Estructuras de datos (ED) avanzadas

- Análisis amortizado de ED
  - conceptos básicos
  - conjuntos disjuntos
  - listas auto-organizativas
  - árboles *splay*
- ED aleatorizadas
  - listas *skip*
  - árboles aleatorizados (*treaps*)
- ED en biología computacional
  - introducción
  - árboles de prefijos
  - árboles de sufijos
  - aplicaciones

Estructuras de datos (ED) avanzadas

- **Análisis amortizado de ED**
  - **conceptos básicos**
  - conjuntos disjuntos
  - listas auto-organizativas
  - árboles *splay*

- ED aleatorizadas
  - listas *skip*
  - árboles aleatorizados (*treaps*)

- ED en biología computacional
  - introducción
  - árboles de prefijos
  - árboles de sufijos
  - aplicaciones

Análisis amortizado de ED: conceptos básicos

- "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… $~€ € %$€ ~€¬& ¬ &@# …

Análisis amortizado de ED: 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 asignaturas anteriores (la probabilidad ahora no interviene)

Análisis amortizado de ED: 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.

Análisis amortizado de ED: conceptos básicos

- **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.

Análisis amortizado de ED: conceptos básicos

- 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≠0 hacer
    desapilar(p); k:=k-1
  fmq
fin
```

Análisis amortizado de ED: conceptos básicos

- 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…

Análisis amortizado de ED: conceptos básicos

- 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: $O(n)/n = O(1)$.

Análisis amortizado de ED: conceptos básicos

- 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*…

Análisis amortizado de ED: conceptos básicos

```
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 en serie con acarreo (“ripple-carry”).

Análisis amortizado de ED: conceptos básicos

– 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
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

en relieve: los bits que cambian para pasar al siguiente valor del contador

el coste de cada incremento es lineal en el número de bits que cambian de valor

nótese que el coste acumulado nunca es mayor del doble del número de incrementos

Análisis amortizado de ED: conceptos básicos

- 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, $i = 0, 1, \ldots, \lfloor \log n \rfloor$

Análisis amortizado de ED: conceptos básicos

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

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

$$\sum_{i=0}^{\infty} c^i = \frac{1}{1-c}$$

- 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 $O(n)/n = O(1)$.

Análisis amortizado de ED: conceptos básicos

- **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:

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

    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.

Análisis amortizado de ED: conceptos básicos

- Sumando el potencial de todas las operaciones:

$$\sum_{i=1}^{n} P(i) = \sum_{i=1}^{n} \left(A(i) - C(i) + P(i-1) \right)$$

$$\Rightarrow \sum_{i=1}^{n} \left(P(i) - P(i-1) \right) = \sum_{i=1}^{n} \left(A(i) - C(i) \right)$$

$$\Rightarrow P(n) - P(0) = \sum_{i=1}^{n} \left(A(i) - C(i) \right) \Rightarrow P(n) - P(0) \geq 0$$

(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.

Análisis amortizado de ED: conceptos básicos

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

Análisis amortizado de ED: conceptos básicos

- 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*.

Análisis amortizado de ED: conceptos básicos

- 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.

Análisis amortizado de ED: conceptos básicos

– 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

Análisis amortizado de ED: conceptos básicos

- **Método potencial**
  - Consideramos, de nuevo, una función de potencial:

    $$P(i) = A(i) - C(i) + P(i-1),\ \ i = 1, 2, \ldots, n \ ^{(*)}$$

    tal que:

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

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

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

  - El gran problema: ¿cómo elegir el potencial?

Análisis amortizado de ED: conceptos básicos

- 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.

Análisis amortizado de ED: conceptos básicos

- 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 $O(n)$.

Análisis amortizado de ED: conceptos básicos

- 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
  fmq;
  si i<length(A) entonces
    A[i]:=1
  fsi
fin
```

Análisis amortizado de ED: conceptos básicos

- Cálculo del coste amortizado (cont.):
  - $A(i) = C(i) + P(i) - P(i-1) \leq (t_i + 1) + (b_{i\text{-}1} - t_i + 1) - b_{i\text{-}1}$
    $= 2$
  - 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.
  - $A(i) = C(i) + P(i) - P(i-1),\ \ i = 1, 2, \ldots, n \Rightarrow$

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

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

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:

$$\sum_{i=1}^{n} C(i) \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j$$

$$\sum_{i=0}^{n} c^i = \frac{c^{n+1} - 1}{c - 1}, \quad c \neq 1 \longrightarrow \begin{aligned} &< n + 2n \\ &= 3n \end{aligned}$$

  - 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: $P(i) = 2 \cdot numdatos(i) - capacidad(i)$
      - Tras una expansión, $numdatos(i) = capacidad(i)/2$, luego $P(i) = 0$.
      - Inmediatamente antes de una expansión, $numdatos(i) = capacidad(i)$, luego $P(i) = numdatos(i)$.
      - El valor inicial es 0, y como la tabla siempre está medio llena, $numdatos(i) \geq capacidad(i)/2$, 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:

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

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

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

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 ½.
    - 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 ¼ de su capacidad**.
    - De esta forma, tras una expansión el factor de carga es ½ y por tanto la mitad de los elementos deben ser borrados para que ocurra una contracción.
    - Tras una contracción el factor de carga es ½ 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, $\alpha(i) = numdatos(i)/capacidad(i)$, crece hacia 1 o disminuye hacia ¼.
    - Como en una tabla vacía $numdatos = capacidad = 0$ y $\alpha = 1$, siempre se tiene que $numdatos(i) = \alpha(i) \cdot capacidad(i)$.
    - Por ejemplo:

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

      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 $\alpha(i) = ½$, 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 $\alpha(i) = ¼$, $capacidad(i) = 4 \cdot numdatos(i)$, 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 $\alpha(i-1) \geq ½$, 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 $\alpha(i-1) < ½$, la tabla no precisa expandirse, y hay dos casos:
      - » Si $\alpha(i) < ½$:

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

Primer ejemplo: análisis de tablas dinámicas

» Si $\alpha(i-1) < ½$ pero $\alpha(i) \geq ½$:

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

- 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 $\alpha(i-1) < ½$ y la operación no provoca una contracción, entonces $capacidad(i) = capacidad(i-1)$ y el coste es:

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

Primer ejemplo: análisis de tablas dinámicas

- Si $\alpha(i-1) < ½$ y la operación provoca una contracción:

  $C(i) = numdatos(i) + 1$, porque se borra en dato y se mueven $numdatos(i)$

  $capacidad(i)/2 = capacidad(i-1)/4 = numdatos(i) + 1$

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

  - Si $\alpha(i-1) \geq ½$ el coste amortizado también se puede acotar con una constante (**ejercicio**).

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

Estructuras de datos (ED) avanzadas

- Análisis amortizado de ED
  - conceptos básicos
  - **conjuntos disjuntos**
  - listas auto-organizativas
  - árboles *splay*
- ED aleatorizadas
  - listas *skip*
  - árboles aleatorizados (*treaps*)
- ED en biología computacional
  - introducción
  - árboles de prefijos
  - árboles de sufijos
  - aplicaciones

Análisis amortizado de ED: conjuntos disjuntos

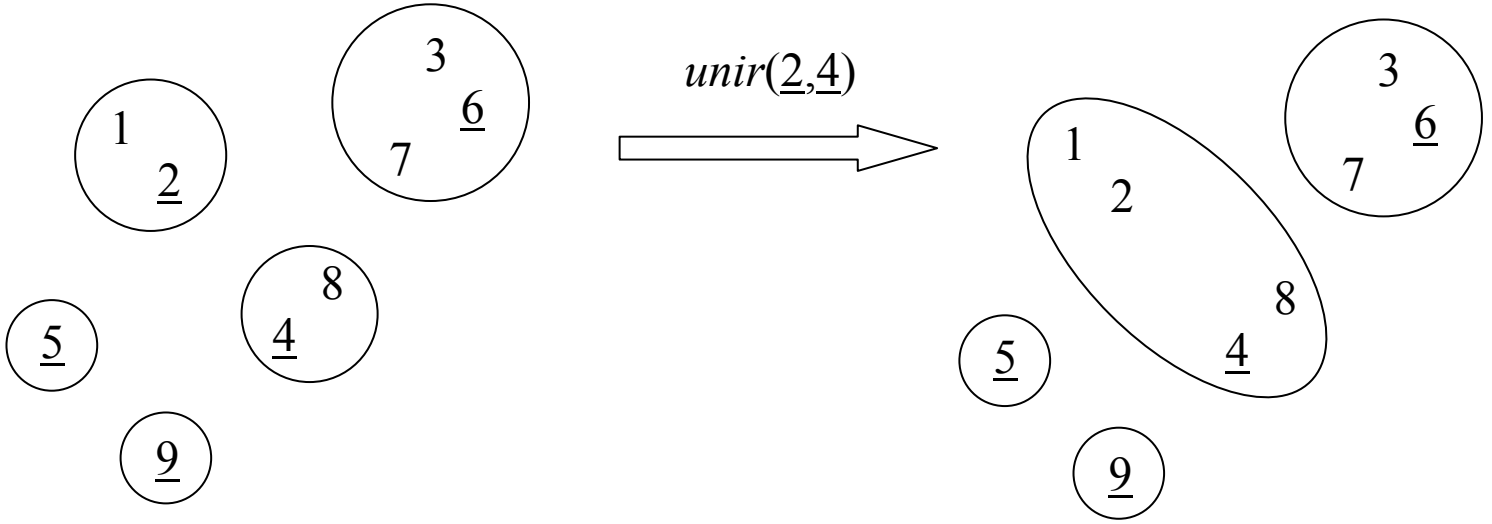
- Motivación:
  - Algoritmo de Kruskal para el cálculo del árbol de recubrimiento de peso mínimo de un grafo

```
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 "Algorirmia Básica")

Análisis amortizado de ED: conjuntos disjuntos

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



encontrar(3)=6

Análisis amortizado de ED: conjuntos disjuntos

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

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

  - 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

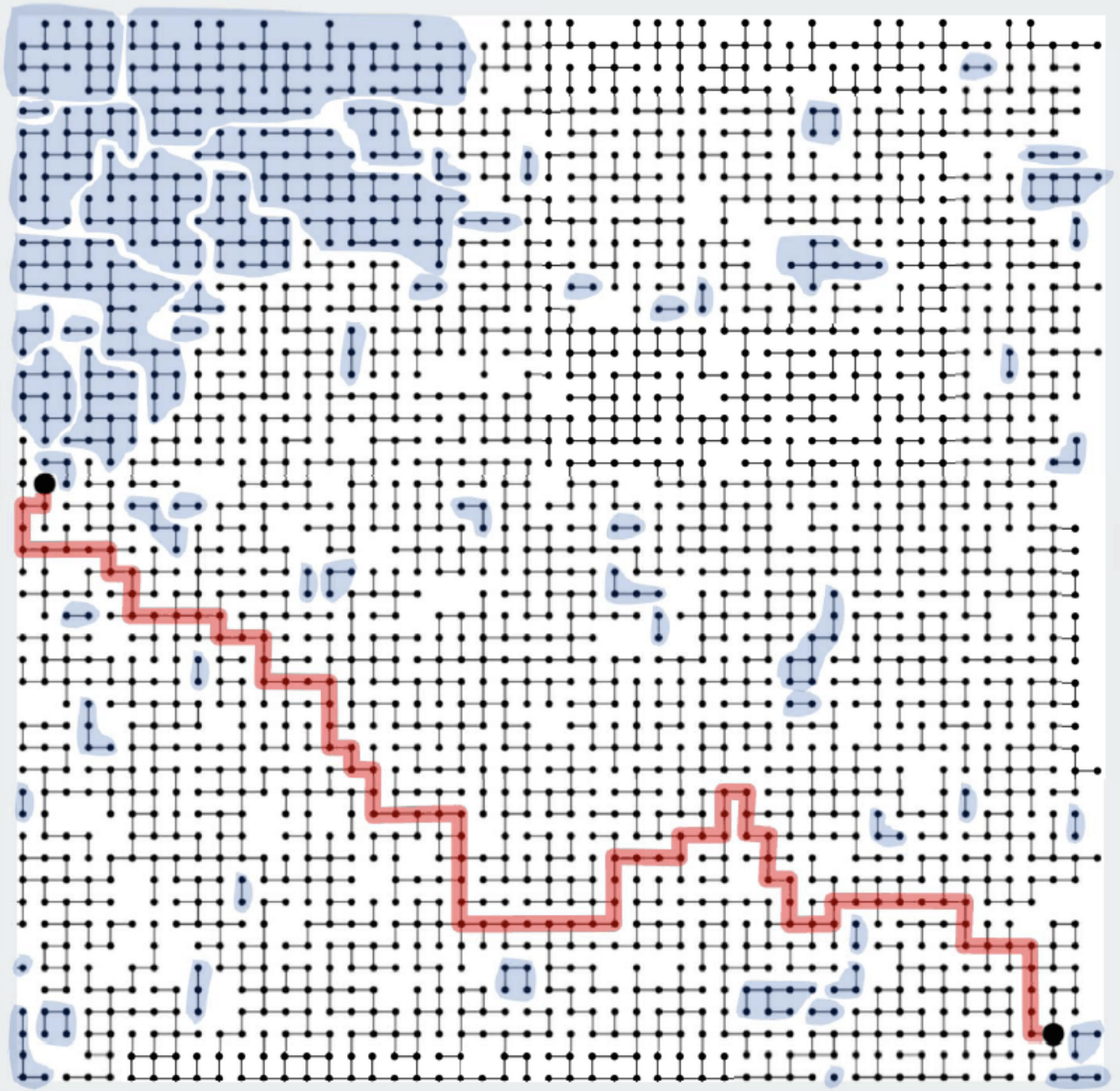
Análisis amortizado de ED: conjuntos disjuntos

- 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.
- Diseño de VLSI.


u
¿encontrar(u)
=
encontrar(v)?
v

¿encontrar(u) = encontrar(v)?

verdad



63 componentes

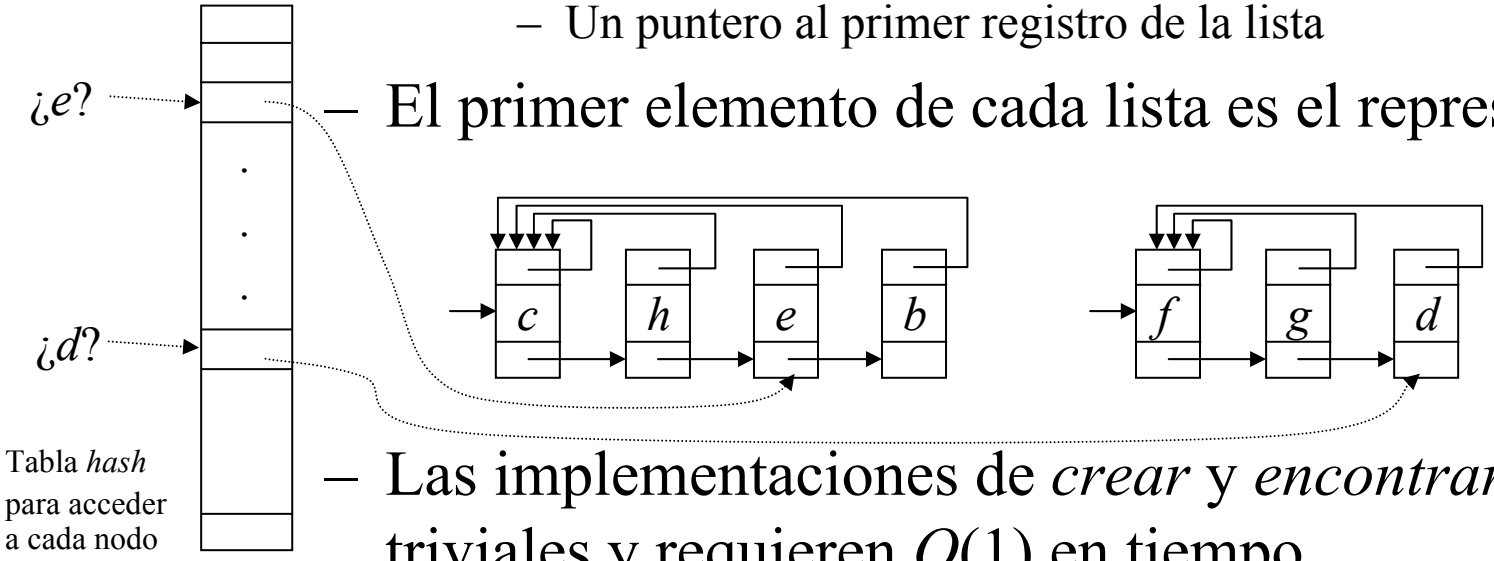
Análisis amortizado de ED: conjuntos disjuntos

- 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
```

Análisis amortizado de ED: conjuntos disjuntos

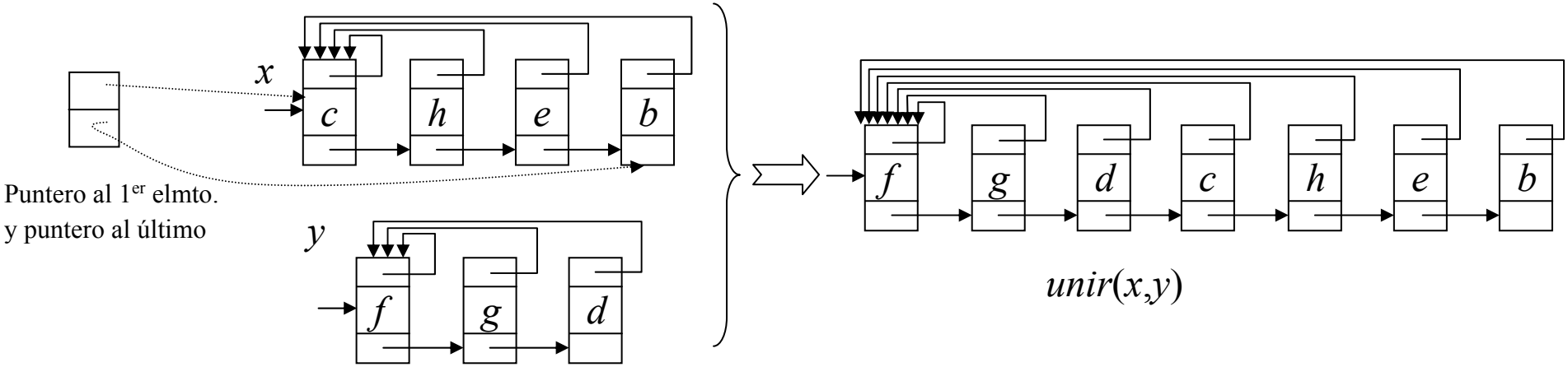
- Primera implementación: listas encadenadas
  - Cada conjunto se almacena en una lista encadenada
    - Cada registro de la lista contiene:
      - Un elemento del conjunto
      - Un puntero al siguiente registro de la lista
      - Un puntero al primer registro de la lista
  - El primer elemento de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



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

Análisis amortizado de ED: 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)…

Análisis amortizado de ED: conjuntos disjuntos

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

operación	nº de objetos modificados
crear(x_1)	1
crear(x_2)	1
…	…
crear(x_n)	1
unir(x_1, x_2)	1
unir(x_2, x_3)	2
unir(x_3, x_4)	3
…	…
unir(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), $\Theta(m)$.

Análisis amortizado de ED: 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.

Análisis amortizado de ED: 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 $O(m + n\log n)$.

Análisis amortizado de ED: conjuntos disjuntos

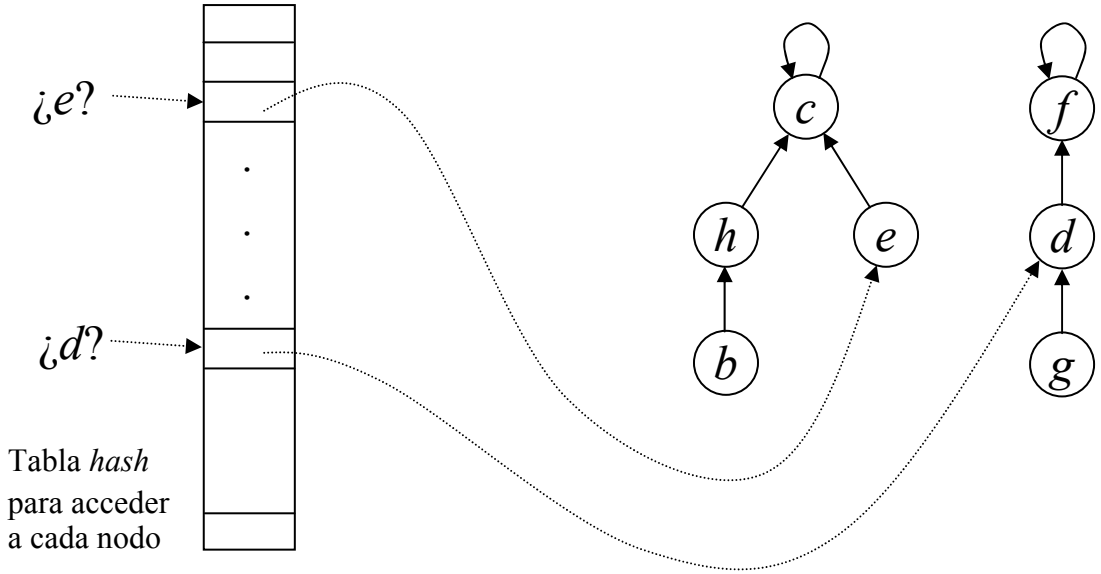
- Ejercicio:

  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 $O(\log n)$.

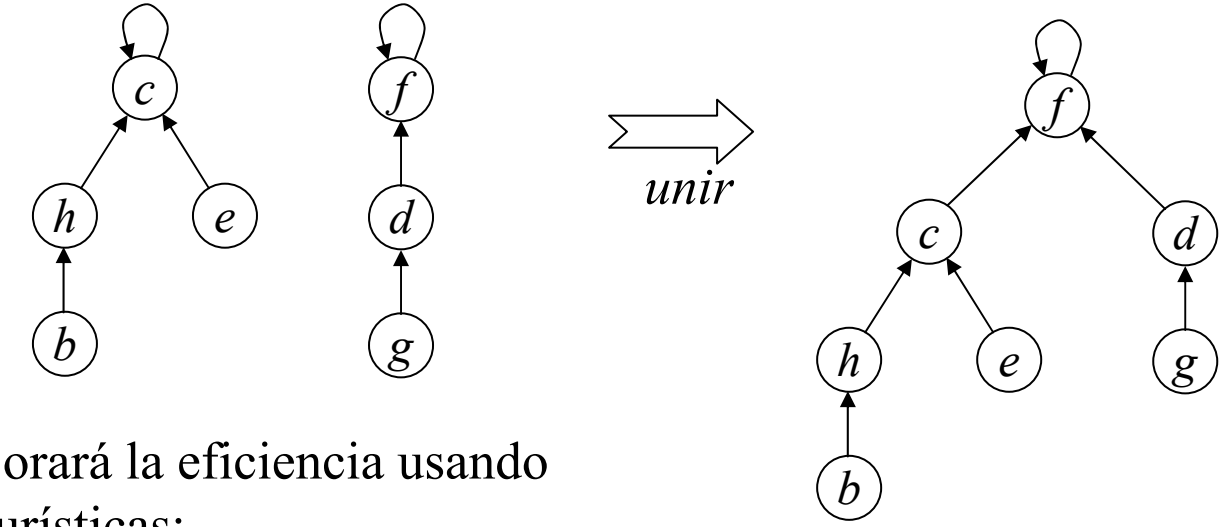
Análisis amortizado de ED: conjuntos disjuntos

- 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.



Análisis amortizado de ED: 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*

Análisis amortizado de ED: 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.

Análisis amortizado de ED: 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
```

Análisis amortizado de ED: conjuntos disjuntos

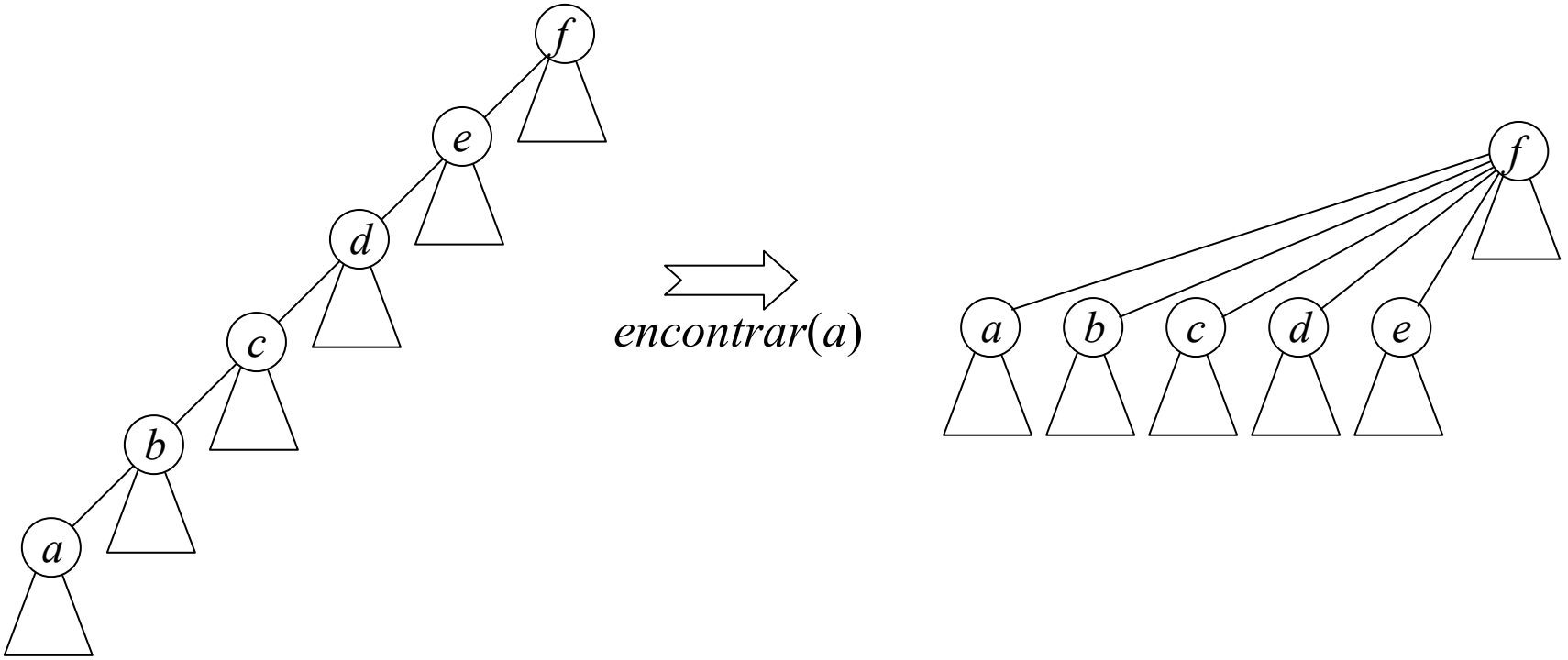
- Implementación en el bosque de la heurística de "compresión de caminos"
  - Muy simple y muy efectiva.
  - Se usa en la operación de *encontrar*.
  - Hace que los nodos recorridos en el *camino de búsqueda* pasen a apuntar directamente a la raíz.
  - No se modifica la información sobre el rango.

```
función 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

Análisis amortizado de ED: conjuntos disjuntos

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



Análisis amortizado de ED: 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…

Análisis amortizado de ED: 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.

Análisis amortizado de ED: conjuntos disjuntos

- ¿Cómo es la función $\alpha(m,n)$?
  - Funciones previas:
    - Para todo entero $i \geq 0$, abreviamos $2^{2^{\cdot^{\cdot^{2}}}}\Big\}\, i$ 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:

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

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

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

Análisis amortizado de ED: conjuntos disjuntos

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

  $A(1,j) = 2^j$, para $j \geq 1$

  $A(i,1) = A(i-1,2)$, para $i \geq 2$

  $A(i,j) = A(i-1,A(i,j-1))$, 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}	$2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}16$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}16$	$2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}2^{2^{\cdot^{\cdot^{\cdot^2}}}}\}16$

Análisis amortizado de ED: 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, $A(4,\lfloor m/n \rfloor) \geq A(4,1) = A(3,2) = 2^{2^{\cdot^{\cdot^{2}}}}\Big\}16$
    - 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.

Análisis amortizado de ED: 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} ($\approx 2*10^{19728}$), 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 $O(m\ \alpha(m,n))$

Análisis amortizado de ED: 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.

Análisis amortizado de ED: conjuntos disjuntos

[P_4] – Si se define el *cardinal*(x) como el nº de nodos del árbol con raíz x (incluido x), entonces

$$cardinal(x) \geq 2^{rango(x)}$$

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

  $$cardinal(x) \geq 1 = 2^0 = 2^{rango(x)}$$

Análisis amortizado de ED: conjuntos disjuntos

- Paso de inducción:

  Suponer que es cierto antes de *enlazar* x e y que $cardinal(x) \geq 2^{rango(x)}$ y $cardinal(y) \geq 2^{rango(y)}$

  – si $rango(x) \neq rango(y)$, asumir que $rango(x) < rango(y)$, entonces y es la raíz del árbol resultante y se tiene que:

  $$cardinal'(y) = cardinal(x) + cardinal(y) \geq$$
  $$\geq 2^{rango(x)} + 2^{rango(y)} \geq 2^{rango(y)} = 2^{rango'(y)},$$

  y no cambian más rangos ni cardinales

  – si $rango(x) = rango(y)$, el nodo y es de nuevo la raíz del nuevo árbol y se tiene que:

  $$cardinal'(y) = cardinal(x) + cardinal(y) \geq$$
  $$\geq 2^{rango(x)} + 2^{rango(y)} = 2^{rango(y)+1} = 2^{rango'(y)}$$

Análisis amortizado de ED: 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

Análisis amortizado de ED: 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

Análisis amortizado de ED: 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'.

Análisis amortizado de ED: 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
```

Análisis amortizado de ED: conjuntos disjuntos

- Veamos la operación *encontrar*, antes… algo de notación:
  - Agrupamos los rangos de los nodos en *bloques*: el rango r está en el bloque que numeramos con $\log^* r$, para $r = 0,1,\ldots,\lfloor \log n \rfloor$ (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 $j \geq -1$,

$$B(j) = \begin{cases} -1, & \text{si } j = -1 \\ 1, & \text{si } j = 0 \\ 2, & \text{si } j = 1 \\ 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}\Big\}{\scriptstyle j\text{-}1}, & \text{si } j \geq 2 \end{cases}$$

  - Entonces, para $j = 0,1,\ldots,\log^* n - 1$, el bloque j-ésimo consiste en el conjunto de rangos $\{B(j-1)+1, B(j-1)+2, \ldots, B(j)\}$.

Bloque 0:
{0,1}
Bloque 1:
{2}
Bloque 2:
{3,4}
Bloque 3:
{5,…,16}
Bloque 4:
{17,…,65536}
Bloque 5:
{65537,..,BIG}

log*2=1
log*4=2
log*16=3
log*65536=4
log*(2^{65536})=5

Análisis amortizado de ED: 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 $x_0, x_1, \ldots, x_l$, con $x_i = padre(x_{i-1})$, $i = 1, 2, \ldots, l$, y $x_l = padre(x_l)$, 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 $j = 0,1,\ldots,\log^* n - 1$; es decir, a todo x_i tal que $\log^* rango(x_i) < \log^* rango(x_{i+1})$; y a
  - el hijo de la raíz, es decir, $x_i \neq x_l$ tal que $padre(x_i) = x_l$.
- 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.

Análisis amortizado de ED: 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 $x_i \neq padre(x_l)$, 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 $\log^* rango(padre(x_i))$ y $\log^* rango(x_i)$;

  – 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.

Análisis amortizado de ED: 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 $0,1,\ldots, \log^* n - 1$, 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…

Análisis amortizado de ED: conjuntos disjuntos

- Acotación del nº 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 $padre(x_i) \neq x_l$ 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 nº 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 $B(j-1)+2$, $B(j-1)+3$, …, $B(j)$.
    - » Es decir, puede ocurrir $B(j) - B(j-1) - 1$ veces.

Análisis amortizado de ED: 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 $j \geq 0$.

Análisis amortizado de ED: 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:

$$P(n) \leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} (B(j) - B(j-1) - 1)$$

$$\leq \sum_{j=0}^{\log^* n - 1} \frac{3n}{2B(j)} B(j) = \frac{3}{2} n \log^* n$$

- 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 $O(m \log^* n)$.

Estructuras de datos (ED) avanzadas

- Análisis amortizado de ED
  - conceptos básicos
  - conjuntos disjuntos
  - **listas auto-organizativas**
  - árboles *splay*
- ED aleatorizadas
  - listas *skip*
  - árboles aleatorizados (*treaps*)
- ED en biología computacional
  - introducción
  - árboles de prefijos
  - árboles de sufijos
  - aplicaciones

Análisis amortizado de ED: listas auto-organizativas

- Motivación:
  - En el uso de tablas de símbolos:
    localidad de referencia temporal = tendencia a arracimarse los accesos a un símbolo determinado en una secuencia de operaciones.

    Ejemplos:
    - En un programa: $n:=n+1$ → 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

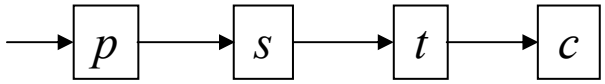
    ➔ estructuras de datos auto-organizativas

Análisis amortizado de ED: listas 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 ése 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*

Análisis amortizado de ED: listas 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).

Análisis amortizado de ED: listas 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)

Análisis amortizado de ED: listas 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

Análisis amortizado de ED: listas 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

Análisis amortizado de ED: listas 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

Análisis amortizado de ED: listas 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

Análisis amortizado de ED: listas 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*

Análisis amortizado de ED: listas 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

Análisis amortizado de ED: listas 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

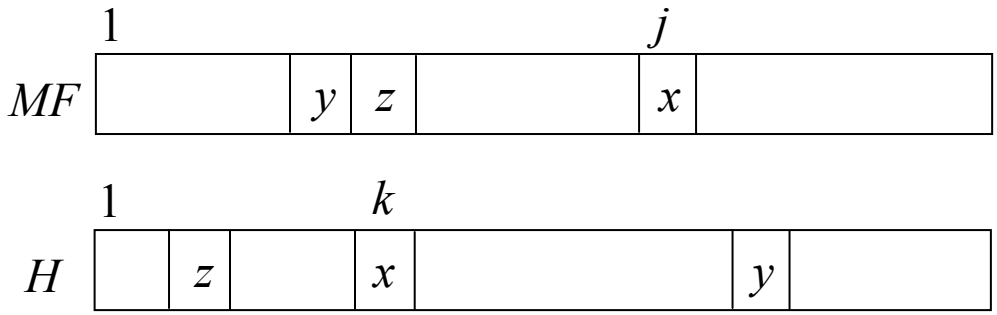
- En las dos listas finales del ejemplo: (a,b), (d,b), (c,b)

	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

Análisis amortizado de ED: listas auto-organizativas

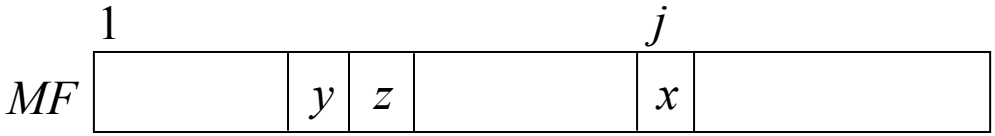
- Cálculo del coste total amortizado de la secuencia de operaciones *p* con la heurística mover al frente:
  - Mediante el método potencial, usaremos como función de potencial el número de inversiones entre la lista *MF* y la lista *H*
  - Suponer que la *i*-ésima operación es 'buscar(*x*)' y que *x* es el *j*-ésimo elemento de la lista *MF* y el *k*-ésimo de la lista *H*

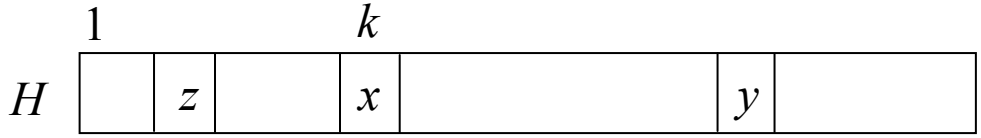


  - El coste de la búsqueda (en *MF*) es *j*
  - Después *x* se desplaza al primer lugar en *MF* y e_i lugares hacia la izquierda en *H*

Análisis amortizado de ED: listas 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

Análisis amortizado de ED: listas auto-organizativas

– Según el método potencial: *(Análisis de la búsqueda)*

$$A(i) = C(i) + P(i) - P(i-1)$$
$$= j - c + (j - 1 - c) - e_i = 2(j - 1 - c) - e_i + 1$$

– 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

Estructuras de datos (ED) avanzadas

- Análisis amortizado de ED
  - conceptos básicos
  - conjuntos disjuntos
  - listas auto-organizativas
  - **árboles *splay***

- ED aleatorizadas
  - listas *skip*
  - árboles aleatorizados (*treaps*)

- ED en biología computacional
  - introducción
  - árboles de prefijos
  - árboles de sufijos
  - aplicaciones

Análisis amortizado de ED: árboles *splay*

- 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 $O(\log n)$

Análisis amortizado de ED: árboles *splay*

- 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

Análisis amortizado de ED: árboles *splay*

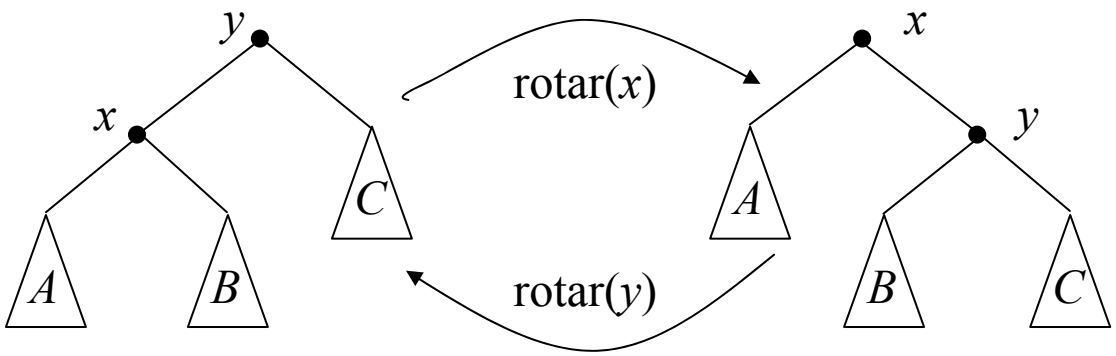
- Operación básica: *splay*
  - splay(i,S): reorganiza el árbol S de manera que la nueva raíz es:
    - el elemento i si i pertenece a S, o
    - $\max\{k \in S \mid k < i\}$ ó $\min\{k \in S \mid k > i\}$, si i no pertenece a S
- Las otras operaciones:
  - buscar(i,S): dice si i está en S
  - insertar(i,S): inserta i en S si no estaba ya
  - borrar(i,S): borra i de S si estaba
  - concatenar(S,S'): une S y S' en un solo árbol, suponiendo que $x < y$ para todo $x \in S$ y todo $y \in S'$
  - separar(i,S): separa S en S' y S'' tales que $x \leq i \leq y$ para todo $x \in S'$ y todo $y \in S''$

Análisis amortizado de ED: árboles *splay*

- 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*(+∞,*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

Análisis amortizado de ED: árboles *splay*

- Implementación de la operación *splay*
  - Recordar las rotaciones:



  - Preservan la estructura de árbol de búsqueda

Análisis amortizado de ED: árboles *splay*

- 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, …, *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

$$\sum_{i=1}^{n-1} i = \Omega(n^2)$$

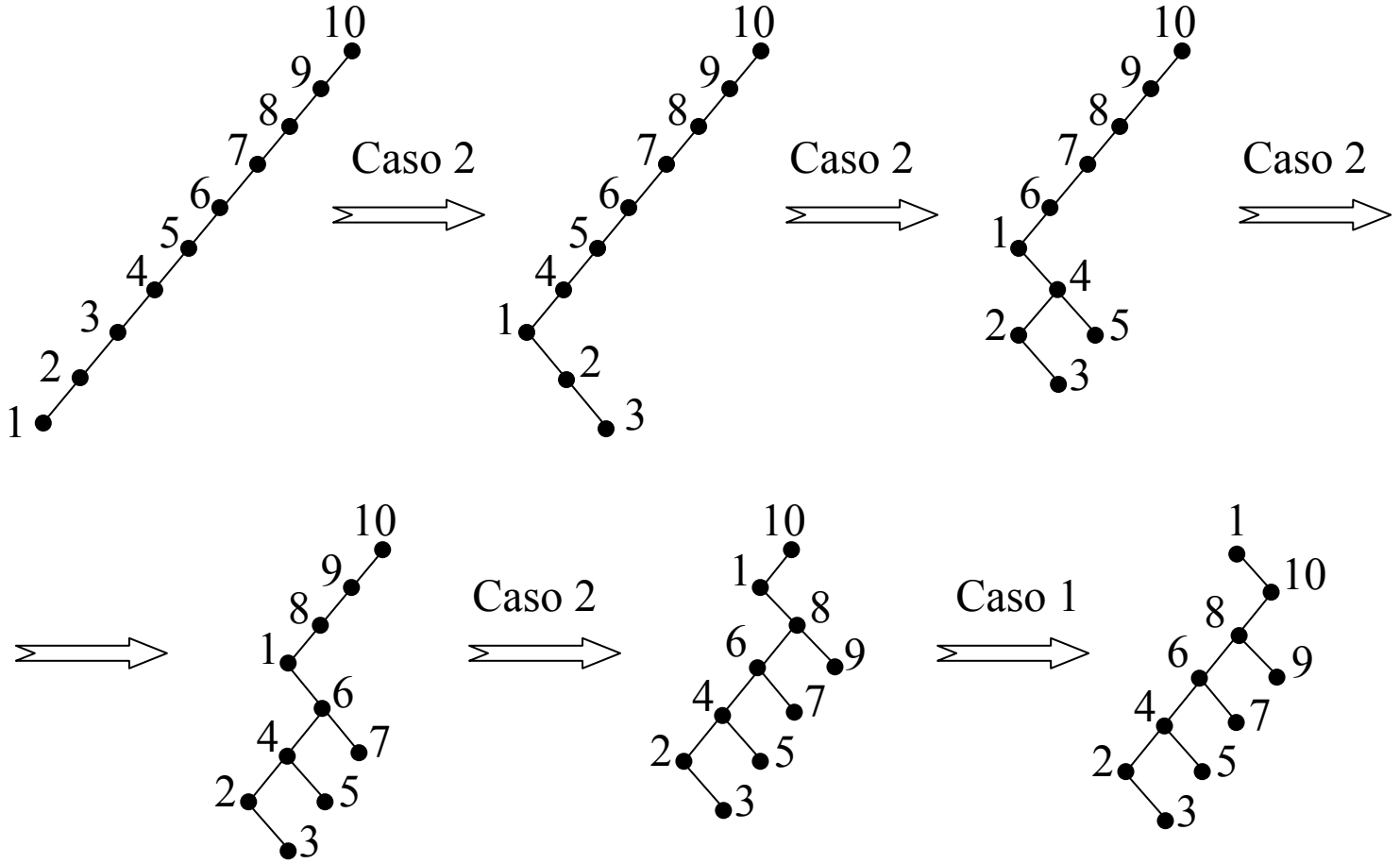
    - tras todas las búsquedas el árbol ha quedado como al principio, así que si se vuelve a repetir la secuencia de búsquedas en orden, se llega a $\Omega(mn)$ para *m* operaciones.

Análisis amortizado de ED: árboles *splay*

- La solución adoptada (para subir x a la raíz) es la siguiente:
  - **Caso 1:** si x tiene padre pero no tiene abuelo, se ejecuta 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 rotar(y) y luego 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 rotar(x) y luego rotar(x) otra vez (con el padre en ambos casos)

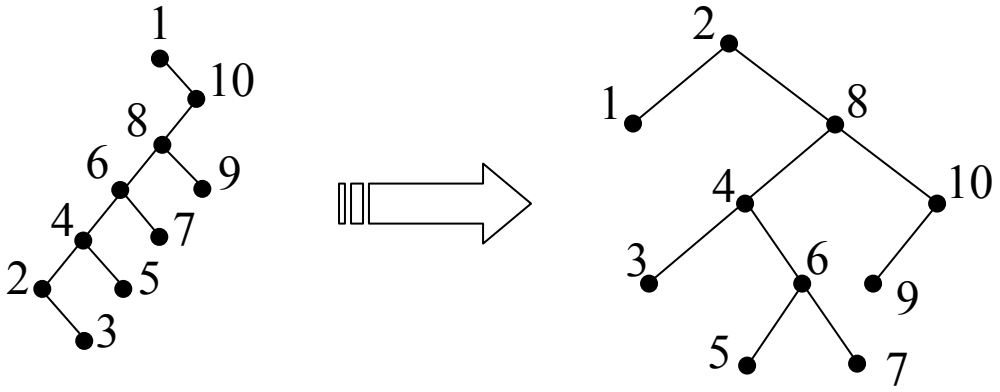
Análisis amortizado de ED: árboles *splay*

– Ejemplo: splay(1,*S*), con *S* el árbol degenerado



Análisis amortizado de ED: árboles *splay*

– Sigue el ejemplo: splay(2,*S*) al resultado anterior…



- Nótese que el árbol está más equilibrado con cada *splay*

Análisis amortizado de ED: árboles *splay*

- 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)

Análisis amortizado de ED: árboles *splay*

– **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.

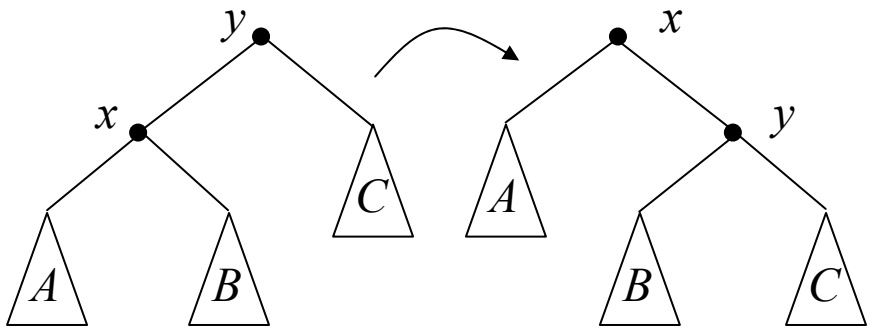
Demostración:

- 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

Análisis amortizado de ED: árboles *splay*

- **Caso 1:** z no existe
  - Es la última rotación en la operación de *splay*
  - Se ejecuta 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, $A(i)$)

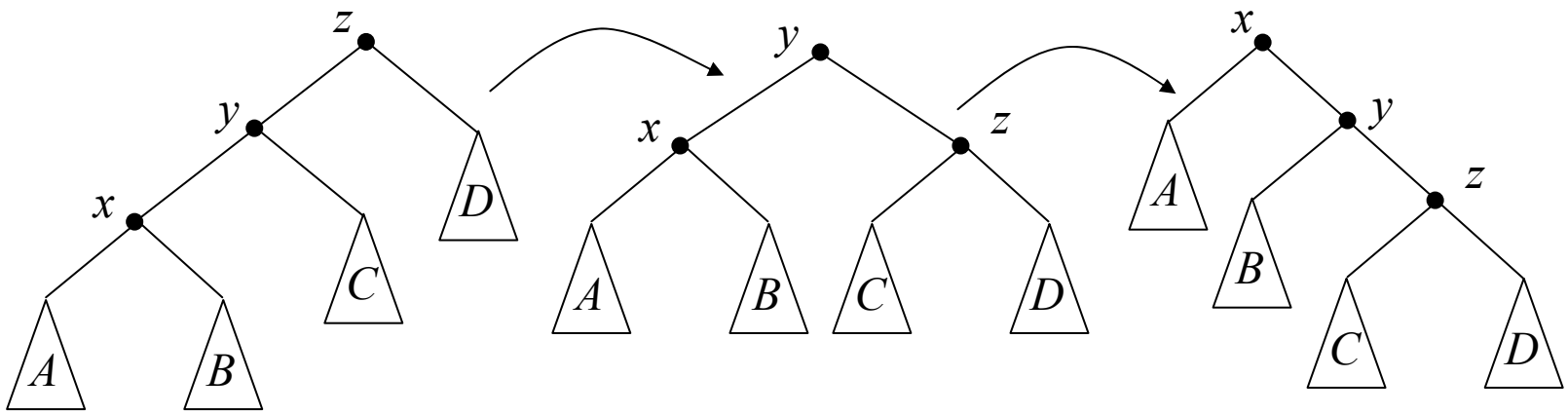
$$3(\mu'(x) - \mu(x)) + 1$$

unidades de crédito por la rotación

recordar que $A(i) = C(i) + P(i) - P(i-1)$

Análisis amortizado de ED: árboles *splay*

- **Caso 2:** x e y son ambos hijos izquierdos o hijos derechos
  - Se ejecuta rotar(y) y luego 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)

Análisis amortizado de ED: árboles *splay*

– 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:

$$\begin{aligned}
&\mu'(x) + \mu'(y) + \mu'(z) - \mu(x) - \mu(y) - \mu(z) \\
&\quad = \mu'(y) + \mu'(z) - \mu(x) - \mu(y) \\
&\quad = (\mu'(y) - \mu(x)) + (\mu'(z) - \mu(y)) \\
&\quad \leq (\mu'(x) - \mu(x)) + (\mu'(x) - \mu(x)) \\
&\quad = 2(\mu'(x) - \mu(x))
\end{aligned}$$

– Aún sobrarían $\mu'(x) - \mu(x)$ unidades para pagar las operaciones elementales de las dos rotaciones (comparaciones y manipulación de punteros), salvo que $\mu'(x) = \mu(x)$ …

Análisis amortizado de ED: árboles *splay*

– Veamos que en este caso ($\mu'(x) = \mu(x)$) la cantidad precisa para mantener el invariante $^{(*)}$ es negativa y por tanto es "gratis" mantenerlo:

$$\left.\begin{array}{l} \mu'(x) = \mu(x) \\ \mu'(x) + \mu'(y) + \mu'(z) \geq \mu(x) + \mu(y) + \mu(z) \end{array}\right\} \Longrightarrow \text{contradicción}$$

En efecto: $\mu(z) = \mu'(x) = \mu(x)$

$$\Rightarrow \quad \mu(x) = \mu(y) = \mu(z)$$

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

$$\mu'(x) = \mu'(y) = \mu'(z)$$

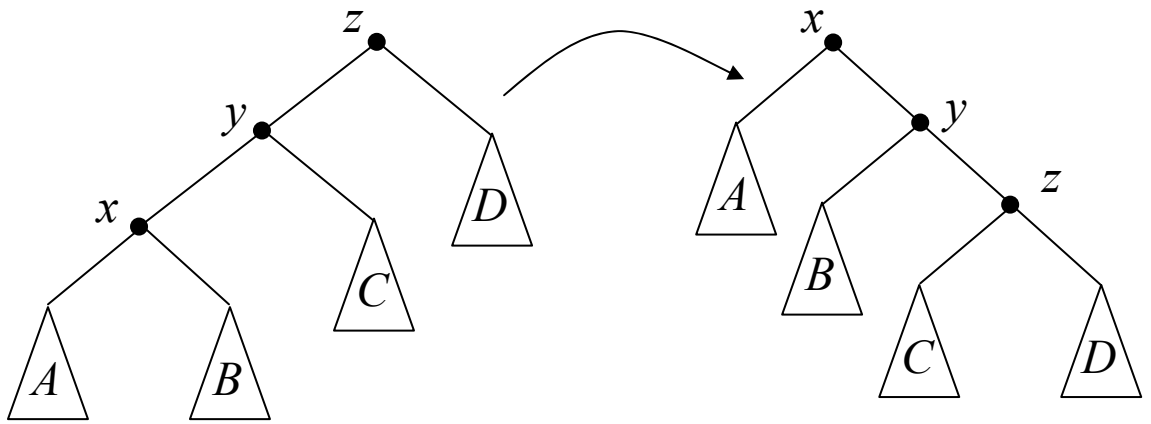
Y como $\mu(z) = \mu'(x)$ entonces

$$\mu(x) = \mu(y) = \mu(z) = \mu'(x) = \mu'(y) = \mu'(z)$$

Análisis amortizado de ED: árboles *splay*

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 $\lfloor \log a \rfloor = \lfloor \log (a + b + 1) \rfloor = \lfloor \log b \rfloor$

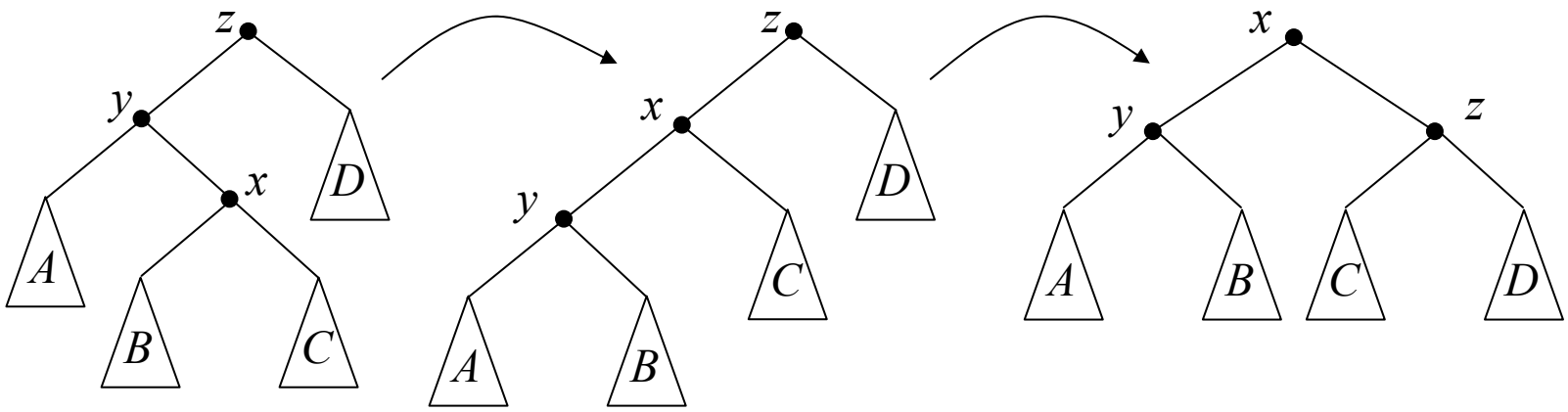


y asumiendo que $a \leq b$ (el otro caso es igual),
$\lfloor \log (a + b + 1) \rfloor \geq \lfloor \log 2a \rfloor = 1 + \lfloor \log a \rfloor > \lfloor \log a \rfloor$

Por tanto se llega a contradicción.

Análisis amortizado de ED: árboles *splay*

- **Caso 3:** x e y son uno hijo derecho y otro hijo izquierdo
  - Se ejecuta dos veces 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

Análisis amortizado de ED: árboles *splay*

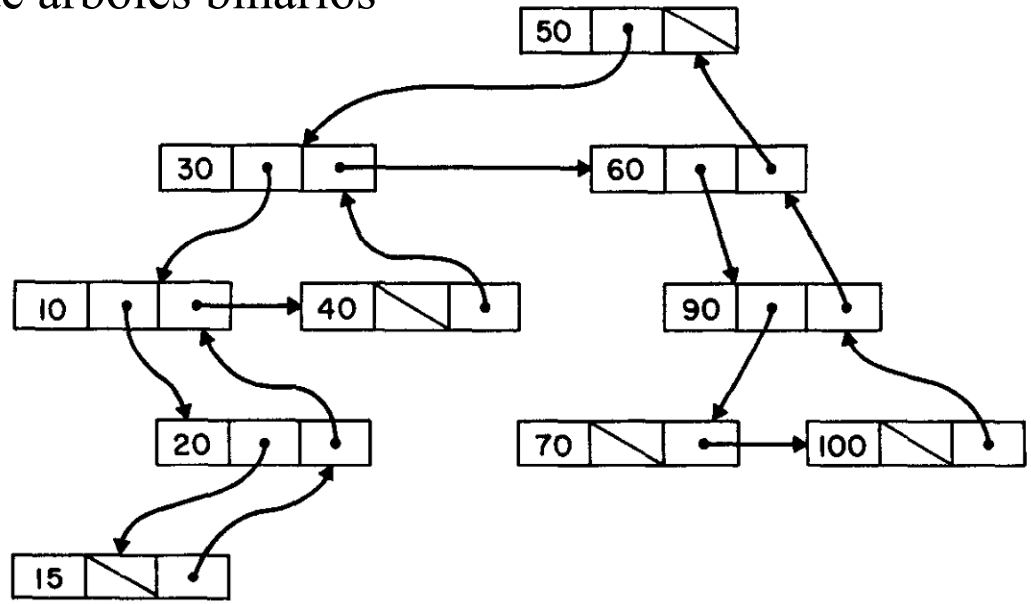
- Recapitulando:
  - El precio (*) de cada operación de *splay* está acotado por $3\lfloor \log n \rfloor + 1$, con n el número de datos del árbol
  - El resto de operaciones se pueden implementar con un número constante de *splays* más un número constante de operaciones sencillas (comparaciones y asignaciones 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

Análisis amortizado de ED: árboles *splay*

- 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



Análisis amortizado de ED: árboles *splay*

- 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: *Data Structures and Algorithm Analysis in Java*, Addison-Wesley, 1999.
    - Puede encontrarse alguna explicación sobre el código en ese libro o en el artículo de Sleator y Tarjan ("Self-adjusting binary search trees")

Análisis amortizado de ED: árboles *splay*

```
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;
```

Análisis amortizado de ED: árboles *splay*

```
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;
```

Análisis amortizado de ED: árboles *splay*

```
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
    ...
```

¡Ojo! Este código no es "de fiar", no está probado. Proviene de simplificar un poco un código de Weiss. Si alguna vez tienes que implementar un splay tree, busca una implementación "top-down" (es más eficiente).

Análisis amortizado de ED: árboles *splay*

```
  ...
  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;
```

Análisis amortizado de ED: árboles *splay*

```
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;
```

Análisis amortizado de ED: árboles *splay*

```
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;
```

Análisis amortizado de ED: árboles *splay*

```
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;
```

Estructuras de datos (ED) avanzadas

- Análisis amortizado de ED
  - conceptos básicos
  - conjuntos disjuntos
  - listas auto-organizativas
  - árboles *splay*

- **ED aleatorizadas**
  - **listas *skip***
  - árboles aleatorizados (*treaps*)

- ED en biología computacional
  - introducción
  - árboles de prefijos
  - árboles de sufijos
  - aplicaciones

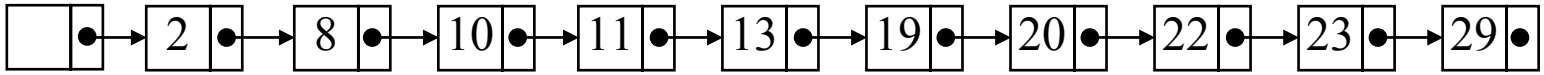
ED aleatorizadas: listas *skip* (*)

- 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.

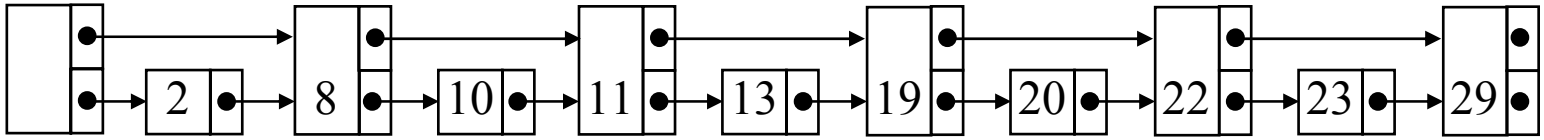
ED aleatorizadas: listas *skip*

- Lista enlazada (ordenada):



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

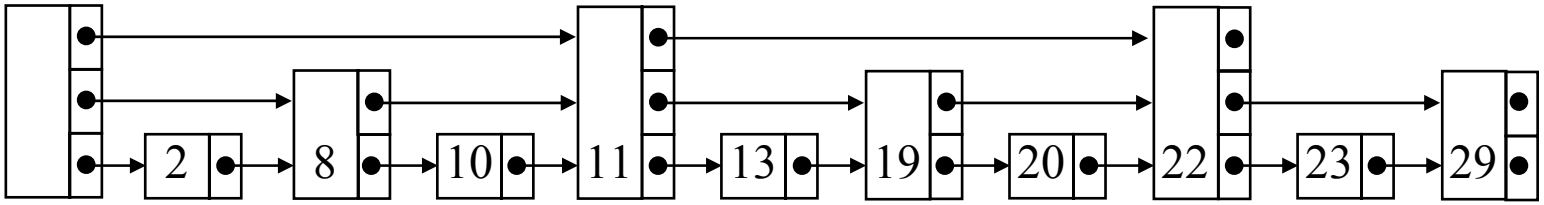
- Añadiendo un puntero a cada nodo par…



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

ED aleatorizadas: listas *skip*

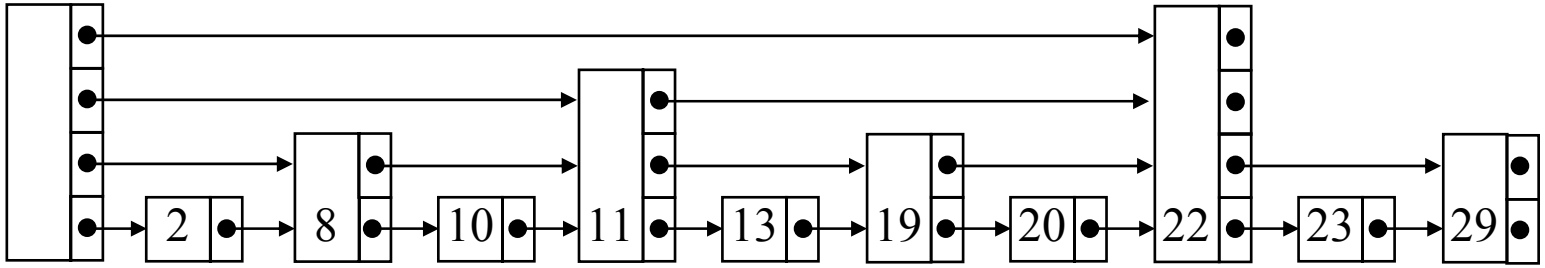
- Y añadiendo otro a cada nodo múltiplo de 4…



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

ED aleatorizadas: listas *skip*

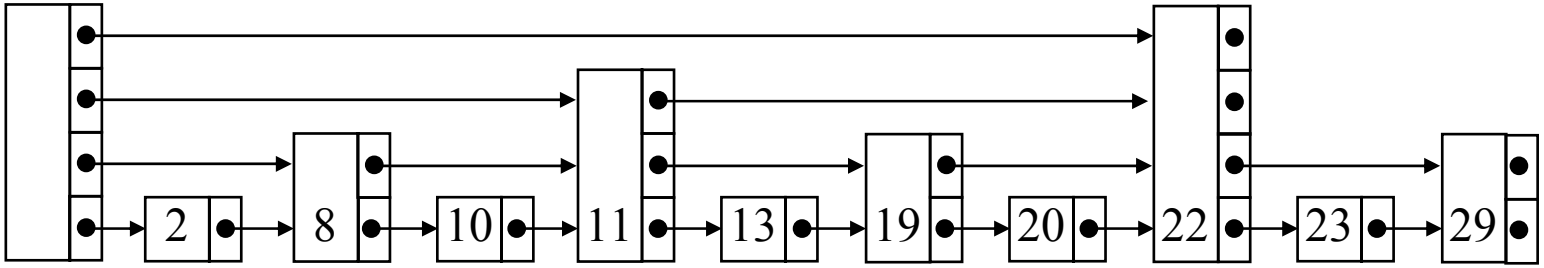
- Y por fin el **caso límite**: cada nodo múltiplo de 2^i apunta al nodo que está 2^i lugares por delante (para todo $i \geq 0$):



  - 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.

ED aleatorizadas: listas *skip*

- En particular…



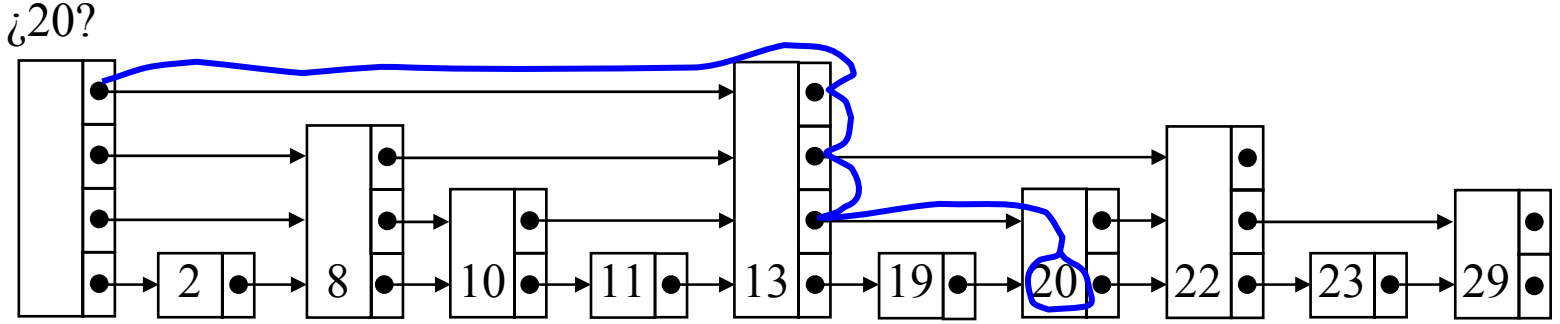
  - Si llamamos nodo de nivel k al que tiene k punteros, se cumple la siguiente propiedad (más débil que la definición del "caso límite"):

> 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).

ED aleatorizadas: listas *skip*

- Un ejemplo de lista *skip*:



- ¿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.

ED aleatorizadas: 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.

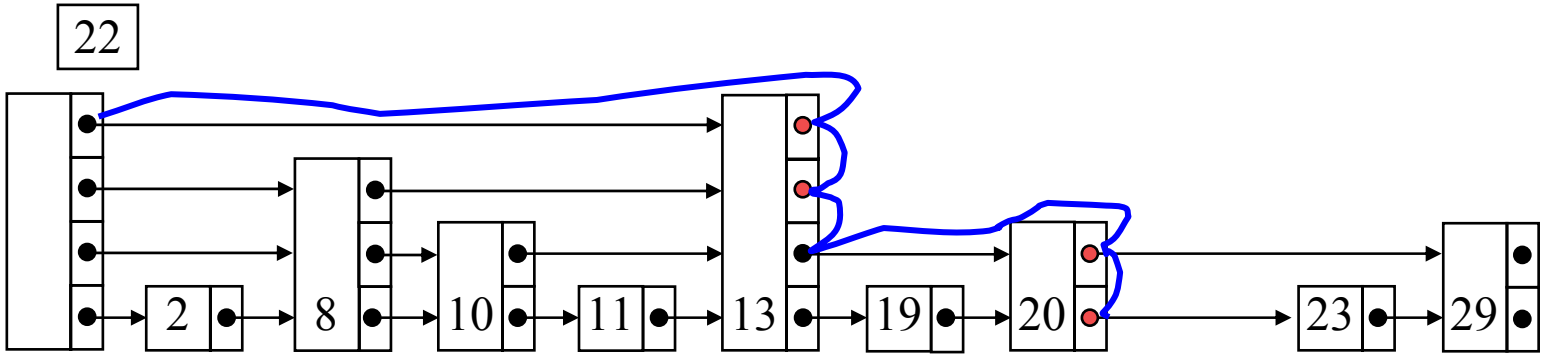
    - Se elige el nivel de un nuevo nodo de acuerdo con esas probabilidades: se tira una moneda al aire hasta que salga cara, y se elige el número total de lanzamientos realizados como nivel del nodo.
      - Distribución geométrica de parámetro $p = 1/2$.

        (En realidad se puede plantear con un parámetro arbitrario, p, y luego seleccionar qué valor de p es más adecuado)

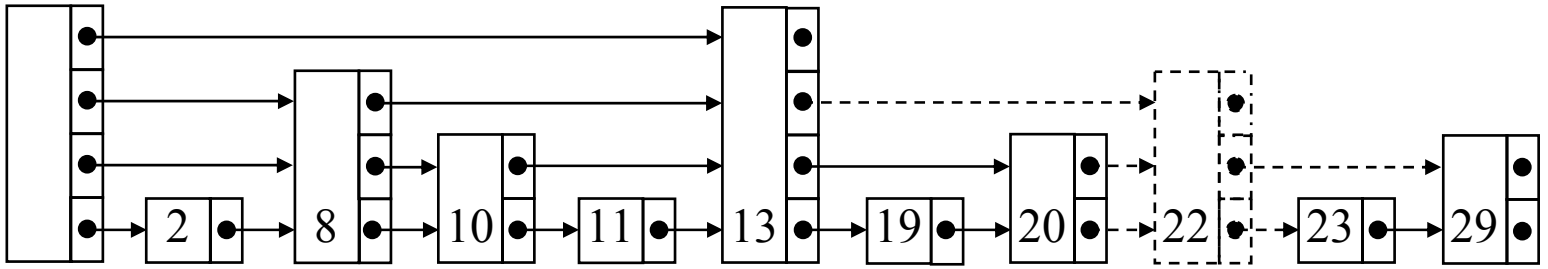
ED aleatorizadas: listas *skip*

– 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.



ED aleatorizadas: 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.

ED aleatorizadas: listas *skip*

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

ED aleatorizadas: 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.

ED aleatorizadas: listas *skip*

```
algoritmo inserta(lista:lista_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}
  . . .
```

ED aleatorizadas: 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
```

ED aleatorizadas: 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}
  . . .
```

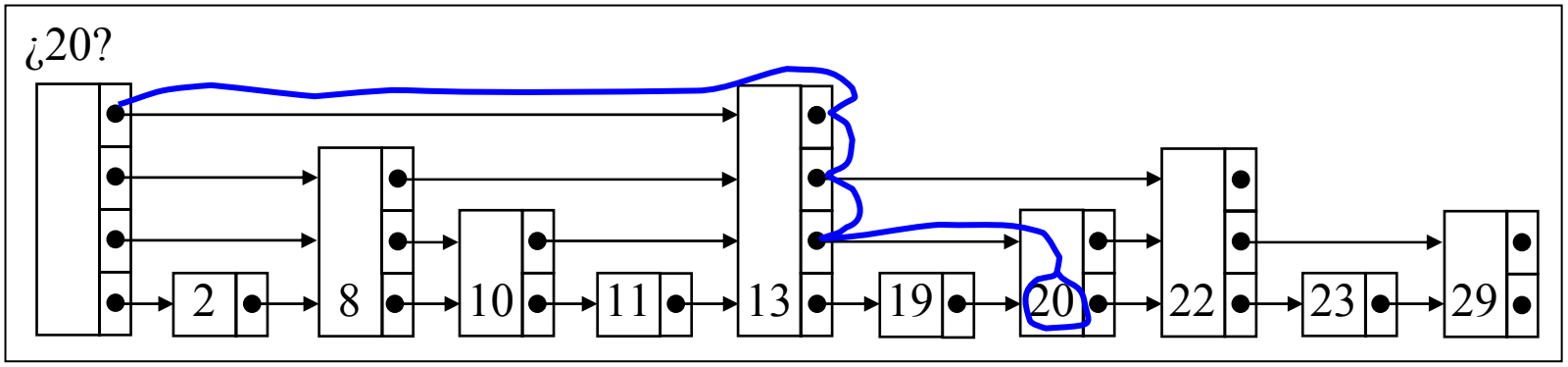
ED aleatorizadas: listas *skip*

```
  . . .
  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
```

ED aleatorizadas: 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.

ED aleatorizadas: listas *skip*



ED aleatorizadas: 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.

ED aleatorizadas: 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 $(L(n)\text{-}1)$-ésimo éxito de la serie de experimentos, es decir, es una binomial negativa $BN(L(n)\text{-}1,p)$.
  - El número de movimientos hacia arriba es exactamente $L(n)\text{-}1$, por tanto:

    coste de escalar al nivel $L(n)$
    en una lista de longitud infinita $=_{prob}$ $(L(n)\text{-}1) + BN(L(n)\text{-}1,p)$

    Nota: $X =_{prob} Y$ si $\Pr\{X>t\} = \Pr\{Y>t\}$, para todo t

    además, $X \leq_{prob} Y$ 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 $n \leq_{prob}$ $(L(n)\text{-}1) + BN(L(n)\text{-}1,p)$

ED aleatorizadas: 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.

ED aleatorizadas: listas *skip*

– Juntando resultados:

número de comparaciones en la búsqueda =

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

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

Su valor medio es

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

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…

ED aleatorizadas: 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 (*splay*)
  - 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.

ED aleatorizadas: 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.

ED aleatorizadas: 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”).

Estructuras de datos (ED) avanzadas

- Análisis amortizado de ED
  - conceptos básicos
  - conjuntos disjuntos
  - listas auto-organizativas
  - árboles *splay*

- ED aleatorizadas
  - listas *skip*
  - **árboles aleatorizados (*treaps*)**

- ED en biología computacional
  - introducción
  - árboles de prefijos
  - árboles de sufijos
  - aplicaciones

ED aleatorizadas: árboles aleatorizados (*treaps*)

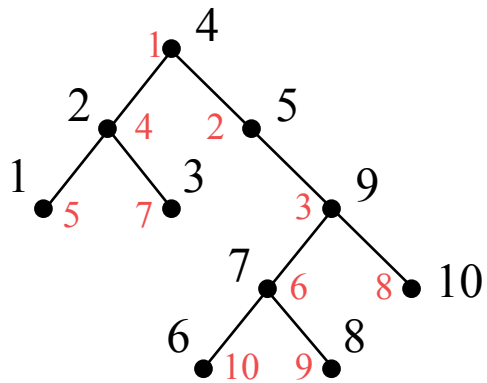
- *Treap* aleatorio (*treap* = *tree* + *heap*)
  - Otra estructura de datos probabilista para almacenar diccionarios con **coste promedio** logarítmico de las operaciones fundamentales
  - Definición. *Treap* es un árbol binario con dos claves:
    - con respecto a una de las claves (que es única) es 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

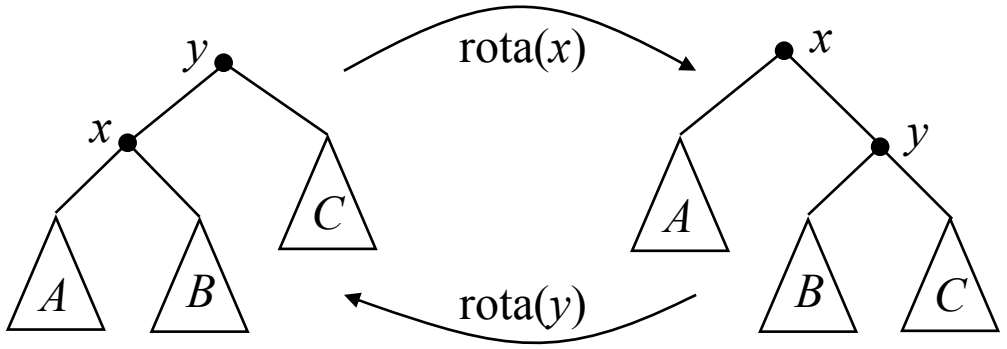
    ¡El *treap* es único!



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

ED aleatorizadas: árboles aleatorizados (*treaps*)

- 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*

ED aleatorizadas: árboles aleatorizados (*treaps*)

- 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)

ED aleatorizadas: árboles aleatorizados (*treaps*)

- 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 $m_{\leq} = \{1,2,\ldots,m\}$, $m_{\geq} = \{m,m+1,\ldots,n\}$, 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"

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

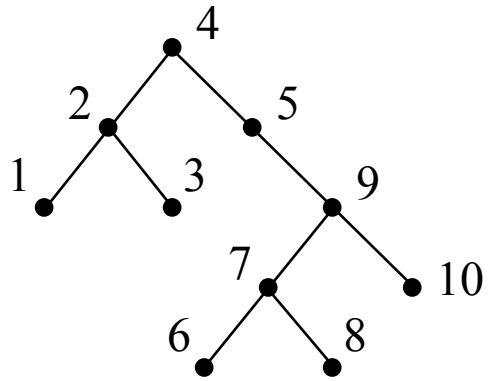
ED aleatorizadas: árboles aleatorizados (*treaps*)

- 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

ED aleatorizadas: árboles aleatorizados (*treaps*)

- Problema: calcular la media de la variable aleatoria…
  - H_m = "número de éxitos obtenidos al recorrer una permutación aleatoria de $\{1,2,\ldots,m\}$ de izquierda a derecha buscando elementos mayores que todos los de su izquierda"
  - Sea σ una permutación aleatoria de $\{1,2,\ldots,m\}$.
  - Sea σ' la permutación de $\{2,3,\ldots,m\}$ 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:

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

ED aleatorizadas: árboles aleatorizados (*treaps*)

- 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

ED aleatorizadas: árboles aleatorizados (*treaps*)

- 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 $\{1,2,\ldots,m\}$ elementos k tales que:
    - k aparece a la derecha de m
    - k es mayor que todos los elementos de $\{1,2,\ldots,m\text{-}1\}$ 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 $\{1,2,\ldots,m\text{-}1\}$ 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

ED aleatorizadas: árboles aleatorizados (*treaps*)

- 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 $E[G_{m\text{-}1}]$.
  - La probabilidad de que haya éxito con el 1 es $1/(m(m\text{-}1))$, 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 $E[G_m] + E[G_{n\text{-}m+1}]$, se tiene que como máximo es 2.

Estructuras de datos (ED) avanzadas

- ED aleatorizadas
  - listas *skip*
  - árboles aleatorizados (*treaps*)

- Análisis amortizado de ED
  - conceptos básicos
  - conjuntos disjuntos
  - listas auto-organizativas
  - árboles *splay*

- **ED en biología computacional**
  - **introducción**
  - árboles de prefijos
  - árboles de sufijos
  - aplicaciones

ED en biología computacional: introducción

- Ejemplo de problema:

  El problema de la subcadena o reconocimiento exacto de un patrón:

  – Dados una cadena madre de n caracteres

  $$S = `s_1\ s_2\ \ldots\ s_n'$$

  y un patrón de m caracteres ($n \geq m$)

  $$P = `p_1\ p_2\ \ldots\ p_m'$$

  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

ED en biología computacional: introducción

- 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…

ED en biología computacional: introducción

- 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.

ED en biología computacional: introducción

- The anatomy of a genome
  - 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.

ED en biología computacional: introducción

- The anatomy of a genome
  - 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.

ED en biología computacional: introducción

- The anatomy of a genome
  - 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

ED en biología computacional: introducción

"Usamos GCG
(una interfaz muy popular para buscar ADN y proteinas 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 algorimo 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)…"

(Extraido del libro de D. Gusfield: *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*, Cambridge University Press, 1997.)

ED en biología computacional: introducción

- Algoritmos (más) clásicos:
  - 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,...,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≤n-m hacer
    ok:=verdad; j:=1;
    mq ok and j≤m hacer
      si P[j]≠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
```

ED en biología computacional: introducción

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 = 'aaaaaaab' P = 'aaab'
  - El número total de comparaciones es

    $\Omega(m(n\text{-}m))$

    es decir, $\Omega(mn)$ si n es sustancialmente mayor que m.

ED en biología computacional: introducción

- Algoritmo Knuth-Morris-Pratt (KMP, 1977)
  - Coste $O(n)$ con un preprocesamiento del patrón con coste $O(m)$
- Algoritmo Boyer-Moore (BM, 1977)
  - Coste $O(n)$, pero es sublineal (no necesariamente examina todos los caracteres de S)
  - Tiende a ser más eficiente cuando m crece
  - En el mejor caso, encuentra todas las apariciones de P en S en un tiempo $O(m+n/m)$
- Y muchos más (variantes de éstos o parecidos):
  - *Applets*: http://www-igm.univ-mlv.fr/~lecroq/string/
  - Libro (de los mismos autores): http://www-igm.univ-mlv.fr/~lecroq/string/string.pdf

ED en biología computacional: introducción

- Algoritmos igual de “antiguos” pero cuyo uso no se ha ido extendiendo hasta hace menos tiempo

  Basados en la utilización de estructuras de datos avanzadas, especialmente adecuadas para resolver problemas de búsqueda en cadenas largas

  – Árboles de prefijos
  – Árboles de sufijos

Estructuras de datos (ED) avanzadas

- ED aleatorizadas
  - listas *skip*
  - árboles aleatorizados (*treaps*)

- Análisis amortizado de ED
  - conceptos básicos
  - conjuntos disjuntos
  - listas auto-organizativas
  - árboles *splay*

- ED en biología computacional
  - introducción
  - **árboles de prefijos**
  - árboles de sufijos
  - aplicaciones

ED en biología computacional: árboles de prefijos

- *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…

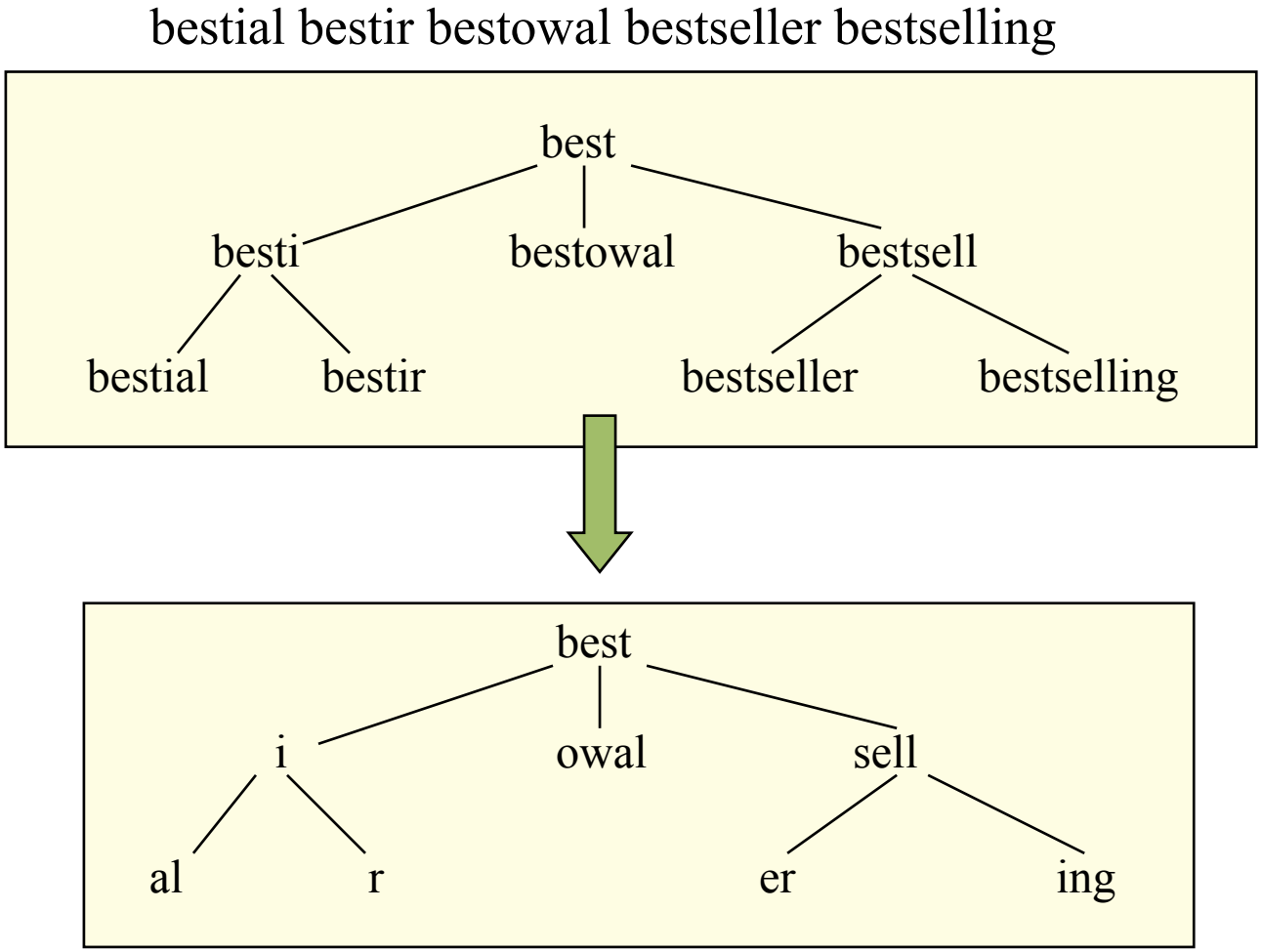
```
best
---- i
---- - al
---- - r
---- owal
---- sell
---- ---- er
---- ---- ing
```

ED en biología computacional: árboles de prefijos

- Trie: definición formal
  - Sea $\Sigma=\{\sigma_1, \ldots, \sigma_m\}$ 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:
    - $\text{trie}(X) = \varnothing$, si $X = \varnothing$
    - $\text{trie}(X) = \langle x \rangle$, si $X = \{x\}$
    - $\text{trie}(X) = \langle \text{trie}(X \setminus \sigma_1), \ldots, \text{trie}(X \setminus \sigma_m) \rangle$, 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.

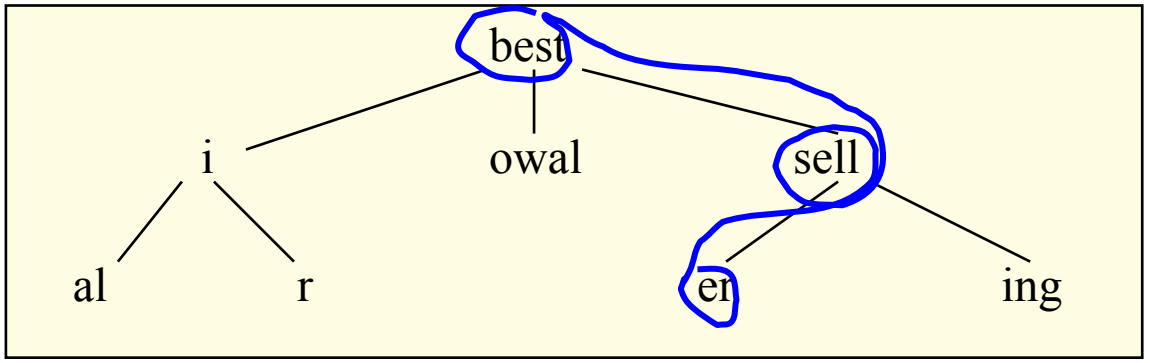
ED en biología computacional: árboles de prefijos

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

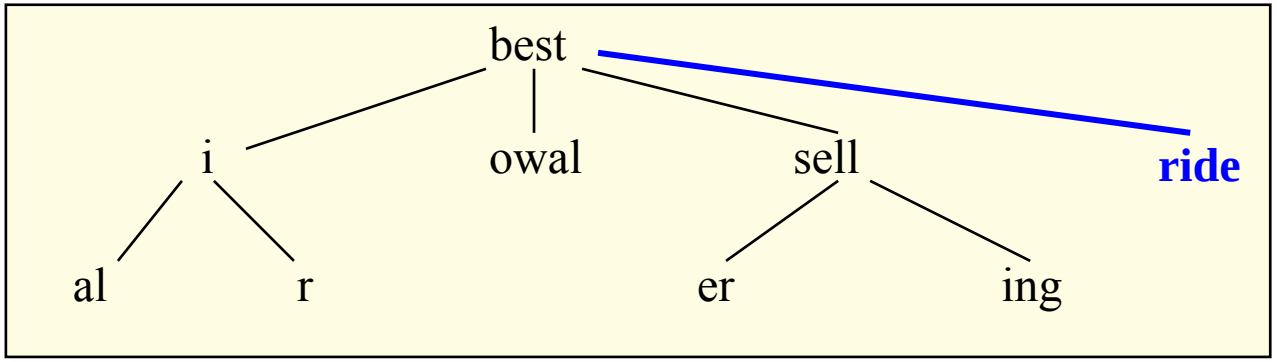


ED en biología computacional: árboles de prefijos

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



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



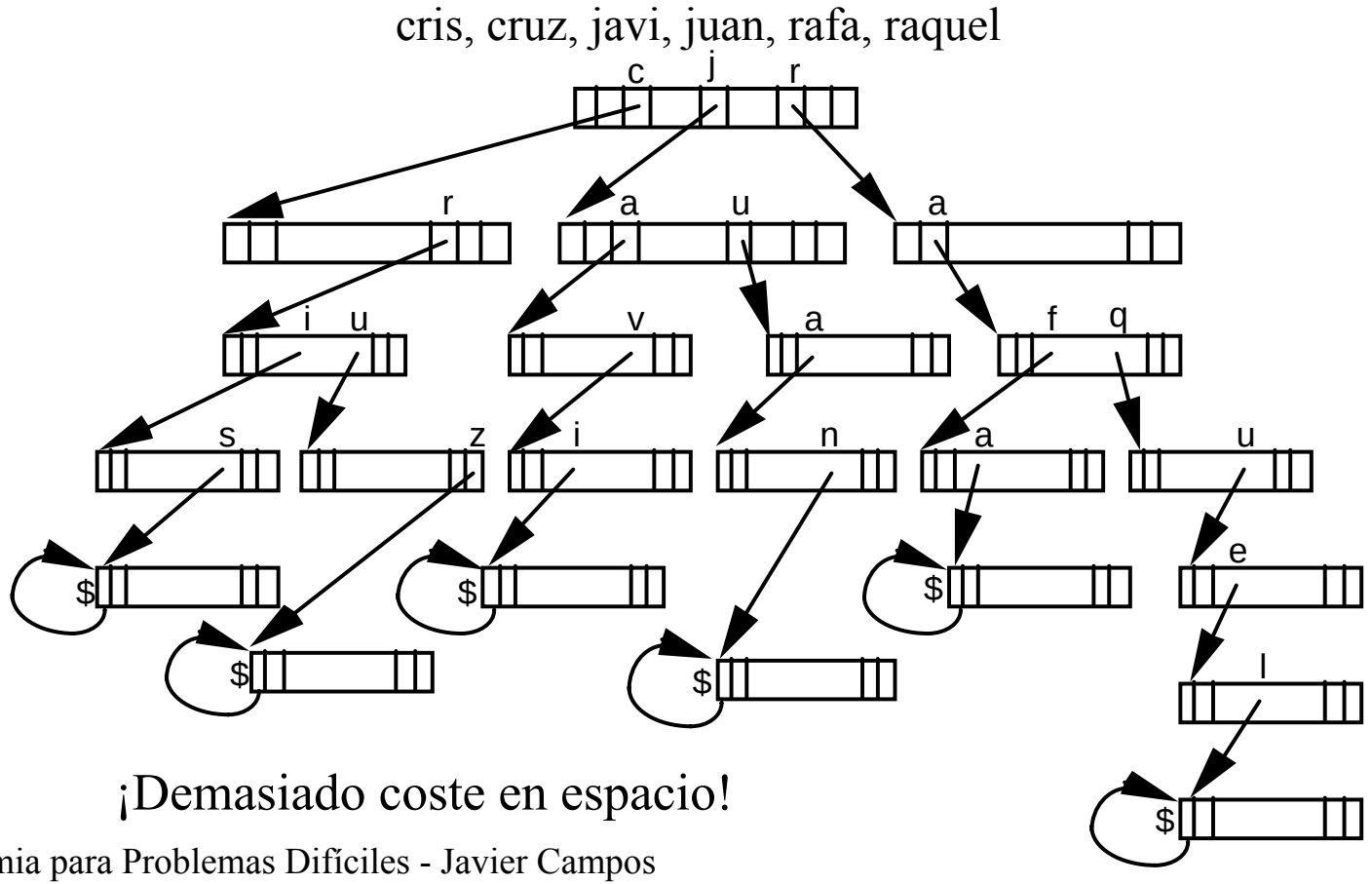
ED en biología computacional: árboles de prefijos

- 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

ED en biología computacional: árboles de prefijos

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

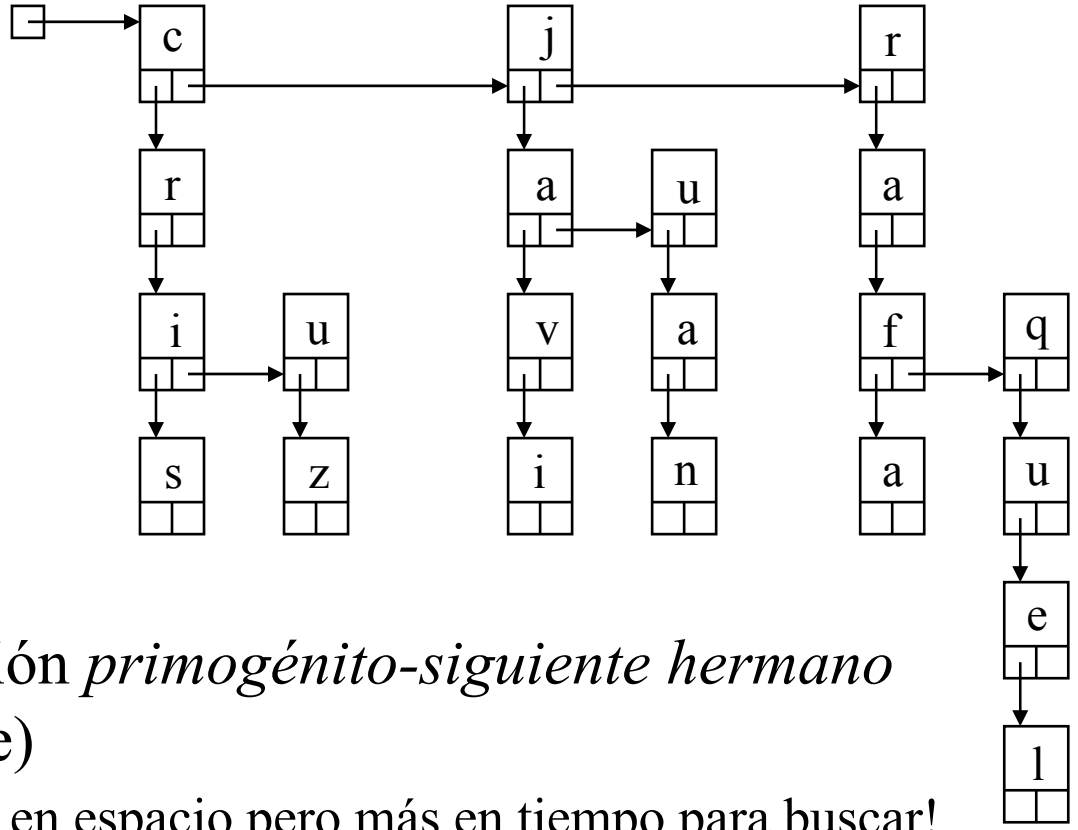


¡Demasiado coste en espacio!

ED en biología computacional: árboles de prefijos

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

cris, cruz, javi, juan, rafa, raquel



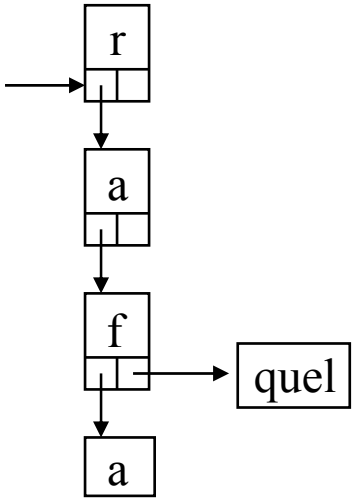
(representación *primogénito-siguiente hermano* de un bosque)

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

ED en biología computacional: árboles de prefijos

– 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…)



ED en biología computacional: árboles de prefijos

– *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.

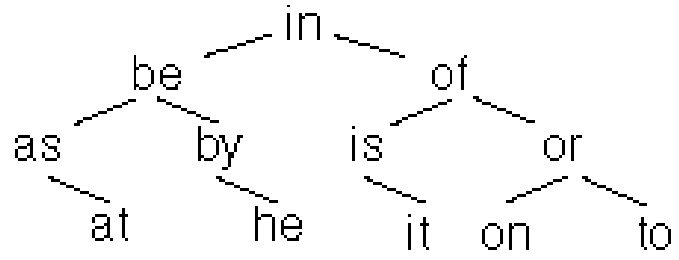
- Una búsqueda compara el carácter actual en la cadena buscada con el carácter del nodo.
- Si el carácter buscado es menor, la búsqueda de ese carácter sigue en el hijo izquierdo.
- Si el carácter buscado es mayor, se sigue en el hijo derecho.
- Si el carácter es igual, se va al hijo central, y se pasa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

ED en biología computacional: árboles de prefijos

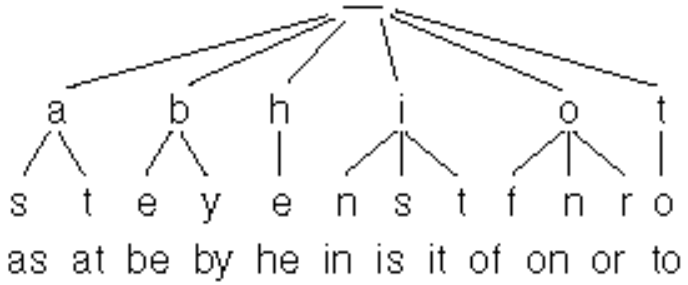
Árbol ternario de búsqueda para las palabras…

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

En un *abb*:

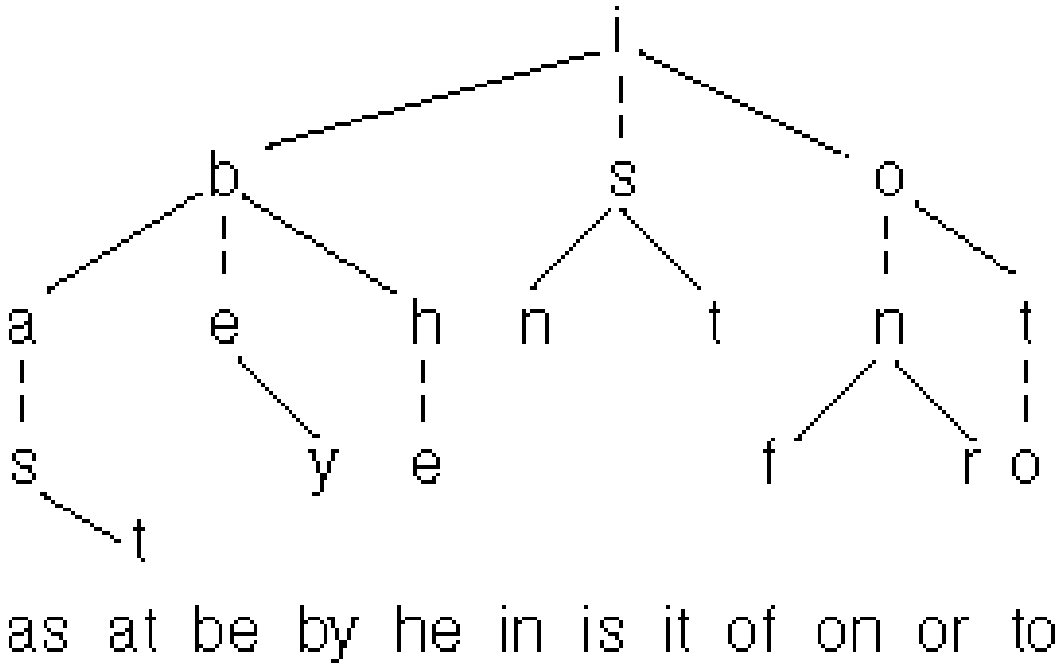


En un trie (representación nodo-vector o nodo-lista):



ED en biología computacional: árboles de prefijos

Árbol ternario de búsqueda:



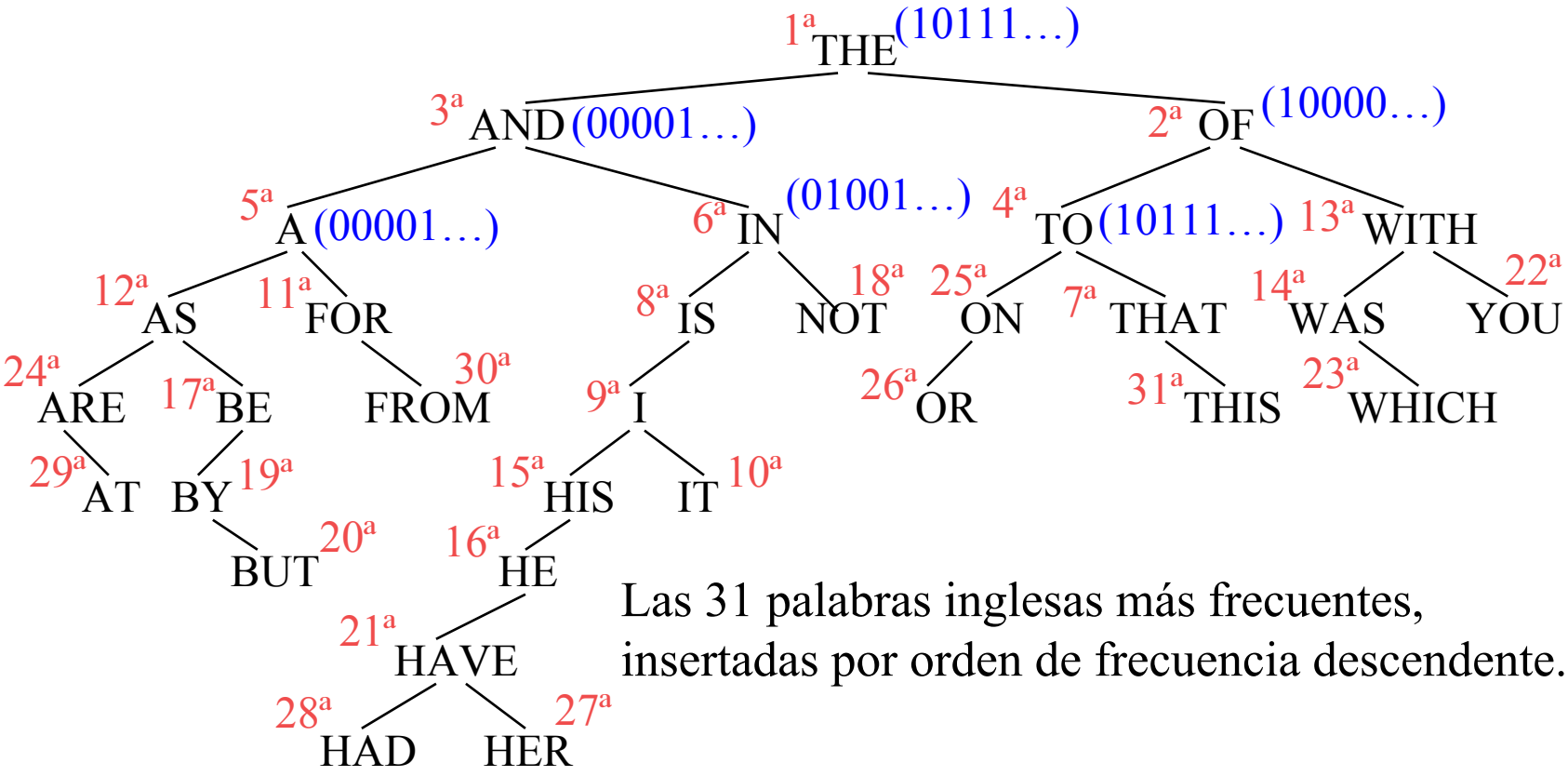
ED en biología computacional: árboles de prefijos

- *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

ED en biología computacional: árboles de prefijos

– Árboles digitales de búsqueda (caso binario):

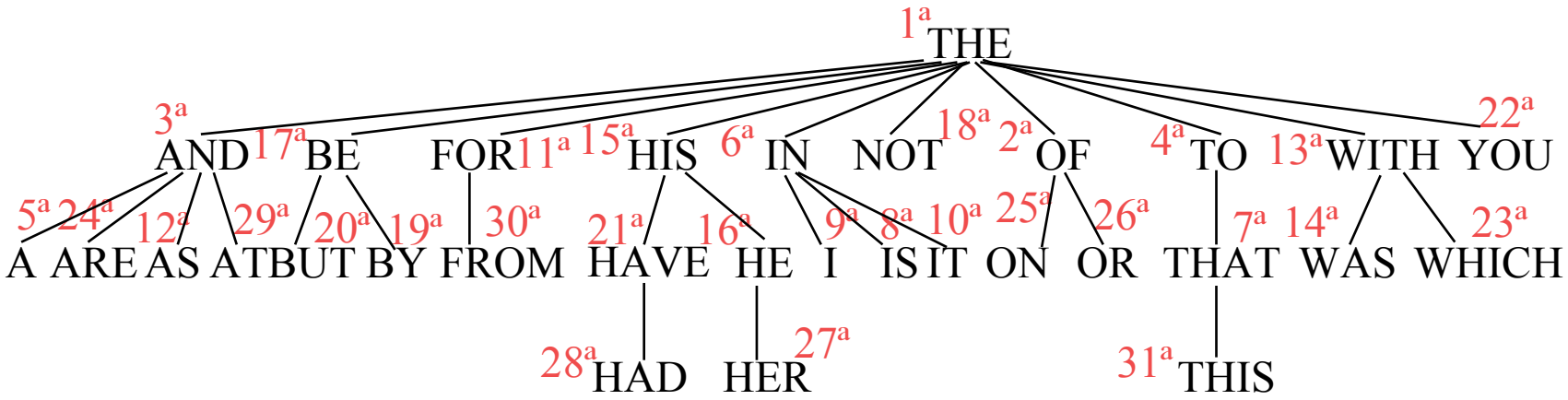


Las 31 palabras inglesas más frecuentes, insertadas por orden de frecuencia descendente.

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

ED en biología computacional: árboles de prefijos

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



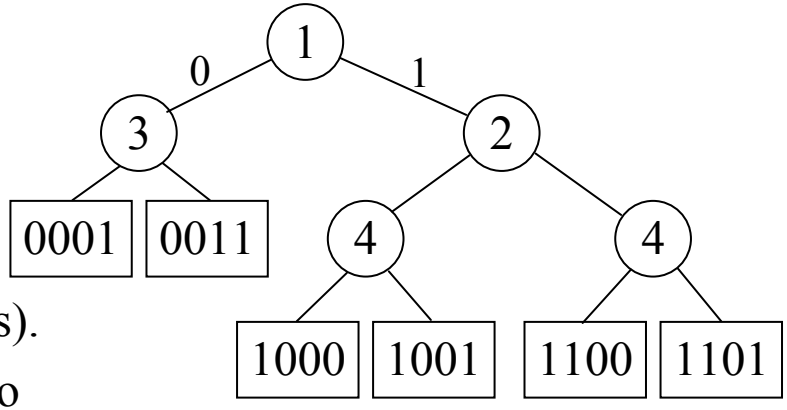
Los mismos datos de antes, insertados en igual orden, pero en un árbol digital de búsqueda de orden 27.

ED en biología computacional: árboles de prefijos

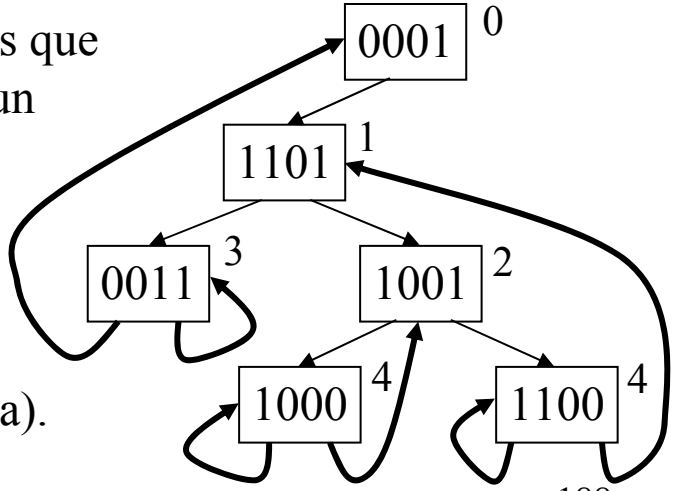
- *Patricia* (*P*ractical *A*lgorithm *T*o *R*etrieve *I*nformation *C*oded *I*n *A*lphanumeric)
  - Problema del trie: si |{claves}| << |{claves potenciales}|, la mayoría de los nodos internos tienen un solo hijo ➔ aumenta el coste en espacio
  - Idea: árbol binario, pero evitando las bifurcaciones 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*)

ED en biología computacional: árboles de prefijos

- 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.

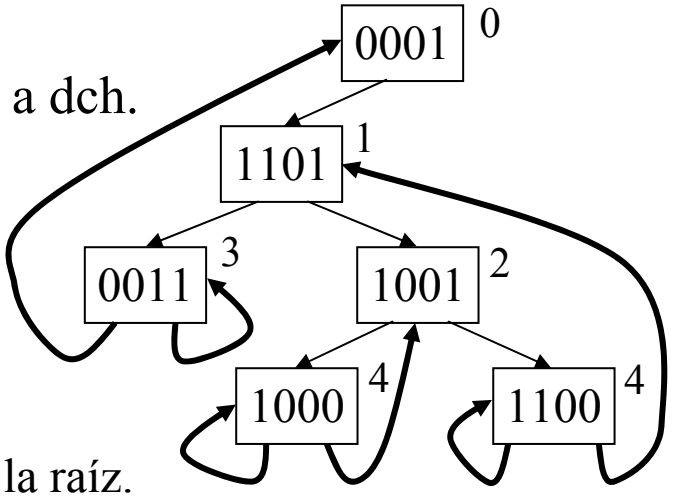


  - En un Patricia, las claves se almacenan en los nodos internos.
    - Como hay un nodo interno menos que nº de elementos, hay que añadir un nuevo nodo (se pone como raíz).
    - Cada nodo sigue guardando el nº de bit usado para bifurcar.
    - Ese nº distingue si el puntero sube o baja (si > el del padre, baja).



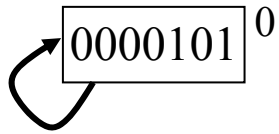
ED en biología computacional: árboles de prefijos

- 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 nº de bit del nodo alcanzado es 2, se sigue hacia abajo según ese bit, como el 2º bit de la clave es 1, vamos al hijo derecho.
    - Ahora se usa el 4º bit de la clave buscada, como es 1 seguimos el puntero hijo dch. y llegamos a la clave 1101.
    - Como el nº de bit es 1 (<4) comparamos la clave actual con la buscada, como coincide, terminamos con éxito.

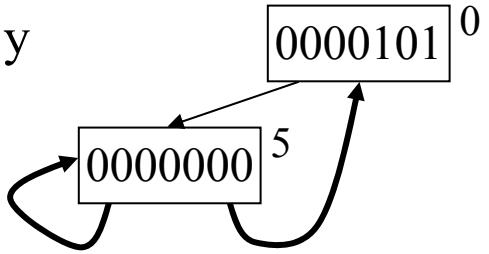


ED en biología computacional: árboles de prefijos

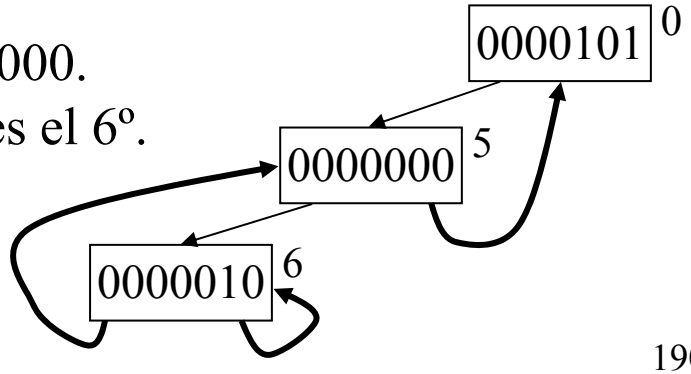
- Inserción de claves:
  - Partimos de árbol vacío (ninguna clave).
  - Insertamos la clave 0000101.



  - Ahora insertamos la clave 0000000. Buscando esa clave llegamos a la raíz y vemos que es distinta. Vemos que el primer bit en que difieren es el 5º. Creamos el hijo izq. etiquetado con el bit 5 y guardamos la clave en él. Como el 5º bit de la clave insertada es 0, el puntero izq. de ese nodo apunta al mismo nodo. El puntero dch. apunta al nodo raíz.

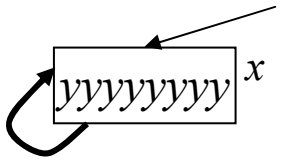


  - Insertamos 0000010. La búsqueda termina en 0000000. El primer bit en que difieren es el 6º.



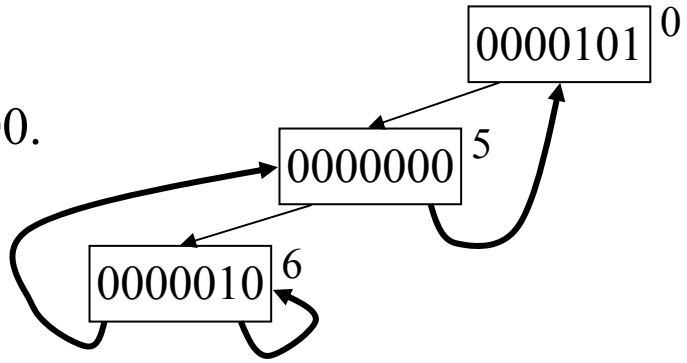
ED en biología computacional: árboles de prefijos

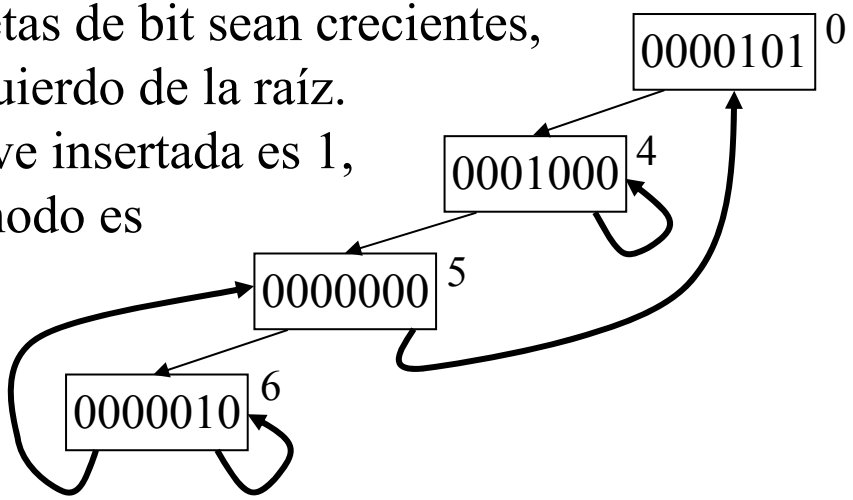
- Estrategia general de inserción (a partir de la 2ª clave):
  - Se busca la clave *C* a insertar, la búsqueda termina en un nodo con clave *C'*.
  - Se calcula el nº *b* de bit más a la izq. en que *C* y *C'* difieren.
  - Se crea un nuevo nodo con la nueva clave, etiquetado con el nº de bit anterior 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*.



ED en biología computacional: árboles de prefijos

- 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.

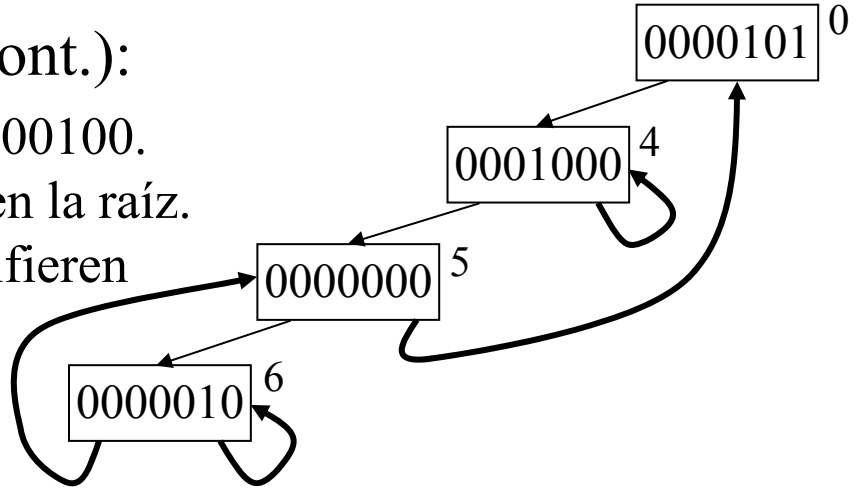




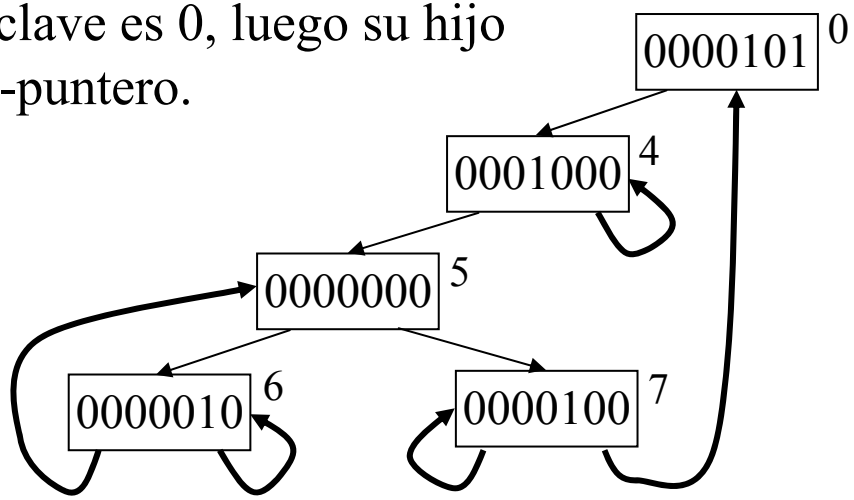
ED en biología computacional: árboles de prefijos

– 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.

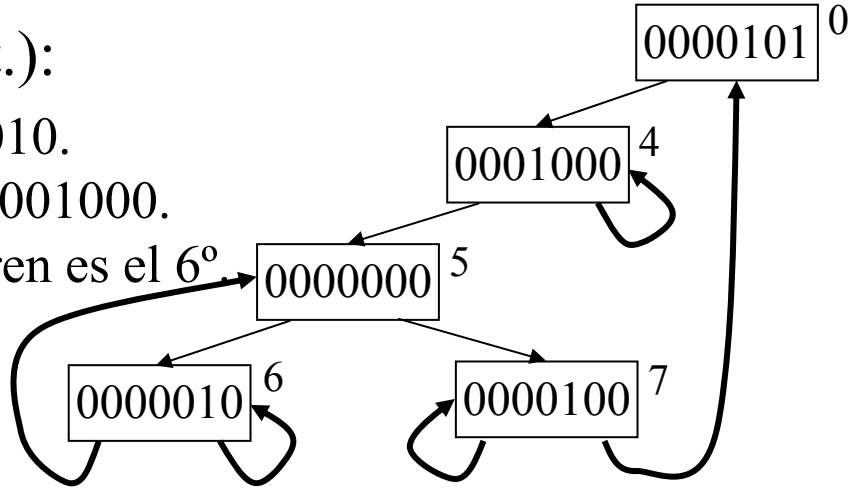


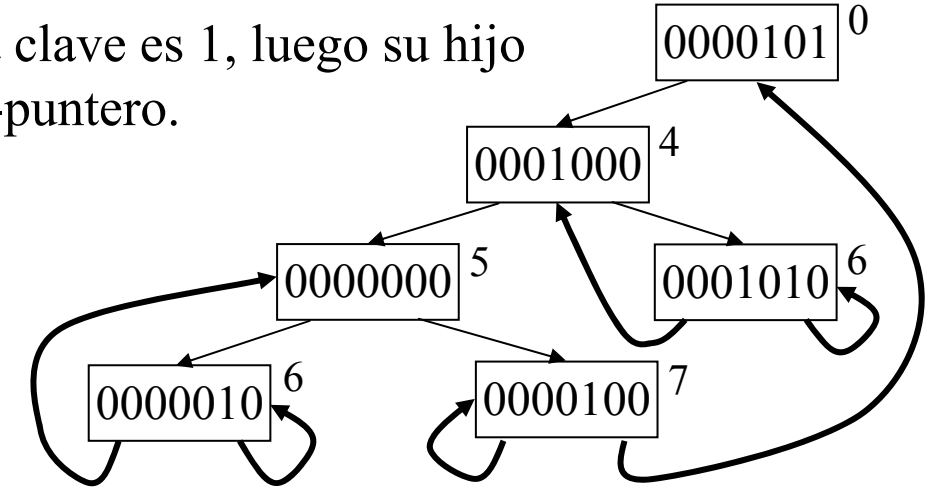
Insertamos el nuevo nodo en el camino de búsqueda de forma que las etiquetas de bit sean crecientes: hijo dch. de 0000000.
El bit 7 de la nueva clave es 0, luego su hijo izquierdo es un auto-puntero.



ED en biología computacional: árboles de prefijos

- 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.





ED en biología computacional: árboles de prefijos

- 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*

ED en biología computacional: árboles de prefijos

- 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?
  - Es un problema **muy difícil** (no vamos a verlo).
  - 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".
    - Mediante esa relación puede demostrarse que…

ED en biología computacional: árboles de prefijos

- Puede demostrarse que 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 $\gamma = 0{,}577215...$ 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 $f(n)$

ED en biología computacional: árboles de prefijos

- 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…"

ED en biología computacional: árboles de prefijos

	Con éxito	Sin éxito
Búsqueda en un trie	log n + 1,33275	log n – 0,10995
Búsqueda en árbol digital	log n – 1,71665	log n – 0,27395
Búsqueda en Patricia	log n + 0,33275	log n – 0,31875

Estructuras de datos (ED) avanzadas

- ED aleatorizadas
  - listas *skip*
  - árboles aleatorizados (*treaps*)

- Análisis amortizado de ED
  - conceptos básicos
  - conjuntos disjuntos
  - listas auto-organizativas
  - árboles *splay*

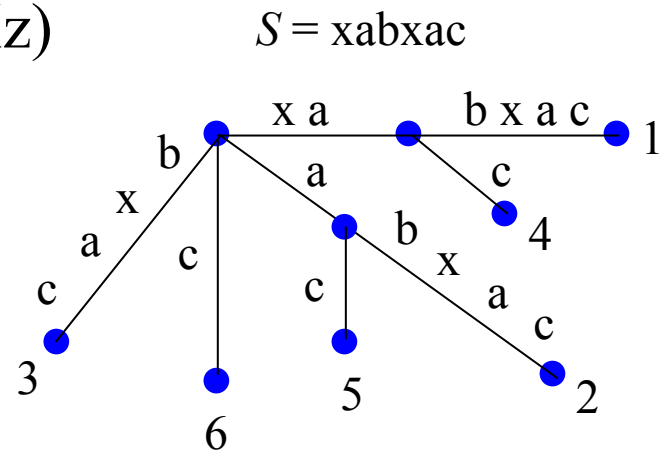
- ED en biología computacional
  - introducción
  - árboles de prefijos
  - **árboles de sufijos**
  - aplicaciones

ED en biología computacional: 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…

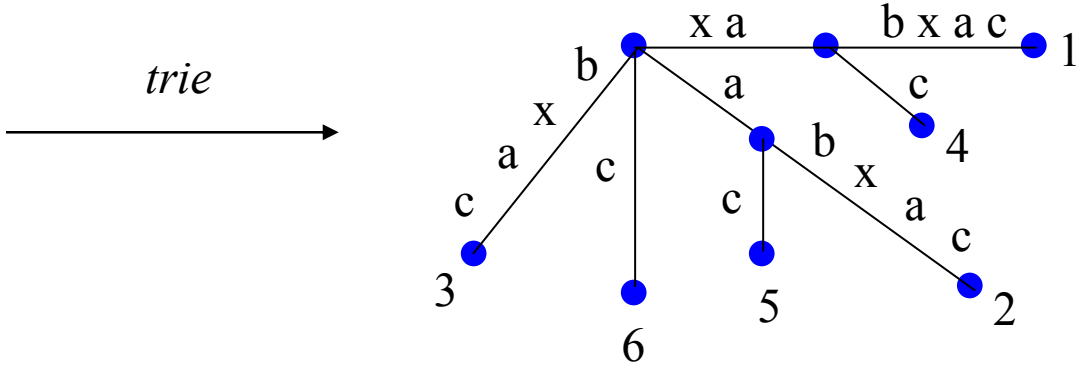
ED en biología computacional: árboles de sufijos

- 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*



ED en biología computacional: árboles de sufijos

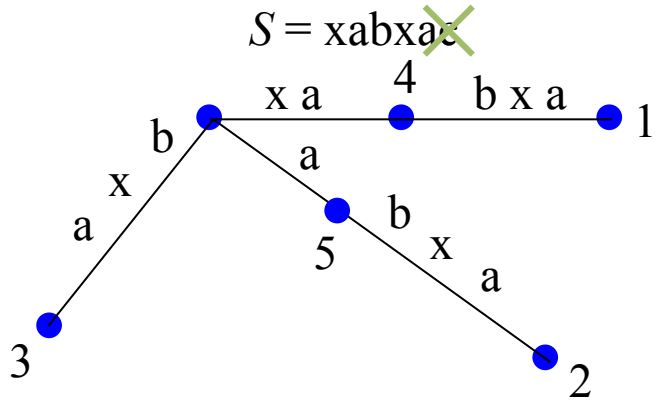
- Es ~~como~~ un *trie* (árbol de prefijos) que almacena todos los sufijos de una cadena
  - Sufijos de la cadena *S* = xabxac :
    - c
    - ac
    - xac
    - bxac
    - abxac
    - xabxac



  - Además: cada nodo interno tiene al menos 2 hijos….
    ➔ es ~~como~~ un Patricia que guarda todos los sufijos de la cadena

ED en biología computacional: 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:



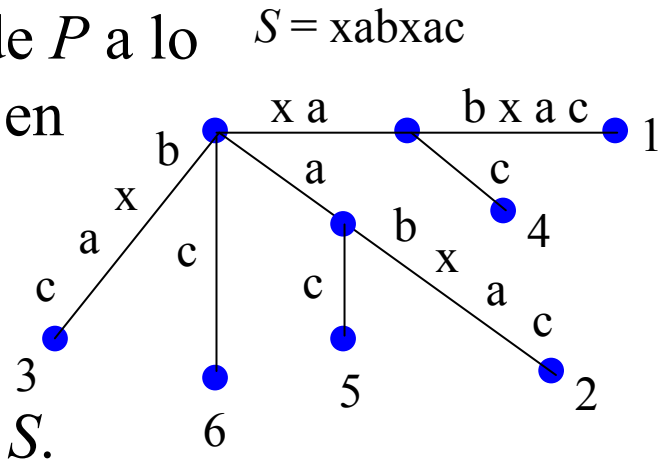
  - Solución fácil: añadir carácter terminador, *S* = xabxa€ (como hicimos con los *tries*)

ED en biología computacional: 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

ED en biología computacional: árboles de sufijos

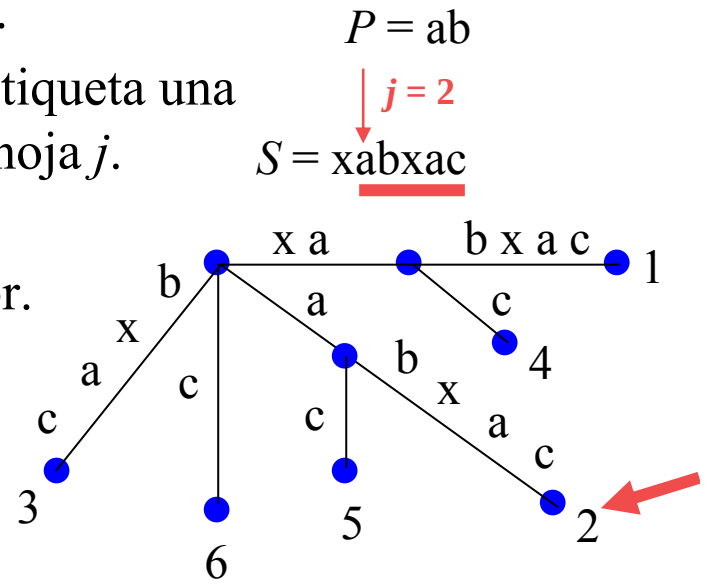
- Solución al problema de la subcadena:
  - Encontrar todas las apariciones de P, de longitud n, en un texto S, de longitud m, en tiempo $O(n + m)$.
  - Construir un árbol de sufijos para S, en tiempo $O(m)$.
  - Hacer coincidir los caracteres de P a lo largo del único camino posible en el árbol hasta 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.

ED en biología computacional: 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*, $O(n)$.

ED en biología computacional: 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.

ED en biología computacional: árboles de sufijos

- Coste (continuación):
  - Es el mismo coste de algoritmos clásicos (como el de Knuth-Morris-Pratt o el de Boyer-Moore), pero:
    - En ese caso se precisa un tiempo $O(n)$ para el pre-procesamiento de P y luego un tiempo $O(m)$ para la búsqueda en la cadena larga.
    - Con árboles de sufijos 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 de Knuth-Morris-Pratt).

ED en biología computacional: 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).

ED en biología computacional: 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.

ED en biología computacional: 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€ borrando cada copia del símbolo terminal € 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 $S[1..i]$€.

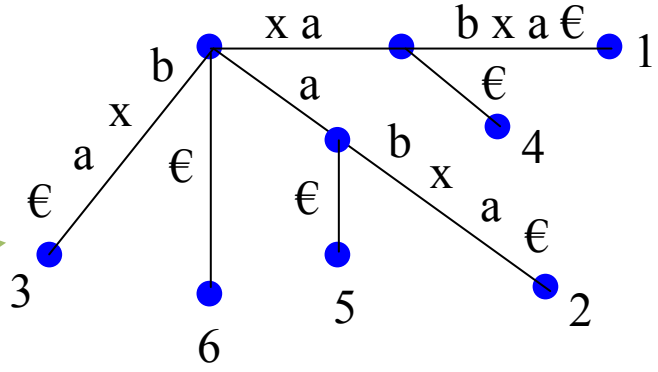
ED en biología computacional: árboles de sufijos

- El árbol de sufijos implícitos de una cadena *S*:
  - Tiene menos hojas que el árbol de sufijos de *S*€ si y sólo si al menos uno de los sufijos de *S* es prefijo de otro sufijo de *S* (precisamente para evitar esa situación se añadió el símbolo €).

S = xabxa€

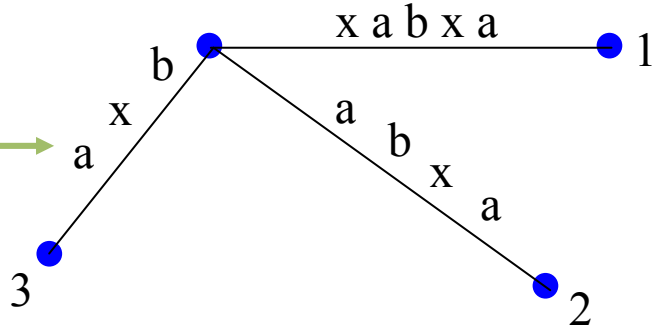
El sufijo xa es prefijo del sufijo xabxa, y también el sufijo a es prefijo de abxa.

Por eso en el árbol de sufijos de S las aristas que llevan a las hojas 4 y 5 están etiquetadas sólo con €.



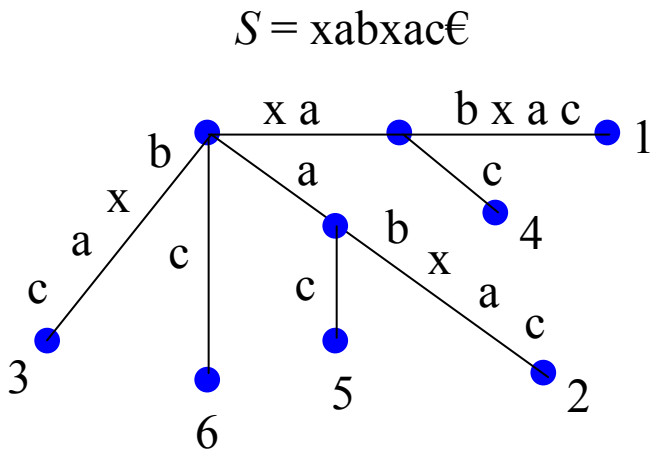
Al borrar esas aristas quedan dos nodos con un solo hijo y se borran también.

El árbol implícito puede no tener una hoja por cada sufijo de *S*, luego tiene menos información que el árbol de sufijos de *S*.



ED en biología computacional: 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.



ED en biología computacional: árboles de sufijos

- El algoritmo en su versión ineficiente, $O(m^3)$

```
algoritmo Ukkonen_alto_nivel(S:cadena; m:nat)
principio
  construir árbol I1;
  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
    fpara
  fpara
fin
```

Es una arista con etiqueta $S(1)$

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

Recordar que I_i es el árbol de sufijos implícitos para el prefijo $S[1..i]$.

ED en biología computacional: 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:

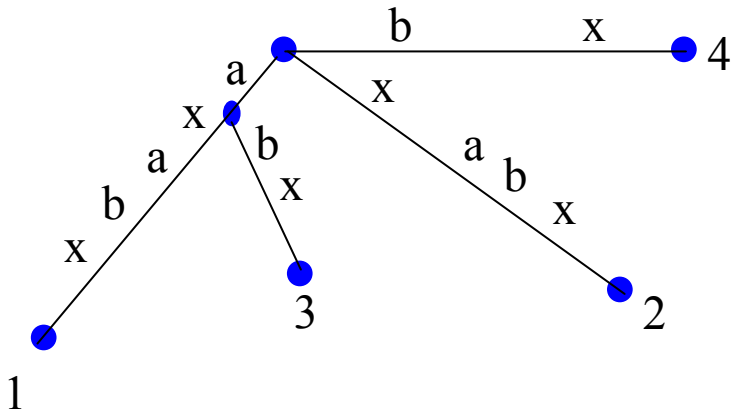
    [Regla 1] Si β termina en una hoja: se añade $S(i+1)$ al final de la etiqueta de la arista de la que cuelga esa hoja

    [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.

ED en biología computacional: árboles de sufijos

- Ejemplo: S = axabx

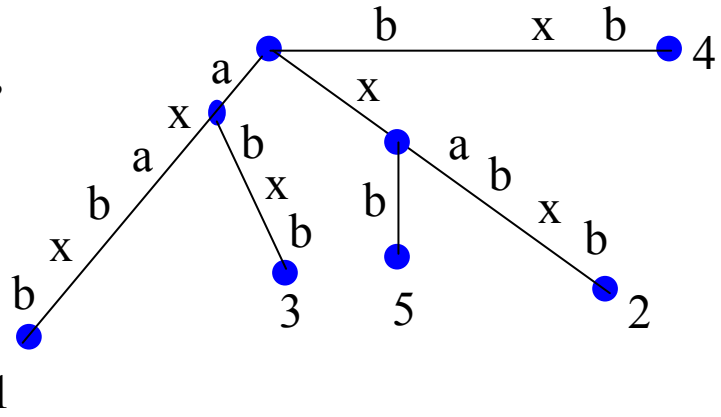


Es un árbol de sufijos implícitos para S.

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.

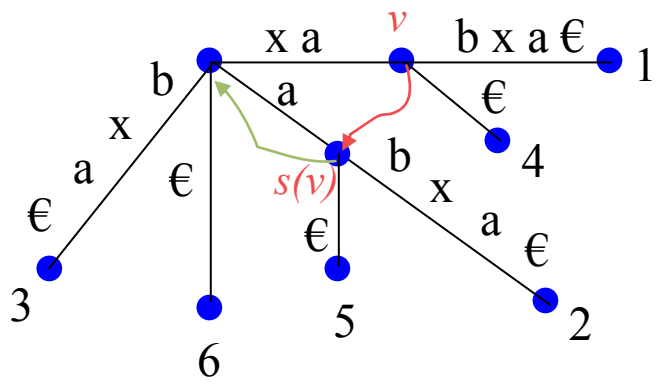


ED en biología computacional: árboles de sufijos

- 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 $O(m^3)$

ED en biología computacional: 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

ED en biología computacional: 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*α, 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*α, entonces hay un nodo *s*(*v*) en I_i con etiqueta α

(demostraciones: libro de D. Gusfield)

ED en biología computacional: 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

ED en biología computacional: 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).

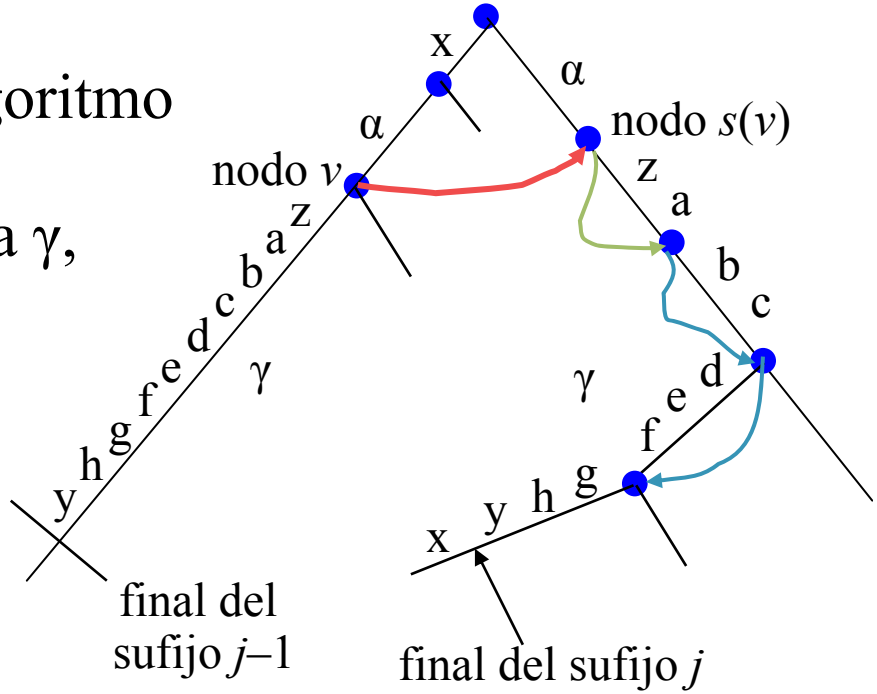
ED en biología computacional: árboles de sufijos

- 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 $O(m^2)$

ED en biología computacional: 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(nº nodos de camino):
    - Sea $g = |\gamma|$
    - El 1er 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



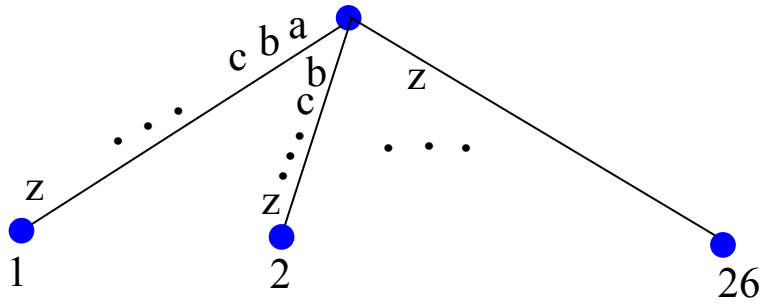
ED en biología computacional: árboles de sufijos

- Terminología: *profundidad* de un nodo es el nº 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)

ED en biología computacional: árboles de sufijos

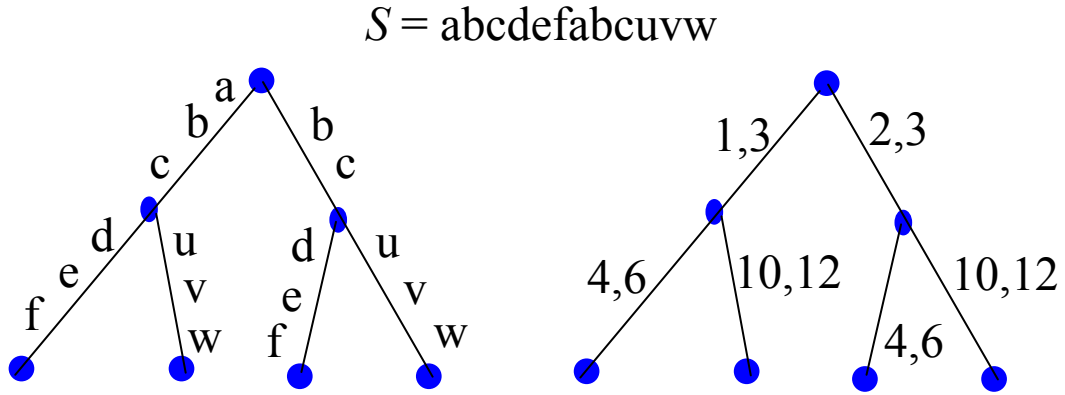
- Problema para bajar de coste $O(m^2)$:
  - El árbol puede requerir $\Theta(m^2)$ en espacio:
    - Las etiquetas de aristas pueden contener $\Theta(m^2)$ caracteres
    - Ejemplo: S = abcdefghijklmnopqrstuvwxyz
      Cada sufijo empieza por distinta letra, luego de la raíz salen 26 aristas y cada una etiquetada con un sufijo completo, por tanto se requieren 26 × 27/2 caracteres en total



  - Se requiere otra forma de guardar las etiquetas…

ED en biología computacional: 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)



  - Número máximo de aristas: $2m - 1$, luego el coste en espacio para almacenar el árbol es $O(m)$

ED en biología computacional: 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.

ED en biología computacional: 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*.

ED en biología computacional: 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.

ED en biología computacional: 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)

ED en biología computacional: 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 $j_i + 1$

ED en biología computacional: 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 € 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

Estructuras de datos (ED) avanzadas

- ED aleatorizadas
  - listas *skip*
  - árboles aleatorizados (*treaps*)

- Análisis amortizado de ED
  - conceptos básicos
  - conjuntos disjuntos
  - listas auto-organizativas
  - árboles *splay*

- ED en biología computacional
  - introducción
  - árboles de prefijos
  - árboles de sufijos
  - **aplicaciones**

ED en biología computacional: aplicaciones

- [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

ED en biología computacional: aplicaciones

- [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

ED en biología computacional: aplicaciones

- [P3] Búsqueda de un patrón en una base de datos (BD) de cadenas
  - 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 Gutemberg 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)

ED en biología computacional: aplicaciones

- 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

ED en biología computacional: aplicaciones

- 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)

ED en biología computacional: aplicaciones

- 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"

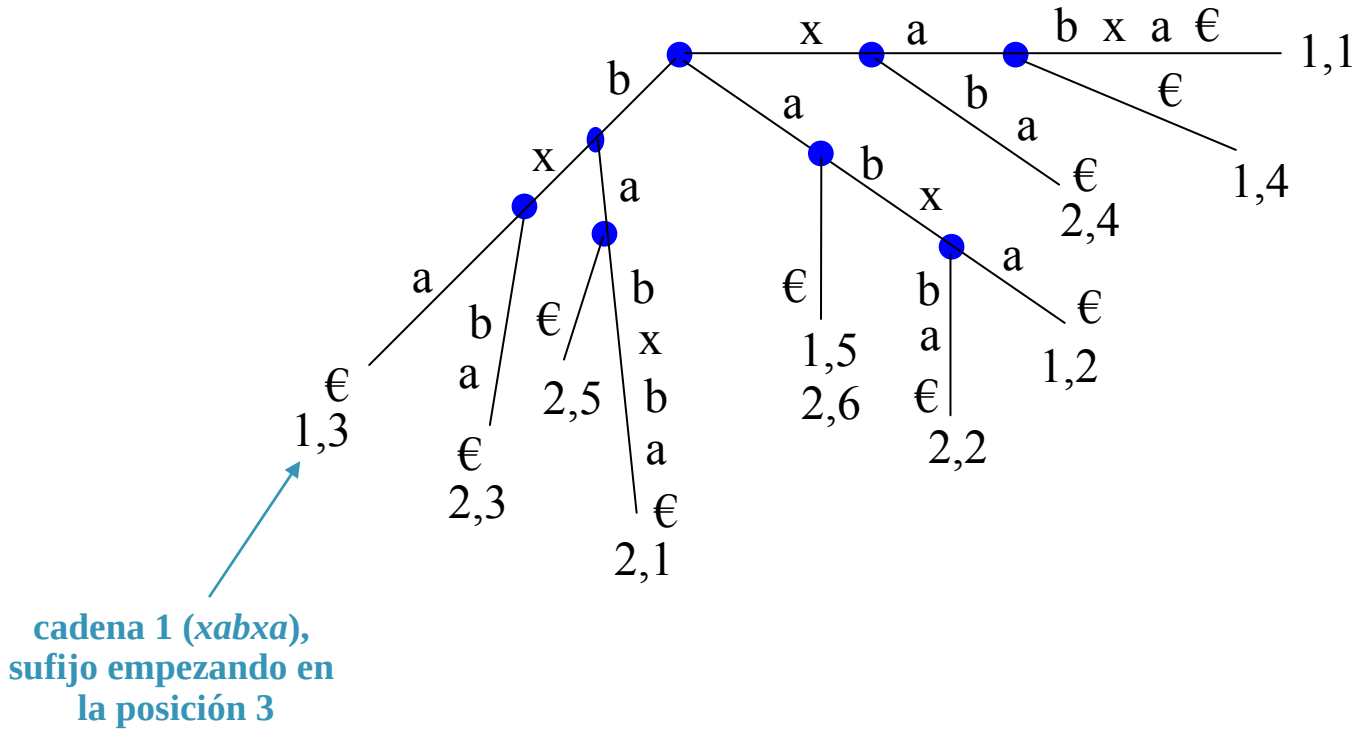
ED en biología computacional: aplicaciones

- Para más cadenas: repitir 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

ED en biología computacional: aplicaciones

– Ejemplo: resultado de añadir *babxba* al árbol de *xabxa*



ED en biología computacional: aplicaciones

- 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

ED en biología computacional: aplicaciones

- [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

ED en biología computacional: aplicaciones

- 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].

ED en biología computacional: aplicaciones

- 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…

ED en biología computacional: aplicaciones

- La investigación continúa…
  - Ejemplo (de artículo algo más 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.

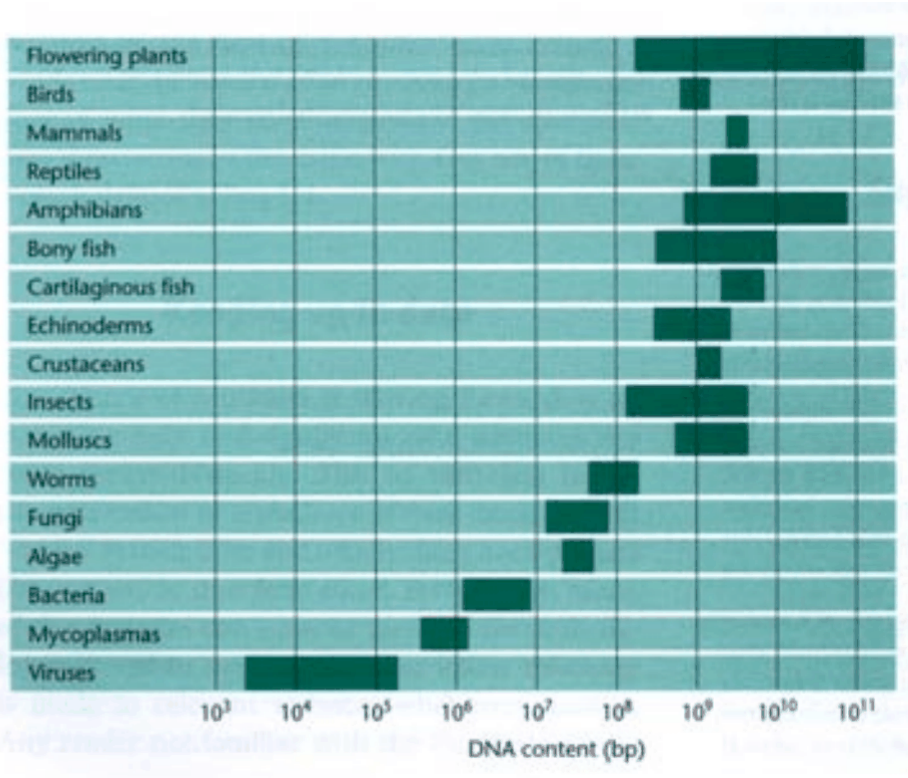
    But, it has been shown as difficult to handle long DNA sequences using suffix trees due to the so-called memory bottleneck problems.

    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 nucleotidos),
3'2 Gbps en el caso del *Homo sapiens*.

ED en biología computacional: aplicaciones

Species	Genome size (kb)
Navicola pelliculosa (diatom)	35,000
Drosophila melanogaster (fruitfly)	180,000
Parameciumaurelia (ciliate)	190,000
Gallus domesticus (chicken)	1,200,000
Cyprinus carpio (carp)	1,700,000
Boa constrictor (snake)	2,100,000
Rattus norvegicus (rat)	3,100,000
Xenopus laevis (toad)	3,100,000
Homo sapiens (bubba)	3,200,000
Nicotiana tabaccum (tobacco)	3,800,000
Paramecium caudatum (ciliate)	8,600,000
Allium cepa (onion)	18,000,000
Lilium formosanum (lily)	36,000,000
Amphiuma means (newt)	68,000,000
Pinus resinosa (pine)	84,000,000
Protopterus aethiopicus (lungfish)	140,000,000
Ophioglossum petiolatum (fern)	160,000,000
Amoeba dubia	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.

ED en biología computacional: aplicaciones

"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.

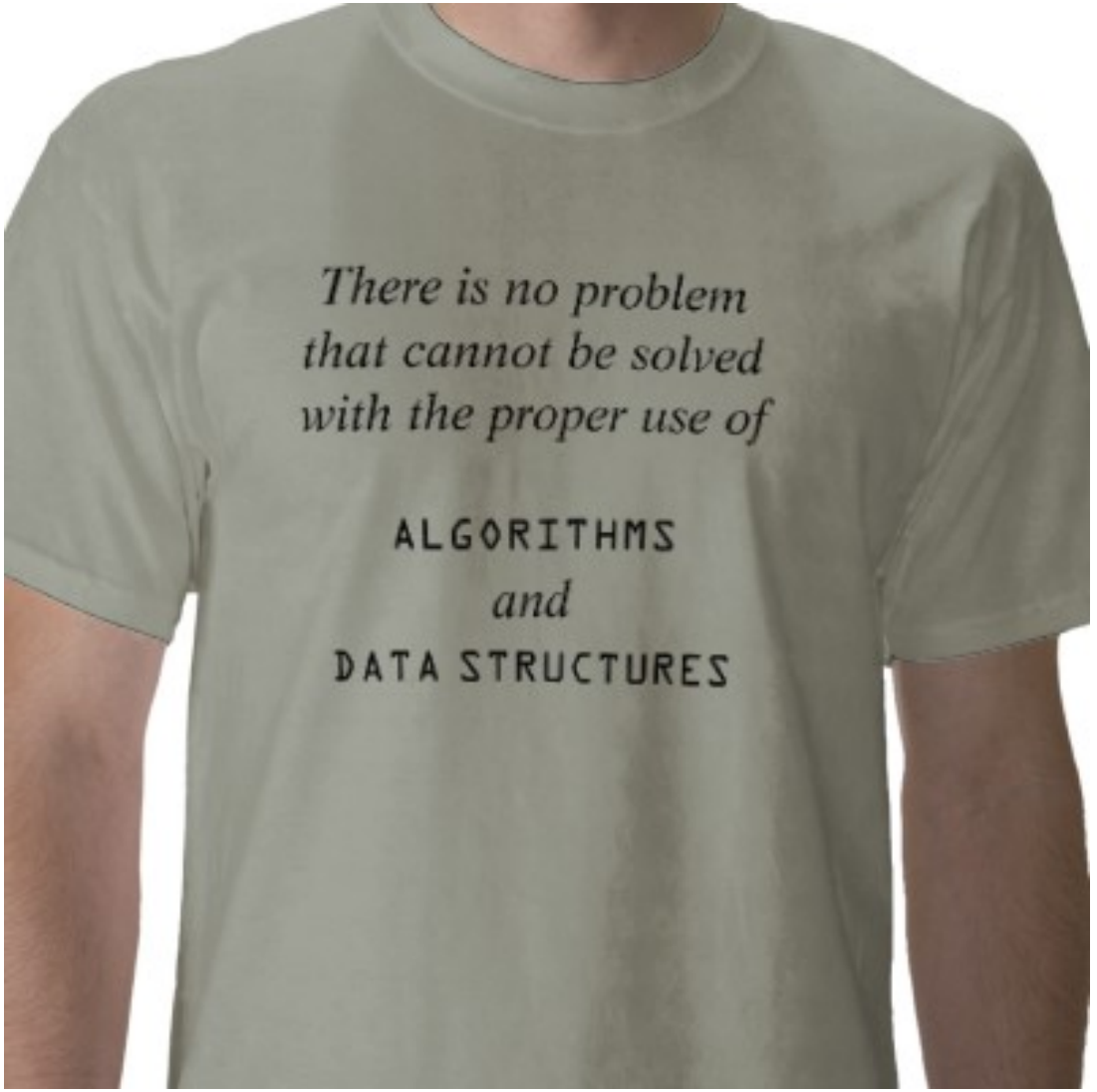
We adopt a two-phase strategy to construct a suffix tree on disk: 1) to construct a diskbase suffix-tree without suffix links and 2) rebuild suffix links upon the suffix-tree being constructed on disk, if needed.

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."

ED en biología computacional: aplicaciones



ED en biología computacional: aplicaciones

Ejemplo: Dr. Sheldon Cooper y su algoritmo para hacer amigos.