

Estructuras de Datos y Algoritmos

Árboles lexicográficos (o *tries*)

LECCIÓN 17

© All wrongs reversed – bajo licencia CC-BY-NC-SA 4.0



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA

Aula 0.04, Edificio Agustín de Betancourt

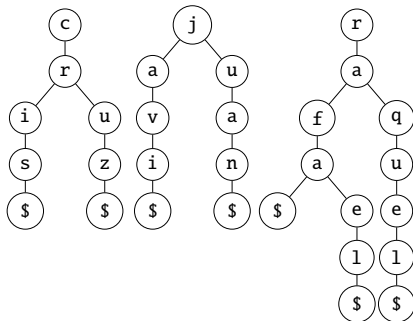


Índice

Índice

Árboles lexicográficos

- Un *trie* es una estructura de datos arborescente que utiliza las partes de la clave para organizar y buscar entre la colección de datos que almacena (TAD diccionario)
 - El nombre *trie* proviene de *retrieval*, y se pronuncia como *try* para distinguirlo de *tree*
 - Se utilizan con claves de tipo cadena, pero también pueden usarse con claves numéricas u otras (en general, cadenas de símbolos de un alfabeto finito)
 - La clave no se almacena entera en un nodo, sino en un camino (normalmente de la raíz a cada hoja)



cris cruz javi juan rafa rafael raquel

Árboles lexicográficos

- Todos los descendientes de un nodo tienen un prefijo común
 - Un *trie* es un **árbol de prefijos**
- Si el alfabeto utilizado tiene definida una relación de orden, el *trie* se llama *árbol lexicográfico*
- La búsqueda en un *trie*, para una clave de longitud m , es de $O(m)$ en el peor caso
- Suelen requerir menos espacio que otros árboles, al no almacenar las claves en los nodos y compartirlas para representar las claves con prefijos comunes
- Los datos asociados a cada clave se localizan al final del camino que forma la clave (hoja)

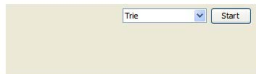
Árboles lexicográficos

- Véase ejemplo de applet en:

<http://people.cis.ksu.edu/~rhowell/viewer/viewer.html>

Search Tree Viewer

Below is an applet for creating and manipulating Binary Search Trees. To start, click the Start button. This will open a window in which the tree



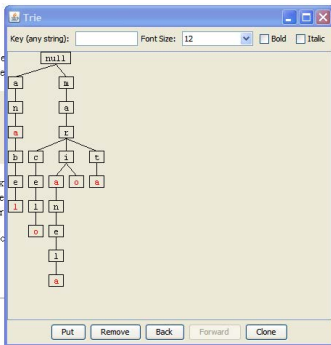
To create and manipulate a tree, enter any string in the text box. The keys will be maintained in lexicographic order. A copy of the tree and history in your current window; the tree

Important Note: This applet treats all keys as strings. You may have reflected a misunderstanding of this fact.

- [More information.](#)

Last updated January 12, 2006.

Rod Howell (rhowell@ksu.edu)



- Otro applet en

http://people.ksp.sk/~kuko/gnarley-trees/?page_id=100

(ir a [Stringology](#) >> [trie](#))

Árboles lexicográficos

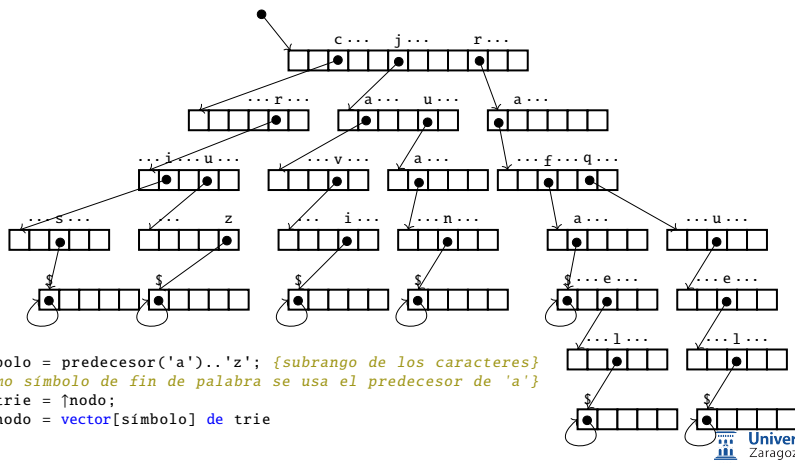
- Para poder almacenar palabras (claves) que sean prefijos de otras, es necesario utilizar un símbolo terminador (ejemplo: carácter '\$')
- Aplicaciones más comunes:
 - Manipulación de diccionarios (búsqueda, inserción, borrado, etcétera; TAD diccionario)
 - Búsqueda de prefijos de palabras o, más general, búsqueda y comparación de subcadenas: procesamiento de textos, completado automático de comandos, etcétera
 - ...

Índice

Implementaciones

Nodo-vector

- Cada nodo es un vector de punteros para acceder a los subárboles
- Rápida selección de hijo versus alto coste en espacio



tipos

```
símbolo = predecesor('a')..'z'; {subrango de los caracteres}  
{como símbolo de fin de palabra se usa el predecesor de 'a'}  
trie = ↑nodo;  
nodo = vector[símbolo] de trie
```


Implementaciones

Nodo-abb

- Esta 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 de los *tries* con la eficiencia de los ABBs (al usarlos en cada “nodo” del *trie*):
 - 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

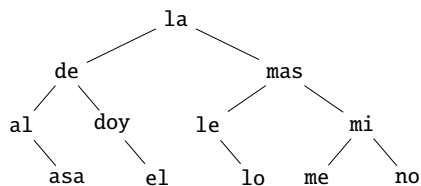
Implementaciones

Nodo-abb

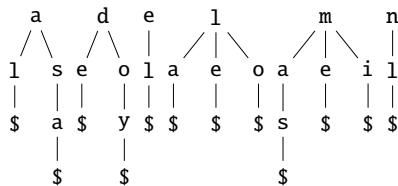
Un árbol ternario de búsqueda para las palabras:

al, asa, de, doy, el, la, le, lo, mas, me, mi, no

En un ABB podrían quedar:



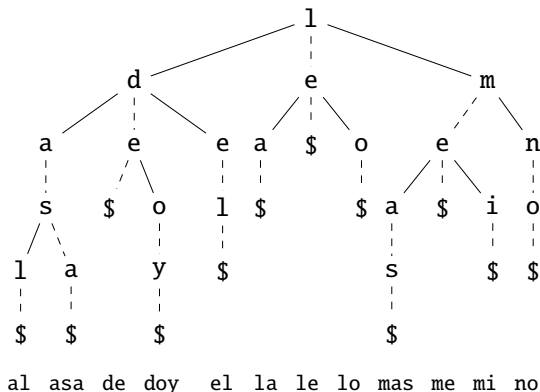
En un *trie* quedarían:



Implementaciones

Nodo-abb

Almacenando el *trie* como árbol ternario de búsqueda podrían quedar:



Implementaciones

Detalles implementación Nodo-lista

tipos

```
trie = ↑nodo;  
nodo = registro  
    dato: carácter;  
    primogenito, sigHermano: trie  
freg
```

procedimiento creaVacio(sal t: trie)

principio

```
t := nil
```

fin

procedimiento plantaPalabra(ent palabra: cadena; e/s t: trie)

{Algoritmo auxiliar para plantar un árbol vertical (lista) con los caracteres de una palabra}

variables

```
taux: trie;  
resto: cadena
```

principio

```
si long(palabra) = 0 entonces {long devuelve la longitud de una cadena}
```

```
nuevoDato(t);  
t↑.dato := '$'; {marca de fin de palabra}  
t↑.primogenito := nil;  
t↑.sigHermano := nil
```

sino

```
resto := palabra[2..long(palabra)]; {trozo de la palabra...}  
plantaPalabra(resto,taux);  
nuevoDato(t);  
t↑.dato := palabra[1]; {primer carácter de la palabra}  
t↑.primogenito := taux;  
t↑.sigHermano := nil
```

fsi

fin

```

procedimiento insertar(ent palabra: cadena; e/s t: trie)
variables
    taux: trie;
    resto: cadena
principio
    si t = nil entonces
        plantaPalabra(palabra, t);
    sino
        si long(palabra) = 0 entonces
            si t↑.dato ≠ '$' entonces
                nuevoDato(taux);
                taux↑.dato := '$';
                taux↑.primogenito := nil;
                taux↑.sigHermano := t;
                t := taux
            fsi
        sino {t ≠ nil and long(palabra) > 0}
            si palabra[1] < t↑.dato entonces
                plantaPalabra(palabra, taux);
                taux↑.sigHermano := t;
                t := taux
            sino_si palabra[1] = t↑.dato entonces
                resto := palabra[2..long(palabra)];
                insertar(resto, t↑.primogenito)
            sino {palabra[1] > t↑.dato}
                insertar(palabra, t↑.sigHermano)
            fsi
        fsi
    fsi
fin

```



```

función pertenece(palabra: cadena; t: trie) devuelve booleano
variable
    resto: cadena
principio
    si t = nil entonces
        devuelve falso
    sino
        si long(palabra) = 0 entonces
            devuelve t↑.dato = '$'
        sino
            si palabra[1] < t↑.dato entonces
                devuelve falso
            sino_si palabra[1] = t↑.dato entonces
                resto := palabra[2..long(palabra)];
                devuelve pertenece(resto, t↑.primogenito)
            sino {palabra[1] > t↑.dato}
                devuelve pertenece(palabra, t↑.sigHermano)
        fsi
    fsi
    fsi
fin

```

```

procedimiento eliminar(ent palabra: cadena; e/s t: trie)
variables
    taux: trie;
    resto: cadena
principio
    si t ≠ nil entonces
        si long(palabra) = 0 entonces
            si t↑.dato = '$' entonces
                taux := t;
                t := t↑.sigHermano;
                disponer(taux)
            fsi
        sino
            si palabra[1] = t↑.dato entonces
                resto := palabra[2..long(palabra)];
                eliminar(resto, t↑.primogenito);
                si t↑.primogenito = nil entonces
                    taux := t;
                    t := t↑.sigHermano;
                    disponer(taux)
                fsi
            sino_si palabra[1] > t↑.dato entonces
                eliminar(palabra, t↑.sigHermano)
            fsi
        fsi
    fsi
fin

```

```

procedimiento preTrie(ent t: trie; ent palabra: cadena)
  {Adaptación del recorrido en pre-orden para un bosque de árboles n-arios visto
  en la lección 15. En el parámetro 'palabra' se recibe la concatenación de caracteres
  del camino que va de la raíz del trie al padre del nodo apuntado por t}
principio
  si t ≠ nil entonces
    preOrden(t, palabra); {recorrido del árbol apuntado por t}
    preTrie(t↑.sigHermano, palabra) {recorrido de los demás árboles hermanos de t}
  fsi
fin

procedimiento preOrden(ent t: trie; ent palabra: cadena)
  {Adaptación del recorrido en pre-orden para un árbol n-ario visto en la lección 15}
principio
  si t↑.dato = '$' entonces
    escribirLínea(palabra)
  sino
    preTrie(t↑.primogenito, palabra + t↑.dato) {+ es aquí la concatenación de cadenas}
  fsi
fin

procedimiento escribe(ent t: trie)
variable
  palabra: cadena
principio {de escribe}
  palabra := ""; {"" representa la cadena vacía}
  preTrie(t, palabra)
fin

```



Estructuras de Datos y Algoritmos

Árboles lexicográficos (o *tries*)

LECCIÓN 17

© All wrongs reversed – bajo licencia CC-BY-NC-SA 4.0



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática
UNIVERSIDAD DE ZARAGOZA

Aula 0.04, Edificio Agustín de Betancourt

