# Análisis del caso promedio

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

# Tries, árboles digitales de búsqueda y Patricia

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

    Hay información redundante:

    bestial bestir bestowal bestseller bestselling
  - Para ahorrar espacio: agrupar los prefijos comunes
  - Para ahorrar tiempo: si las palabras son más cortas es más rápida la búsqueda…

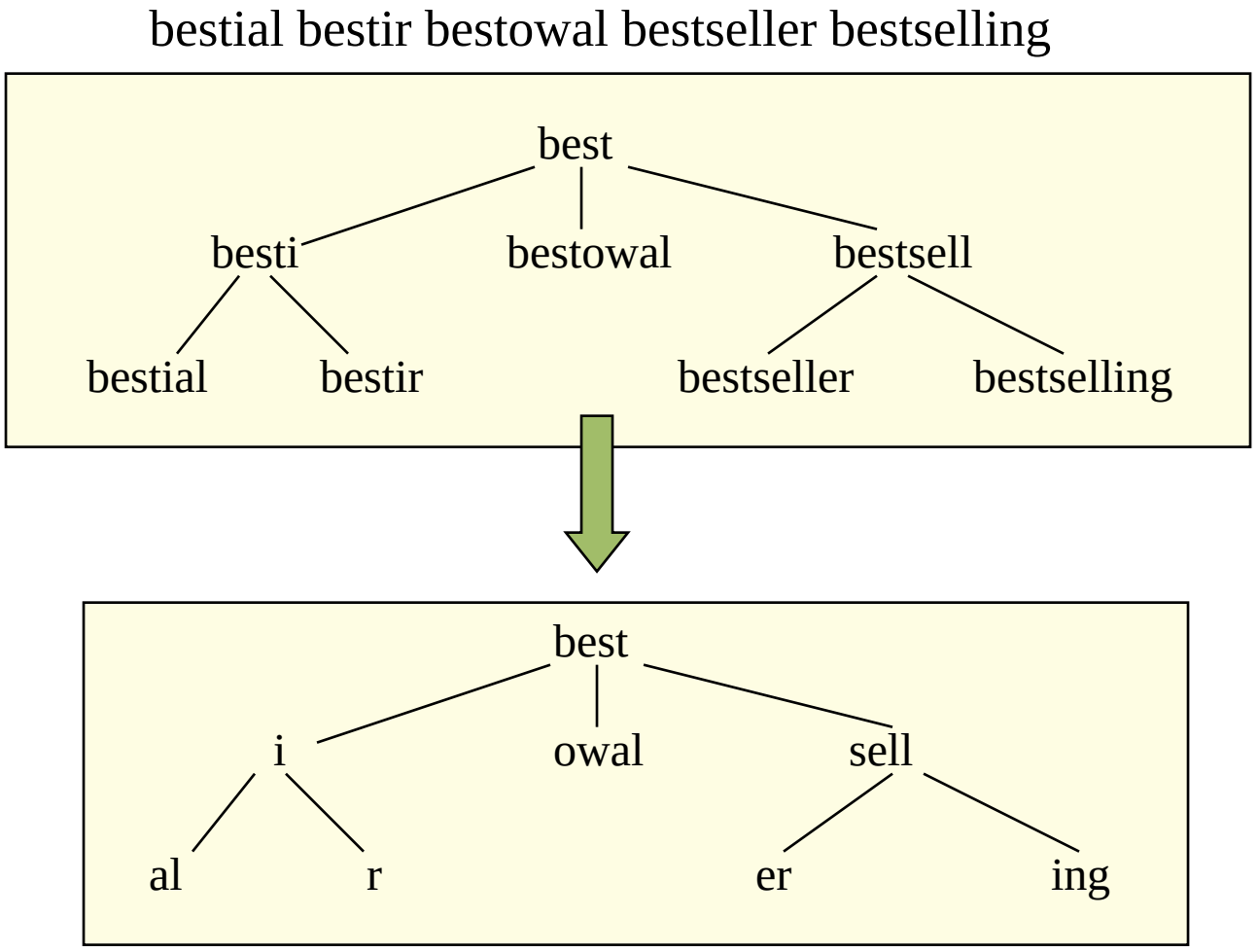
```
best
---- i
---- - al
---- - r
---- owal
---- sell
---- ---- er
---- ---- ing
```

# Tries, árboles digitales de búsqueda y Patricia

- Trie: definición formal
  - Sea $\Sigma=\{\sigma_1, \ldots, \sigma_m\}$ un **alfabeto** finito ($m > 1$).
  - Sea $\Sigma^*$ el conjunto de las **palabras** (o secuencias) de símbolos de $\Sigma$, y $X$ un subconjunto de $\Sigma^*$ (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 $\sigma$ quitándoles la primera letra.
  - Si el alfabeto tiene definida una relación de orden (caso habitual), el trie se llama árbol lexicográfico.

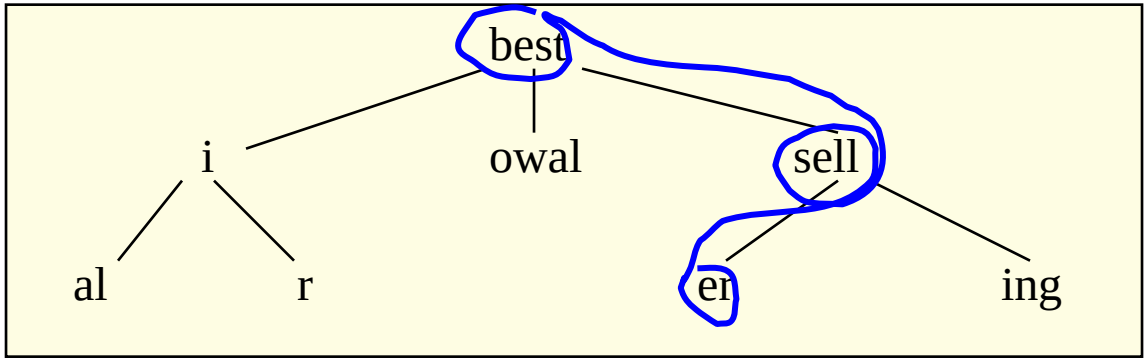
# Tries, árboles digitales de búsqueda y Patricia

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

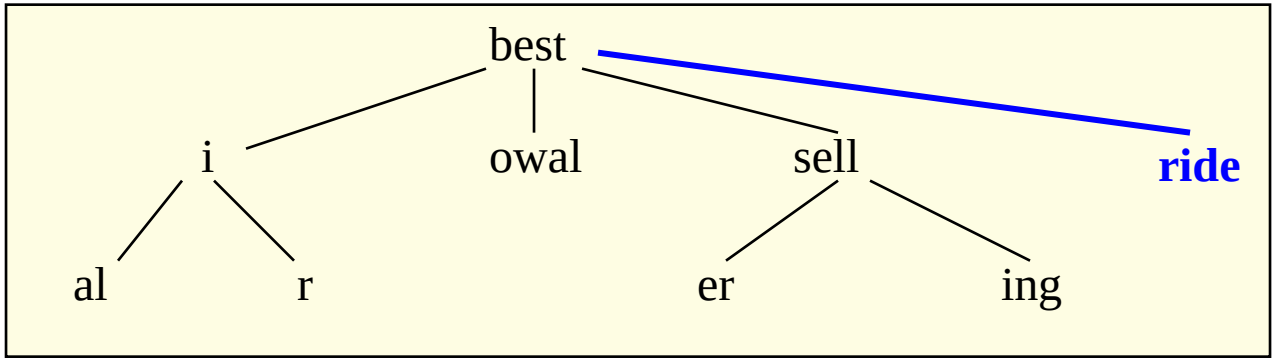


# Tries, árboles digitales de búsqueda y Patricia

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




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



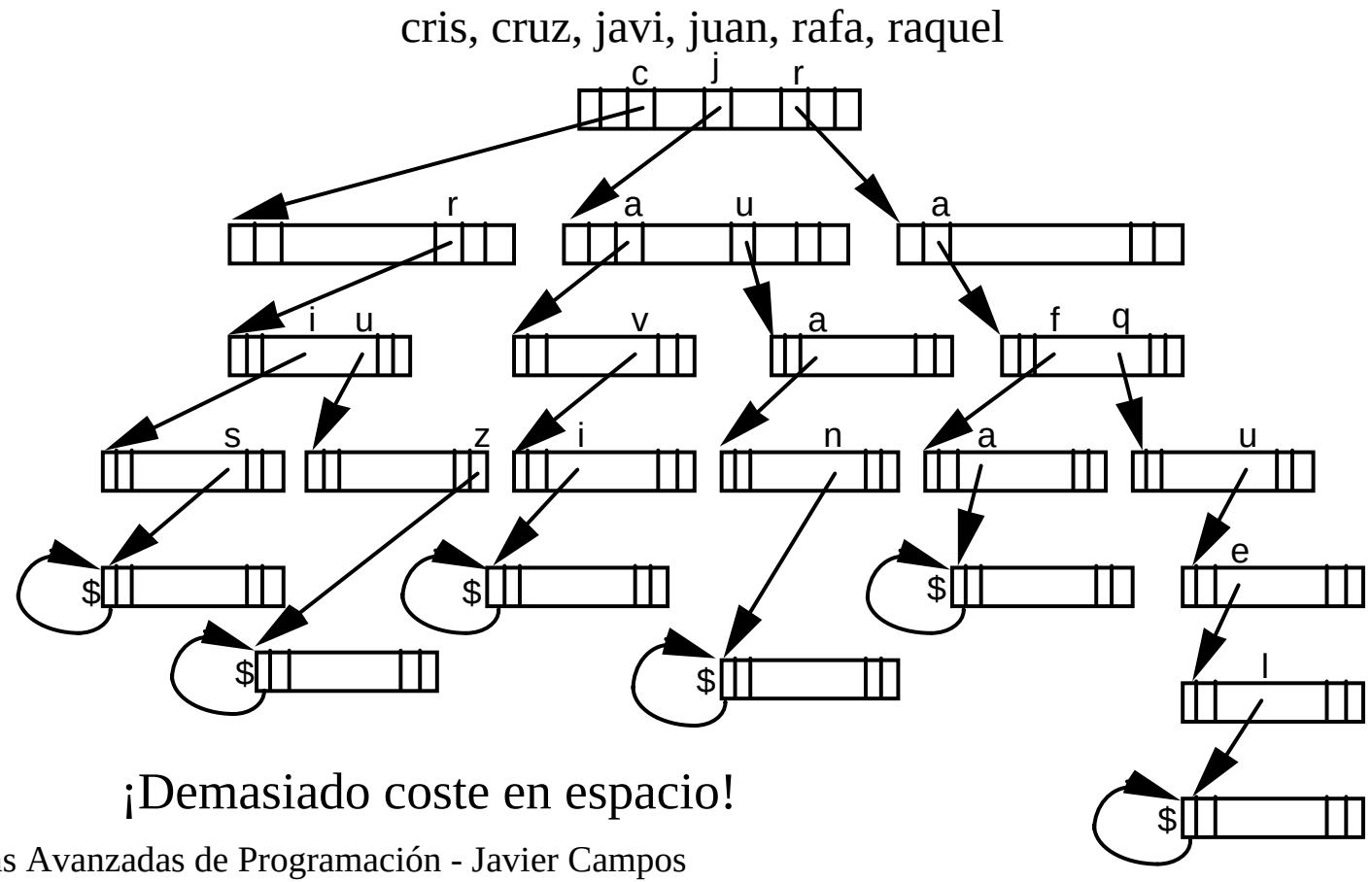
# Tries, árboles digitales de búsqueda y Patricia

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

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

# Tries, árboles digitales de búsqueda y Patricia

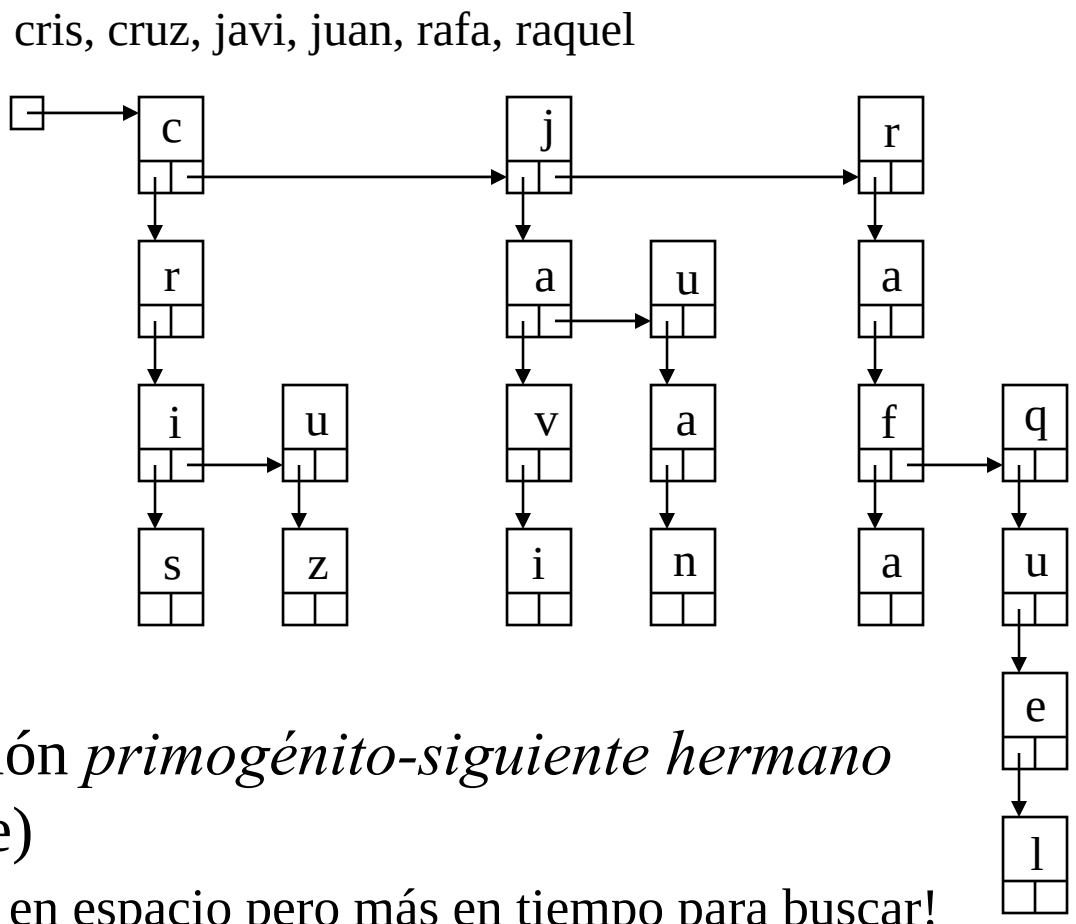
- Implementaciones de tries:
  - *Nodo-vector*: cada nodo es un vector de punteros para acceder a los subárboles directamente




¡Demasiado coste en espacio!

# Tries, árboles digitales de búsqueda y Patricia

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



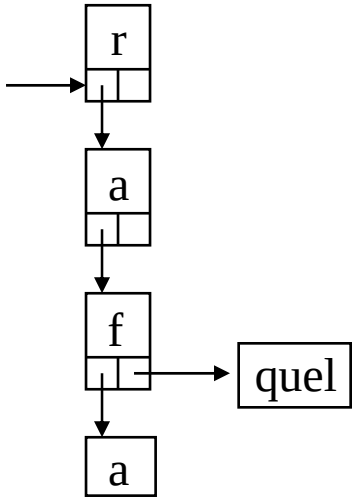

(representación *primogénito-siguiente hermano* de un bosque)

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

# Tries, árboles digitales de búsqueda y Patricia

– Precisión sobre las implementaciones anteriores:

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



# Tries, árboles digitales de búsqueda y Patricia

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

Cada nodo contiene:

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

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

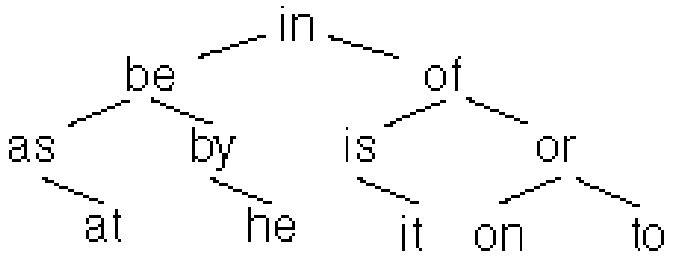
- Una búsqueda compara el carácter actual en la cadena buscada con el carácter del nodo.
- Si el carácter buscado es menor, la búsqueda de ese carácter sigue en el hijo izquierdo.
- Si el carácter buscado es mayor, se sigue en el hijo derecho.
- Si el carácter es igual, se va al hijo central, y se pasa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

# Tries, árboles digitales de búsqueda y Patricia

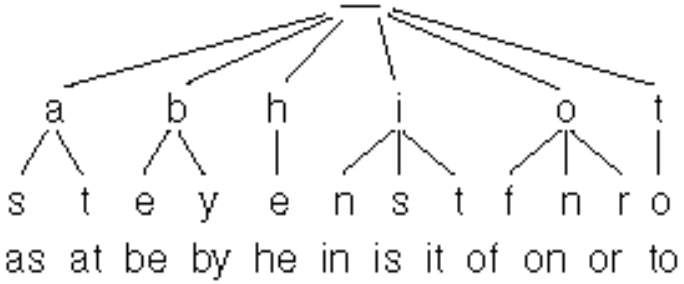
Árbol ternario de búsqueda para las palabras…

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

En un *abb*:

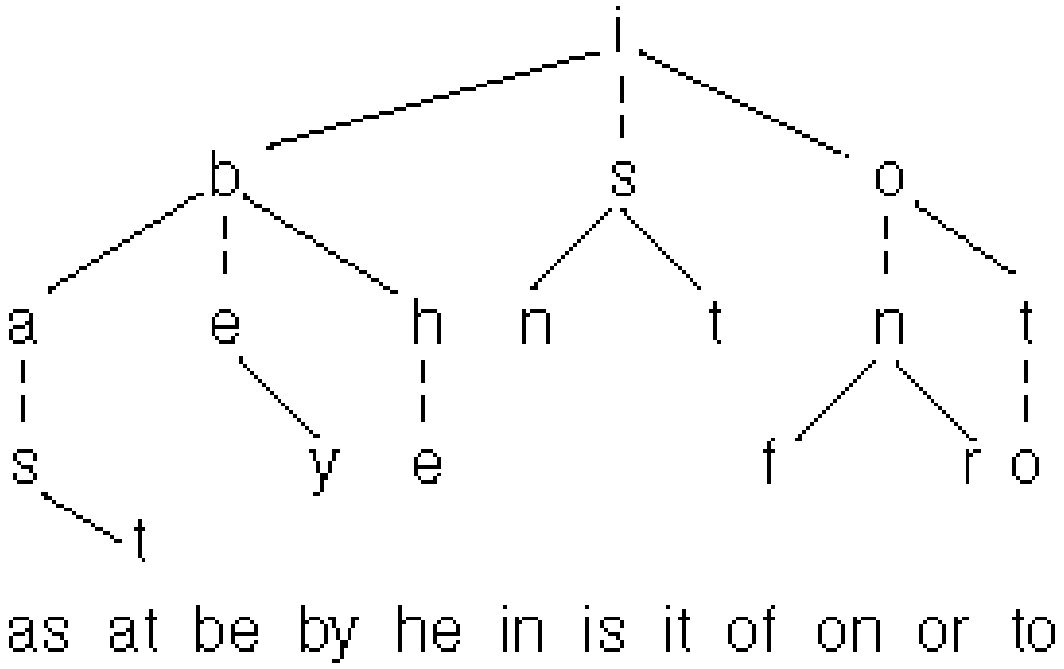



En un trie (representación nodo-vector o nodo-lista):



# Tries, árboles digitales de búsqueda y Patricia

Árbol ternario de búsqueda:



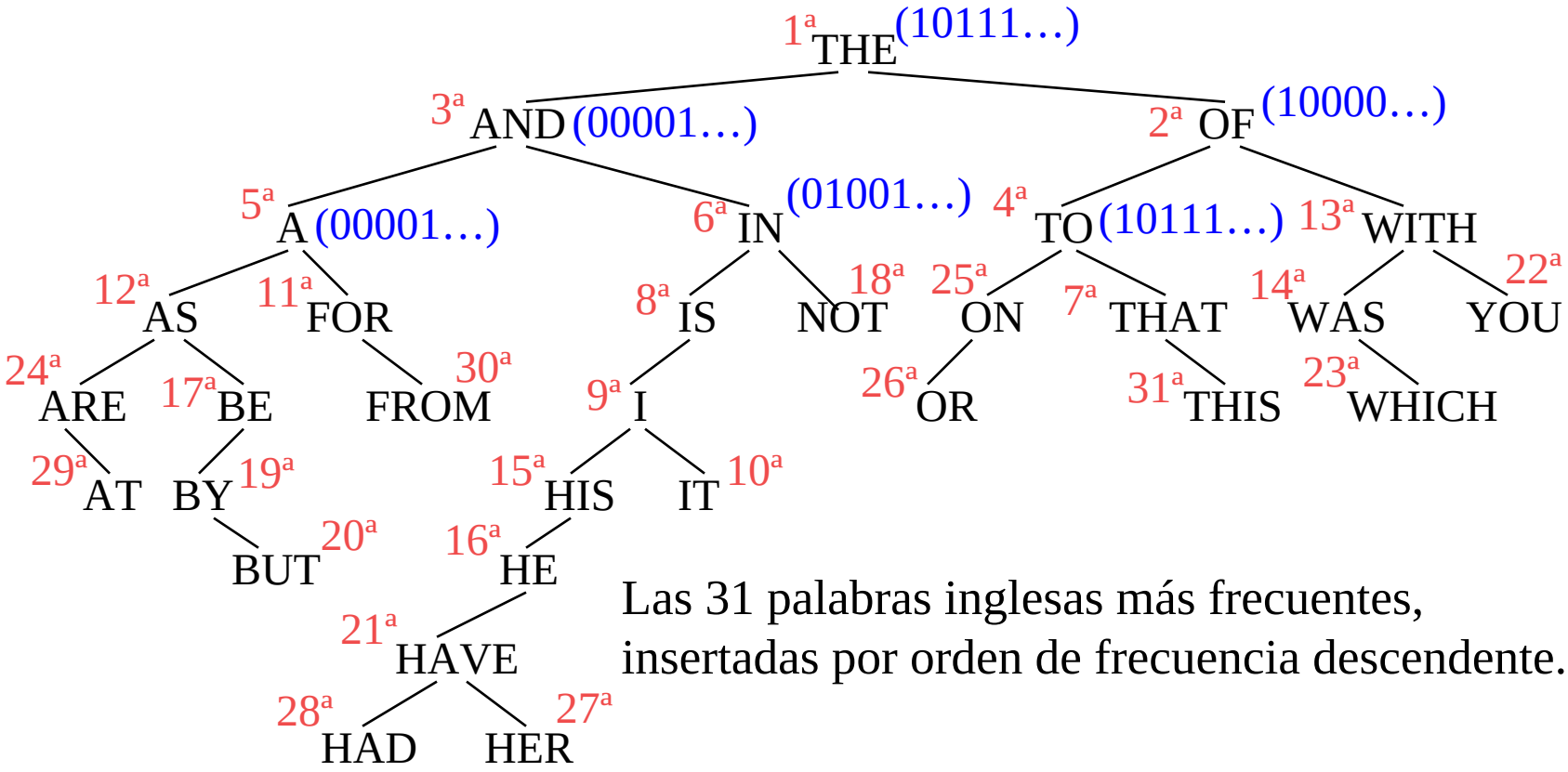
# Tries, árboles digitales de búsqueda y Patricia

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

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

# Tries, árboles digitales de búsqueda y Patricia

– Árboles digitales de búsqueda (caso binario):

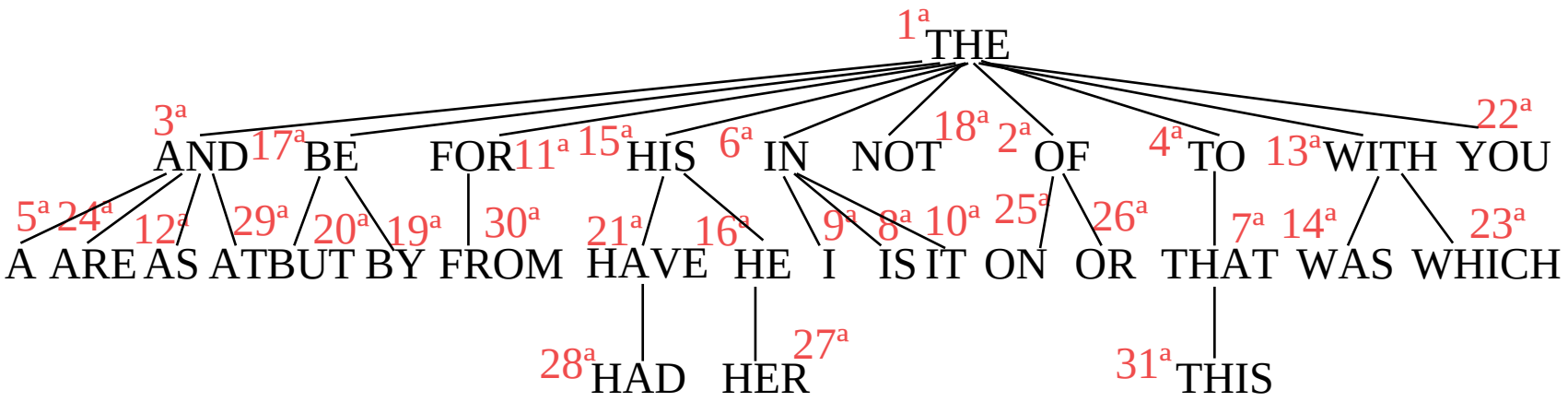



Las 31 palabras inglesas más frecuentes, insertadas por orden de frecuencia descendente.

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

# Tries, árboles digitales de búsqueda y Patricia

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



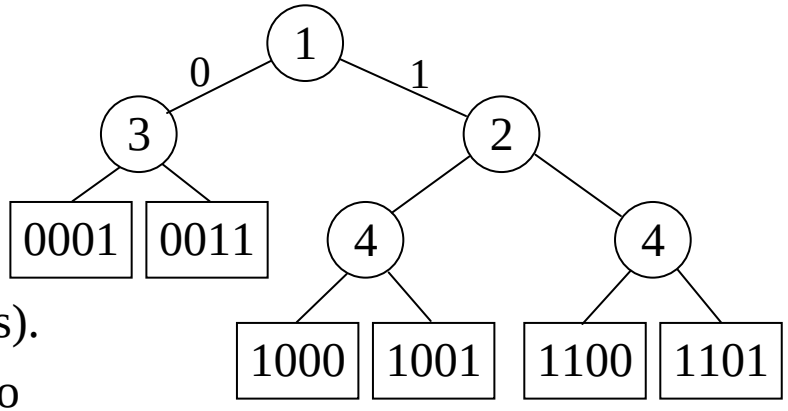

Los mismos datos de antes, insertados en igual orden, pero en un árbol digital de búsqueda de orden 27.

# Tries, árboles digitales de búsqueda y Patricia

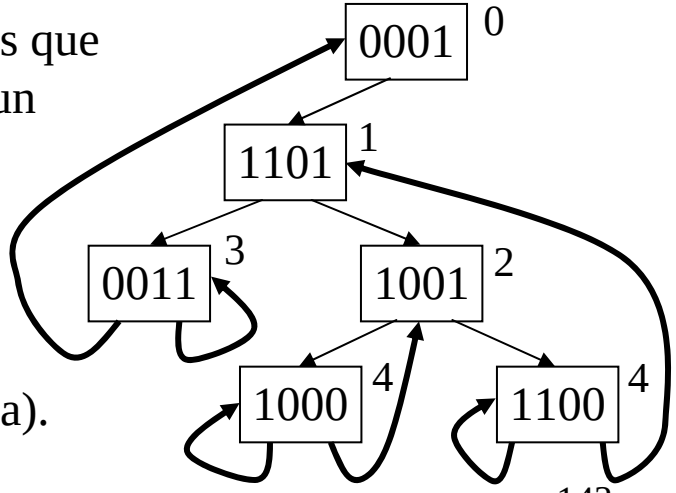
- *Patricia* (*P*ractical *A*lgorithm *T*o *R*etrieve *I*nformation *C*oded *I*n *A*lphanumeric)
  - Problema del trie: si |{claves}| << |{claves potenciales}|, la mayoría de los nodos internos tienen un solo hijo ➔ aumenta el coste en espacio
  - Idea: árbol binario, pero evitando las bifurcaciones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

# Tries, árboles digitales de búsqueda y Patricia

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

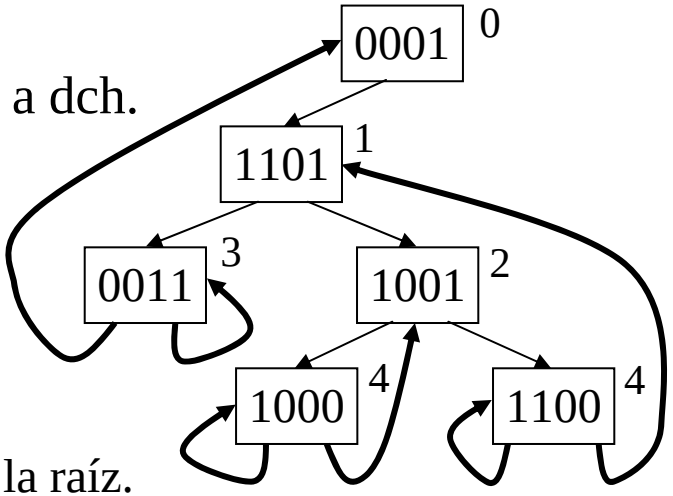



  - En un Patricia, las claves se almacenan en los nodos internos.
    - Como hay un nodo interno menos que nº de elementos, hay que añadir un nuevo nodo (se pone como raíz).
    - Cada nodo sigue guardando el nº de bit usado para bifurcar.
    - Ese nº distingue si el puntero sube o baja (si > el del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



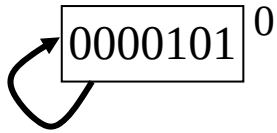
# Tries, árboles digitales de búsqueda y Patricia

- Búsqueda de clave:
  - Se usan los bits de la clave de izq. a dch. Bajando en el árbol.
  - Cuando el puntero que se ha seguido va hacia arriba se compara la clave con la del nodo.
  - Ejemplo, búsqueda de 1101:
    - Se empieza yendo al hijo izq. de la raíz.
    - El puntero que se ha seguido es hacia abajo (se sabe porque el bit que etiqueta el nodo 1101, 1, es mayor que el del padre, 0).
    - Se busca según el valor del bit 1 de la clave **buscada**, como es un 1, vamos al hijo derecho (el 1001).
    - El 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.

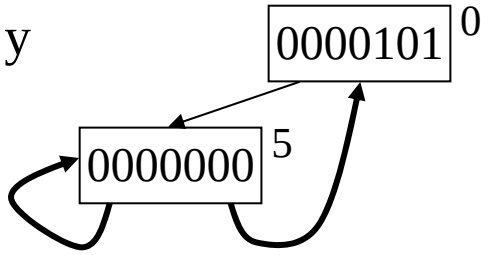


# Tries, árboles digitales de búsqueda y Patricia

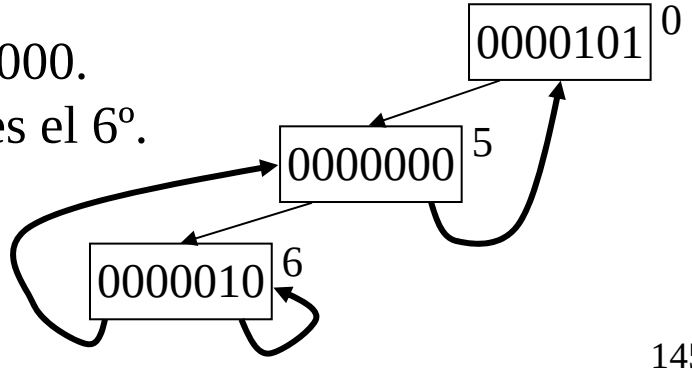
- Inserción de claves:
  - Partimos de árbol vacío (ninguna clave).
  - Insertamos la clave 0000101.




  - 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.

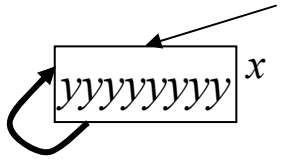



  - Insertamos 0000010. La búsqueda termina en 0000000. El primer bit en que difieren es el 6º.



# Tries, árboles digitales de búsqueda y Patricia

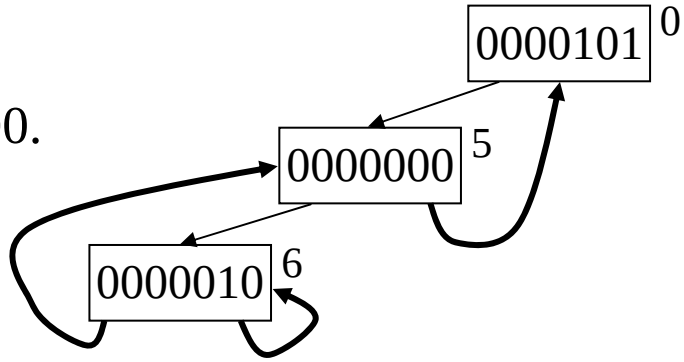
- Estrategia general de inserción (a partir de la 2ª clave):
  - Se busca la clave *C* a insertar, la búsqueda termina en un nodo con clave *C'*.
  - Se calcula el nº *b* de bit más a la izq. en que *C* y *C'* difieren.
  - Se crea un nuevo nodo con la nueva clave, etiquetado con el nº de bit anterior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

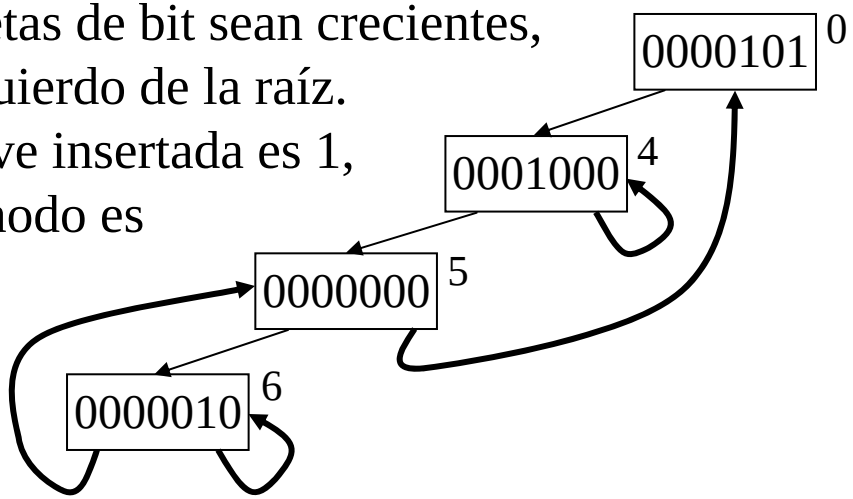


# Tries, árboles digitales de búsqueda y Patricia

– Inserción de claves (cont.):

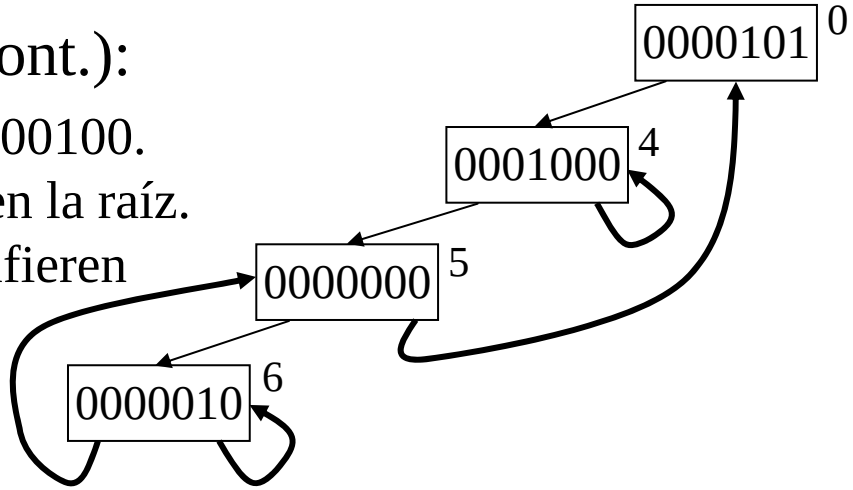
- Insertamos la clave 0001000.
  La búsqueda termina en 0000000.
  El primer bit en que difieren es el 4.
  Creamos un nuevo nodo con etiqueta 4 y ponemos en él la nueva clave.
  Se inserta ese nodo en el camino de la raíz a 0000000 de forma que las etiquetas de bit sean crecientes, es decir, como hijo izquierdo de la raíz.
  Como el bit 4 de la clave insertada es 1, el hijo dch. del nuevo nodo es un auto-puntero.



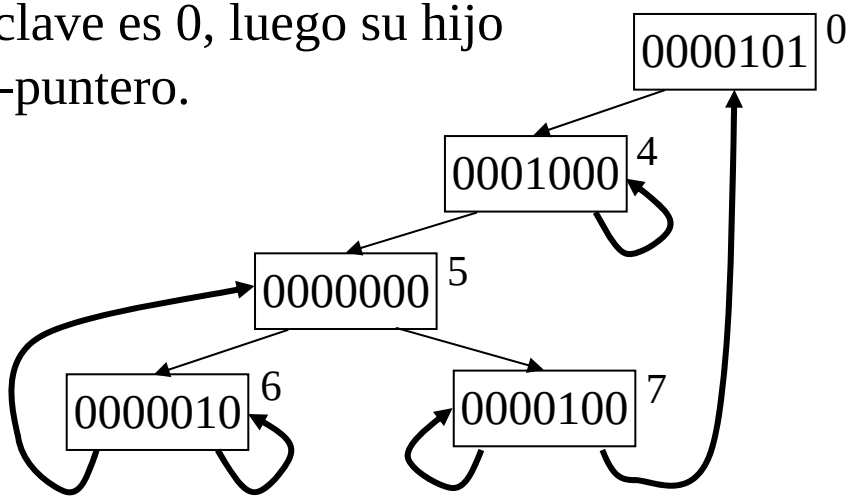



– Inserción de claves (cont.):

- Insertamos la clave 0000100.
  La búsqueda termina en la raíz.
  El primer bit en que difieren es el 7º.
  Creamos nuevo nodo con etiqueta 7.



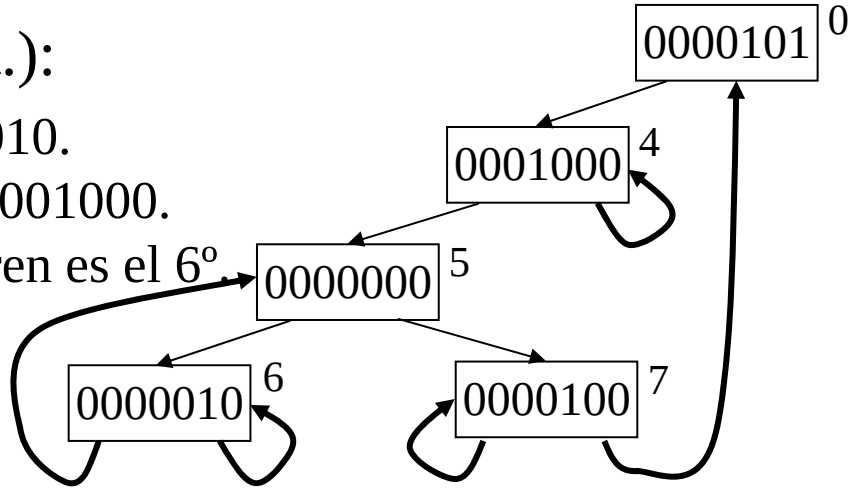

  Insertamos el nuevo nodo en el camino de búsqueda de forma que las etiquetas de bit sean crecientes: hijo dch. de 0000000.
  El bit 7 de la nueva clave es 0, luego su hijo izquierdo es un auto-puntero.

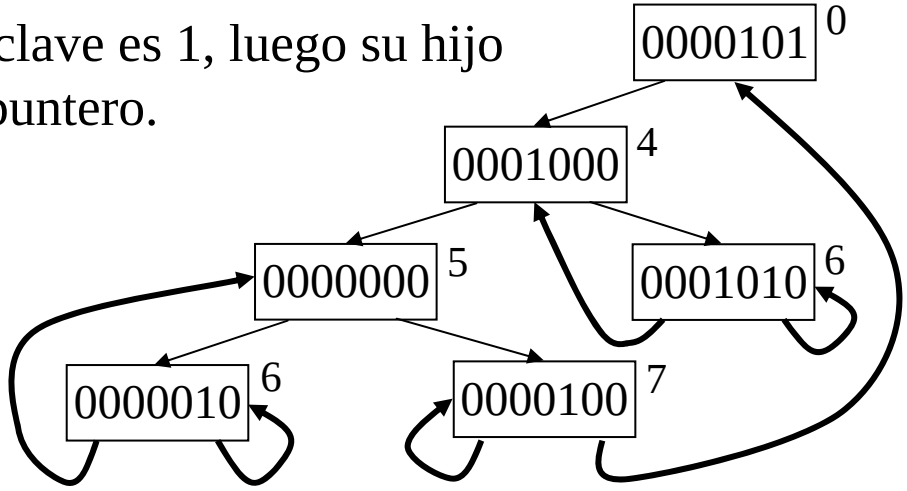


# Tries, árboles digitales de búsqueda y Patricia

– Inserción de claves (cont.):

- Insertamos la clave 0001010.
  La búsqueda termina en 0001000.
  El primer bit en que difieren es el 6º.
  Creamos nuevo nodo
  con etiqueta 6.
  Insertamos el nuevo nodo
  en el camino de búsqueda de forma que las etiquetas de bit sean crecientes: hijo dch. de 0001000.
  El bit 6 de la nueva clave es 1, luego su hijo derecho es un auto-puntero.



# Tries, árboles digitales de búsqueda y Patricia

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

# Tries, árboles digitales de búsqueda y Patricia

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

# Tries, árboles digitales de búsqueda y Patricia

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

# Tries, árboles digitales de búsqueda y Patricia

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

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

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

# Tries, árboles digitales de búsqueda y Patricia

- Una extensión más elaborada: *radix sort*

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

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

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

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

# Tries, árboles digitales de búsqueda y Patricia

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

# Tries, árboles digitales de búsqueda y Patricia

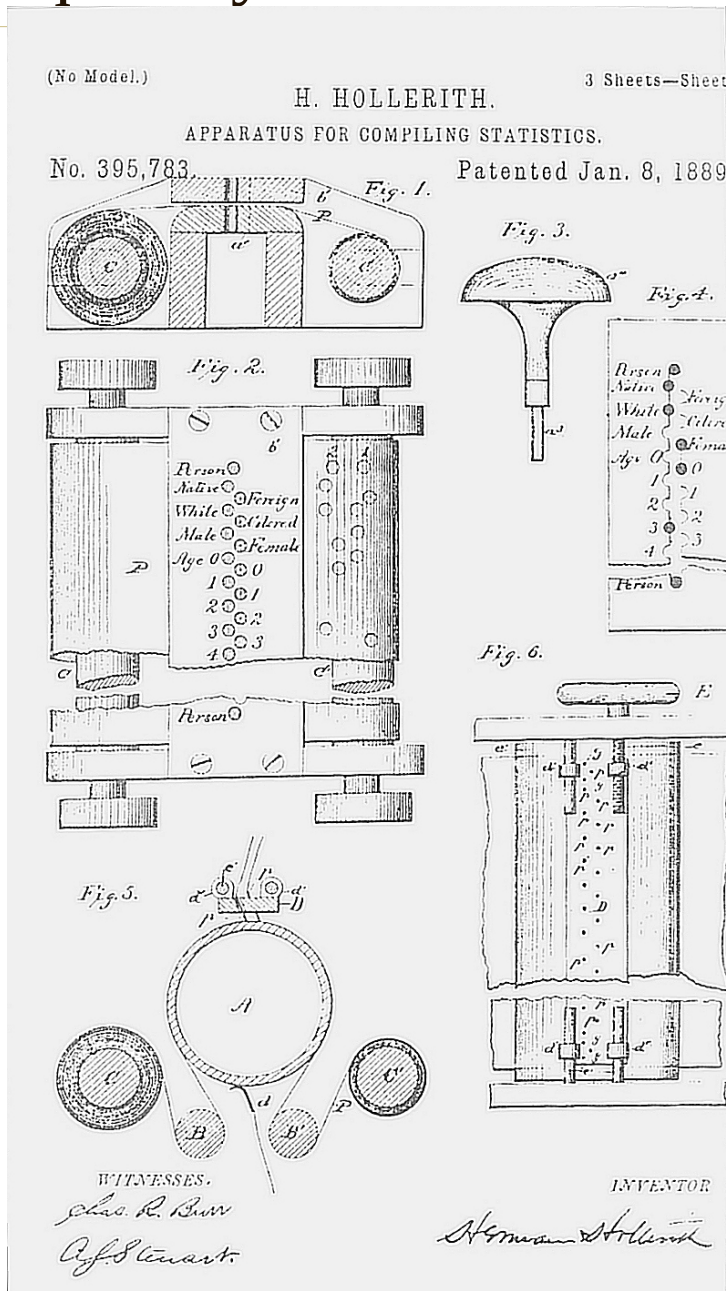
```
algoritmo radix(X,n,k)  {pre: X=vector[1..n] de claves,
              cada una con k dígitos;  post: X ordenado}
principio
  colocar los elmtos. de X en cola GQ {puede usarse X};
  para i:=1 hasta d hacer {d=base de numeración usada)
    colaVacía(Q[i])
  fpara;
  para i:=k descendiendo hasta 1 hacer
    mq not esVacía(GQ) hacer
      x:=primero(GQ); eliminar(GQ);
      d:=dígito(i,x); insertar(x,Q[d])
    fmq;
    para t:=1 hasta d hacer insertarCola(Q[t],GQ) fpara
  fpara;
  para i:=1 hasta n hacer
    X[i]:=primero(GQ); eliminar(GQ)
  fpara
fin
```

- Análisis del método *radix sort*:
  - En tiempo: $O(kn)$, es decir, considerando que $k$ es una constante, es un método lineal en el tamaño del vector
  - Nótese que a mayor base de numeración menor coste
  - En espacio: $O(n)$ espacio adicional.

    (Es posible hacerlo in-situ, y sólo se precisa un espacio adicional $O(\log n)$ para guardar posiciones del vector)

# Tries, árboles digitales de búsqueda y Patricia

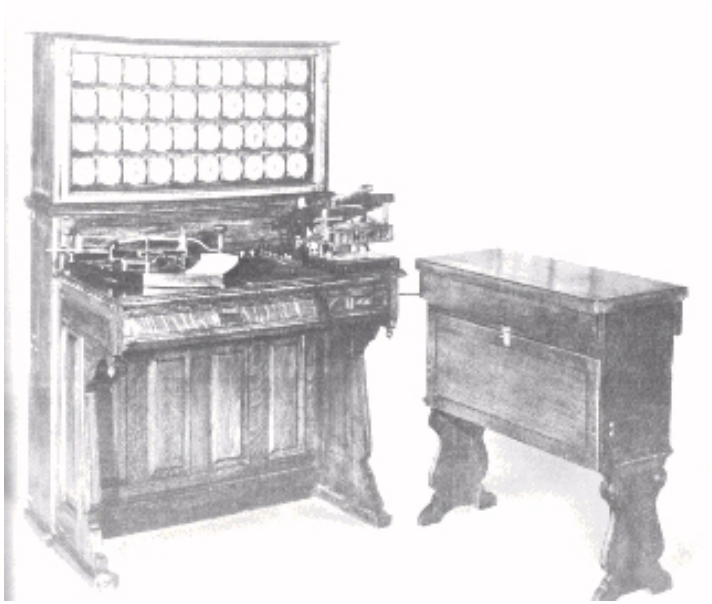
- Un poco de historia: orígenes del *radix sort*
  - EE.UU., 1880: no se puede terminar el censo de la década anterior (en concreto, no se llega a contar el número de habitantes solteros)
  - Herman Hollerith (empleado de la oficina del censo, de 20 años de edad) inventa una máquina tabuladora eléctrica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



# Tries, árboles digitales de búsqueda y Patricia

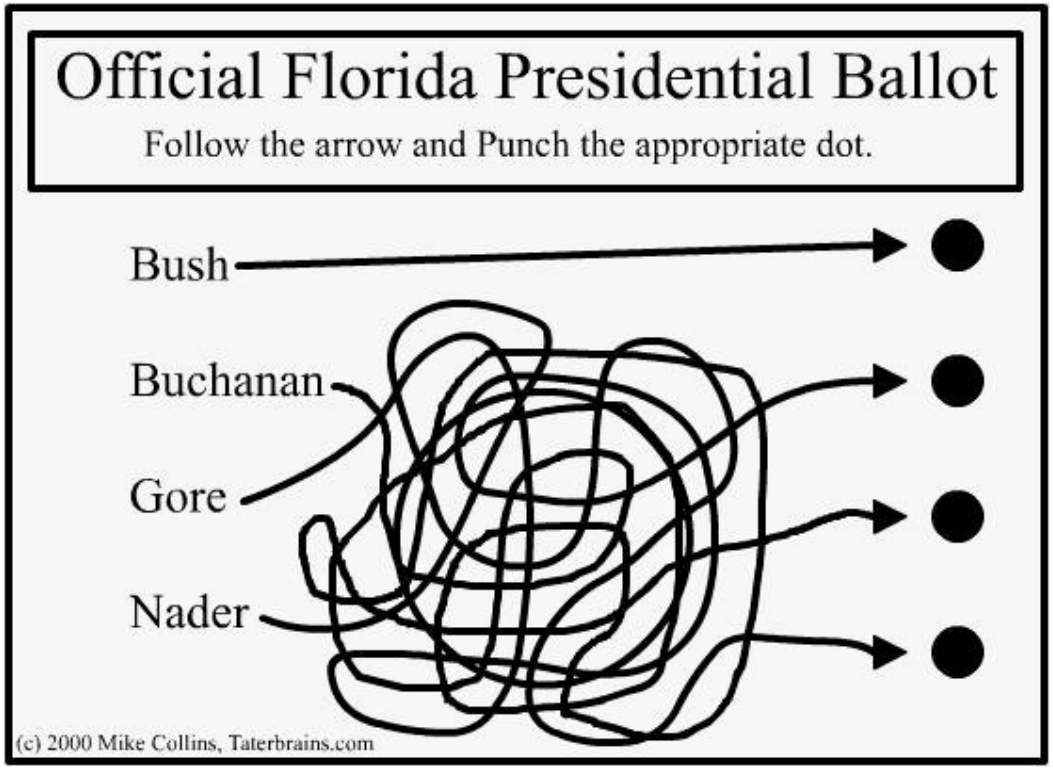
- Un poco de historia: orígenes del *radix sort* (continuación)
  - 1890: se usan unas 100 máquinas de Hollerith para tabular las listas del censo de la década (un operador experto procesaba 19.071 tarjetas en una jornada laboral de 6’5 horas, unas 49 tarjetas por minuto)
  - 1896: Hollerith crea la empresa *Tabulating Machine Company*

- Un poco de historia: orígenes del *radix sort* (continuación)
  - 1900: Hollerith resuelve otra crisis federal inventando una nueva máquina con alimentación automática de tarjetas (útil, con más o menos variaciones, hasta 1960)
  - 1911: la empresa de Hollerith se fusiona con otras dos, creando la *Calculating-Tabulating-Recording Company* (CTR)



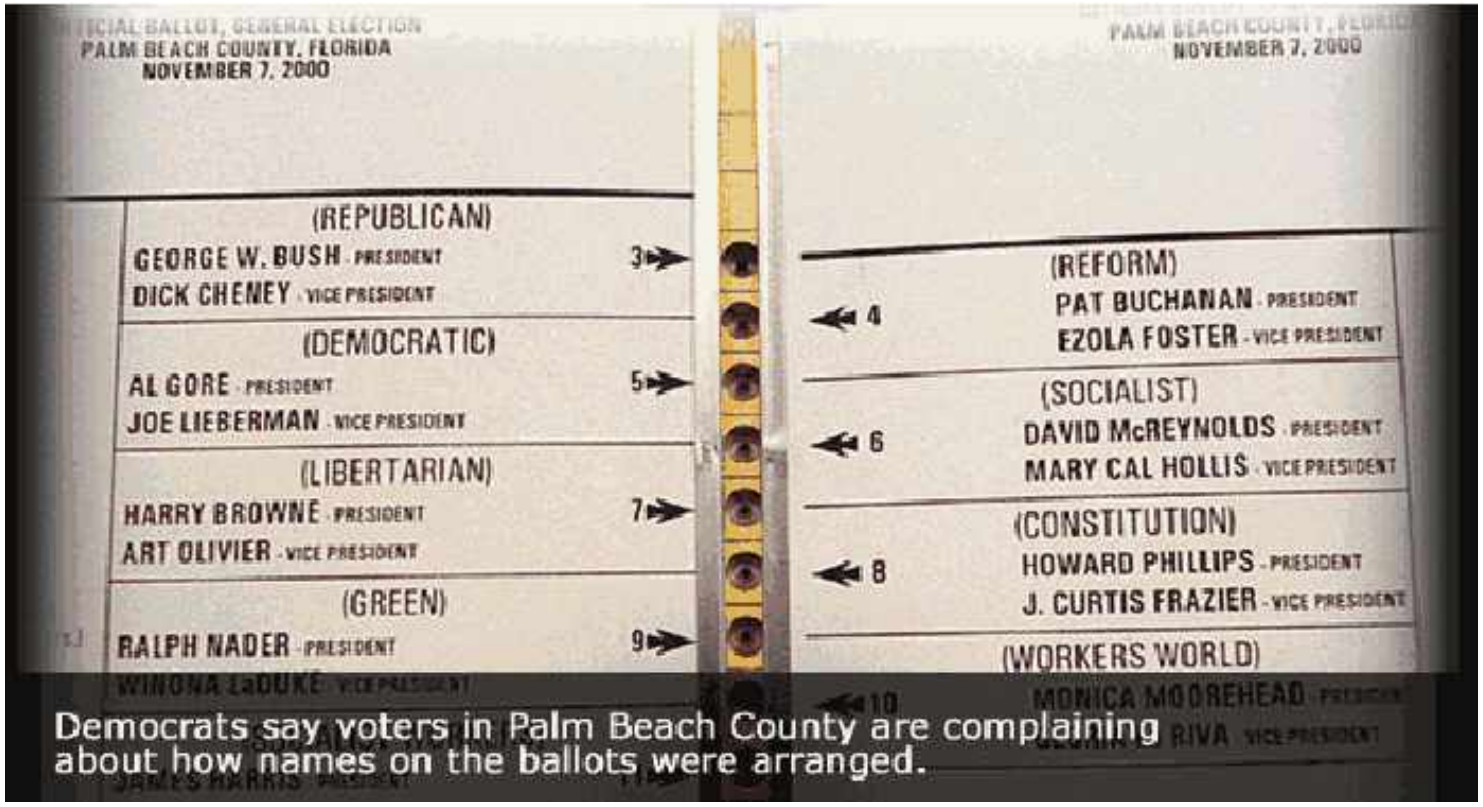
  - 1924: Thomas Watson cambia el nombre a la CTR y la llama *International Business Machines* (IBM)

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



# Tries, árboles digitales de búsqueda y Patricia

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



# Tries, árboles digitales de búsqueda y Patricia

- Método de ordenación *radix*-intercambio:

  (versión del libro de D. Knuth)

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

# Tries, árboles digitales de búsqueda y Patricia

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

# Tries, árboles digitales de búsqueda y Patricia

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

Según Knuth(*),

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

con $\gamma = 0{,}577215...$ la constante de Euler

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

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

# Tries, árboles digitales de búsqueda y Patricia

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

# Tries, árboles digitales de búsqueda y Patricia

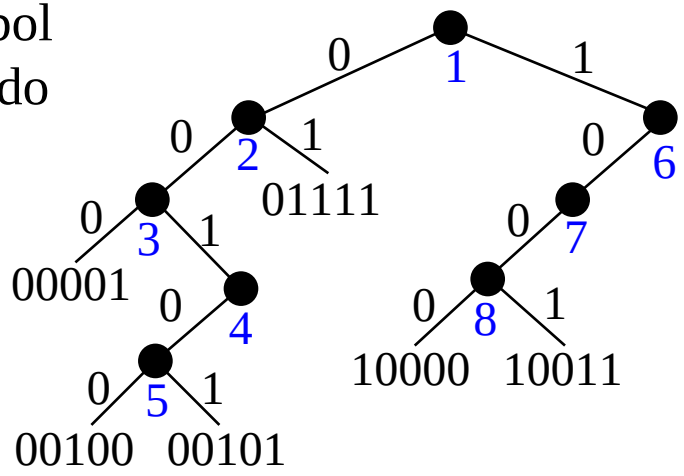
– Ejemplo: con 6 claves, las letras de ‘ORDENA’

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

| | | | | | | | i | d | bit |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 10000 | 10011 | 00100 | 00101 | 01111 | 00001 | 1 | 6 | 1 |
| 2 | 00001 | 01111 | 00100 | 00101 | 10011 | 10000 | 1 | 4 | 2 |
| 3 | 00001 | 00101 | 00100 | 01111 | 10011 | 10000 | 1 | 3 | 3 |
| 4 | 00001 | 00101 | 00100 | 01111 | 10011 | 10000 | 2 | 3 | 4 |
| 5 | 00001 | 00101 | 00100 | 01111 | 10011 | 10000 | 2 | 3 | 5 |
| 6 | 00001 | 00100 | 00101 | 01111 | 10011 | 10000 | 5 | 6 | 2 |
| 7 | 00001 | 00100 | 00101 | 01111 | 10011 | 10000 | 5 | 6 | 3 |
| 8 | 00001 | 00100 | 00101 | 01111 | 10011 | 10000 | 5 | 6 | 4 |
| 9 | 00001 | 00100 | 00101 | 01111 | 10000 | 10011 | | | |

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

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



# Tries, árboles digitales de búsqueda y Patricia

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

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

con $\gamma = 0{,}577215...$ la constante de Euler

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

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

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

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

# Tries, árboles digitales de búsqueda y Patricia

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

  "posiblemente el hueso asintótico más duro que hemos tenido que roer…"