(j)$.
 - » Es decir, puede ocurrir $B(j) - B(j-1) - 1$ veces.



Estructuras de conjuntos disjuntos

- Siguiente paso: acotar el nº de nodos cuyo rango está en el bloque j , para cada $j \geq 0$ (recordar, propiedad P_2 , que el rango de un nodo es fijo una vez que pasa a ser hijo de otro nodo).

- » $N(j)$ = nº nodos cuyo rango es del bloque j .

- » Por la propiedad P_5 :

$$N(j) \leq \sum_{r=B(j-1)+1}^{B(j)} \frac{n}{2^r}$$

- » Para $j = 0$: $N(0) = n/2^0 + n/2^1 = 3n/2 = 3n/2B(0)$

- » Para $j \geq 1$:

$$N(j) \leq \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{B(j)-(B(j-1)+1)} \frac{1}{2^r}$$

$$< \frac{n}{2^{B(j-1)+1}} \sum_{r=0}^{\infty} \frac{1}{2^r}$$

$$= \frac{n}{2^{B(j-1)}} = \frac{n}{B(j)}$$

- » Por tanto: $N(j) \leq 3n/2B(j)$, para todo entero $j \geq 0$.

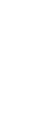


Estructuras de conjuntos disjuntos

- Sumando para todos los bloques el producto del máximo nº de nodos con rango en ese bloque por el máximo nº de unidades de coste de camino por nodo de ese bloque, se tiene:

$$\begin{aligned} P(n) &\leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} (B(j) - B(j-1) - 1) \\ &\leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} B(j) = \frac{3}{2} n \log^* n \end{aligned}$$

- Por tanto, el nº máximo de unidades de coste por todas las operaciones de *encontrar* es $O(m(\log^* n + 1) + n \log^* n)$, es decir, $O(m \log^* n)$, porque $m \geq n$.
- Como el número total de operaciones de *crear* y de *enlazar* es $O(n)$, el coste total de las m operaciones con una ECD con n elementos distintos es $O(m \log^* n)$.



Análisis amortizado

- El plan:
 - Conceptos básicos:
 - Método agregado
 - Método contable
 - Método potencial
 - Primer ejemplo: análisis de tablas *hash* dinámicas
 - Montículos agregables (binomiales y de Fibonacci)
 - Estructuras de conjuntos disjuntos
 - **Listas lineales auto-organizativas**
 - Árboles auto-organizativos ("splay trees")



Listas lineales auto-organizativas

- Motivación:
 - En el uso de tablas de símbolos:
localidad de referencia = tendencia a arracimarse los accesos a un símbolo determinado en una secuencia de operaciones.
Ejemplos:
 - En un programa: $n := n + 1 \rightarrow$ dos accesos al símbolo n
 - Clientes de un banco en una visita a su oficina o a un cajero automático: varias operaciones sobre su cuenta en una sola visita (y luego varias semanas sin volver a operar)
 - Si se conoce de antemano esta localidad de referencia: adaptar la implementación para mejorar la eficiencia
 \rightarrow estructuras de datos auto-organizativas



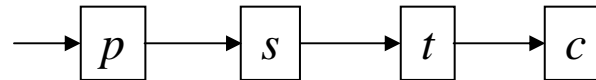
Listas lineales auto-organizativas

- Forma sencilla de “auto-organización”:
 - Recordar la dirección del último dato “accedido” (consultado o insertado)
 - Si se solicita una nueva búsqueda, en primer lugar comparar con el último dato accedido, y si éste es el buscado se evita la búsqueda en la estructura de datos
 - Se pueden almacenar no una sino varias direcciones (de los últimos datos accedidos)
 - En arquitectura de computadores eso se llama memoria *cache*

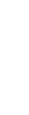


Listas lineales auto-organizativas

- Aplicaremos la idea anterior a listas encadenadas no ordenadas



- Para insertar un dato:
 - recorrer la lista entera para asegurarse de que el dato no está y
 - si el dato no está, añadirlo al final (tras el dato c).
 - Diremos que el coste es $n + 1$ (si había n datos en la lista), aunque el número real de comparaciones es n .
- Para buscar un dato:
 - se recorre la lista secuencialmente.
 - El coste de encontrar el dato i -ésimo es i (comparaciones entre claves).



Listas lineales auto-organizativas

- Heurísticas para mejorar el rendimiento:
 - 1) *mover al frente*: después de acceder a un dato, colocarlo el primero de la lista
 - 2) *trasponer*: después de acceder a un dato, si no es el primero, intercambiarlo con el predecesor en la lista
 - 3) *contar frecuencias*: guardar para cada dato el número de veces que se ha accedido a él y mantener la lista ordenada (de manera no creciente) según estos números
- Se llama lista auto-organizativa a una lista encadenada dotada de alguna de estas heurísticas (o de otra similar)



Listas lineales auto-organizativas

- Comparación de las tres heurísticas:
 - Mover al frente
 - es “optimista”: cree de verdad que un dato accedido será accedido de nuevo y pronto, y si esto ocurre, el segundo acceso tiene un coste $O(1)$
 - la ordenación de la lista se aproxima muy rápidamente a la ordenación óptima (el dato recién accedido se coloca el primero)
 - si ese dato luego no se vuelve a consultar en mucho tiempo, muy rápidamente retrocede en la lista y se coloca al final
 - Trasponer
 - es más “pesimista”: se duda de que el dato recién accedido vaya a ser consultado de nuevo, así que, “tímidamente” se le adelanta una posición
 - Contar frecuencias
 - se basa la decisión en toda la historia de operaciones con la secuencia
 - es una especie de compromiso entre las otras dos heurísticas



Listas lineales auto-organizativas

- El coste en el caso peor en cualquier lista auto-organizativa es siempre $O(n)$:
 - Siempre hay un último elemento en la lista y ese podría ser el accedido en un momento dado
- Por este motivo no se usan para implementar tablas de símbolos en compilación
- Si que se suelen usar para implementar las listas de desbordamiento para la resolución de colisiones por encadenamiento en una tabla dispersa



Listas lineales auto-organizativas

- Análisis:
 - Terminología
 - Heurística *admisible* para listas auto-organizativas:
 - Si tiene la forma “después de acceder a un dato, moverlo un lugar o más hacia el principio de la lista (o dejarlo igual)”
 - *Mover al frente*, *trasponer* y *contar frecuencias* son tres heurísticas admisibles
 - Llamamos *intercambio* al movimiento de un dato *una* posición hacia su izquierda
 - Para una heurística H y una secuencia p de operaciones de manipulación de la lista (búsqueda o inserción):
 - $C_H(p)$ es el coste total de toda la secuencia de operaciones aplicadas a una lista auto-organizativa con heurística H
 - $E_H(p)$ es el número total de intercambios realizados



Listas lineales auto-organizativas

– *Teorema:*

Para toda heurística admisible H y toda secuencia p de m operaciones de inserción y de búsquedas con éxito empezando con una lista vacía se tiene:

$$C_{MF}(p) \leq 2 C_H(p) - E_H(p) - m$$

donde MF es la heurística de mover al frente.

- Es decir, el coste total con la heurística de mover al frente nunca es más del doble del coste con cualquier otra heurística admisible



Listas lineales auto-organizativas

– Demostración del teorema

- Se trata de ejecutar en paralelo ambas heurísticas, MF y H , para la secuencia p , y comparar los costes
- En cada instante ambas listas contendrán los mismos datos pero en distinto orden

	MF	T (trasponer)
insertar(a)	a	a
insertar(b)	b, a	b, a
insertar(c)	c, b, a	b, c, a
insertar(d)	d, c, b, a	b, c, d, a
buscar(d)	d, c, b, a	b, d, c, a
buscar(a)	a, d, c, b	b, d, a, c
buscar(a)	a, d, c, b	b, a, d, c



Listas lineales auto-organizativas

	MF	T (trasponer)
insertar(a)	a	a
insertar(b)	b, a	b, a
insertar(c)	c, b, a	b, c, a
insertar(d)	d, c, b, a	b, c, d, a
buscar(d)	d, c, b, a	b, d, c, a
buscar(a)	a, d, c, b	b, d, a, c
buscar(a)	a, d, c, b	b, a, d, c

$$C_{MF}(p) = 1 + 2 + 3 + 4 + 1 + 4 + 1 = 16$$

$$C_T(p) = 1 + 2 + 3 + 4 + 3 + 4 + 3 = 20$$

$$E_T(p) = 0 + 1 + 1 + 1 + 1 + 1 + 1 = 6$$

$$m = 7$$

$$16 \leq 2 \cdot 20 - 6 - 7$$

- Si los datos están en igual orden en ambas listas el coste de la siguiente operación será el mismo con ambas heurísticas
- Las diferencias surgen si el orden de los datos es muy diferente



Listas lineales auto-organizativas

- Una *inversión* es un par de datos distintos (x,y) tales que x precede a y en una lista y que y precede a x en la otra

- En las dos listas finales del ejemplo:
 (a,b) , (d,b) , (c,b)

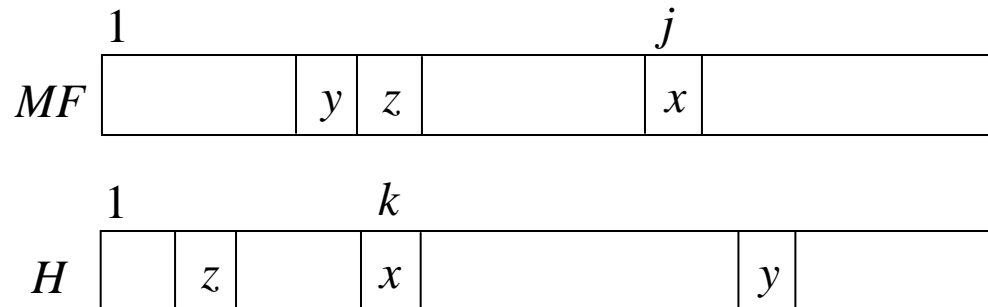
	MF	T (trasponer)
insertar(a)	a	a
insertar(b)	b,a	b,a
insertar(c)	c,b,a	b,c,a
insertar(d)	d,c,b,a	b,c,d,a
buscar(d)	d,c,b,a	b,d,c,a
buscar(a)	a,d,c,b	b,d,a,c
buscar(a)	a,d,c,b	b,a,d,c

- El número total de inversiones es una medida de cuánto difiere el orden



Listas lineales auto-organizativas

- Cálculo del coste total amortizado de la secuencia de operaciones p con la heurística mover al frente:
 - Mediante el método potencial, usaremos como función de potencial el número de inversiones entre la lista MF y la lista H
 - Suponer que la i -ésima operación es ‘buscar(x)’ y que x es el j -ésimo elemento de la lista MF y el k -ésimo de la lista H

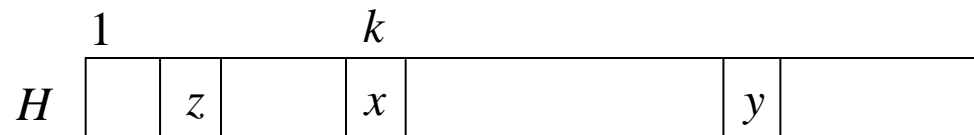
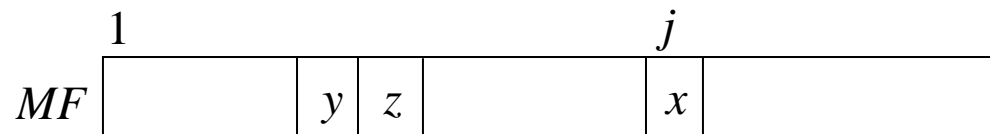


- El coste de la búsqueda (en MF) es j
- Después x se desplaza al primer lugar en MF y e_i lugares hacia la izquierda en H



Listas lineales auto-organizativas

- Efecto en el potencial P del movimiento de x en MF :
 - » x se adelanta a $j - 1$ datos
 - » c de esos datos, como y , que formaban una inversión con x antes del movimiento ya no la forman
 - » el resto de los datos, $j - 1 - c$, que, como z , no formaban inversión, ahora si la forman



- » con x al frente de MF , todo dato que en H preceda a x forma una inversión con él
- » al mover x en H e_i posiciones a la izquierda e_i de las inversiones se destruyen



Listas lineales auto-organizativas

- Según el método potencial: *(Análisis de la búsqueda)*

$$\begin{aligned} A(i) &= C(i) + P(i) - P(i-1) \\ &= j - c + (j - 1 - c) - e_i = 2(j - 1 - c) - e_i + 1 \end{aligned}$$

- Si en MF había $j - 1 - c$ datos que no formaban inversión con x , entonces esos $j - 1 - c$ datos debían preceder a x también en H , por tanto $j - 1 - c \leq k - 1$, y

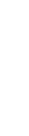
$$A(i) \leq 2(k - 1) - e_i + 1 = 2k - e_i - 1$$

- El análisis de la inserción es muy parecido, se añade primero el dato al final de ambas listas, lo que no cambia el potencial y después se procede como en una búsqueda

- Sumando: $C_{MF}(p) \leq \sum_{i=1}^m A(i) \leq 2C_H(p) - E_H(p) - m$

por definición de
coste amortizado

k = coste de **una** operación
con H porque es la posición
hasta la que se llega en la búsqueda



Análisis amortizado

- El plan:
 - Conceptos básicos:
 - Método agregado
 - Método contable
 - Método potencial
 - Primer ejemplo: análisis de tablas *hash* dinámicas
 - Montículos agregables (binomiales y de Fibonacci)
 - Estructuras de conjuntos disjuntos
 - Listas lineales auto-organizativas
 - **Árboles auto-organizativos ("splay trees")**



Árboles auto-organizativos

- Ideas básicas de los *splay trees*:
 - Árbol binario de búsqueda para almacenar conjuntos de elementos para los que existe una relación de orden, con las operaciones de búsqueda, inserción, borrado, y algunas otras...
 - Renunciar a un equilibrado “estricto” tras cada operación como el de los árboles AVL, los 2-3, los B, o los rojinegros
 - No son equilibrados; la altura puede llegar a ser $n - 1$
 - ¡Que el coste en el caso peor de una operación con un árbol binario de búsqueda sea $O(n)$ no es tan malo, siempre que eso ocurra con poca frecuencia!
 - Al realizar cada operación, el árbol tiende a equilibrarse, de manera que el coste amortizado de cada operación sea $O(\log n)$



Árboles auto-organizativos

- Observaciones:
 - Si una operación, por ejemplo de búsqueda, puede tener un coste de $O(n)$ en el caso peor, y se quiere conseguir un coste amortizado de $O(\log n)$ para todas las operaciones, entonces es imprescindible que los nodos a los que se accede con la búsqueda se muevan tras la búsqueda
 - En otro caso, bastaría con repetir esa búsqueda un total de m veces para obtener un coste amortizado total de $O(mn)$ y por tanto no se conseguiría $O(\log n)$ amortizado para cada operación
 - Después de acceder a un nodo, lo empujaremos hacia la raíz mediante rotaciones (similares a las usadas con árboles AVL o rojinegros)
 - La re-estructuración depende sólo del camino recorrido en el acceso al nodo
 - Igual que en las listas auto-organizativas, los accesos posteriores serán más rápidos



Árboles auto-organizativos

- Operación básica: *splay*
 - $\text{splay}(i, S)$: reorganiza el árbol S de manera que la nueva raíz es:
 - el elemento i si i pertenece a S , o
 - $\max\{k \in S \mid k < i\}$ ó $\min\{k \in S \mid k > i\}$, si i no pertenece a S
- Las otras operaciones:
 - $\text{buscar}(i, S)$: dice si i está en S
 - $\text{insertar}(i, S)$: inserta i en S si no estaba ya
 - $\text{borrar}(i, S)$: borra i de S si estaba
 - $\text{concatenar}(S, S')$: une S y S' en un solo árbol, suponiendo que $x < y$ para todo $x \in S$ y todo $y \in S'$
 - $\text{separar}(i, S)$: separa S en S' y S'' tales que $x \leq i \leq y$ para todo $x \in S'$ y todo $y \in S''$



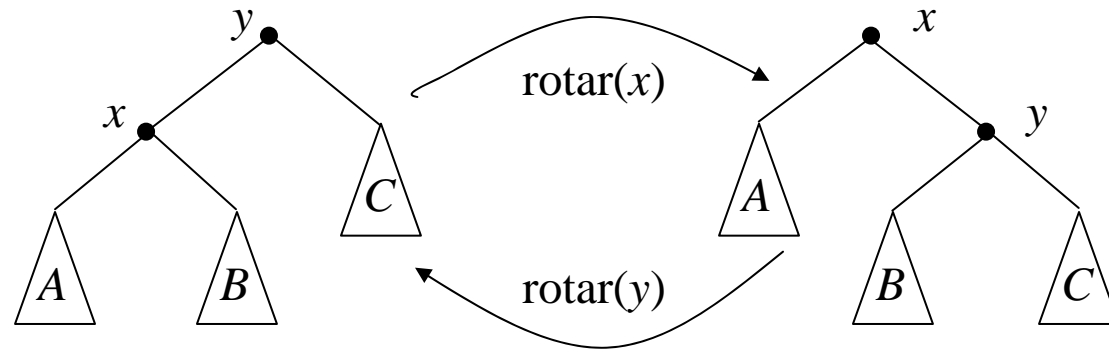
Árboles auto-organizativos

- Todas las operaciones pueden implementarse con un número constante de *splays* más un número constante de operaciones sencillas (comparaciones y asignaciones de punteros)
 - Ejemplo: concatenar(S, S') puede conseguirse mediante $splay(+\infty, S)$ que deja el elemento mayor de S en su raíz y el resto en su subárbol izquierdo y después haciendo que S' sea el subárbol derecho de la raíz de S
 - Otro ejemplo: borrar(i, S) puede hacerse mediante $splay(i, S)$ para colocar i en la raíz, después se borra i y se ejecuta concatenar para unir los subárboles izquierdo y derecho



Árboles auto-organizativos

- Implementación de la operación *splay*
 - Recordar las rotaciones:



- Preservan la estructura de árbol de búsqueda



Árboles auto-organizativos

- Una primera posibilidad para subir un nodo i hasta la raíz es rotarlo repetidamente
 - No sirve: puede provocar que otros nodos bajen demasiado
 - Se puede demostrar que hay una secuencia de m operaciones que requieren $\Omega(mn)$:
 - generar un árbol insertando las claves $1, 2, 3, \dots, n$ en uno vacío, llevando el último nodo insertado hasta la raíz mediante rotaciones
 - » el árbol resultante sólo tiene hijos izquierdos
 - » el coste total hasta ahora es $O(n)$
 - buscar la clave 1 precisa n operaciones, después la clave 1 se lleva hasta la raíz mediante rotaciones ($n - 1$ rotaciones)
 - buscar la clave 2 precisa n operaciones y luego $n - 1$ rotaciones
 - iterando el proceso, la búsqueda de todas las claves en orden requiere
$$\sum_{i=1}^{n-1} i = \Omega(n^2)$$
 - tras todas las búsquedas el árbol ha quedado como al principio, así que si se vuelve a repetir la secuencia de búsquedas en orden, se llega a $\Omega(mn)$ para m operaciones.



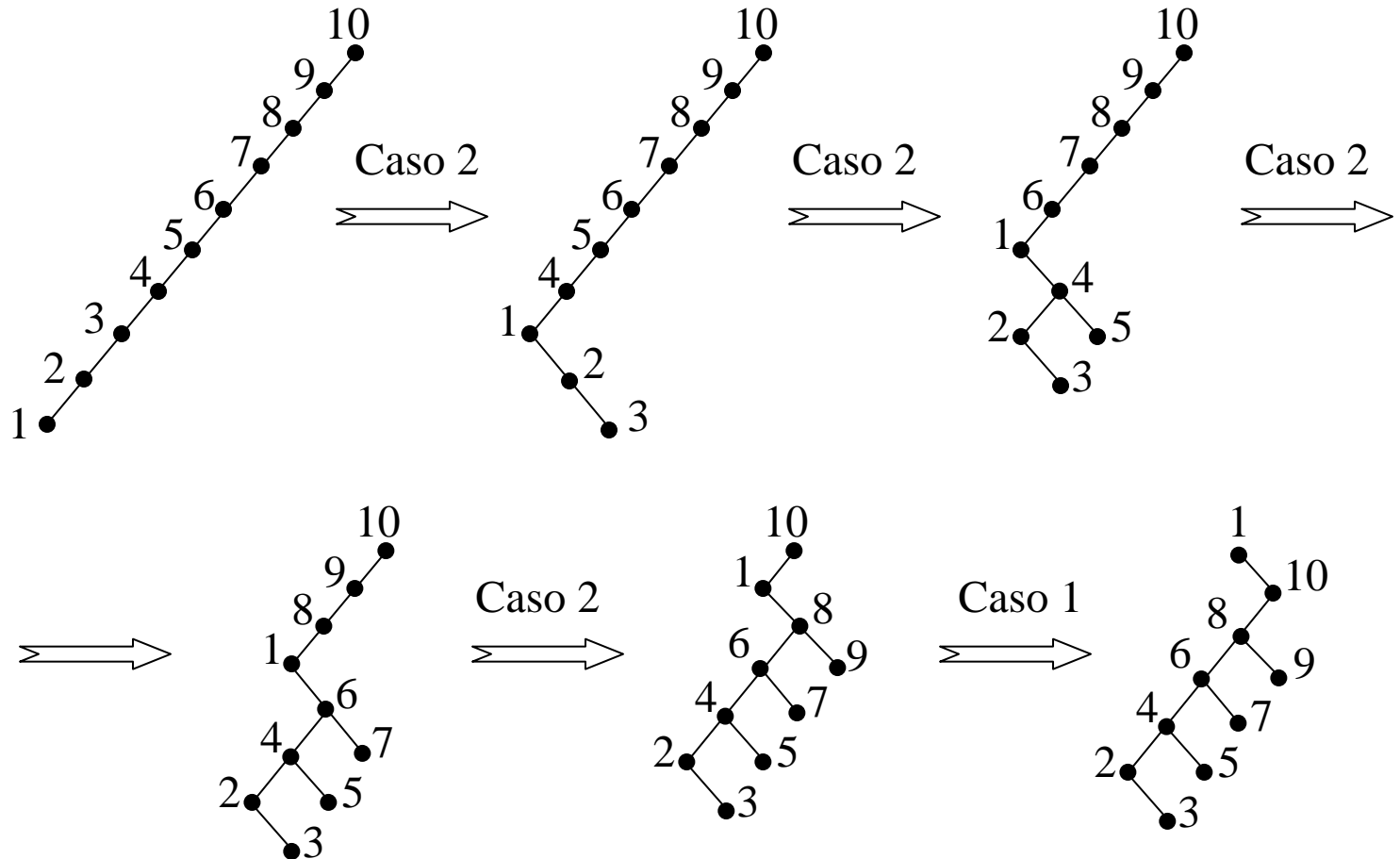
Árboles auto-organizativos

- La solución adoptada (para subir x a la raíz) es la siguiente:
 - **Caso 1:** si x tiene padre pero no tiene abuelo, se ejecuta $\text{rotar}(x)$ (con su padre)
 - **Caso 2:** si x tiene padre (y), y tiene abuelo, y tanto x como y son ambos hijos izquierdos o ambos hijos derechos, entonces se ejecuta $\text{rotar}(y)$ y luego $\text{rotar}(x)$ (con el padre en ambos casos)
 - **Caso 3:** si x tiene padre (y), y tiene abuelo, y x e y son uno hijo derecho y otro izquierdo, entonces se ejecuta $\text{rotar}(x)$ y luego $\text{rotar}(x)$ otra vez (con el padre en ambos casos)



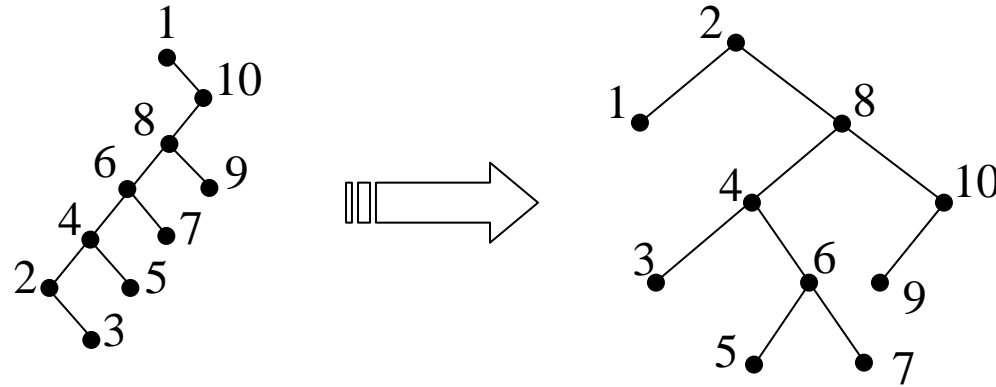
Árboles auto-organizativos

- Ejemplo: $\text{splay}(1, S)$, con S el árbol degenerado



Árboles auto-organizativos

- Sigue el ejemplo: $\text{splay}(2, S)$ al resultado anterior...



- Nótese que el árbol está más equilibrado con cada *splay*

Árboles auto-organizativos

- Análisis del coste de *splay* mediante el método contable (o potencial, que es más o menos igual):
 - Cada nodo x lleva asociado un crédito
 - Cuando se crea x el precio de esa creación se asocia a x como crédito, y se usará para re-estructuraciones posteriores
 - Sea $S(x)$ el subárbol de raíz x y $|S|$ su número de nodos
 - Sean $\mu(S) = \lfloor \log |S| \rfloor$ y $\mu(x) = \mu(S(x))$
 - Exigiremos el siguiente *invariante del crédito* (que no es otra cosa que la definición del potencial):

El nodo x siempre tiene como mínimo un crédito $\mu(x)$ (de esta forma el potencial es siempre no negativo)



Árboles auto-organizativos

- **Lema:** Cada operación *splay* requiere a lo sumo un precio igual a

$$3(\mu(S) - \mu(x)) + 1$$

unidades para ejecutarse y mantener el invariante del crédito.

Demostración:

- Sea y el padre de x y z , si existe, el padre de y .
- Sean μ y μ' los valores de μ antes y después del *splay*.
- Hay tres casos (por la forma de definir el *splay*):
 - (a) z no existe
 - (b) x e y son ambos hijos izquierdos o ambos hijos derechos
 - (c) x e y son uno hijo derecho y otro hijo izquierdo



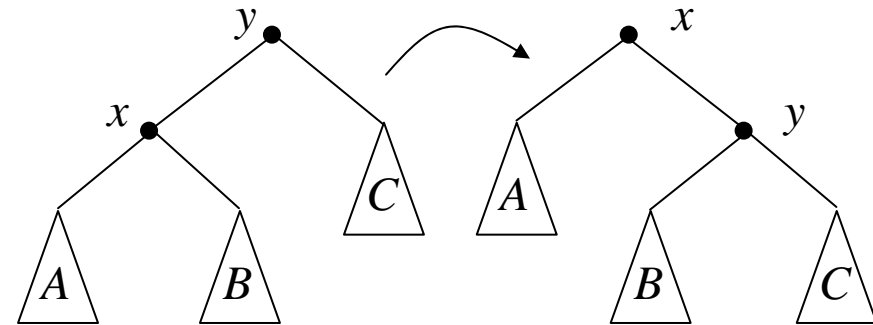
Árboles auto-organizativos

- **Caso 1:** z no existe

- Es la última rotación en la operación de *splay*
- Se ejecuta $\text{rotar}(x)$ y se tiene:

$$\mu'(x) = \mu(y)$$

$$\mu'(y) \leq \mu'(x)$$



- Para mantener el invariante del crédito hay que gastar:

incremento del
potencial

$$\mu'(x) + \mu'(y) - \mu(x) - \mu(y) = \mu'(y) - \mu(x)$$

$$\leq \mu'(x) - \mu(x) \leq 3(\mu'(x) - \mu(x)) = 3(\mu(S) - \mu(x))$$

- Se añade una unidad de crédito por las operaciones de coste constante (comparaciones y manipulaciones de punteros)
- Por tanto, se paga a lo sumo (coste amortizado, $A(i)$)

recordar que

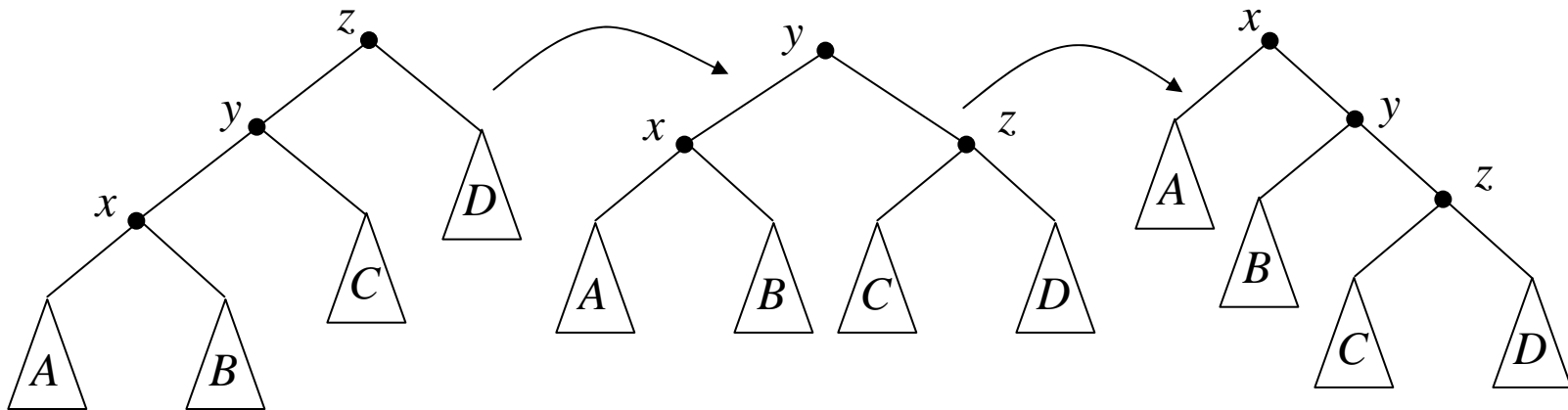
$$A(i) = C(i) + P(i) - P(i-1)$$

$$3(\mu'(x) - \mu(x)) + 1$$

unidades de crédito por la rotación

Árboles auto-organizativos

- **Caso 2:** x e y son ambos hijos izquierdos o hijos derechos
 - Se ejecuta $\text{rotar}(y)$ y luego $\text{rotar}(x)$



- Veremos primero que el precio de estas dos rotaciones para mantener el invariante no es mayor que $3(\mu'(x) - \mu(x))$
- Después, si se ejecuta una secuencia de ellas para subir x a la raíz, resulta una suma telescópica, y el coste total no es mayor que $3(\mu(S) - \mu(x)) + 1$ (el '+1' viene de la última rotación)



Árboles auto-organizativos

– Precio de las dos rotaciones:

» Para mantener el invariante se precisan

$\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z)$ unidades (*)

» Como $\mu'(x) = \mu(z)$, se tiene:

$$\begin{aligned}\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z) \\&= \mu'(y) + \mu'(z) - \mu(x) - \mu(y) \\&= (\mu'(y) - \mu(x)) + (\mu'(z) - \mu(y)) \\&\leq (\mu'(x) - \mu(x)) + (\mu'(x) - \mu(x)) \\&= 2(\mu'(x) - \mu(x))\end{aligned}$$

– Aún sobrarían $\mu'(x) - \mu(x)$ unidades para pagar las operaciones elementales de las dos rotaciones (comparaciones y manipulación de punteros), salvo que $\mu'(x) = \mu(x) \dots$

Árboles auto-organizativos

- Veamos que en este caso ($\mu'(x) = \mu(x)$) la cantidad precisa para mantener el invariante (*) es negativa y por tanto es “gratis” mantenerlo:

$$\left. \begin{array}{l} \mu'(x) = \mu(x) \\ \mu'(x) + \mu'(y) + \mu'(z) \geq \mu(x) + \mu(y) + \mu(z) \end{array} \right\} \Rightarrow \text{contradicción}$$

En efecto: $\mu(z) = \mu'(x) = \mu(x)$

$$\Rightarrow \mu(x) = \mu(y) = \mu(z)$$

Por tanto: $\mu'(x) + \mu'(y) + \mu'(z) \geq 3\mu(z) = 3\mu'(x)$

$$\Rightarrow \mu'(y) + \mu'(z) \geq 2\mu'(x)$$

Y como $\mu'(y) \leq \mu'(x)$ y $\mu'(z) \leq \mu'(x)$ entonces

$$\mu'(x) = \mu'(y) = \mu'(z)$$

Y como $\mu(z) = \mu'(x)$ entonces

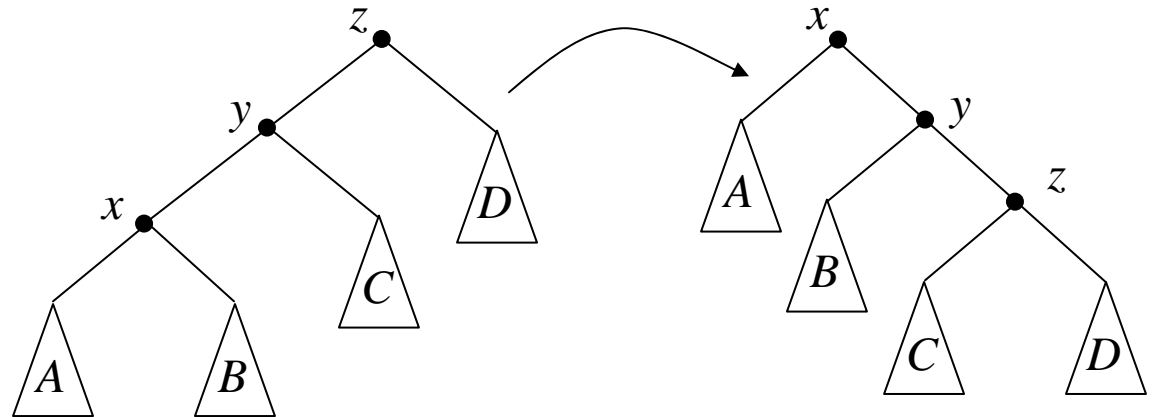
$$\mu(x) = \mu(y) = \mu(z) = \mu'(x) = \mu'(y) = \mu'(z)$$



Árboles auto-organizativos

Pero $\mu(x) = \mu(y) = \mu(z) = \mu'(x) = \mu'(y) = \mu'(z)$ es imposible por la propia definición de μ y μ' ...

» En efecto, si $a = |S(x)|$ y $b = |S'(z)|$, entonces se tendría
 $\lfloor \log a \rfloor = \lfloor \log (a + b + 1) \rfloor = \lfloor \log b \rfloor$

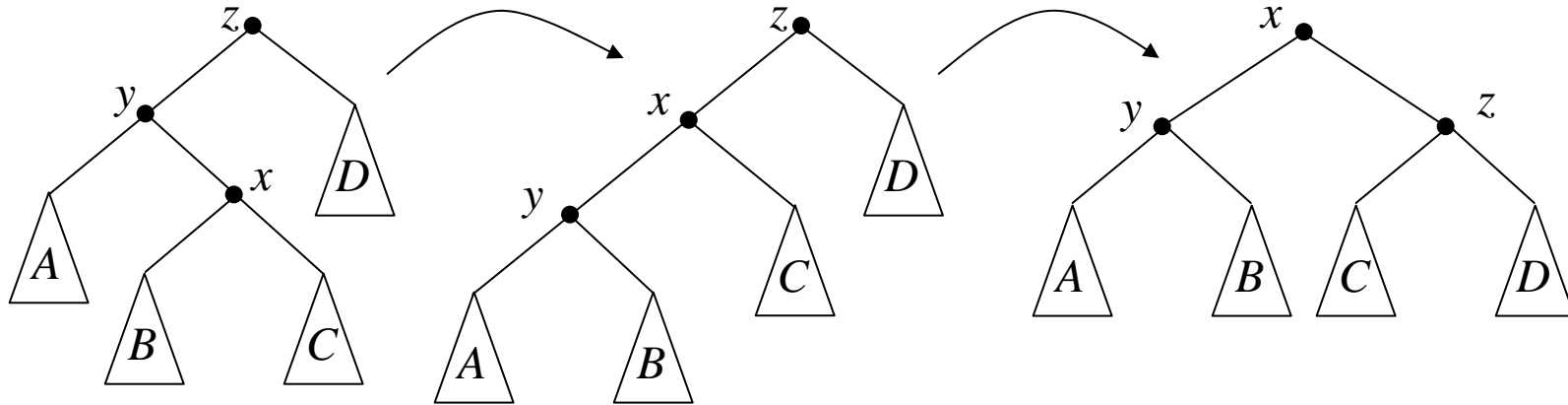


y asumiendo que $a \leq b$ (el otro caso es igual),
 $\lfloor \log (a + b + 1) \rfloor \geq \lfloor \log 2a \rfloor = 1 + \lfloor \log a \rfloor > \lfloor \log a \rfloor$

Por tanto se llega a contradicción.

Árboles auto-organizativos

- **Caso 3:** x e y son uno hijo derecho y otro hijo izquierdo
 - Se ejecuta dos veces $\text{rotar}(x)$



- Igual que en el caso anterior, el precio de estas dos rotaciones para mantener el invariante no es mayor que $3(\mu'(x) - \mu(x))$
- La demostración es similar

Árboles auto-organizativos

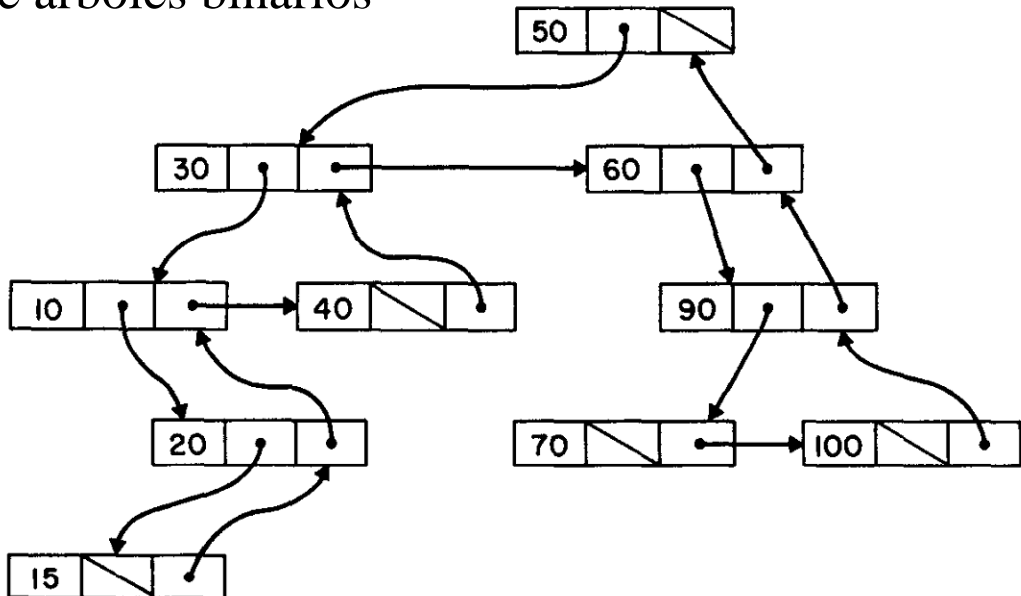
- Resapitulando:
 - El precio ^(*) de cada operación de *splay* está acotado por $3\lfloor \log n \rfloor + 1$, con n el número de datos del árbol
 - El resto de operaciones se pueden implementar con un número constante de *splays* más un número constante de operaciones sencillas (comparaciones y asignaciones de punteros)
 - Por tanto: el tiempo total requerido para una secuencia de m operaciones con un árbol auto-organizativo es $O(m \log n)$, con n el número de operaciones de inserción y de concatenación

^(*) coste amortizado



Árboles auto-organizativos

- Detalles de implementación
 - Se ha descrito el *splay* como una operación desde abajo hacia arriba (*bottom-up splay trees*)
 - Es lo más útil si se tenía acceso directo al nodo a *despatarrar*
 - Se requiere puntero al padre (➔ 3 punteros por nodo) o bien... representación “hijo más a la izquierda – hermano derecho” de árboles binarios



Árboles auto-organizativos

- Detalles de implementación (cont.)
 - En general, la implementación más conveniente lo hace de arriba hacia abajo (*top-down splay trees*)
 - Mientras se busca un dato o la posición para insertarlo se puede ir haciendo el *splay* del camino
 - En las transparencias siguientes se incluye una posible implementación tomada –más o menos– del libro de M.A. Weiss [Wei99]
 - Puede encontrarse alguna explicación sobre el código en ese libro o en el artículo de Sleator y Tarjan ([“Self-adjusting binary search trees”](#))



Árboles auto-organizativos

```
generic
  type dato is private;
  with function "<"(d1,d2:dato) return boolean;
package splay_tree_modulo is
  type splay_tree is limited private;
  procedure crear(s:in out splay_tree);
  procedure buscar(d:dato; s:in out splay_tree; éxito:out boolean);
  procedure insertar(d:dato; s:in out splay_tree);
  procedure borrar(d:dato; s:in out splay_tree);
private
  type nodo;
  type ptNodo is access nodo;
  type splay_tree is
    record laRaiz:ptNodo; nodoNulo:ptNodo; end record;
    -- nodoNulo es un centinela para simplificar el código
  type nodo is
    record elDato:dato;
      izq:ptNodo; der:ptNodo; end record;
end splay_tree_modulo;
```



Árboles auto-organizativos

```
procedure crear(s:in out splay_tree) is
begin
    s.nodoNulo:=new nodo;
    s.nodoNulo.izq:=s.nodoNulo;
    s.nodoNulo.der:=s.nodoNulo;
    s.laRaiz:=s.nodoNulo;
end crear;
```

```
procedure buscar(d:dato; s:in out splay_tree; exito:out boolean) is
begin
    if d.laRaiz/=d.nodoNulo then
        exito:= false;
    else
        splay(d,s.laRaiz,s.nodoNulo);
        exito:=(s.laRaiz.elDato=d);
    end if;
end buscar;
```



Árboles auto-organizativos

```
procedure insertar(d: dato; s: in out splay_tree) is
    nuevoNodo: ptNodo := null;
begin
    nuevoNodo := new nodo;
    nuevoNodo.elDato := d;
    nuevoNodo.izq := null;
    nuevoNodo.der := null;
    if s.laRaiz = s.nodoNulo then
        nuevoNodo.izq := s.nodoNulo;
        nuevoNodo.der := s.nodoNulo;
        s.laRaiz := nuevoNodo;
    else
        ...
```



Árboles auto-organizativos

```
...
else
  splay(d,s.laRaiz,s.nodoNulo);
  if d<s.laRaiz.elDato then
    nuevoNodo.izq:=s.laRaiz.izq;
    nuevoNodo.der:=s.laRaiz;
    s.laRaiz.izq:=s.nodoNulo;
    s.laRaiz:=nuevoNodo;
  elsif s.laRaiz.elDato<d then
    nuevoNodo.der:=s.laRaiz.der;
    nuevoNodo.izq:=s.laRaiz;
    s.laRaiz.der:=s.nodoNulo;
    s.laRaiz:=nuevoNodo;
  else
    return;  -- no se duplican datos
  end if;
end if;
end insertar;
```



Árboles auto-organizativos

```
procedure liberar is new unchecked_deallocation(nodo,ptNodo);

procedure borrar(d:dato; s:in out splay_tree) is
    nuevoArbol:ptNodo; éxito:boolean;
begin
    buscar(d,s,exito);  -- splay de d a la raíz
    if éxito then
        if s.laRaiz.izq=s.nodoNulo then
            nuevoArbol:=s.laRaiz.der;
        else
            -- buscar el máximo en el subárbol izquierdo y hacerle
            -- splay a la raíz y añadirle el hijo derecho
            nuevoArbol:=s.laRaiz.izq;
            splay(d,nuevoArbol,s.nodoNulo);
            nuevoArbol.der:=s.laRaiz.der;
        end if;
        liberar(s.laRaiz);
        s.laRaiz :=nuevoArbol;
    end if;
end borrar;
```



Árboles auto-organizativos

```
cabeza:ptNodo:=new nodo;
procedure splay(d:dato; p:in out ptNodo; elNodoNulo:in out ptNodo) is
    max_hijo_izq:ptNodo:=cabeza; min_hijo_der:ptNodo:=cabeza;
begin
    cabeza.izq:=elNodoNulo; cabeza.der:=elNodoNulo; elNodoNulo.elDato:=d;
    loop
        if d<p.elDato then
            if d<p.izq.elDato then rotar_con_hijo_izq(p); end if;
            exit when p.izq=elNodoNulo;
            min_hijo_der.izq:=p; min_hijo_der:=p; p:=p.izq;
        elsif p.elDato<d then
            if p.der.elDato<d then rotar_con_hijo_der(p); end if;
            exit when p.der=elNodoNulo;
            max_hijo_izq.der:=p; max_hijo_izq:=p; p:=p.der;
        else exit;
        end if;
    end loop;
    max_hijo_izq.der:=p.izq; min_hijo_der.izq:=p.der;
    p.izq:=cabeza.der; p.der:=cabeza.izq;
end splay;
```



Árboles auto-organizativos

```
function "="(d1,d2:dato) return boolean is
begin
    return not(d2<d1) and then not(d1<d2);
end "=";
```

```
procedure rotar_con_hijo_izq(p2:in out ptNodo) is
    p1:ptNodo:=p2.izq;
begin
    p2.izq:=p1.der;
    p1.der:=p2;
    p2:=p1;
end rotar_con_hijo_izq;
```

```
procedure rotar_con_hijo_der(p1:in out ptNodo) is
    p2:ptNodo:=p1.der;
begin
    p1.der:=p2.izq;
    p2.izq:=p1;
    p1:=p2;
end rotar_con_hijo_der;
```



Introducción a los algoritmos de biología computacional



Introducción a los algoritmos de biología computacional



Introducción a la biología computacional

- El plan:
 - **Algoritmos de reconocimiento de patrones:**
 - Knuth-Morris-Pratt
 - Boyer-Moore
 - Árboles de sufijos
 - Primeras aplicaciones de los árboles de sufijos



Reconocimiento de patrones

- El problema de la subcadena o reconocimiento exacto de un patrón:

- Dados una cadena madre de n caracteres

$$S = 's_1 s_2 \dots s_n'$$

y un patrón de m caracteres ($n \geq m$)


$$P = 'p_1 p_2 \dots p_m'$$

se quiere saber si P es una subcadena de S y, en caso afirmativo, en qué posición(es) de S se encuentra.

- Instrucción crítica para medir la eficiencia de las soluciones:
 - número de comparaciones entre pares de caracteres



Reconocimiento de patrones

- Importancia del problema:
 - Imprescindible en gran número de aplicaciones:
 - Procesadores de textos,
 - utilidades como el *grep* de unix,
 - programas de recuperación de información textual,
 - programas de búsqueda en catálogos de bibliotecas,
 - buscadores de internet,
 - lectores de news en internet,
 - bibliotecas digitales,
 - revistas electrónicas,
 - directorios telefónicos,
 - enciclopedias electrónicas,
 - • búsquedas en bases de datos de secuencias de ADN o ARN,
 - ...
 - Problema bien resuelto en determinados casos pero...



The anatomy of a genome (1)

- Genome = set of all DNA contained in a cell.
- Formed by one or more long stretches of DNA strung together into *chromosomes*.
- Chromosomes are faithfully replicated by a cell when it divides.
- The set of chromosomes in a cell contains the DNA necessary to synthesize the *proteins* and other molecules needed to survive, as well as much of the information necessary to finely regulate their synthesis
 - Each protein is coded for by a specific *gene*, a stretch of DNA containing the information necessary for that purpose.

The anatomy of a genome (2)

- DNA molecules consist of a chain of smaller molecules called *nucleotides* that are distinct from each other only in a chemical element called a *base*.
- For biochemical reasons, DNA sequences have an orientation
 - It is possible to distinguish a specific direction in which to read each chromosome or gene
 - The directions are often represented as the left and right end of the sequence
- A DNA sequence can be single-stranded or double-stranded.
- The double-stranded nature is caused by the *pairing* of bases (base pairs, bp).
- When it is double-stranded, the two strands have opposite direction and are complementary to one another.
- This complementarity means that for each A, C, G, T in one strand, there is a T, G, C, or A, respectively, in the other strand.

The anatomy of a genome (3)

- Chromosomes are double-stranded (➔“double helix”)
- Information about a gene can be contained in either strand.
- This pairing introduces a complete redundancy in the encoding
 - allows the cell to reconstitute the entire genome from just one strand (enables faithful replication)
 - for simple convenience, we usually just write out the single strand of DNA sequence we are interested in from left to right
- The letters of the DNA alphabet are variously called nucleotides (nt), bases, or base pairs (bp) for double stranded DNA.
- The length of a DNA sequence can be measured in bases, or in kilobases (1000 bp or Kb), megabases (1000000 bp or Mb), or Gb.
- The genomes present in different organisms range in size from kilobases to gigabases
 - Mammalian genomes are typically 3Gbps (*gigabase pairs*) in size = 3 thousand million nucleotides

Reconocimiento de patrones

“Usamos GCG

(una interfaz muy popular para buscar ADN y proteínas en bancos de datos;

<http://bioinformatics.unc.edu/software/gcg/index.htm>)

para buscar en Genbank

(la mayor base de datos de ADN en USA;

<http://www.ncbi.nlm.nih.gov/Genbank/>)

una cadena de 30 caracteres

(tamaño pequeño en esa aplicación)

y tardó 4 horas usando una copia local de la base de datos para determinar que no existía esa cadena...”

“Repetimos luego el test usando el algoritmo de Boyer-Moore y la búsqueda tardó menos de 10 minutos (la mayor parte del tiempo fue debida al movimiento de datos del disco a la memoria, pues realmente la búsqueda se hizo en menos de 1 minuto)...” (D. Gusfield)



Reconocimiento de patrones

- Importancia del problema (continuación)
 - El problema de la subcadena todavía no tiene una solución tan eficiente y universal que lo haga carecer de interés.
 - El tamaño de las bases de datos seguirá creciendo y la búsqueda de una subcadena seguirá siendo una subtarea necesaria para muchas aplicaciones.
 - Además, tiene interés estudiar el problema y las soluciones conocidas para entender las ideas subyacentes en ellas.
 - Existen otros muchos problemas más difíciles relacionados con cadenas y con enorme interés práctico.



Reconocimiento de patrones

- Método directo

```
función subcadena(S,P:cadena; n,m:nat) dev nat
{Devuelve r si la primera aparición de P en S empieza en la
  posición r (i.e. es el entero más pequeño tal que  $S_{r+i-1}=P_i$ 
  para  $i=1,2,\dots,m$ ), y devuelve 0 si P no es subcadena de S}
variables ok:booleano; i,j:natural
principio
  ok:=falso; i:=0;
  mq not ok and  $i \leq n-m$  hacer
    ok:=verdad; j:=1;
    mq ok and  $j \leq m$  hacer
      si  $P[j] \neq S[i+j]$  ent ok:=falso sino  $j:=j+1$  fsi
    fmq;
     $i:=i+1$ ;
  fmq;
  si ok entonces dev i sino dev 0 fsi
fin
```



Reconocimiento de patrones

- Análisis del método directo:
 - Intenta encontrar el patrón P en cada posición de S .
 - Peor caso:
 - Hace m comparaciones en cada posición para comprobar si ha encontrado una aparición de P .

E.g.: $S = \text{'aaaaaaab'}$ $P = \text{'aaab'}$

- El número total de comparaciones es

$$\Omega(m(n-m))$$

es decir, $\Omega(mn)$ si n es sustancialmente mayor que m .



Reconocimiento de patrones

- El algoritmo Knuth-Morris-Pratt (KMP, 1977)
 - Veamos primero las razones de la ineficiencia del método directo con un ejemplo: $S = \text{'xyxxxyxyxyxyxyxyxyxyxyxyxy'}$ $P = \text{'xyxyxyxyxyxy'}$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
1:	x	y	x	y																			
2:		x																					
3:			x	y																			
4:				x	y	x	y	y															
5:					x																		
6:						x	y	x	y	y	x	y	x	y	x	x							
7:							x																
8:								x	y	x													
9:									x														
10:										x													
11:											x	y	x	y	y								
12:												x											
13:													x	y	x	y	y	x	y	x	y	x	x



Reconocimiento de patrones

- Notar que hay muchas comparaciones redundantes: el método directo compara el mismo subpatrón en el mismo lugar de la cadena madre varias veces.
 - Por ejemplo, verificamos dos veces que la subcadena ‘xyxy’ está en la posición 11 de S (en las líneas 6 y 11).
 - En la aplicación de buscar una palabra en un fichero de texto, el método directo no es muy malo porque las discrepancias ocurren enseguida y, por tanto, los retrocesos son pequeños.
 - Sin embargo, en otras aplicaciones (por ejemplo, biología molecular) el alfabeto es muy pequeño y hay muchas repeticiones, por tanto los retrocesos son mayores.



Reconocimiento de patrones

- Otro ejemplo:

$S = \text{'yyyyyyyyyyyyyyx'}$

$P = \text{'yyyyyx'}$

- Con el método directo, las cinco ‘y’ del patrón se comparan con la cadena madre, se encuentra la discrepancia de la ‘x’, se “desplaza” el patrón un lugar a la derecha y se hacen cuatro comparaciones redundantes:

¡las cuatro primeras ‘y’ ya sabemos que están ahí!

Se desplaza el patrón un lugar más a la derecha y ¡de nuevo cuatro comparaciones redundantes!

Etcétera.



Reconocimiento de patrones

- Un ejemplo más:

$S = \text{'xyyyyxyxyxyyyy'}$

$P = \text{'xyyyy'}$

- Buscamos la ocurrencia de una 'x' seguida de cinco 'y'.

Si el número de 'y' no es suficiente, no hay necesidad de volver atrás y desplazar una posición a la derecha.

Las cuatro 'y' encontradas no valen para nada y hay que buscar una nueva 'x'.



Reconocimiento de patrones

- Volviendo al ejemplo inicial:

$S = 'xyxxyxyxyxyxyxyxyxyxyxyxx'$ $P = 'xyxyxyxyxyxx'$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
1:	x	y	x	y																			
2:		x																					
3:			x	y																			
4:				x	y	x	y	y															
5:					x																		
6:						x	y	x	y	y	x	y	x	y	x	x							
7:

- Ocurre una discrepancia en p_5 (con s_8 , la línea 4).
- Los dos caracteres precedentes de S han tenido que ser 'xy' (es decir, p_3p_4 , porque hasta p_4 hubo coincidencia).
- Pero los dos primeros caracteres de P también son 'xy', luego no hace falta volverlos a comparar con p_3p_4 .
- Lo ideal sería desplazar P a la derecha el máximo número posible de posiciones (para ahorrar comparaciones) pero sin perder ninguna posibilidad de encontrar el patrón en S .

En el ejemplo, hay que desplazar P dos posiciones y continuar comparando s_8 con p_3 (ahorramos tres comparaciones).




Reconocimiento de patrones

- Notar que la discusión anterior es **independiente de la cadena madre S** .

Conocemos los últimos caracteres de S porque han coincidido con los anteriores del patrón P .

- Sigamos con el ejemplo...

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
.
6:						x	y	x	y	y	x	y	x	y	x								
7:							x																
8:								x	y	x													
9:									x														
10:										x													
11:											x	y	x	y	y								
12:												x											
13:													x	y	x	y	y	x	y	x	y	x	x

- La discrepancia en la línea 6 es en p_{11} .
- Ahora podemos ahorrarnos 15 comparaciones...



Reconocimiento de patrones

- La discrepancia fue entre p_{11} y s_{16} .
- Consideremos el subpatrón $p_1 p_2 \dots p_{10}$.
 - Sabemos que $p_1 p_2 \dots p_{10} = s_6 s_7 \dots s_{15}$.
 - Queremos saber cuántas posiciones hay que desplazar P hacia la derecha hasta que vuelva a existir la posibilidad de que coincida con una subcadena de S .
 - Nos fijamos en el **máximo sufijo de $p_1 p_2 \dots p_{10}$ que coincide con un prefijo de P** .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
.
6:						x	y	x	y	y	x	y	x	y	x	x							

- En este caso, el sufijo es de longitud 3: 'xyx'.
- Luego se puede continuar comparando s_{16} con p_4 .

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	x	y	x	x	y	x	y	x	y	y	x	y	x	y	x	y	y	x	y	x	y	x	x
.
6:						x	y	x	y	y	x	y	x	y	x	x							
13:													x	y	x	y	y	x	y	x	y	x	x

Reconocimiento de patrones

- Notar, de nuevo, que **toda la información necesaria para saber cuánto hay que desplazar a la derecha el patrón está incluida en el propio patrón.**
- Se puede “preprocesar” (o precondicionar) el patrón para acelerar el método directo.
- La idea es la siguiente:
 - La cadena madre S siempre se recorre hacia la derecha (no hay retrocesos), aunque un mismo carácter de S puede compararse con varios del patrón P (cuando haya discrepancias).
 - Cuando haya una discrepancia se consultará una **tabla** para saber cuánto hay que retroceder en el patrón o, dicho de otra forma, cuántos desplazamientos del patrón hacia la derecha pueden hacerse.
 - En la tabla hay un entero por cada carácter de P , e indica cuántos desplazamientos hacia la derecha deben hacerse cuando ese carácter discrepe con uno de la cadena madre.



Reconocimiento de patrones

- Definición precisa de la tabla:
 - Para cada p_i de P , hay que calcular el sufijo más largo $p_{i-j}p_{i-j+1}\dots p_{i-1}$ que es igual al prefijo de P :
$$\text{sig}(i) = \max \{ j \mid 0 < j < i-1, p_{i-j}p_{i-j+1}\dots p_{i-1} = p_1p_2\dots p_j \}$$
$$0 \text{ si no existe tal } j$$
 - Así, si $\text{sig}(i) = j$, el caracter discrepante de S con p_i puede pasar a compararse directamente con p_{j+1} (sabemos que los j caracteres más recientes de S coinciden con los j primeros de P).
 - Por convenio, $\text{sig}(1) = -1$, para distinguir ese caso especial.
 - Además, es obvio que siempre $\text{sig}(2) = 0$ (no $\exists j$ tal que $0 < j < 2-1$).
- En el ejemplo anterior:

$i =$	1	2	3	4	5	6	7	8	9	10	11
$P =$	x	y	x	y	y	x	y	x	y	x	x
$\text{sig} =$	-1	0	0	1	2	0	1	2	3	4	3



Reconocimiento de patrones

```
función KMP(S,P:cadena; n,m:natural) devuelve natural
{Idéntica especificación a 'subcadena'.}
variables i,j,pos:natural
principio
  j:=1; i:=1; pos:=0;
  mq pos=0 and i≤n hacer
    si P[j]=S[i]
      entonces
        j:=j+1; i:=i+1
      sino
        j:=sig[j]+1;
        si j=0 entonces
          j:=1; i:=i+1
        fsi
      fsi;
    si j=m+1 entonces pos:=i-m fsi
  fmq;
  devuelve pos
fin
```



Reconocimiento de patrones

- Falta el cálculo de la tabla *sig*:
 - Lo haremos por inducción. En primer lugar, $sig(2) = 0$.
 - Suponer que *sig* ya está calculada para $1, 2, \dots, i-1$.
 - Como mucho, $sig(i)$ puede ser $sig(i-1)+1$.

Esto ocurre si $p_{i-1} = p_{sig(i-1)+1}$.

$i =$	1	2	3	4	5	6	7	8	9	10	11
$P =$	<u>x</u>	<u>y</u>	<u>x</u>	<u>y</u>	<u>y</u>	<u>x</u>	<u>y</u>	<u>x</u>	<u>y</u>	x	x
$sig =$	-1	0	0	1	2	0	1	2	3	<u>4</u>	

- Si $p_{i-1} \neq p_{sig(i-1)+1}$:

$i =$	1	2	3	4	5	6	7	8	9	10	11
$P =$	<u>x</u>	<u>y</u>	<u>x</u>	<u>y</u>	<u>y</u>	<u>x</u>	<u>y</u>	<u>x</u>	<u>y</u>	<u>x</u>	x
$sig =$	-1	0	0	1	2	0	1	2	3	4	?

Es como una búsqueda de un patrón...

$i =$	1	2	3	4	5	6	7	8	9	10	11
$P =$	x	y	x	y	y	x	y	x	y	<u>x</u>	x
$p_1 \dots p_{sig(i-1)+1} =$						x	y	x	y	y	
$p_1 \dots p_{sig(sig(i-1)+1)+1} =$								x	y	<u>x</u>	

$\Rightarrow sig(11) = 3$



Reconocimiento de patrones

```
tipo vect = vector[1..m] de entero

algoritmo calculaSig(ent P:cadena; ent m:natural;
                    sal sig:vect)
{Cálculo de la tabla 'sig' para el patrón 'P'.}
variables i,j:natural
principio
    sig[1] := -1;
    sig[2] := 0;
    para i:=3 hasta m hacer
        j:=sig[i-1]+1;
        mq j>0 and entonces P[i-1]≠P[j] hacer
            j:=sig[j]+1
        fmq;
        sig[i] := j
    fpara
fin
```



Reconocimiento de patrones

- Análisis de la función KMP:
 - La cadena madre S se recorre sólo una vez, aunque un carácter s_i de S puede que haya que compararlo con varios de P .
 - ¿Cuántas veces se puede llegar a comparar un carácter s_i de S con otros de P ?
 - Supongamos que la primera comparación de s_i se hace con p_k . Por tanto, en particular, se ha avanzado k veces en P hasta hacer esa comparación y sin retroceder ninguna vez.
 - Si $s_i \neq p_k$, se retrocede en P (usando la tabla *sig*). Sólo se puede retroceder un máximo de k veces.
- Si se suma el coste de los retrocesos al de los movimientos de avance, únicamente se dobla el coste de los avances.

Pero el nº de avances en P coincide con el nº de avances en S y es n .
Luego el número de comparaciones es $O(n)$.
- Con similares argumentos se demuestra que el coste del cálculo de la tabla *sig* es $O(m)$, luego el coste total es $O(n)$.



Reconocimiento de patrones

- El algoritmo de Boyer y Moore (BM, 1977)
 - Como el KMP, el BM puede encontrar todas las apariciones de un patrón P (de longitud m) en una cadena madre S (de longitud n) en un tiempo $O(n)$ en el caso peor.
 - KMP examina cada carácter de S al menos una vez, luego realiza un mínimo de n comparaciones.
 - BM es **sublineal**: no examina necesariamente todos los caracteres de S y el número de comparaciones es, a menudo, inferior a n .
 - Además, BM tiende a ser más eficiente cuando m crece.
 - En el mejor caso, BM encuentra todas las apariciones de P en S en un tiempo $O(m+n/m)$.



Reconocimiento de patrones

- Descripción del BM:

- Como en KMP, desplazamos P sobre S de izquierda a derecha examinando los caracteres enfrentados.
- Pero ahora la verificación de los caracteres de P se hace de derecha a izquierda después de cada desplazamiento del patrón.
- Se utilizan dos reglas para decidir el desplazamiento que hay que hacer después de una discrepancia:

Regla A. Después de un desplazamiento, se compara p_m con un carácter c de S y son distintos:

- Si c aparece más a la izquierda en el patrón, se desplaza éste para alinear la última aparición de c en el patrón con el carácter c de S .
- Si c no está en P , se coloca éste inmediatamente detrás de la aparición de c en S .

Regla B. Si un cierto nº de caracteres al final de P se corresponde con caracteres de S , aprovechamos este conocimiento parcial de S (como en KMP) para desplazar P a una nueva posición compatible con la información que poseemos.



Reconocimiento de patrones

- Ejemplo:

$S = \text{'se espera cielo nublado para mañana'}$
 $P = \text{'lado'}$



Se detecta la primera discrepancia en la primera comparación. Se aplica la Regla A:

$S = \text{'se espera cielo nublado para mañana'}$
 $P = \text{'lado'}$



De nuevo una discrepancia. Regla A:

$S = \text{'se espera cielo nublado para mañana'}$
 $P = \text{'lado'}$



Lo mismo. Regla A:

$S = \text{'se espera cielo nublado para mañana'}$
 $P = \text{'lado'}$



Reconocimiento de patrones

$S = \text{'se espera cielo nublado para mañana'}$

$P = \text{'lado'}$



Lo mismo. Regla A:

$S = \text{'se espera cielo nublado para mañana'}$

$P = \text{'lado'}$



Discrepancia. Regla A. Ahora el carácter enfrentado a p_m aparece en P . Se desplaza P para alinear ambas apariciones.

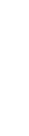
$S = \text{'se espera cielo nublado para mañana'}$

$P = \text{'lado'}$



Se hace la verificación (de derecha a izquierda) y se encuentra el patrón.

- Sólo se ha usado la Regla A.
- Se han hecho sólo 9 comparaciones.



tap

- Técnicas Avanzadas de Programación - Javier Campos

Reconocimiento de patrones

$S = \text{'babcbabcbabcaabcbabcbacabc'}$
 $P = \text{'abcabcacab'}$

↑

– Regla A:

$S = \text{'babcbabcbabcaabcbabcbacabc'}$
 $P = \text{'abcabcacab'}$

↑

– Otra vez, Regla A:

$S = \text{'babcbabcbabcaabcbabcbacabc'}$
 $P = \text{'abcabcacab'}$

↑↑↑↑↑↑↑↑↑↑

– Se han hecho 21 comparaciones para encontrar P , 2 de ellas redundantes.



Reconocimiento de patrones

- Precondicionamiento o preproceso necesario para implementar BM:

- Se necesita dos vectores:

$d1[\{\text{juego de caracteres}\}]$

$d2[1..m-1]$

el 1º para implementar la Regla A y el 2º la B.

- El cálculo de $d1$ es fácil:

```
para todo  $c \in \{\text{juego caract.}\}$  hacer  
  si  $c \notin P[1..m]$   
    entonces  $d1[c] := m$   
    sino  $d1[c] := m - \max\{i \mid P[i] = c\}$   
  fsi  
fpara
```



Reconocimiento de patrones

- El cálculo de d_2 es más complicado (no veremos los detalles).
- La interpretación de d_2 es:

Después de una discrepancia en la posición i del patrón, re comenzamos la comprobación en la posición m del patrón $d_2[i]$ caracteres más a la derecha en S .

- Veamos un ejemplo:

$S = \text{'?????xe????????'}$ con $x \neq t$

$P = \text{'ostente'}$



Como $x \neq t$ no se puede alinear la e de S con la otra e de P , luego hay que desplazar P completamente, es decir, $d_2[6] := 7$:

$S = \text{'?????xe????????'}$ con $x \neq t$

$P = \text{'ostente'}$



Reconocimiento de patrones

- De igual forma:

$S = \text{'?????xte?????'} \quad \text{con } x \neq n$

$P = \text{'ostente'}$

↑↑↑

En este caso, es claro que $d2[5] := 3$:

$S = \text{'?????xte?????'} \quad \text{con } x \neq n$

$P = \text{'ostente'}$

↑

Etcétera: $d2 := [7, 7, 7, 3, 7, 7]$



Reconocimiento de patrones

- Hay veces que según lo dicho hasta ahora hay que aplicar la Regla B, sin embargo es más eficiente aplicar la Regla A:

$S = \text{'??virte?????'}$

$P = \text{'ostente'}$



Si se aplica la Regla B ($d2[5]=3$):

$S = \text{'??virte?????'}$

$P = \text{'ostente'}$



Sin embargo, como 'r' no aparece en P , se puede desplazar directamente en ($d1[r]=7$):

$S = \text{'??virte?????'}$

$P = \text{'ostente'}$



- Detalles: ¡consultar bibliografía!



Introducción a la biología computacional

- El plan:
 - Algoritmos de reconocimiento de patrones:
 - Knuth-Morris-Pratt
 - Boyer-Moore
 - **Árboles de sufijos**
 - Primeras aplicaciones de los árboles de sufijos



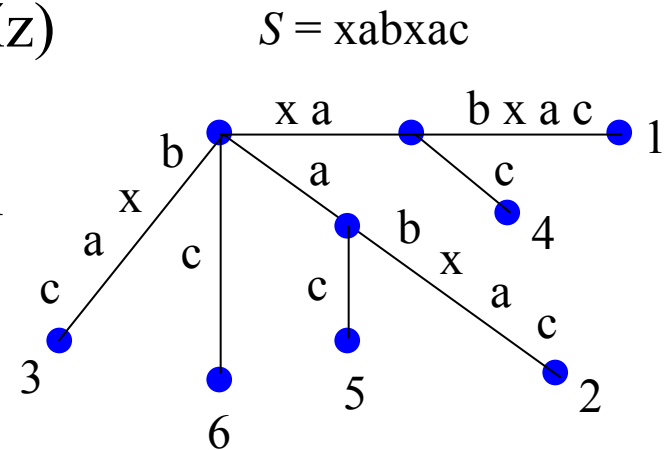
Árboles de sufijos

- ¿Qué es un árbol de sufijos?
 - Una estructura de datos que sirve para almacenar una cadena de caracteres con “información pre-procesada” sobre su estructura interna.
 - Esa información es útil, por ejemplo, para resolver el problema de la subcadena en tiempo lineal:
 - Sea un texto S de longitud m
 - Se pre-procesa (se construye el árbol) en tiempo $O(m)$
 - Para buscar una subcadena P de longitud n basta con $O(n)$.
Esta cota no la alcanza ni el KMP ni el BM (requieren $O(m)$)
 - Sirve además para otros muchos problemas más complejos, como por ejemplo:
 - Dado un conjunto de textos $\{S_i\}$ ver si P es subcadena de algún S_i
 - Reconocimiento inexacto de patrones...



Árboles de sufijos

- Definición: árbol de sufijos para una cadena S de longitud m
 - Árbol con raíz y con m hojas numeradas de 1 a m
 - Cada nodo interno (salvo la raíz) tiene al menos 2 hijos
 - Cada arista está etiquetada con una subcadena no vacía de S
 - Dos aristas que salen del mismo nodo no pueden tener etiquetas que empiecen por el mismo carácter
 - Para cada hoja i , la concatenación de etiquetas del camino desde la raíz reproduce el sufijo de S que empieza en la posición i

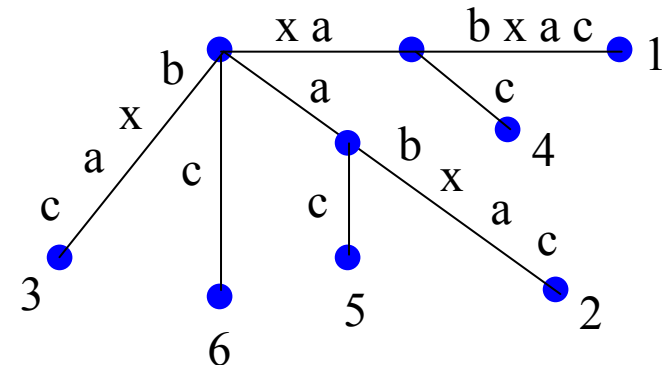
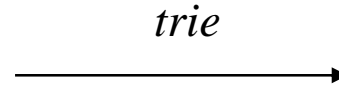


Árboles de sufijos

- Es como un *trie* (árbol de prefijos) que almacena todos los sufijos de una cadena

– Sufijos de la cadena $S = \text{xabxac}$:

- c
- ac
- xac
- bxac
- abxac
- xabxac



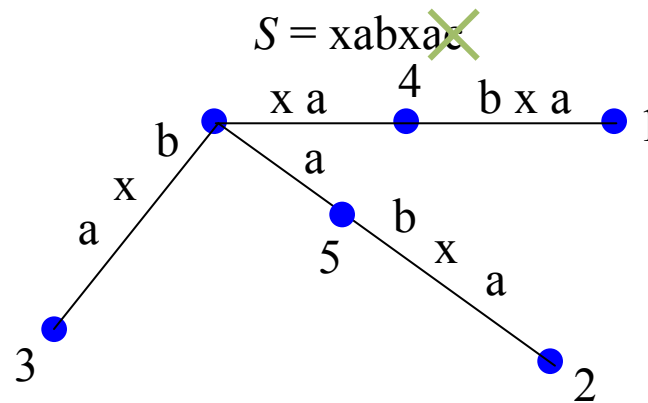
- Además: cada nodo interno tiene al menos 2 hijos....
➔ es como un Patricia que guarda todos los sufijos de la cadena



Árboles de sufijos

- Un problema...
 - La definición no garantiza que exista un árbol de sufijos para cualquier cadena S .
 - Si un sufijo de S coincide con el prefijo de algún otro sufijo de S , el camino para el primer sufijo no terminaría en una hoja.

– Ejemplo:



- Solución fácil: añadir carácter terminador, $S = \text{xabxabc}\epsilon$ (como hicimos con los *tries*)

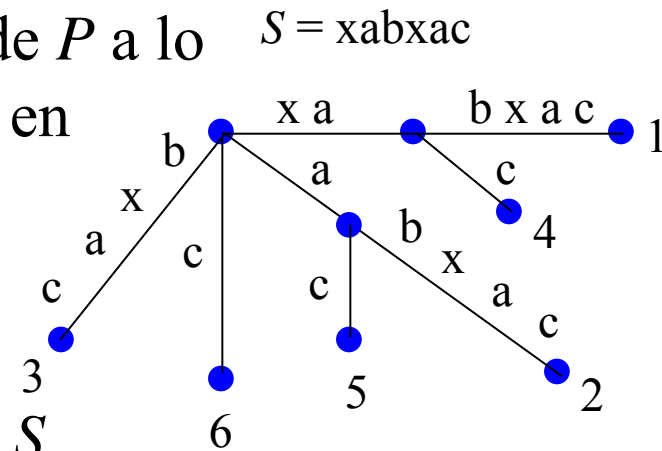
Árboles de sufijos

- Terminología:
 - *Etiqueta de un nodo*: concatenación ordenada de las etiquetas de las aristas del camino desde la raíz a ese nodo
 - *Profundidad en la cadena* de un nodo: número de caracteres en la etiqueta del nodo



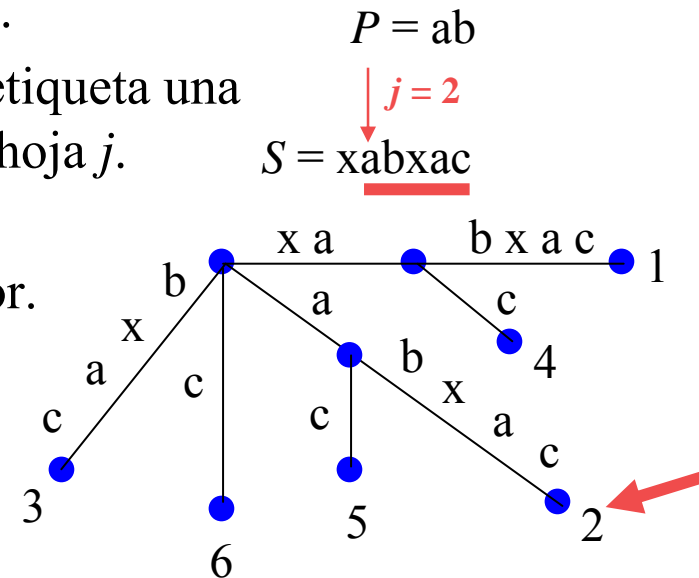
Árboles de sufijos

- Solución al problema de la subcadena:
 - Encontrar todas las apariciones de P , de longitud n , en un texto S , de longitud m , en tiempo $O(n + m)$.
 - Construir un árbol de sufijos para S , en tiempo $O(m)$.
 - Hacer coincidir los caracteres de P a lo largo del único camino posible en el árbol hasta que:
 - a) se acaban los caracteres de P , o
 - b) no hay más coincidencias.
 - En el caso (b), P no aparece en S .
 - En el caso (a), cada hoja del subárbol por debajo de la arista de la última coincidencia tiene la posición del inicio de una aparición de P en S , y no hay más.



Árboles de sufijos

- Explicación del caso (a):
 - P aparece en S desde la posición j si y sólo si P es un prefijo de $S[j..m]$ (que es uno de los sufijos de S).
 - Y esto ocurre si y sólo si la cadena P etiqueta una parte inicial del camino de la raíz a la hoja j .
 - Y esa parte inicial es precisamente la que hace coincidir el algoritmo anterior.
 - Esa parte coincidente es única porque no hay dos aristas que salgan de un mismo nodo y tengan etiquetas que empiecen por el mismo carácter.
- Como se supone que el alfabeto es finito, el coste del trabajo en cada nodo es constante, luego el coste de hacer coincidir P con la etiqueta de un camino del árbol es proporcional a la longitud de P , $O(n)$.



Árboles de sufijos

- Coste de enumerar todas las apariciones en el caso (a):
 - Una vez terminados los caracteres de P , basta recorrer el subárbol bajo el final del camino de S con el que han coincidido, recopilando los números que etiquetan las hojas.
 - Como cada nodo interno tiene al menos 2 hijos, el nº de hojas es proporcional al nº de aristas recorridas, luego el tiempo del recorrido es $O(k)$, si k es el nº de apariciones de P en S .
 - Por tanto el coste de calcular todas las apariciones es $O(n + m)$, es decir, $O(m)$ para construir el árbol y $O(n + k)$ para la búsqueda.




Árboles de sufijos

- Coste (continuación):
 - Es el mismo coste de los algoritmos de la sección anterior (por ejemplo, el KMP), pero:
 - En ese caso se precisaba un tiempo $O(n)$ para el pre-procesamiento de P y luego un tiempo $O(m)$ para la búsqueda.
 - Ahora se usa $O(m)$ pre-procesando y luego $O(n + k)$ buscando, donde k es el numero de ocurrencias de P en S .
 - Si sólo se precisa una aparición de P , la búsqueda se puede reducir de $O(n + k)$ a $O(n)$ añadiendo a cada nodo (en el pre-procesamiento) el nº de una de las hojas de su subárbol.
 - Hay otro algoritmo que permite usar los árboles de sufijos para resolver el problema de la subcadena que precisa $O(n)$ para el pre-procesamiento y $O(m)$ para la búsqueda (es decir, igual que el KMP).



Árboles de sufijos

- Construcción de un árbol de sufijos:
 - Veremos la idea de uno de los tres métodos de coste lineal (en la longitud del texto) conocidos para construir el árbol
 - Weiner, 1973: “el algoritmo de 1973”, según Knuth
 - McCreight, 1976: más eficiente en espacio
 -  • Ukkonen, 1995: igual de eficiente que el anterior pero más “fácil” de entender (14 páginas del libro de D. Gusfield...)
 - Pero primero veamos un método *naif*, de coste cuadrático (y realmente fácil, i.e., una transparencia).



Árboles de sufijos

- Construcción del árbol para la cadena S (de longitud m) con coste cuadrático:
 - Crear árbol N_1 con una sola arista entre la raíz y la hoja numerada con 1, y etiqueta $S€$, es decir, $S[1..m]€$.
 - El árbol N_{i+1} (hasta llegar al N_m) se construye añadiendo el sufijo $S[i+1..m]€$ a N_i :
 - Empezando desde la raíz de N_i , encontrar el camino más largo cuya etiqueta coincide con un prefijo de $S[i+1..m]€$ (como cuando se busca un patrón en un árbol), y al acabar:
 - se ha llegado a un nodo, w , o
 - se está en mitad de una etiqueta de una arista (u,v) ; en este caso, se parte la arista en dos (por ese punto) insertando un nuevo nodo, w
 - Crear una nueva arista $(w,i+1)$ desde w a una nueva hoja y se etiqueta con la parte final de $S[i+1..m]€$ que no se pudo encontrar en N_i .



Árboles de sufijos

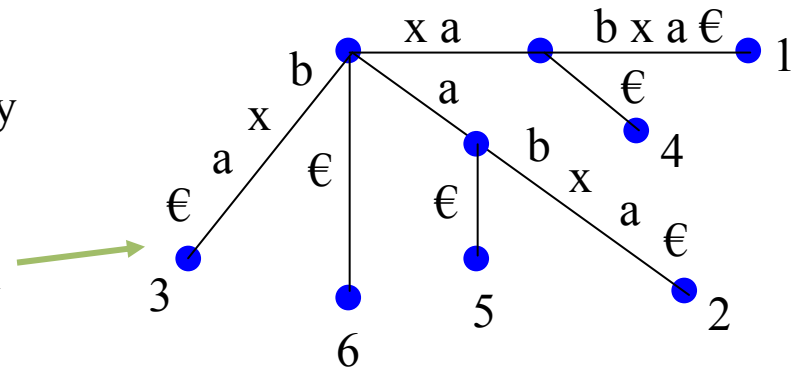
- Un algoritmo lineal para construir un árbol de sufijos (Ukkonen, 1995)
- Sobre la forma de presentarlo:
 - Primero se presenta en su forma más simple, aunque ineficiente
 - Luego se puede mejorar su eficiencia con varios trucos de sentido común
- El método: construir una secuencia de *árboles de sufijos implícitos*, y el último de ellos convertirlo en árbol de sufijos de la cadena S
 - *Arbol de sufijos implícitos*, I_m , para una cadena S : se obtiene del árbol de sufijos de $S\epsilon$ borrando cada copia del símbolo terminal ϵ de las etiquetas de las aristas y después eliminando cada arista que no tenga etiqueta y cada nodo interno que no tenga al menos dos hijos.
 - Se define de manera análoga el árbol de sufijos implícitos, I_i , para un prefijo $S[1..i]$ a partir del árbol de sufijos de $S[1..i]\epsilon$.



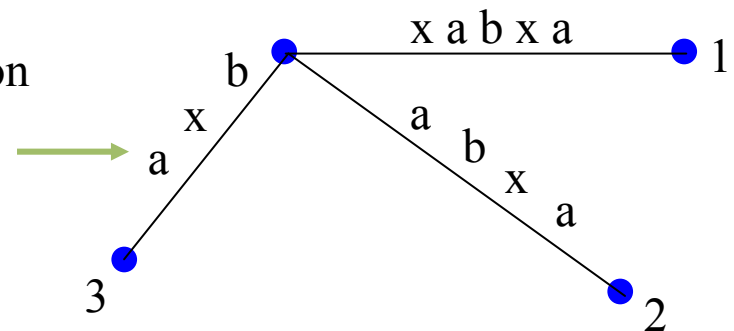
tap

- $$S = xabxa \in$$

Por eso en el árbol de sufijos de S las aristas que llevan a las hojas 4 y 5 están etiquetadas sólo con €.

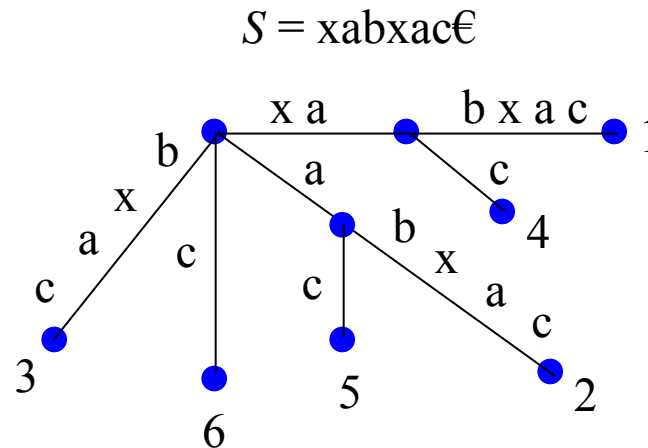


El árbol implícito puede no tener una hoja por cada sufijo de S , luego tiene menos información que el árbol de sufijos de S .



Árboles de sufijos

- En caso contrario, si S termina con un carácter que no apareció antes en S , entonces el árbol de sufijos implícitos de S tendrá una hoja por cada sufijo y por tanto es realmente un árbol de sufijos.



Árboles de sufijos

- El algoritmo en su versión ineficiente, $O(m^3)$

algoritmo Ukkonen_alto_nivel(S :cadena; m :nat)

principio

construir árbol I_1 ;

→ Es una arista con etiqueta $S(1)$

para $i:=1$ hasta $m-1$ **hacer**

para $j:=1$ hasta $i+1$ **hacer**

encontrar el final del
camino desde la raíz
etiquetado con $S[j..i]$
en el árbol actual;

si se precisa **entonces**

extender ese camino con el
carácter $S(i+1)$ para
asegurar que la cadena
 $S[j..i+1]$ está en el árbol

Extensión j :
Se añade al
árbol la cad.
 $S[j..i+1]$, para
 $j=1..i$.

Finalmente
(ext. $j+1$),
se añade la
cad. $S(i+1)$

Fase $i+1$:
Se calcula el
árbol I_{i+1} a
partir de I_i .

La fase $i+1$
tiene $i+1$
extensiones

fpara

fpara

fin

Recordar que I_i es el árbol de sufijos
implícitos para el prefijo $S[1..i]$.



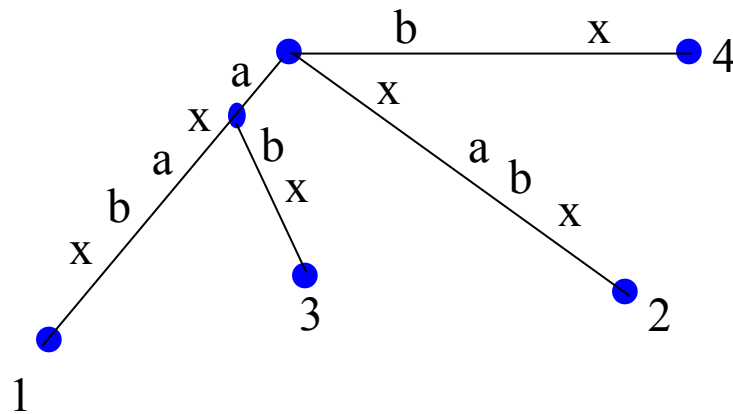
Árboles de sufijos

- Detalles sobre la extensión j de la fase $i+1$:
 - Se busca $S[j..i] = \beta$ en el árbol y al llegar al final de β se debe conseguir que el sufijo $\beta S(i+1)$ esté en el árbol, para ello se pueden dar tres casos:
 - [Regla 1]** Si β termina en una hoja: se añade $S(i+1)$ al final de la etiqueta de la arista de la que cuelga esa hoja
 - [Regla 2]** Si ningún camino desde el final de la cadena β empieza con $S(i+1)$, pero al menos hay un camino etiquetado que continúa desde el final de β se crea una nueva arista a una hoja (que se etiqueta con j) colgando desde allí y etiquetada con $S(i+1)$.
Si β termina en mitad de una etiqueta de una arista hay que insertar un nuevo nodo partiendo la arista y colgar de él una nueva hoja. La nueva hoja se etiqueta con j .
 - [Regla 3]** Si algún camino desde el final de β empieza por $S(i+1)$, la cadena $\beta S(i+1)$ ya estaba en el árbol. No hay que hacer nada.



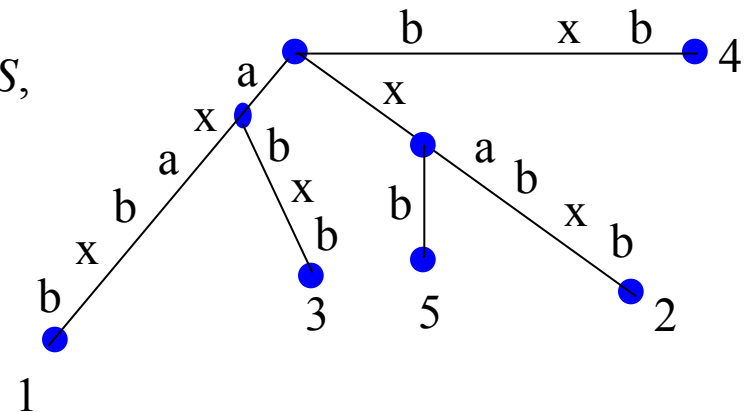
Árboles de sufijos

- Ejemplo: $S = axabx$



Es un árbol de sufijos implícitos para S . Los primeros 4 sufijos terminan en hoja. El último, x , termina en medio de una arista.

Si añadimos b como sexta letra de S , los primeros 4 sufijos se extienden mediante la regla 1, el 5º por la regla 2, y el 6º por la regla 3.



Árboles de sufijos

- Coste de esta primera versión:
 - Una vez alcanzado el final de un sufijo β de $S[1..i]$ se precisa tiempo constante para la extensión (asegurar que el sufijo $\beta S(i+1)$ está en el árbol).
 - La clave, por tanto, está en localizar los finales de todos los $i+1$ sufijos de $S[1..i]$.
 - Lo fácil:
 - localizar el final de β en tiempo $O(|\beta|)$ bajando desde la raíz;
 - así, la extensión j de la fase $i+1$ lleva un tiempo $O(i+1-j)$;
 - por tanto el árbol I_{i+1} se puede crear a partir de I_i en $O(i^2)$, y así I_m se crea en $O(m^3)$



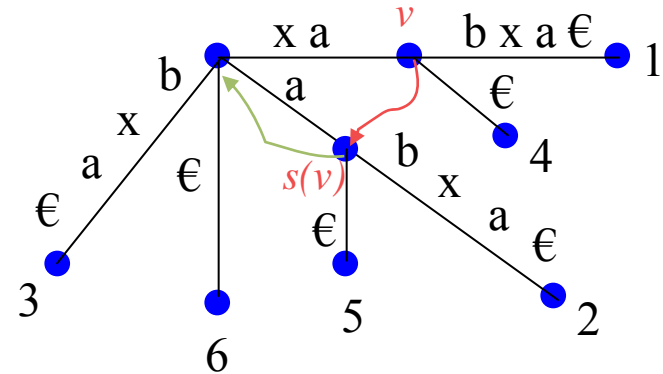
Árboles de sufijos

- Reduciendo el coste...: *punteros a sufijos*

- Sea v un nodo interno con etiqueta (desde la raíz) $x\alpha$ (es decir, un carácter x seguido de una cadena α) y otro nodo $s(v)$ con etiqueta α , entonces un

puntero de v a $s(v)$ se llama un puntero a sufijo

- Caso especial: si α es la cadena vacía entonces el puntero a sufijo desde un nodo interno con etiqueta $x\alpha$ **apunta al nodo raíz**
- El nodo raíz no se considera “interno” y por eso no sale ningún puntero a sufijo desde él



Árboles de sufijos

- **Lema 1:** si al árbol actual se le añade en la extensión j de la fase $i+1$ un nodo interno v con etiqueta $x\alpha$, entonces, una de dos:
 - el camino etiquetado con α ya termina en un nodo interno en el árbol, o
 - por las reglas de extensión se creará un nodo interno al final de la cadena α en la extensión $j+1$ de la fase $i+1$
- **Corolario 1:** cualquier nodo interno creado en el algoritmo de Ukkonnen será el origen de un puntero a sufijo al final de la siguiente extensión
- **Corolario 2:** en cualquier árbol de sufijos implícitos I_i , si un nodo interno v tiene etiqueta $x\alpha$, entonces hay un nodo $s(v)$ en I_i con etiqueta α

(demostraciones: libro de D. Gusfield)



Árboles de sufijos

- Fase $i+1$, extensión j (para $j=1..i+1$): localiza el sufijo $S[j..i]$ de $S[1..i]$
 - La versión naif recorre un camino desde la raíz
 - Se puede simplificar usando los punteros a sufijos...
 - Primera extensión ($j=1$):
 - El final de la cadena $S[1..i]$ debe terminar en una hoja de I_i porque es la cadena más larga del árbol
 - Basta con guardar siempre un puntero a la hoja que corresponde a la cadena completa $S[1..i]$
 - Así, se accede directamente al final del sufijo $S[1..i]$ y añadir el carácter $S(i+1)$ se resuelve con la regla 1, con coste constante



Árboles de sufijos

- Segunda extensión ($j=2$): encontrar el final de $S[2..i]$ para añadir $S(i+1)$.
 - Sea $S[1..i] = x\alpha$ (con α vacía o no) y sea $(v,1)$ la arista que llega a la hoja 1.
 - Hay que encontrar el final de α en el árbol.
 - El nodo v es la raíz o es un nodo interno de I_i
 - Si v es la raíz, para llegar al final de α hay que recorrer el árbol hacia abajo siguiendo el camino de α (como el algoritmo naif).
 - Si v es interno, por el Corolario 2, hay un puntero a sufijo desde v hasta $s(v)$.

Más aún, como $s(v)$ tiene una etiqueta que es prefijo de α , el final de α debe terminar en el subárbol de $s(v)$

Entonces, en la búsqueda del final de α , no hace falta recorrer la cadena completa, sino que se puede empezar el camino en $s(v)$ (usando el puntero a sufijo).



Árboles de sufijos

- El resto de extensiones de $S[j..i]$ a $S[j..i+1]$ con $j > 2$:
 - Empezando desde el final de $S[j-1..i]$ (al que se ha tenido que llegar en la extensión anterior) ascender un nodo para llegar bien a la raíz bien a un nodo interno v desde el que sale un puntero a sufijo que llega a $s(v)$.
 - Si v no es la raíz, seguir el puntero a sufijo y descender en el subárbol de $s(v)$ hasta el final de $S[j..i]$, y extender con $S(i+1)$ según las reglas de extensión.
- Coste resultante: de momento... el mismo, pero veamos ahora un truco que lo reduce a $O(m^2)$

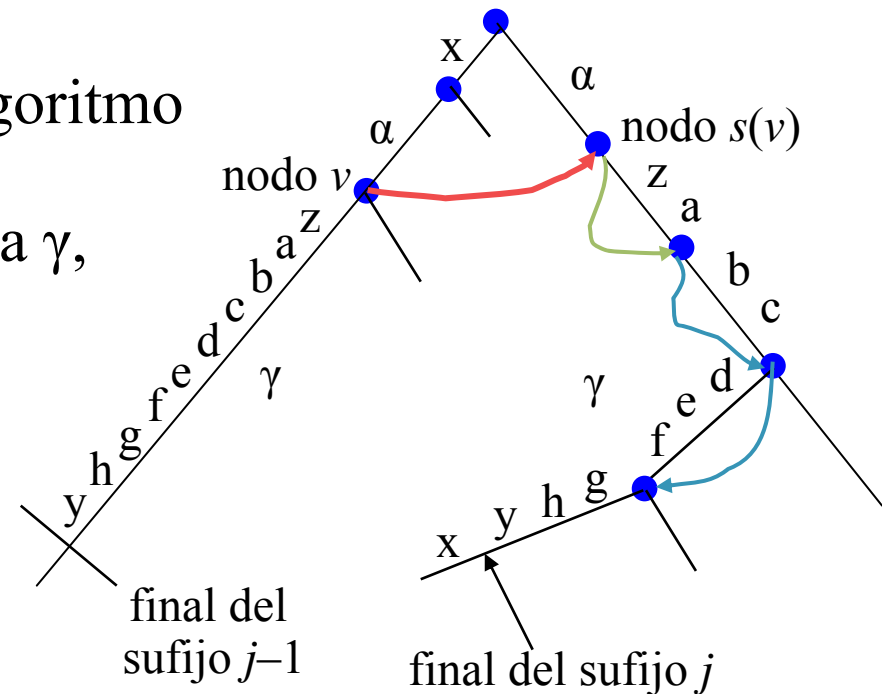


Árboles de sufijos

- **Truco nº 1:**

- En la extensión $j+1$ el algoritmo desciende desde $s(v)$ a lo largo de la subcadena, sea γ , hasta el final de $S[j..i]$
- Implementación directa: $O(|\gamma|)$
- Se puede reducir a $O(\text{nº nodos de camino})$:

- Sea $g = |\gamma|$
- El 1^{er} carácter de γ debe aparecer en una (y sólo una) arista de las que salen de $s(v)$; sea g' el nº de caracteres de esa arista
- Si $g' < g$ entonces se puede **saltar al nodo final de la arista**
- Se hace $g = g - g'$, se asigna a una variable h la posición de $g' + 1$ y **se sigue mirando hacia abajo**



Árboles de sufijos

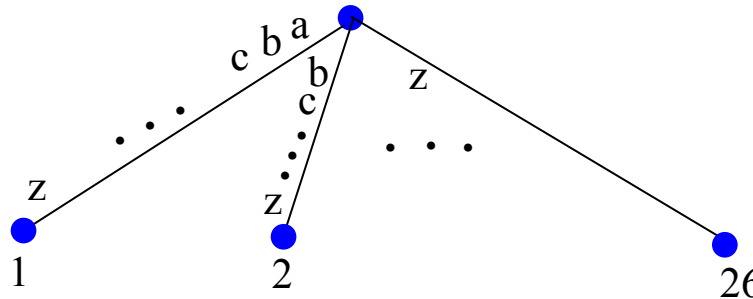
- Terminología: *profundidad* de un nodo es el n° de nodos del camino de la raíz a ese nodo
- **Lema 2:** Sea $(v, s(v))$ un puntero a sufijo recorrido en el algoritmo de Ukkonen. En ese instante, la profundidad de v es, como mucho, uno más que la de $s(v)$.
- **Teorema 1:** usando el truco 1, el coste en tiempo de cualquier fase del algoritmo de Ukkonen es $O(m)$.
- **Corolario 3:** el algoritmo de Ukkonen puede implementarse con punteros a sufijos para conseguir un coste en tiempo en $O(m^2)$.

(demostraciones: libro de D. Gusfield)



Árboles de sufijos

- Problema para bajar de coste $O(m^2)$:
 - El árbol puede requerir $\Theta(m^2)$ en espacio:
 - Las etiquetas de aristas pueden contener $\Theta(m^2)$ caracteres
 - Ejemplo: $S = \text{abcdefghijklmnopqrstuvwxyz}$
Cada sufijo empieza por distinta letra, luego de la raíz salen 26 aristas y cada una etiquetada con un sufijo completo, por tanto se requieren $26 \times 27/2$ caracteres en total



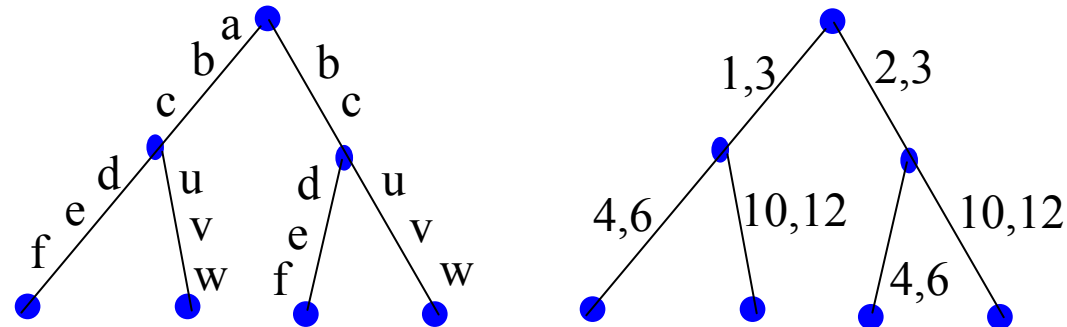
- Se requiere otra forma de guardar las etiquetas...



Árboles de sufijos

- Compresión de etiquetas
 - Guardar un *par de índices*: principio y fin de la subcadena en la cadena S
 - El coste para localizar los caracteres a partir de las posiciones es constante (si se guarda una copia de S con acceso directo)

$S = \text{abcdefabcuvw}$



- Número máximo de aristas: $2m - 1$, luego el coste en espacio para almacenar el árbol es $O(m)$

Árboles de sufijos

- Observación 1: recordar la Regla 3

Detalles sobre la extensión j de la fase $i+1$:

Se busca $S[j..i] = \beta$ en el árbol y al llegar al final de β se debe conseguir que el sufijo $\beta S(i+1)$ esté en el árbol, para ello se pueden dar tres casos:

...

[Regla 3] Si algún camino desde el final de β empieza por $S(i+1)$, la cadena $\beta S(i+1)$ ya estaba en el árbol. No hay que hacer nada.

- Si se aplica la regla 3 en alguna extensión j , se aplicará también en el resto de extensiones desde $j + 1$ hasta $i + 1$.
- Más aún, sólo se añade un puntero a sufijo tras aplicar la regla 2...

[Regla 2] Si ningún camino desde el final de la cadena β empieza con $S(i+1)$, pero al menos hay un camino etiquetado que continúa desde el final de β se crea una nueva arista a una hoja colgando desde allí y etiquetada con $S(i+1)$. Si β termina en mitad de una etiqueta de una arista hay que insertar un nuevo nodo partiendo la arista y colgar de él una nueva hoja.



Árboles de sufijos

- **Truco nº 2:**
 - Se debe terminar la fase $i + 1$ la primera vez que se aplique la regla 3 para una extensión.
 - Las extensiones de la fase $i + 1$ tras la primera ejecución de la regla 3 se dirán *implícitas* (no hay que hacer nada en ellas, luego no hay que hacerlas)
 - Por el contrario, una extensión j en la que se encuentra explícitamente el final de $S[j..i]$ (y por tanto se aplica la regla 1 o la 2), se llama *explícita*.



Árboles de sufijos

- Observación 2:

- Una vez que se crea una hoja y se etiqueta con j (por el sufijo de S que empieza en la posición j), se queda como hoja en toda la ejecución del algoritmo.
- En efecto, si hay una hoja etiquetada con j la regla 1 de extensión se aplicará en todas las fases sucesivas a la extensión j .

[Regla 1] Si β termina en una hoja: se añade $S(i+1)$ al final de la etiqueta de la arista de la que cuelga esa hoja

- Por tanto, tras crear la hoja 1 en la fase 1, en toda fase i hay una secuencia inicial de extensiones consecutivas (empezando desde la extensión 1) en las que se aplican las reglas 1 ó 2.
Sea j_i la última extensión de esta secuencia.



Árboles de sufijos

- Como cada aplicación de la regla 2 crea una nueva hoja, se sigue de la observación 2 que $j_i \leq j_{i+1}$,
 - es decir, la longitud de la secuencia inicial de extensiones en las que se aplican las reglas 1 ó 2 no puede reducirse en fases sucesivas;
 - por tanto, se puede aplicar el siguiente truco de implementación:
 - en la fase $i + 1$ evitar todas las extensiones explícitas desde la 1 a la j_i (así se consume tiempo constante en hacer todas esas extensiones implícitamente);
 - lo vemos en detalle a continuación
(recordar que la etiqueta de una arista se representa por 2 índices, p, q , que especifican la subcadena $S[p..q]$, y que la arista a una hoja en el árbol I_i tendrá $q = i$, y por tanto en la fase $i + 1$ q se incrementará a $i + 1$, indicando el añadido del carácter $S(i + 1)$ al final de cada sufijo)



Árboles de sufijos

- **Truco nº 3:**

- En la fase $i + 1$, cuando se crea una arista a una hoja que se debería etiquetar con los índices $(p, i + 1)$ representando la cadena $S[p..i + 1]$, en lugar de eso, etiquetarla con (p, e) , donde e es un símbolo que denota el “final actual”
- e es un índice global al que se da valor $i + 1$ una vez en cada fase
- En la fase $i + 1$, puesto que el algoritmo sabe que se aplicará la regla 1 en las extensiones de la 1 a la j_i como mínimo, no hace falta más trabajo adicional para implementar esas j_i extensiones; en su lugar, basta coste constante para incrementar la variable e y luego el trabajo necesario para las extensiones a partir de la $j_i + 1$



Árboles de sufijos

- Coste: el algoritmo de Ukkonen, usando los punteros a sufijos e implementando los trucos 1, 2 y 3, construye los árboles de sufijos implícitos I_1 a I_m en tiempo total $O(m)$.
(detalles: libro de D. Gusfield y “web:material adicional”)
- El árbol de sufijos implícitos I_m se puede transformar en el árbol de sufijos final en $O(m)$:
 - Se añade el símbolo terminador ϵ al final de S y se hace continuar el algoritmo de Ukkonen con ese carácter
 - Como ningún sufijo es ahora prefijo de otro sufijo, el algoritmo crea un árbol de sufijos implícitos en el que cada sufijo termina en una hoja (luego es un árbol de sufijos)
 - Hay que cambiar cada índice e de las aristas a las hojas por m



Introducción a la biología computacional

- El plan:
 - Algoritmos de reconocimiento de patrones:
 - Knuth-Morris-Pratt
 - Boyer-Moore
 - Árboles de sufijos
 - **Primeras aplicaciones de los árboles de sufijos**



Primeras aplicaciones de los árboles de sufijos

- [P1] Reconocimiento exacto de un patrón
 - Si el patrón, $P(1..m)$, y el texto, $T(1..n)$, se conocen a la vez, el coste con árboles de sufijos es igual que con el algoritmo KMP o el BM: $O(n+m)$
 - Si hay que hacer búsquedas de varios patrones en un mismo texto, tras un preprocesamiento (creación del árbol de sufijos del texto) con coste $O(n)$, cada búsqueda de las k apariciones de un patrón P en T lleva $O(m+k)$; en cambio, los algoritmos basados en preprocesar el patrón (como KMP) precisan $O(n+m)$ en cada búsqueda
 - Este problema fue el origen de los árboles de sufijos



Primeras aplicaciones de los árboles de sufijos

- [P2] Reconocimiento exacto de un conjunto de patrones
 - Existe un algoritmo de Aho y Corasick (no trivial, sección 3.4 del libro de D. Gusfield) para encontrar todas (las k) apariciones de un conjunto de patrones de longitud total m en un texto de longitud n con coste $O(n+m+k)$
 - Con un árbol de sufijos se obtiene exactamente la misma cota



Primeras aplicaciones de los árboles de sufijos

- [P3] Búsqueda de una subcadena en una base de datos (BD) de patrones
 - Ejemplo: identificación de un sospechoso en la escena del crimen
 - Se guarda (se “secuencia”) un pequeño intervalo del ADN de cada persona llamado *huella genética* (→ se tiene la BD de “fichados”)
 - El intervalo seleccionado es tal que:
 - puede ser aislado con fiabilidad mediante una PCR (“reacción en cadena de polimerasa”: proceso usado para hacer copias de un intervalo de ADN; es a los genes lo que la imprenta de Gutenberg es a la prensa escrita)
 - es una cadena muy variable (de manera que es un “identificador casi único” de la persona → “huella”)
 - Para identificar los restos encontrados en la escena del crimen se extrae ese mismo intervalo (o en muchos casos una subcadena, si el estado de los restos no permite extraer el intervalo completo) y se busca en la BD de “fichados”
(en realidad, se busca la subcadena común más larga entre el intervalo extraído de la escena y los intervalos de la BD, pero ese problema lo veremos más tarde)



Primeras aplicaciones de los árboles de sufijos

- Solución a [P3]: un *árbol de sufijos generalizado*
 - Sirve para guardar los sufijos de un conjunto de textos
 - Forma conceptualmente fácil de construirlo:
 - añadir un terminador distinto (y no perteneciente al alfabeto de caracteres de los textos) a cada texto del conjunto,
 - concatenar todas las cadenas resultantes, y
 - generar el árbol de sufijos para la cadena resultante.
 - Consecuencias:
 - El árbol resultante tendrá una hoja por cada sufijo de la cadena concatenada y se construye en tiempo proporcional a la suma de las longitudes de todos los textos
 - Los números (etiquetas) de las hojas pueden sustituirse por un par de números: uno identificando el texto al que pertenecen y el otro la posición de inicio en ese texto



Primeras aplicaciones de los árboles de sufijos

– Defecto del método:

- El árbol guarda sufijos que abarcan a más de uno de los textos originales, y que por tanto no interesan (“sintéticos”)

– Solución:

- Como el terminador de cada texto es diferente y no aparece en los textos originales, la etiqueta de todo camino de la raíz a cualquier nodo interno es subcadena de uno de los textos originales
- Por tanto los sufijos que no interesan siempre están en caminos de la raíz a las hojas, luego reduciendo el segundo índice (el que marca el fin de la subcadena) de la etiqueta de la arista que lleva a cada hoja se pueden eliminar todos los sufijos que no interesan (que abarcan a más de un texto)



Primeras aplicaciones de los árboles de sufijos

– Implementación más hábil:

- Simular el método anterior sin concatenar antes las cadenas
- Con dos cadenas S_1 y S_2 distintas:
 - 1º construir el árbol de sufijos de S_1 (suponiendo que tiene 1 carácter terminador)
 - 2º partiendo de la raíz del árbol anterior hacer coincidir S_2 con algún camino en el árbol hasta que haya una discrepancia
 - » suponer que los i primeros caracteres de S_2 coinciden
 - » entonces el árbol guarda todos los sufijos de S_1 y los de $S_2[1..i]$
 - » esencialmente, se han hecho las i primeras fases del algoritmo de Ukkonen para S_2 sobre el árbol de S_1
 - 3º continuar con ese árbol el alg. de Ukkonen para S_2 desde la fase $i+1$
 - » al acabar, el árbol tiene todos los sufijos de S_1 y de S_2 pero sin añadir sufijos “sintéticos”



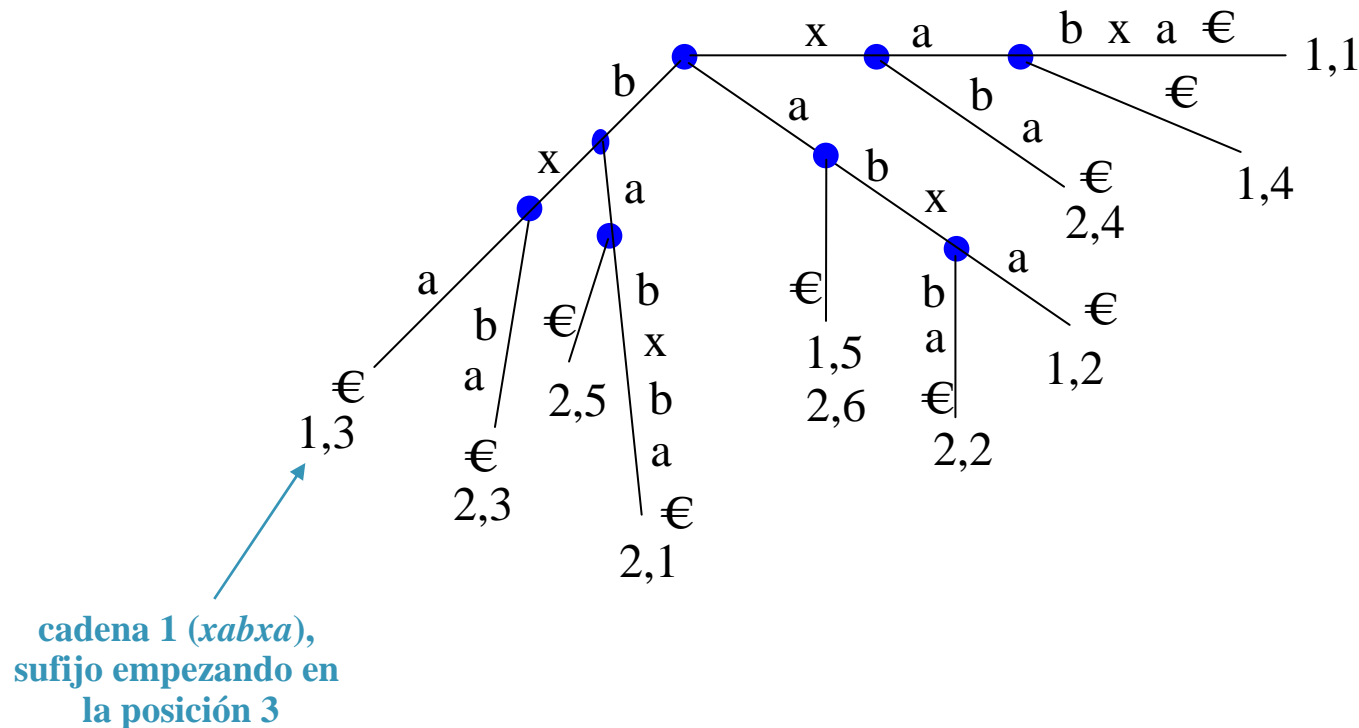
Primeras aplicaciones de los árboles de sufijos

- Para más cadenas: repetir el proceso para todas ellas
 - se crea el árbol de sufijos generalizado de todas en tiempo lineal en la suma de las longitudes de todas las cadenas
- Problemillas:
 - Las etiquetas comprimidas de las aristas pueden referirse a varias cadenas → hay que añadir un símbolo más a cada arista (ahora ya son tres)
 - Puede haber sufijos idénticos en dos (o más) cadenas, en ese caso una hoja representará todas las cadenas y posiciones de inicio del correspondiente sufijo



Primeras aplicaciones de los árboles de sufijos

- Ejemplo: resultado de añadir *babxba* al árbol de *xabxa*



cadena 1 (*xabxa*),
sufijo empezando en
la posición 3

Primeras aplicaciones de los árboles de sufijos

- Recapitulando, para resolver [P3]:
 - Construir el árbol de sufijos generalizado de las cadenas S_i de la BD en tiempo $O(m)$, con m la suma de las longitudes de todas las cadenas, y en espacio también $O(m)$
 - Una cadena P de longitud n se encuentra (o puede afirmarse que no está) en la BD en tiempo $O(n)$
 - Se consigue haciendo coincidir esa subcadena con un camino del árbol
 - Si P es una subcadena de una o varias cadenas de la BD, el algoritmo puede encontrar todas las cadenas de la BD que la contienen en tiempo $O(n+k)$ donde k es el nº de ocurrencias de la subcadena (recorriendo el subárbol de debajo del camino seguido haciendo coincidir P)
 - Si la cadena P no coincide con un camino en el árbol entonces ni P está en la BD ni es subcadena de ninguna cadena de la BD; en ese caso, el camino que ha coincidido define el prefijo más largo de P que es subcadena de una cadena de la BD



Primeras aplicaciones de los árboles de sufijos

- [P4] Subcadena común más larga de dos cadenas
 - Construir el árbol de sufijos generalizado de S_1 y S_2
 - Cada hoja del árbol representa o un sufijo de una de las dos cadenas o un sufijo de ambas cadenas
 - Marcar cada nodo interno v con un 1 (resp., 2) si hay una hoja en el subárbol de v que representa un sufijo de S_1 (respectivamente, S_2)
 - La etiqueta del camino de cualquier nodo interno marcado simultáneamente con 1 y 2 es una subcadena común de S_1 y S_2 y la más larga de ellas es la subcadena común más larga
 - Luego el algoritmo debe buscar el nodo marcado con 1 y 2 con etiqueta del camino más larga
 - El coste de construir el árbol y de la búsqueda es lineal



Primeras aplicaciones de los árboles de sufijos

- Por tanto: “Teorema. La subcadena común más larga de dos cadenas se puede calcular en tiempo lineal usando un árbol de sufijos generalizado.”
 - Aunque ahora parece fácil, D. Knuth en los 70’s conjeturó que sería imposible encontrar un algoritmo lineal para este problema...
- El problema de identificación del sospechoso [P3] reducido a encontrar la subcadena más larga de una cadena dada que sea también subcadena de alguna de las cadenas de una base de datos puede resolverse fácilmente extendiendo la solución del problema [P4].



Primeras aplicaciones de los árboles de sufijos

- Etcétera, etcétera:

- [P5] Reconocer contaminación de ADN

Dada una cadena S_1 que podría estar contaminada y S_2 (que es la posible fuente de contaminación) encontrar todas las subcadenas de S_2 que ocurren en S_1 y tienen longitud mayor que l .

- [P6] Subcadenas comunes a más de dos cadenas
- [P7] Compactación de un árbol de sufijos, usando un grafo dirigido y acíclico de palabras
- [P8] Uso inverso: árbol de sufijos del patrón buscado
- ... (ver libro de D. Gusfield)
- Aplicaciones concretas en proyectos “genoma”...
- Codificación de mínima longitud de ADN...
- Reconocimiento inexacto de patrones...
- Comparación múltiple de cadenas...



La investigación continúa...

- Ejemplo (de artículo reciente):

C.F. Cheung, J.X. Yu, H. Lu, “Constructing Suffix Tree for Gigabyte Sequences with Megabyte Memory”, *IEEE Transactions on Knowledge and Data Engineering*, Vol. 17, No. 1, January 2005.

“Suffix trees are widely acknowledged as a data structure to support exact/approximate sequence matching queries as well as repetitive structure finding efficiently when they can reside in main memory.

But, it has been shown as difficult to handle long DNA sequences using suffix trees due to the so-called **memory bottleneck problems**.

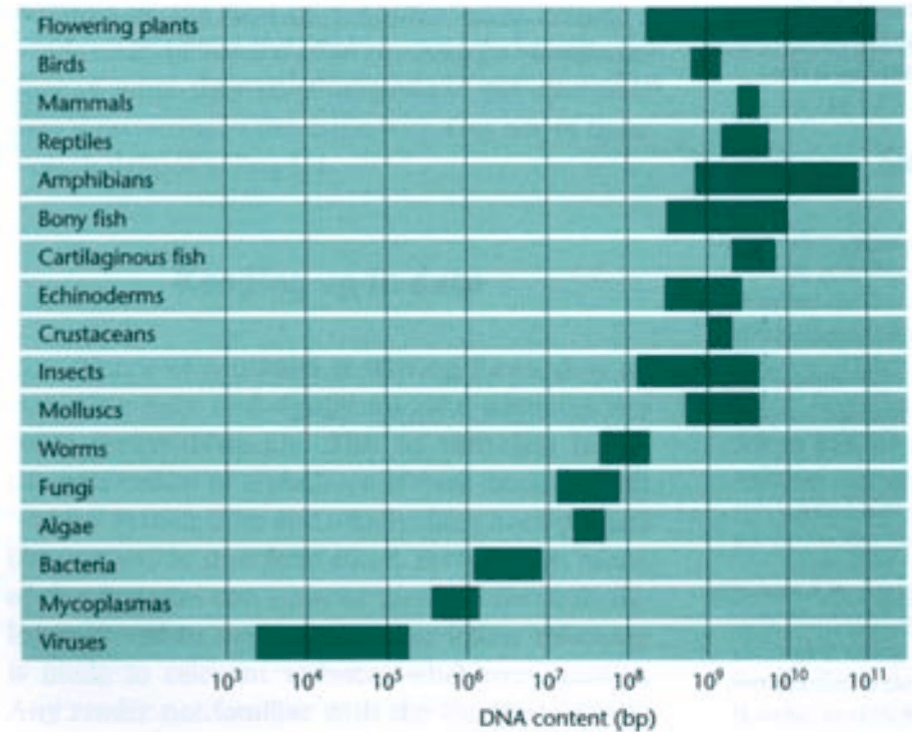
The most space efficient main-memory suffix tree construction algorithm takes nine hours and 45 GB memory space to index the human genome.”

El genoma de los mamíferos tiene habitualmente unos 3Gbps (*gigabase pairs* = 3 mil millones de nucleótidos),
3’2 Gbps en el caso del *Homo sapiens*.



La investigación continúa...

SPECIES	GENOME SIZE (KB)
<i>NAVICOLA PELLICULOSA</i> (DIATOM)	35,000
<i>DROSOPHILA MELANOGASTER</i> (FRUITFLY)	180,000
<i>PARAMECIUM AURELIA</i> (CILIATE)	190,000
<i>GALLUS DOMESTICUS</i> (CHICKEN)	1,200,000
<i>CYPRINUS CARPIO</i> (CARP)	1,700,000
<i>BOA CONSTRICTOR</i> (SNAKE)	2,100,000
<i>RATTUS NORVEGICUS</i> (RAT)	3,100,000
<i>XENOPUS LAEVIS</i> (TOAD)	3,100,000
<i>HOMO SAPIENS</i> (BUBBA)	3,200,000
<i>NICOTIANA TABACUM</i> (TOBACCO)	3,800,000
<i>PARAMECIUM CAUDATUM</i> (CILATE)	8,600,000
<i>ALLIUM CEPA</i> (ONION)	18,000,000
<i>LILIUM FORMOSANUM</i> (LILY)	36,000,000
<i>AMPHIUMA MEANS</i> (NEWT)	68,000,000
<i>PINUS RESINOSA</i> (PINE)	84,000,000
<i>PROTOPTERUS AETHIOPICUS</i> (LUNGFISH)	140,000,000
<i>OPHIOGLOSSUM PETIOLATUM</i> (FERN)	160,000,000
<i>AMOEBEA DUBIA</i>	670,000,000



Las variaciones en la longitud del genoma se deben fundamentalmente a las diferencias entre la cantidad y tipo de **secuencias repetidas (!!!)**

➡ En lo que a genoma se refiere, el tamaño no importa, lo que importa es la cantidad de información.

La investigación continúa...

“In this paper, we show that suffix trees for long DNA sequences can be **efficiently constructed on disk** using small bounded main memory space and, therefore, **all existing algorithms based on suffix trees can be used to handle long DNA sequences that cannot be held in main memory.**

We adopt a two-phase strategy to construct a suffix tree on disk: 1) to construct a diskbase suffix-tree without suffix links and 2) rebuild suffix links upon the suffix-tree being constructed on disk, if needed.

We propose a new disk-based suffix tree construction algorithm, called DynaCluster, which shows $O(n \log n)$ experimental behavior regarding CPU cost and linearity for I/O cost.

DynaCluster needs 16MB main memory only to construct more than 200Mbps DNA sequences and significantly outperforms the existing disk-based suffix-tree construction algorithms using prepartitioning techniques in terms of both construction cost and query processing cost.

We conducted extensive performance studies and report our findings in this paper.”



Fin

