Algorítmica

Búsqueda y Ordenación Digital

Conrado Martínez
U. Politècnica Catalunya

ALG Q1-2022–2023

Temario

Parte I: Repaso de Conceptos Algorítmicos

Parte II: Búsqueda y Ordenación Digital

Parte III: Algoritmos Voraces

Parte IV: Programación Dinámica

Parte V: Flujos sobre Redes y Programación Lineal

Parte II

Búsqueda y Ordenación Digital

Parte II

Búsqueda y Ordenación Digital

Tries

Las claves que identifican a los elementos de una colección están formadas por una secuencia de símbolos (p.e. caracteres, dígitos, bits), siendo está descomposición más o menos “natural”, y dicha circunstancia puede aprovecharse ventajosamente para implementar las operaciones típicas de un diccionario de manera notablemente eficiente.
Adicionalmente, es frecuente que necesitemos ofertar operaciones del TAD basadas en esta descomposición de las claves en símbolos: por ejemplo, podemos querer una operación que dada una colección de palabras C y una palabra p nos devuelva la lista de todas las palabras de C que contienen a la palabra p como subcadena.

Consideremos un alfabeto finito $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$ de cardinalidad $m \geq 2$. Mediante Σ^* denotaremos, como es habitual en muchos contextos, el conjunto de las secuencias o cadenas formadas por símbolos de Σ. Dadas dos secuencias u y v denotaremos $u \cdot v$ la secuencia resultante de concatenar u y v. Para la secuencia de longitud 0, es decir, la secuencia vacía usaremos la notación λ.

Definición

Dado un conjunto finito de secuencias $X \subset \Sigma^*$ de idéntica longitud, el *trie* T correspondiente a X es un árbol m-ario definido recursivamente de la siguiente manera:

1. Si X contiene un sólo elemento o ninguno entonces T es un árbol consistente en un único nodo que contiene al único elemento de X o está vacío.
2. Si $|X| \geq 2$, sea T_i el trie correspondiente a $X_i = \{y \,|\, x = \sigma_i \cdot y \in X \wedge \sigma_i \in \Sigma\}$. Entonces T es un árbol m-ario constituido por una raíz $\circ$ y los m subárboles $T_1, T_2, \ldots, T_m$.

$x_1 = 000101$

$x_2 = 010001$

$x_3 = 101000$

$x_4 = 010101$

$x_5 = 110101$

$x_6 = 100111$

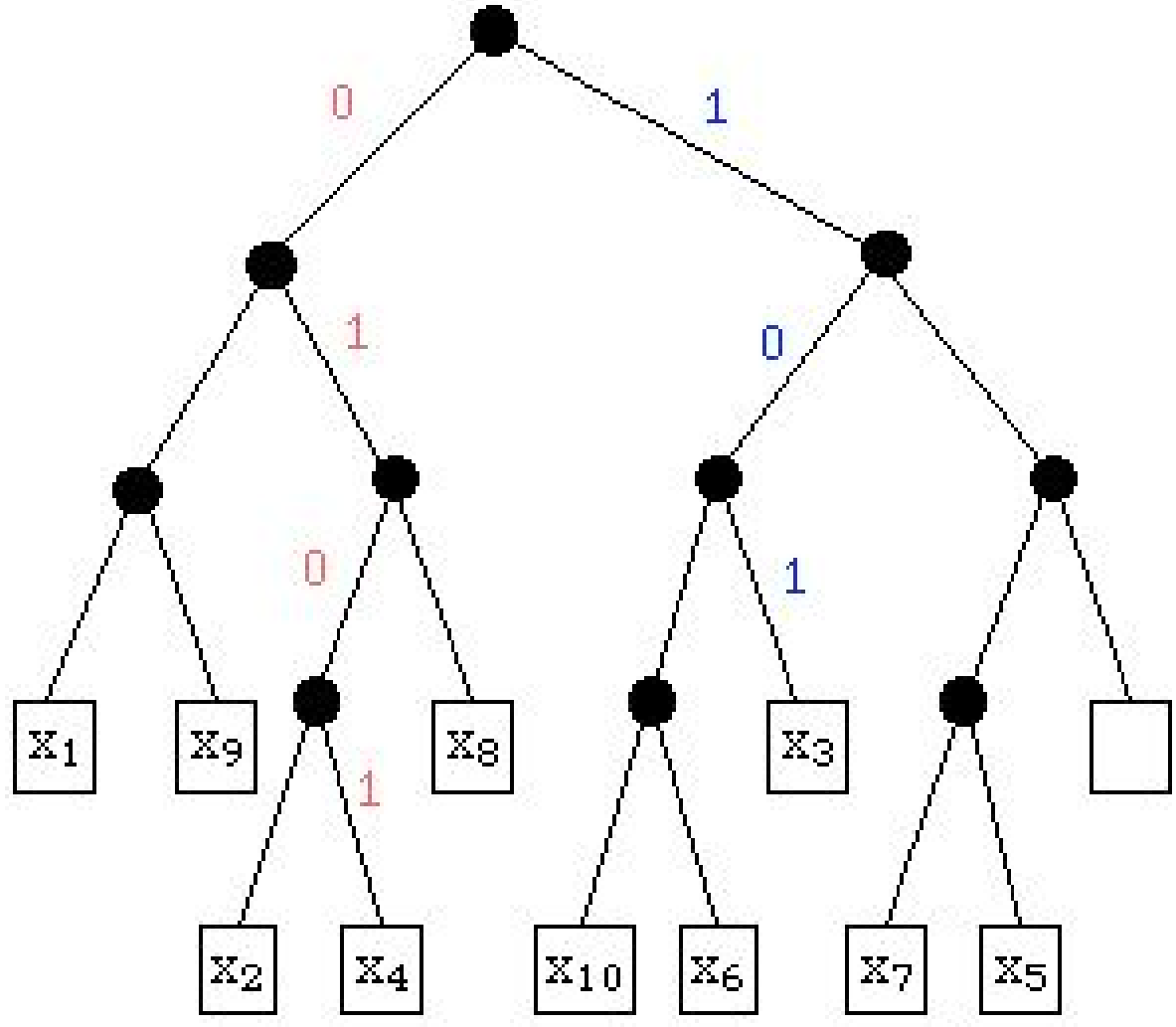
$x_7 = 110001$

$x_8 = 011111$

$x_9 = 001110$

$x_{10} = 100001$

$m = 2$



Lema

Si las aristas del trie T correspondiente a un conjunto X se etiquetan mediante los símbolos de Σ de tal modo que la arista que une la raíz con su primer subárbol se etiqueta σ_1, la que une la raíz con su subárbol se etiqueta σ_2, etc. entonces las etiquetas del camino que nos llevan desde la raíz hasta una hoja no vacía que contiene a x constituyen el prefijo más corto que distingue unívocamente a x; es decir, ningún otro elemento de X empieza con el mismo prefijo.

Lema

Sea p la etiqueta correspondiente a un camino que va de la raíz de un trie T hasta un cierto nodo (interno u hoja) de T. Entonces el subárbol enraizado en dicho nodo contiene todos los elementos de X que tienen en común al prefijo p (y no más elementos).

Lema

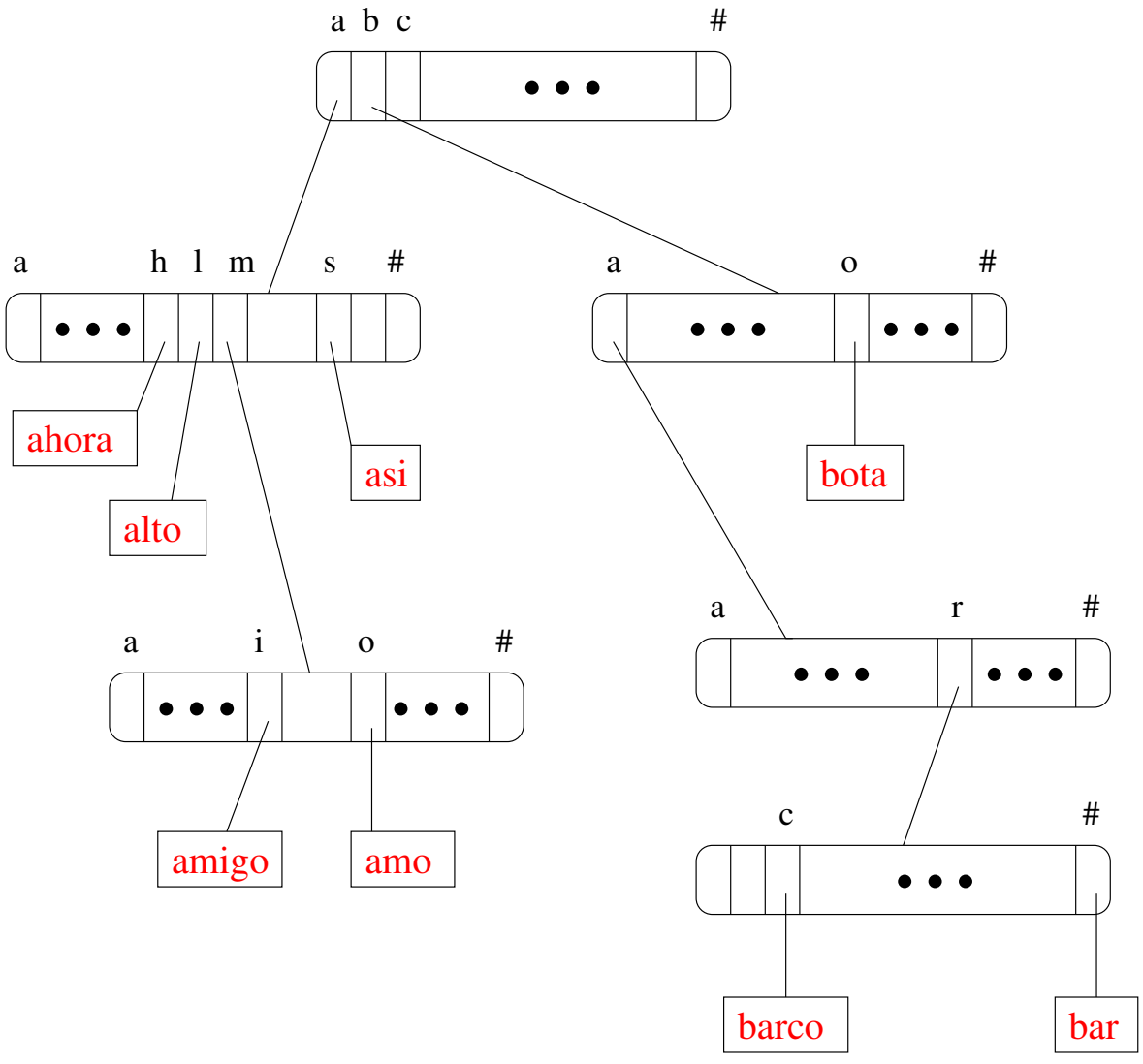
Dado un conjunto $X \subset \Sigma^*$ de secuencias de igual longitud, su trie correspondiente es único. En particular T no depende del orden en que se "presenten" los elementos de X.

Lema

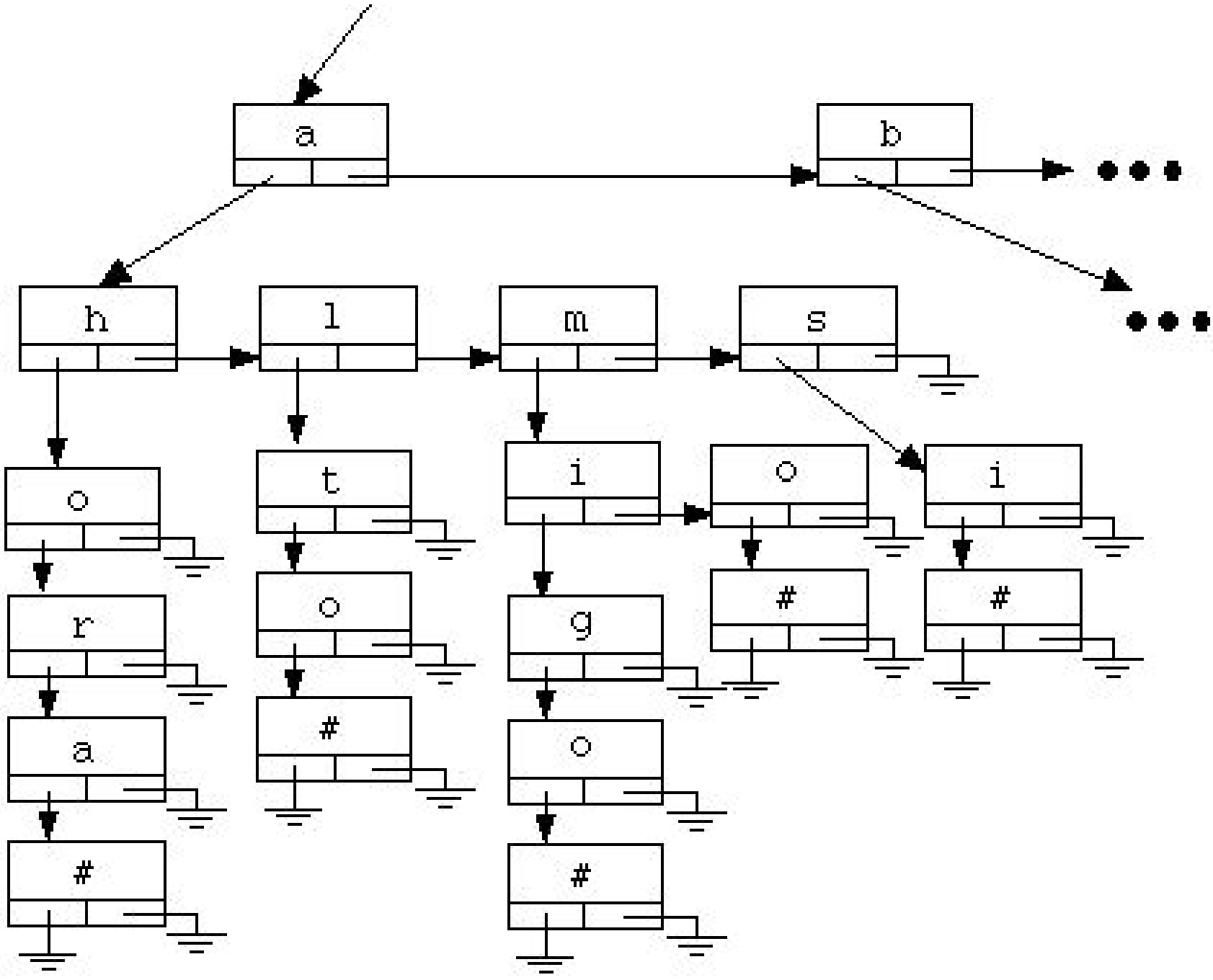
La altura de un trie T es igual a la longitud mínima de prefijo necesaria para distinguir cualesquiera dos elementos del conjunto al que corresponde el trie. En particular, si ℓ es la longitud de las secuencias en X, la altura de T será $\leq \ell$.

La definición de tries impone que todas las secuencias sean de igual longitud, lo cual es muy restrictivo. Pero si no exigimos esta condición entonces tenemos un problema que habremos de afrontar: si un elemento x es prefijo propio de otro elemento y, cómo podremos distinguirlo? Cómo diferenciamos las situaciones en la que x e y pertenecen ambos a X de las situaciones en la que sólo y está en X?
Una solución habitual consiste en ampliar Σ con un símbolo especial (p.e. ♯) de fin de secuencia, y marcar cada una de las secuencias en X con dicho símbolo. Ello garantiza que ninguna de las secuencias (marcadas) es prefijo propio de otra. El "precio" a pagar es que hay que trabajar con un alfabeto de $m + 1$ símbolos.

X = {ahora, alto, amigo, amo, asi, bar, barco, bota, ...}



Las técnicas de implementación de los tries son las convencionales para árboles. Si se utiliza un vector de apuntadores por nodo, los símbolos de Σ suelen poderse utilizar directamente como índices (eventualmente, se habrá de utilizar una función $ind : \Sigma \to \{1, \ldots, m\}$). Las hojas que contienen los elementos de X pueden almacenar exclusivamente los sufijos restantes, ya que el prefijo está ímplicitamente codificado en el camino desde la raíz a la hoja. En el caso en que se utilice la representación primogénito-siguiente hermano, cada nodo almacena un símbolo y sendos apuntadores al primogénito y al siguiente hermano. Puesto que suele haber definido un orden sobre Σ la lista de hijos de cada nodo suele ordenarse de acuerdo a áquel.


a
b
h
l
m
s
o
t
i
o
i
r
o
g
#
#
a
#
o
#
#

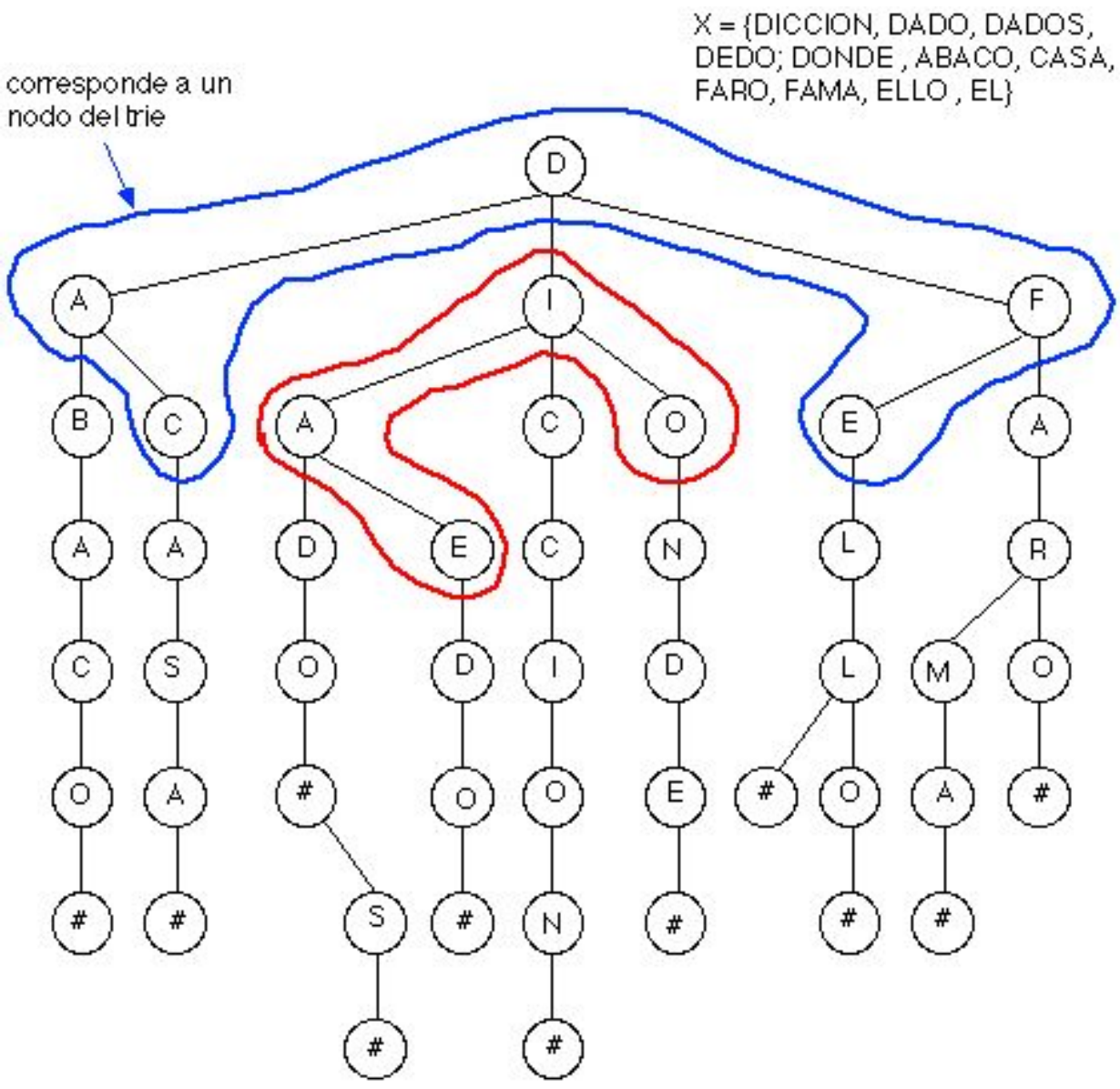
Aunque es más costoso en espacio emplear nodos del trie para representar las palabras completas, resulta ventajoso evitar la necesidad de nodos de distinto tipo, apuntadores a nodos de diferentes tipos, o representaciones poco eficientes para los nodos (p.e. *unions*).

```
// La clase Clave debe soportar las siguientes
// operaciones:
// x.length() devuelve la longitud >= 0 de una clave x
template <typename Clave>
int length() throw();

// x[i] devuelve el i-ésimo símbolo de x,
// lanza un error si i < 0 o i >= x.length()
template <typename Simbolo, typename Clave>
Simbolo operator[](const Clave& x, int i) throw(error);

template <typename Simbolo, typename Clave,typename Valor>
class DiccDigital {
public:
   ...
private:
   struct nodo_trie {
      Simbolo _c;
      nodo_trie* _primg;
      nodo_trie* _sigher;
      Valor _v;
   };
   nodo_trie* raiz;
   ...
};
```

```
template  <typename Simbolo,
          typename Clave,
          typename Valor>
void DiccDigital<Simbolo,Clave,Valor>::busca(
     const Clave& k, bool& esta, Valor& v) const
       throw(error) {

   nodo_trie* p = busca_en_trie(raiz, k, 0);
   if (p == nullptr)
      esta = false;
   else {
      esta = true;
      v = p -> _v;
   }
}

// Pre: p es la raíz del subárbol conteniendo
// las claves cuyos i-1 primeros símbolos
// coinciden con los i-1 símbolos iniciales de k
// Coste: Θ(longitud(k))
template  <typename Simbolo, typename Clave,
          typename Valor>
DiccDigital<Simbolo,Clave,Valor>::nodo_trie*
   DiccDigital<Simbolo,Clave,Valor>::busca_en_trie(
     nodo_trie* p, const Clave& k,
     int i) const throw() {

   if (p == nullptr)        return nullptr;
   if (i == k.length()) return p;
   if (p -> _c > k[i])  return nullptr;
   if (p -> _c < k[i])
      return busca_en_trie(p -> _sigher, k, i);
   // p -> _c == k[i]
   return busca_en_trie(p -> _primg, k, i+1);
}
```

Una solución que intenta combinar eficiencia en el acceso a los subárboles y en espacio consiste en implementar cada “nodo” del trie como un BST. La estructura resultante se denomina *árbol ternario de búsqueda* ya que cada uno de sus nodos contiene tres apuntadores: dos apuntadores al hijo izquierdo y derecho, respectivamente, en el BST, y un apuntador a la ráiz del subárbol al que da acceso el nodo.

```
template <typename Simbolo,
          typename Clave,
          typename Valor>
class DiccDigital {
public:
   ...
   void inserta(const Clave& k, const Valor& v)
        throw(error);
   ...
private:
   struct nodo_tst {
      Simbolo _c;
      nodo_tst* _izq
      nodo_tst* _cen;
      nodo_tst* _der;
      Valor _v;
   };
   nodo_tst* raiz;
   ...
static  nodo_tst* inserta_en_tst(nodo_tst* t,
        int i, const Clave& k, const Valor& v)
        throw(error);
   ...
};
```


X = {DICCION, DADO, DADOS,
DEDO; DONDE , ABACO, CASA,
FARO, FAMA, ELLO , EL}
corresponde a un
nodo del trie
D
A
I
F
B
C
A
C
O
E
A
A
A
D
E
C
N
L
R
C
S
O
D
I
D
L
M
O
O
A
#
O
E
#
O
A
#
#
#
S
#
N
#
#
#
#
#
#
#

```
template <typename Simbolo,
          typename Clave,
          typename Valor>
void DiccDigital<Simbolo,Clave,Valor>::inserta(
     const Clave& k, const Valor& v) throw(error) {
   // Simbolo() es un simbolo nulo,
   // p.e. si Simbolo==char entonces
   // Simbolo() == '\0'
   k[k.length()] = Simbolo(); // añadir centinela
                              // al final de la clave
   raiz = inserta_en_tst(raiz, 0, k, v);
}
```

```
template <typename Simbolo,
          typename Clave,
          typename Valor>
DiccDigital<Simbolo,Clave,Valor>::nodo_tst*
DiccDigital<Simbolo,Clave,Valor>::inserta_en_tst(
          nodo_tst* t, int i,
          const Clave& k, const Valor& v)
          throw(error) {

   if (t == nullptr) {
      t = new nodo_tst;
      t -> _izq = t -> _der = t -> cen = nullptr;
      t -> _c = k[i];
      if (i < k.length() - 1) {
         t -> _cen = inserta_en_tst(t -> _cen, i + 1, k, v);
      } else { // i == k.length() - 1; k[i] == Simbolo()
         t -> _v = v;
      }
   } else {
      if (t -> _c == k[i])
         t -> _cen = inserta_en_tst(t -> _cen, i + 1, k, v);
      if (k[i] < t -> _c)
         t -> _izq = inserta_en_tst(t -> _izq, i, k, v);
      if (t -> _c < k[i])
         t -> _der = inserta_en_tst(t -> _der, i, k, v);
   }
   return t;
}
```

Parte II

Búsqueda y Ordenación Digital

Ordenación digital

La descomposición digital de las claves puede emplearse además de para la búsqueda para la ordenación. Los algoritmos de ordenación basados en estos principios se denominan de ordenación digital.

- *Counting sort*
- *Bucket sort*
- *Radix sort*

Estos algoritmos ordenan un vector de tamaño n que contienen enteros en un rango $[0..r]$, por ejemplo, con coste inferior a $\Theta(n \log n)$, a menudo $\Theta(n)$. También pueden usarse estas ideas para ordenar n strings de longitud ℓ cada uno y formados por símbolos de un cierto alfabeto finito $\Sigma = \{\sigma_1, \ldots, \sigma_U\}$.

Counting sort

Supongamos que todos los elementos de A están en un cierto rango $[0..r]$, siendo r un valor razonablemente pequeño, $r = \Theta(n)$. Entonces el algoritmo simplemente cuenta, para cada $j \in [0..r]$ cuántos j's hay en A; sea c_j dicho número. Una vez hemos hecho esto, basta generar un vector con c_0 ceros, c_1 unos, etc.
El algoritmo usará un vector auxiliar B de tamaño n para la salida y otro vector auxiliar C de tamaño r para los conteos.

Counting sort

Require: $0 \leq A[i] \leq r$ para todo i, $0 \leq i < n$
  procedure CountingSort($A[0..n-1]$, r)
    for $j := 0$ **to** r **do** ▷ $\mathcal{O}(r)$
      $C[j] := 0$
    end for
    for $i := 0$ **to** $n-1$ **do** ▷ $\mathcal{O}(n)$
      $C[A[i]] := C[A[i]] + 1;$
    end for
    ▷ $C[j] =$ número de j's en A, $0 \leq j \leq r$
    for $j := 1$ **to** r **do** ▷ $\mathcal{O}(r)$
      $C[j] := C[j] + C[j-1]$
    end for
    ▷ $C[j] =$ número de valores $\leq j$'s en A, $0 \leq j \leq r$
    for $i := n-1$ **downto** 0 **do** ▷ $\mathcal{O}(n)$
      $B[C[A[i]]] := A[i];\ C[A[i]] := C[A[i]] - 1;$
    end for
  end procedure

Counting sort

El coste de counting sort es obviamente $\Theta(n + r)$. Si $r = \mathcal{O}(n)$ entoncesel coste es $\Theta(n)$.
Dado que se necesita espacio auxiliar $\Theta(r)$ el algoritmo de counting sort es justamente interesante solo cuando $r = \mathcal{O}(n)$, de otro modo se incurre en un tiempo mayor que lineal y lo peor, espacio superlineal también.
Es importante destacar que la implementación dada hace que *counting sort* sea estable: en caso de empate, se respeta el orden original.

Bucket sort

El algoritmo de bucket sort se basa en un principio similar a *counting sort*: se usa una estructura de datos auxiliar con B cubetas (*buckets*) y los n objetos a ordenar se reparten entre las B cubetas. Por ejemplo una cubeta puede contener todos los strings que comienzan por una cierta letra. Todo objeto en la cubeta B_i ha de ser menor o igual que cualquier objeto en la cubeta B_{i+1}, con arreglo al orden que nos interesa.
Después se ordena cada cubeta utilizando un algoritmo de coste lineal, y la salida final se obtiene escribiendo en el vector de salida los contenidos (ya ordenados) de las sucesivas cubetas.

Bucket sort

El coste de la primera fase será $\Theta(n)$ asumiendo que determinar y colocar cada objeto $A[i]$ en la cubeta B_j que le corresponde se hace en tiempo constante.
Si b_j denota el número de objetos en la cubeta B_j, ordenar los objetos de b_j tendrá coste $\Theta(b_j)$. Y añadir los contenidos ordenados en la salida también es $\Theta(b_j)$.
En total:

$$\Theta(n) + \sum_{j=1}^{B} \Theta(b_j) = \Theta(n) + \Theta\left(\sum_{j=1}^{B} b_j\right) = \Theta(n).$$

LSD Radix sort

Consideremos un vector de n elementos cada uno de los cuales es una secuencia de ℓ símbolos.
Si ordenamos repetidamente el vector con arreglo a los sucesivos símbolos de derecha a izquierda (en caso de ser bits diríamos del bit de menor peso al bit de mayor peso) mediante un algoritmo de ordenación estable de coste lineal entonces el vector queda ordenado en tiempo $\Theta(n \cdot \ell)$.
Este el algoritmo de LSD radix sort (LSD=Least Significant Digit).

LSD Radix sort

procedure LSDRADIXSORT(A, ℓ)
    for $i := 0$ **to** $\ell - 1$ **do**
        Ordenar A con un *sort* estable de coste lineal, de acuerdo los i-ésimos símbolos
    end for
end procedure

LSD Radix sort

Ejemplo

$\ell = 3$

329		720		720		329
475		475		329		355
657		355		436		436
839	$\Longrightarrow$	436	$\Longrightarrow$	838	$\Longrightarrow$	475
436		657		355		657
720		329		657		720
355		839		475		839

LSD Radix sort

Teorema

LSD Radix sort ordena correctamente el vector de n elementos.

Demostración

Por inducción sobre ℓ.
Base: Si $\ell = 1$ se ordenan los n elementos de A con arreglo a su único símbolo.
Hipótesis de inducción: Los n elementos están correctamente ordenados si nos fijamos exclusivamente en sus últimos $\ell - 1$ símbolos. Sean $A[i]$ y $A[j]$ dos elementos cualesquiera del vector tras ordenar respecto a $\ell - 1$ símbolos.

LSD Radix sort

Demostración (continúa)

Si ordenamos con ordenación estable respecto al símbolo ℓ-ésimo (de mayor peso) y $A[i][\ell] = A[j][\ell]$ entonces la ordenación estable mantendrá el orden que tuvieran $A[i]$ y $A[j]$ ya que empatan respecto al símbolo ℓ-ésimo, lo cual es correcto.
Si $A[i][\ell] < A[j][\ell]$ entonces es seguro que $A[i] < A[j]$ y el algoritmo de ordenación estable sobre el símbolo ℓ-ésimo pondrá $A[i]$ precediendo a $A[j]$. □

LSD Radix sort

Supongamos que los elementos del vector son enteros y los "símbolos" que componen los elementos son los dígitos del elemento en una cierta base b, es decir, los símbolos son números en $\{0, \ldots, b-1\}$. Entonces si usamos *counting sort* cada iteración cuesta $\Theta(n+b)$ y el coste total de LSD radix sort es $R(n, b, \ell) = \Theta((n+b) \cdot \ell)$.

- Supongamos que cada entero en el vector es un número positivo $< f(n)$.
- Entonces $\ell = \lceil \log_b(f(n)) \rceil$ y el coste de LSD radix sort es $R(n, \ell, b) = \Theta((n+b) \log f(n))$.
- Si $f(n) = \omega(1)$ entonces el coste es $R(n, \ell, b) = \omega(n)$, mayor que lineal.

Podemos mejorar la eficiencia? Sí: cambiando la base b.

LSD Radix sort

Por ejemplo, si en vez de usar $b = 2$ tomamos $b' = 2^r$ entonces $\ell' = \log_{2^r} f(n) = \frac{\log_2 f(n)}{r}$ y el coste del LSD radix sort se rebaja en un factor $1/r$. En vez de trabajar con símbolos que son bits, nuestros símbolos pasan a ser bloques de r bits.
En general, pasaremos de ℓ bits a $\ell' = \lceil \ell/r \rceil$ dígitos en base 2^r. Entonces el coste de LSD radix sort es $\Theta((n + 2^r)\ell/r)$; tomando $r = c \log_2 n$ para alguna $c < 1$ conseguimos que el coste sea $\Theta(n)$

LSD Radix sort

No obstante, se necesita especial atención para mantener el coste de counting sort lineal, ya que en la nueva situación estaremos trabajando con números de $\Theta(\log n)$ bits y no podemos asumir que mover o copiar números de ese tamaño nos tome tiempo constante.

MSD Radix sort

Consideremos ahora un vector de n elementos cada uno de los cuales es una secuencia de ℓ bits. Dado un elemento x, $\texttt{bit}(x, i)$ denotará su i-ésimo bit.
Ordenamos el vector con relación al bit de mayor peso y obtenemos dos bloques. Los elementos cuyo bit de mayor peso es 0 y a continuación los elementos con bit de mayor peso 1.
Los dos bloques se ordenan separada y recursivamente respecto al bit de segundo mayor peso, y así sucesivamente.
Este es el algoritmo de MSD Radix sort (MSD=Most Significant Digit).

MSD Radix sort

Puede generalizarse fácilmente a elementos formados por ℓ símbolos cada uno, donde cada símbolo es un dígito entre 0 y $b - 1$ (o puede verse de esta forma, por ejemplo, letras con códigos ASCII de 0 a 255); cada etapa no recursiva subdivide el bloque en curso en $\leq b$ bloques, cada bloque correspondiendo a los elementos cuyo i-ésimo bit es un cierto dígito/símbolo.

MSD Radix sort

```
// Llamada inicial:
// radix_sort(A, 0, A.size()-1, ℓ − 1);
template <typename Elem, typename Symb>
void radix_sort(vector<Elem>& A, int i, int j, int r) {

   if (i < j and r >= 0) {
      int k;
      radix_split(A, i, j, r, k);
      radix_sort(A, i, k, r - 1);
      radix_sort(A, k + 1, j, r - 1);
   }
}
```

MSD Radix sort

```
// bit(x, r) devuelve el bit r-ésimo de x
// r == 0 => bit de menor peso
// r == ℓ − 1 => bit de mayor peso
template <typename Elem, typename Symb>
void radix_split(vector<Elem>& A, int i, int j,
                 int r, int& k) {

   int u = i; int v = j;
   while (u < v + 1) {
      while (u < v + 1 and bit(A[u],r) == 0) ++u;
      while (u < v + 1 and bit(A[v],r) == 1) --v;
      if (u < v + 1) swap(A[u], A[v]);
   }
   k = v;
}
```

MSD Radix sort

Cada etapa de *radix sort* tiene un coste no recursivo lineal; puesto que la profundidad del árbol de recursión es ℓ, el coste del algoritmo es $\Theta(n \cdot \ell)$. Otra forma de deducir el coste consiste en considerar el coste asociado a cada elemento de A: un elemento cualquiera de A es examinado (y eventualmente intercambiado con otro) a lo sumo ℓ veces, de ahí que el coste total sea $\Theta(n \cdot \ell)$.

MSD Radix sort

MSD radix sort está en correspondencia con los tries de forma análoga a cómo quicksort está en correspondencia con los árboles binarios de búsqueda.

En el análisis de MSD radix sort podemos aplicar consideraciones semejantes a las que hacíamos para LSD radix sort: cambiando la base b haremos hasta b bloques con cada `radix_split` y la profundidad de la recursión se reducirá en un factor $1/r$.

MSD Radix sort

El algoritmo que hace la partición en bloques `radix_split` **no** respeta el orden original y en consecuencia MSD radix sort no es estable, a diferencia de lo que sucede con LSD radix sort.

Por otro lado si aumentamos la base b (=número de símbolos distintos posibles) se nos complica enormemente el algoritmo de partición a no ser que se use espacio extra auxiliar—y una de las ventajas de MSD radix sort es no necesitar espacio auxiliar!

Quicksort for Strings

Una solución intermedia entre quicksort y MSD radix sort que funciona de manera muy eficiente en la práctica para ordenar n strings se inspira en las ideas que fundamentan los TSTs.

```
// Cada string está acabado con un carácter end-of-string
// menor que cualquier otro carácter.
void qsort(vector<string>& A, int i, int j, int r) {
   if (i < j) {
     int p,q;
     char c = A[i][r]; // usamos la letra r-ésima de A[i]
                       // como pivote
     3-way-partition(A, c, i, j, p, q);
     // A[i..p-1] = elementos cuya r-ésima letra es < c
     // A[p..q] = elementos cuya r-ésima letra es = c
     // A[q+1..j] = elementos cuya r-ésmia letra es > c
     qsort(A, i, p-1, r);
     qsort(A, p, q  , r+1);
     qsort(A, q+1, j, r);
   }
}
```