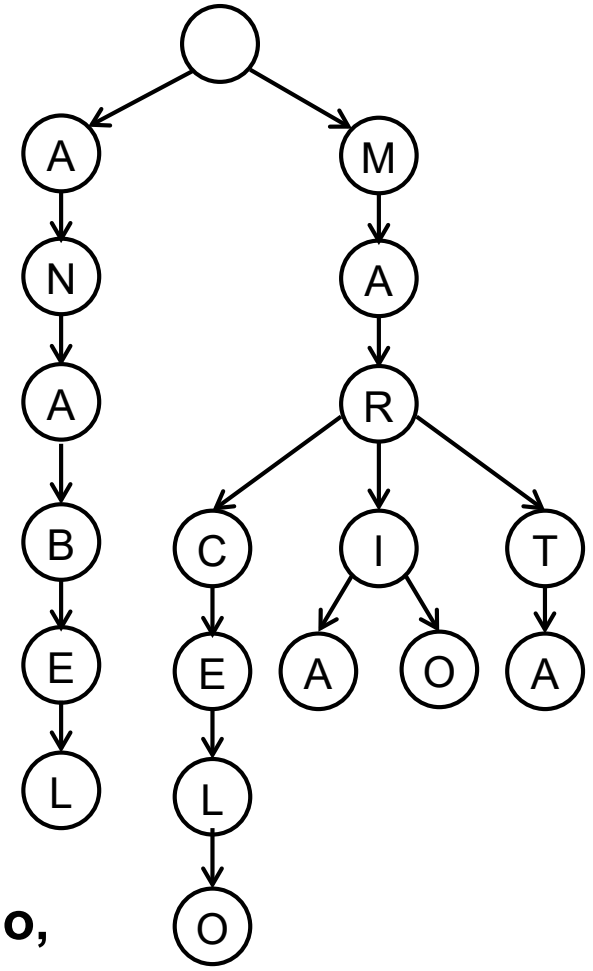
Árboles Lexicográficos

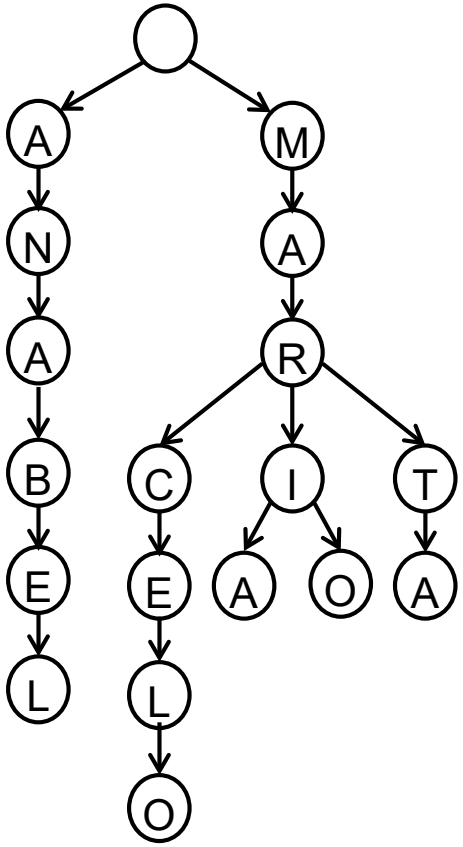
Lección 17

Tries

- 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 o similares*)
  - El nombre *trie* proviene de "retrieval", y se pronuncia como "try" para distinguirlo de "tree"
  - Normalmente los *trie* se utilizan con claves de tipo *string* o cadena, pero también pueden usarse con claves numéricas, alfanuméricas, etc.
  - **La clave no se almacena entera en un nodo, sino en un camino**



Tries



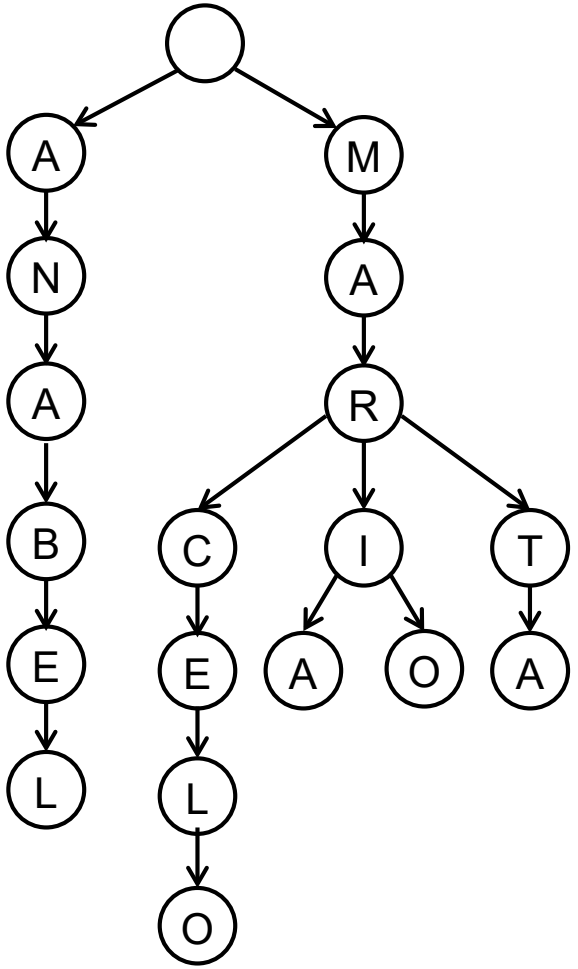
- Un **trie** es una estructura de datos arbórea que utiliza las partes de la clave para organizar y buscar entre su colección de datos (TAD diccionario)
  - ....
  - **La clave no se almacena entera en un nodo, sino en un camino**
    - Cada nodo del camino tiene una **parte** de la clave
      - La **raíz** del trie suele asociarse con la *cadena vacía*
    - El conjunto de valores posibles que puede tomar cada parte de la clave forma un **alfabeto** finito (suele ser numérico, alfanumérico, etc.)
      - En general las claves serán secuencias (cadenas) de símbolos de dicho alfabeto finito
    - **Los datos asociados a una clave** suelen localizarse al final del camino que forma la clave (que suelen ser las hojas)

Tries

- Aplicaciones comunes:
  - Manipulación de diccionarios (inserción, borrado, búsqueda, etc. … TAD diccionario)
  - Búsqueda de palabras, o de prefijos comunes, búsqueda y comparación de subcadenas: procesamiento de textos, corrección o sugerencia de palabras, completado automático de comandos, etc…
  - Búsquedas en BDs, enrutamiento por IPs, o similares…
  - …

Tries

- Todos los descendientes de un nodo tienen un **prefijo común** *(trie* es un ***árbol de prefijos****)*
- Si el alfabeto utilizado tiene definida una **relación de orden**, el *trie* se llama ***árbol lexicográfico***
- Suelen requerir menos espacio que otros árboles, al no almacenar las claves en los nodos y compartirlos para representar las claves con prefijos comunes
- La búsqueda en un trie, para una clave de longitud *m*, es de $\Theta(m)$ en el *peor caso* *(si no está, se suele detectar antes)*
  - *Análisis teóricos (Knuth) lo aproximan a Θ(log N) siendo N el número total de claves que se pueden construir, y la base del algoritmo es el tamaño del alfabeto que define para las partes de la clave…*

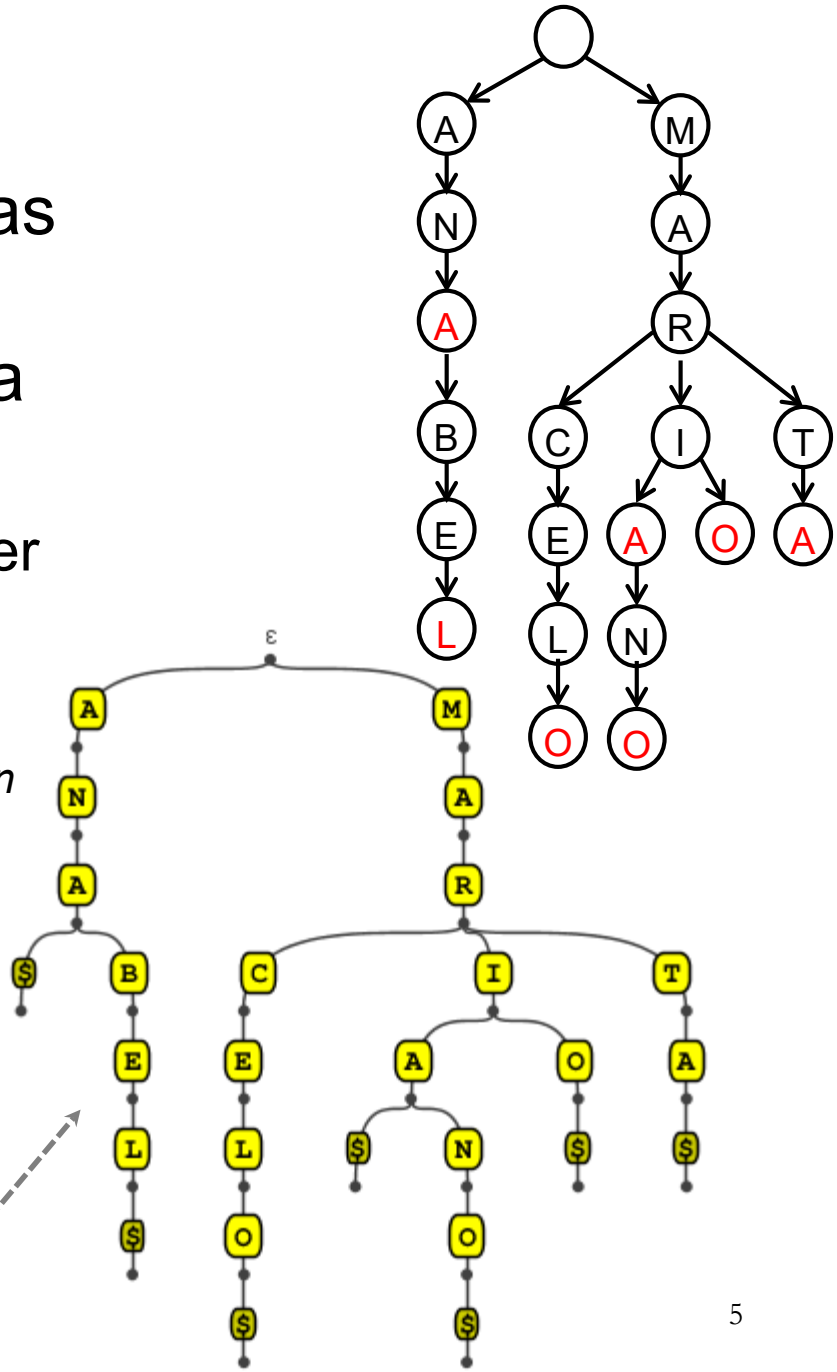


Tries

- Para poder almacenar palabras (claves) que sean prefijos de otras, es necesario utilizar una señal de terminación
  - Por ejemplo utilizando un carácter especial: '$', predecesor('a'), …

→*Si hijos ordenados (árbol lexicográfico):*
recorrido en preorden → claves en orden

- En general, desde el nodo con el último trozo de la clave, o desde el nodo con la marca de finalización, se podrá acceder a los datos que acompañan a dicha la clave

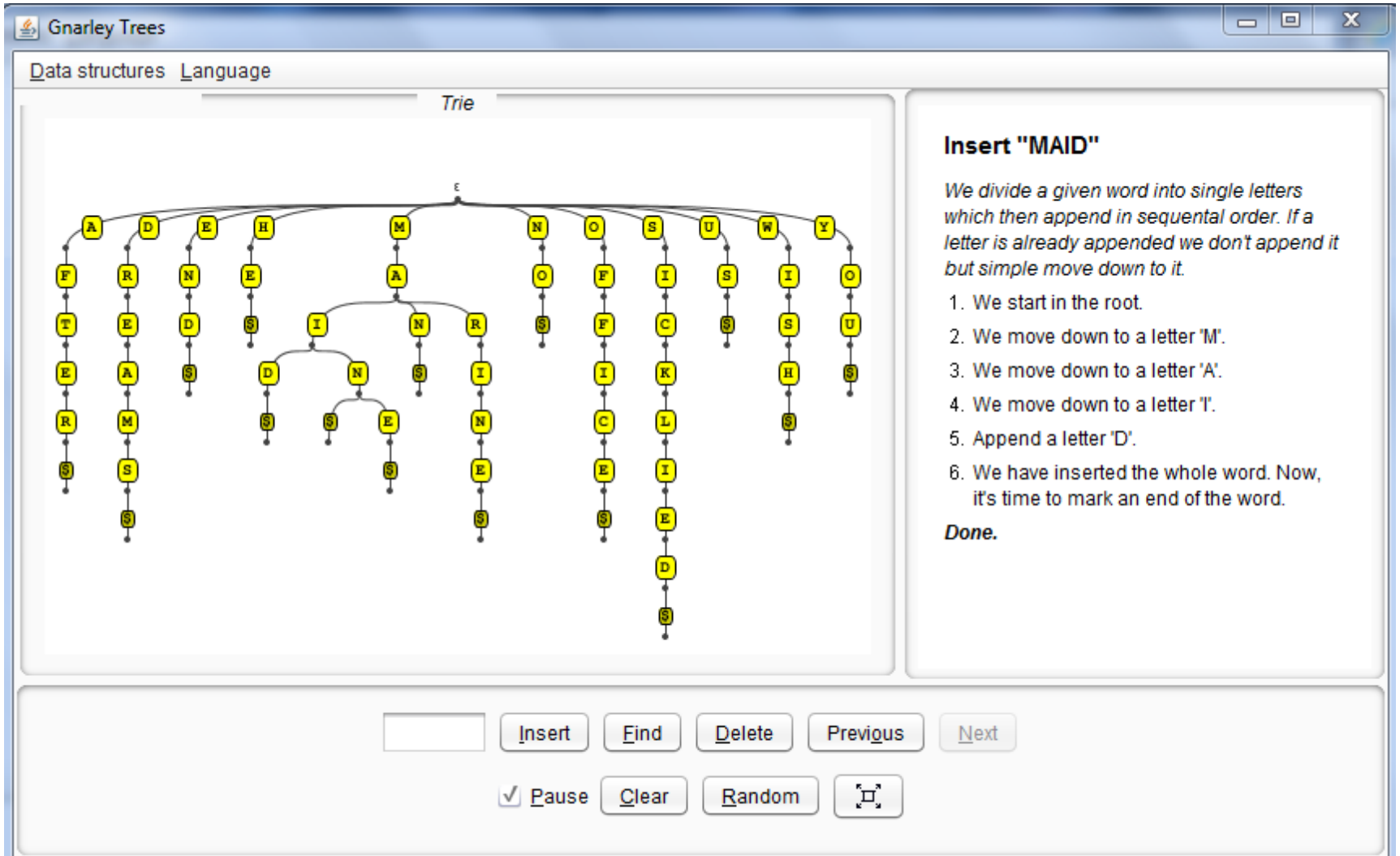


Esta imagen es un recorte de pantalla de una ejecución de la animación que se presenta en la siguiente transparencia

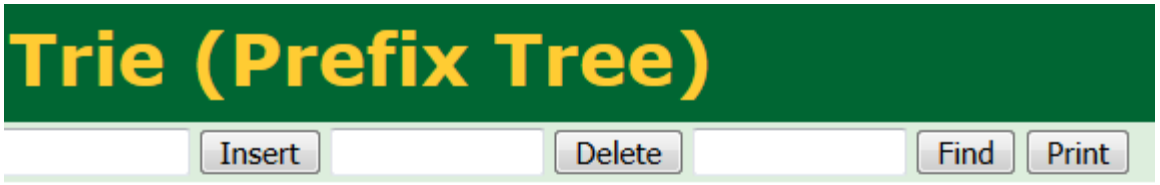
Tries

Ejemplo de animación en: https://kubokovac.eu/gnarley-trees/Trie.html

(Data Structures> Stringology > Trie)

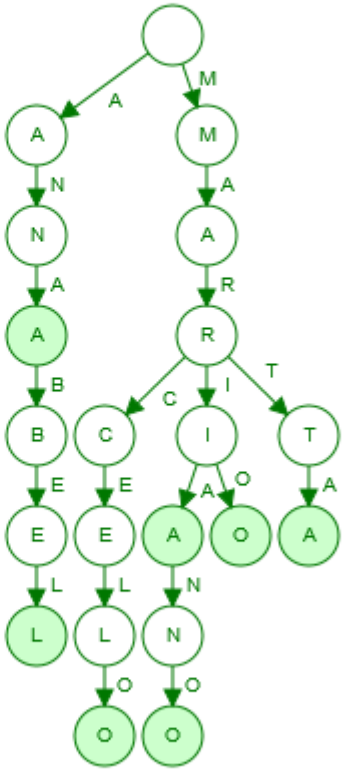


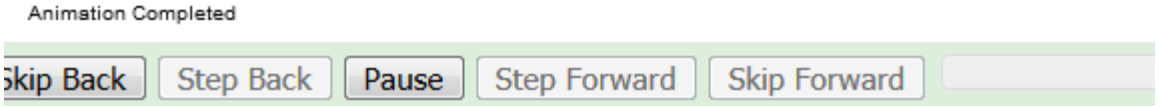
Tries



Ver ejemplo de animación en:

https://www.cs.usfca.edu/~galles/visualization/Trie.html



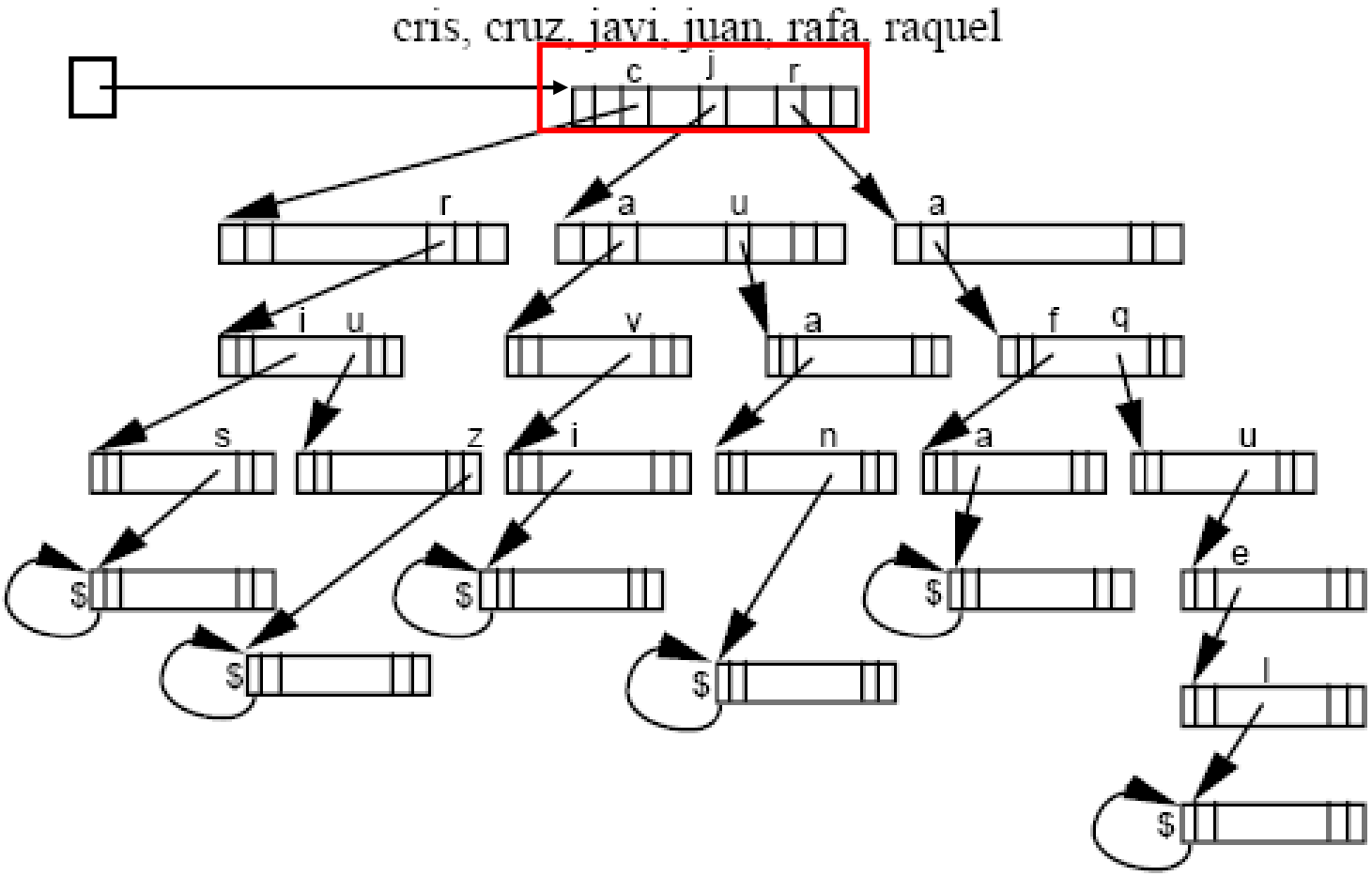


Implementaciones: Nodo-vector

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

  El alfabeto se refleja en el tipo índice para el vector

  – Rápida selección de hijo versus alto coste en espacio



Implementaciones: Nodo-vector

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

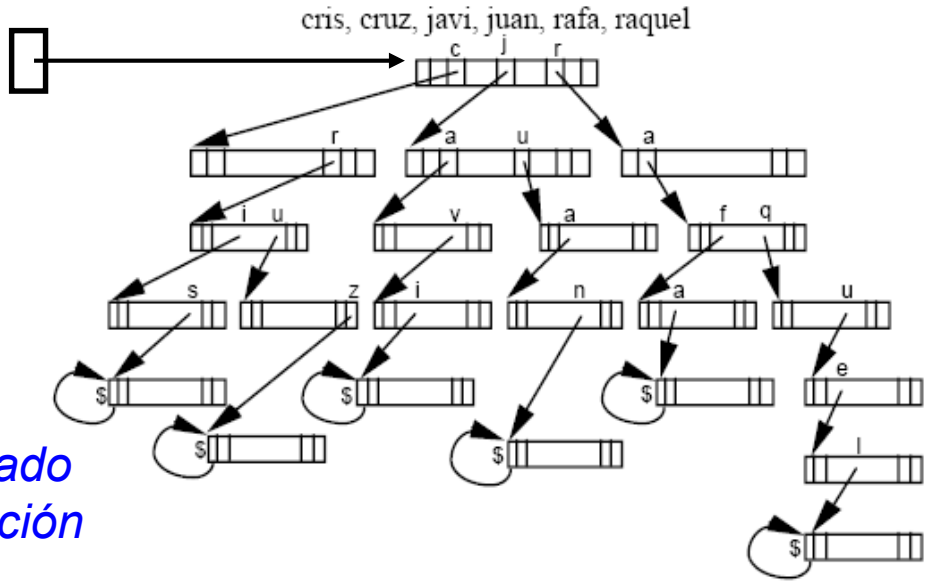
tipos

símbolo **=** predecesor('a') ..'z'; *{subrango de los caracteres}*

{como símbolo de fin de palabra se usará el predecesor de 'a'}

trie = ↑nodo;

nodo = **vector**[símbolo] **de** trie

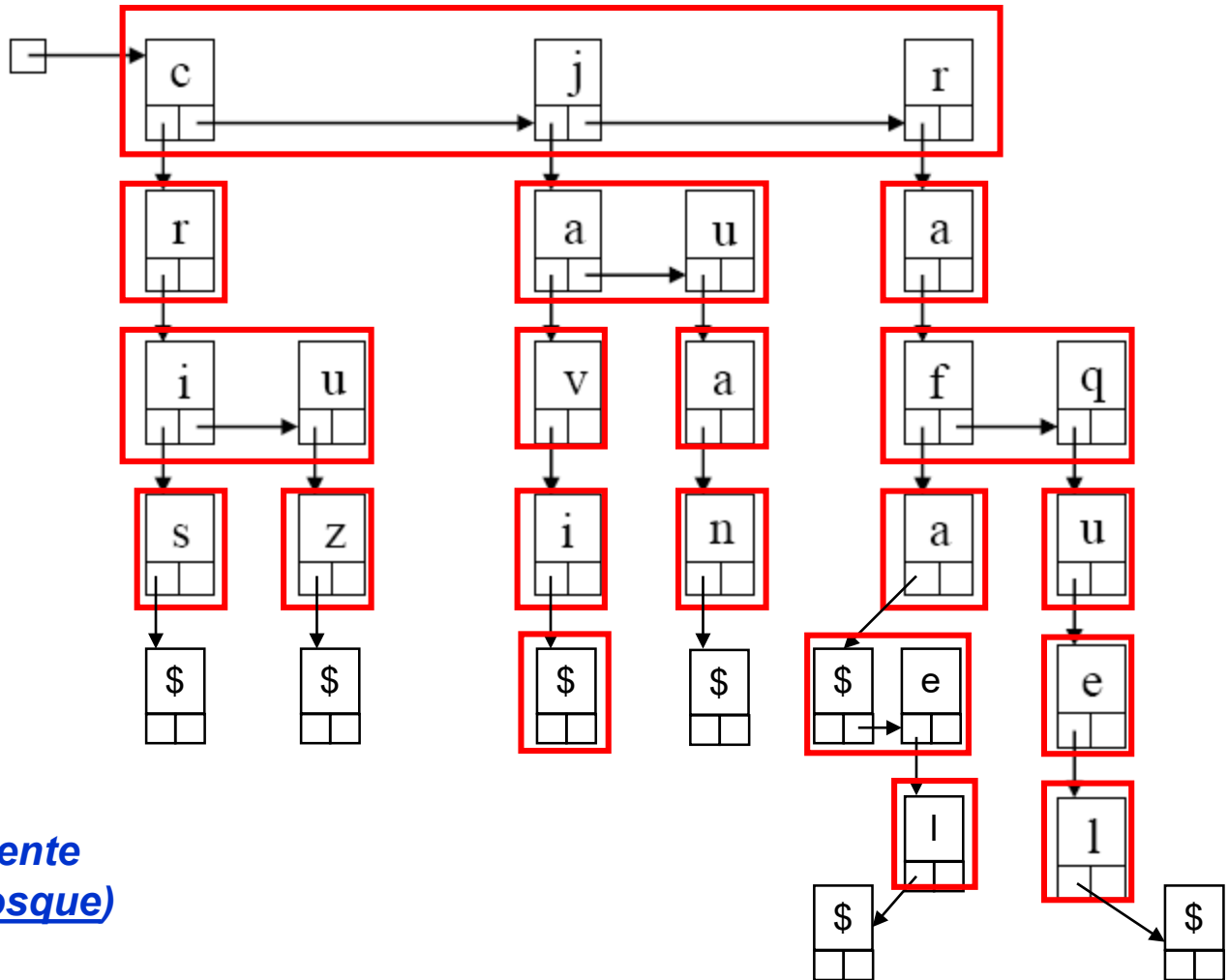


predecesor('a') será usado como marca de terminación

Implementaciones: Nodo-lista

- **Nodo-lista:** cada nodo es una lista enlazada por punteros, que contiene las raíces de los subárboles (tries)
  - Menor coste en espacio, pero mayor tiempo para acceder a los hijos

cris, cruz, javi, juan, rafa, rafael, raquel



(Representación primogénito-siguiente hermano de un bosque)

Implementaciones: Nodo-abb

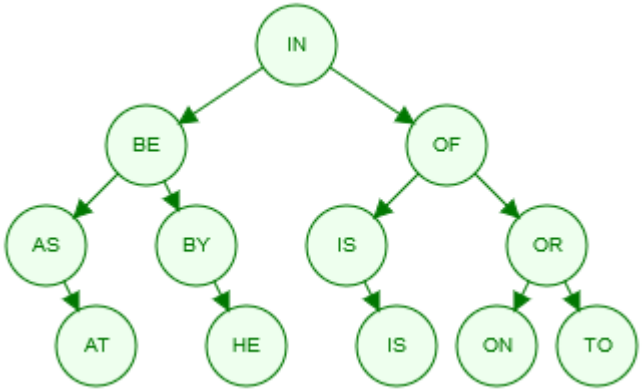
- **Nodo-abb**: la estructura se llama también ***árbol ternario de búsqueda***.
  - Cada nodo contiene:
    - Dos punteros, al hijo izquierdo y derecho (como en un árbol binario de búsqueda).
    - Un puntero, central, a la raíz del trie al que da acceso el nodo.
  - Objetivo: combinar la eficiencia en espacio de los tries, con la eficiencia en búsqueda de los *abb*'s (*al usarlos en cada nivel de una rama del trie*):
    - Una búsqueda compara el carácter actual en la cadena (clave) 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

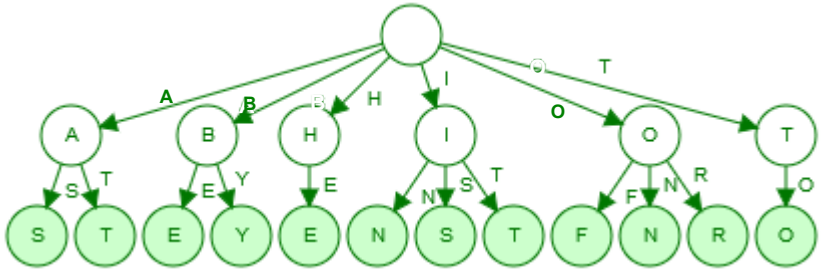
Árbol ternario de búsqueda para las palabras:

AS AT BE BY HE IN IS IT OF ON OR TO

- En un *abb* quedarían:



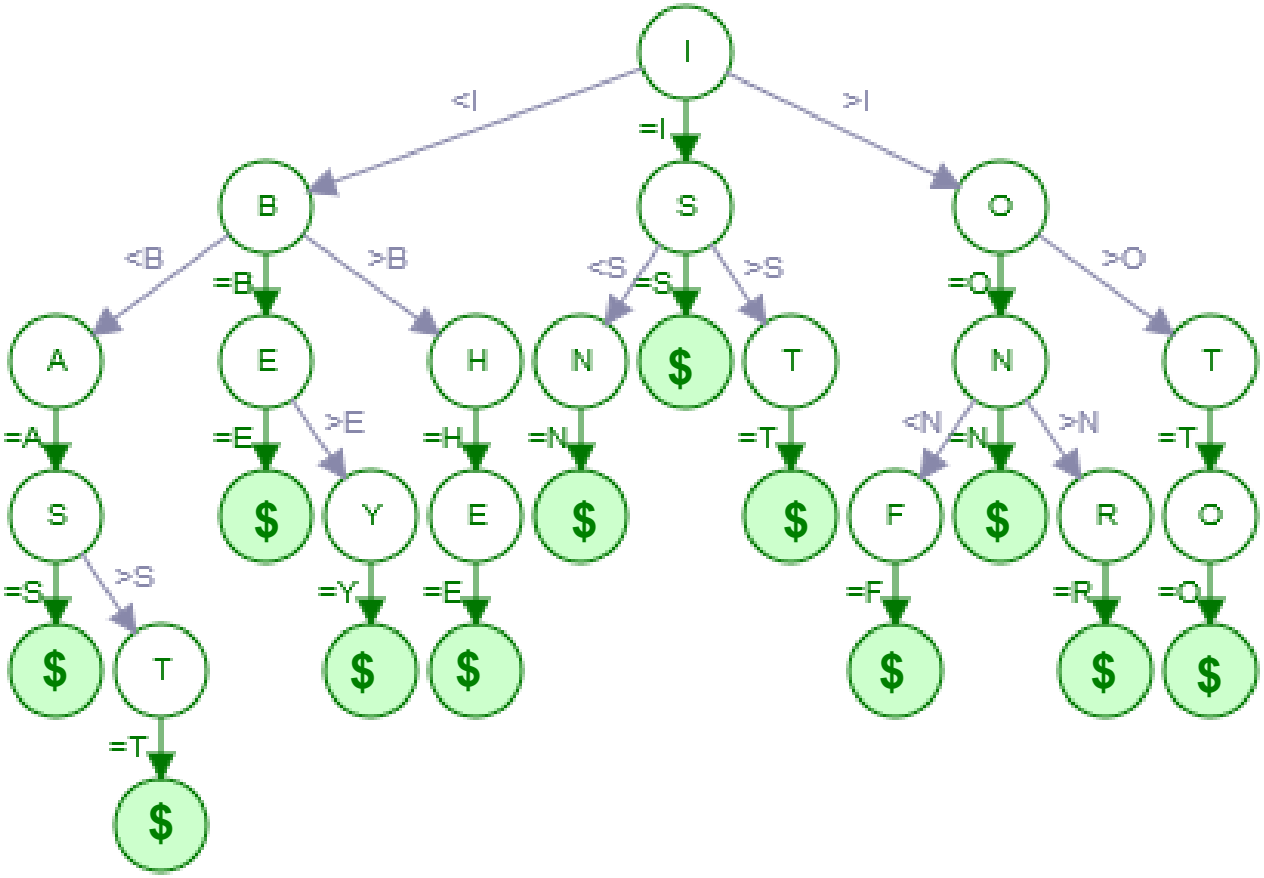
- En un *trie* quedarían como:



Estas imágenes son recortes de pantalla de ejecuciones de las animaciones disponibles respectivamente en:
https://www.cs.usfca.edu/~galles/visualization/BST.html
https://www.cs.usfca.edu/~galles/visualization/Trie.html

Implementaciones: Nodo-abb

- Como árbol ternario de búsqueda quedan:

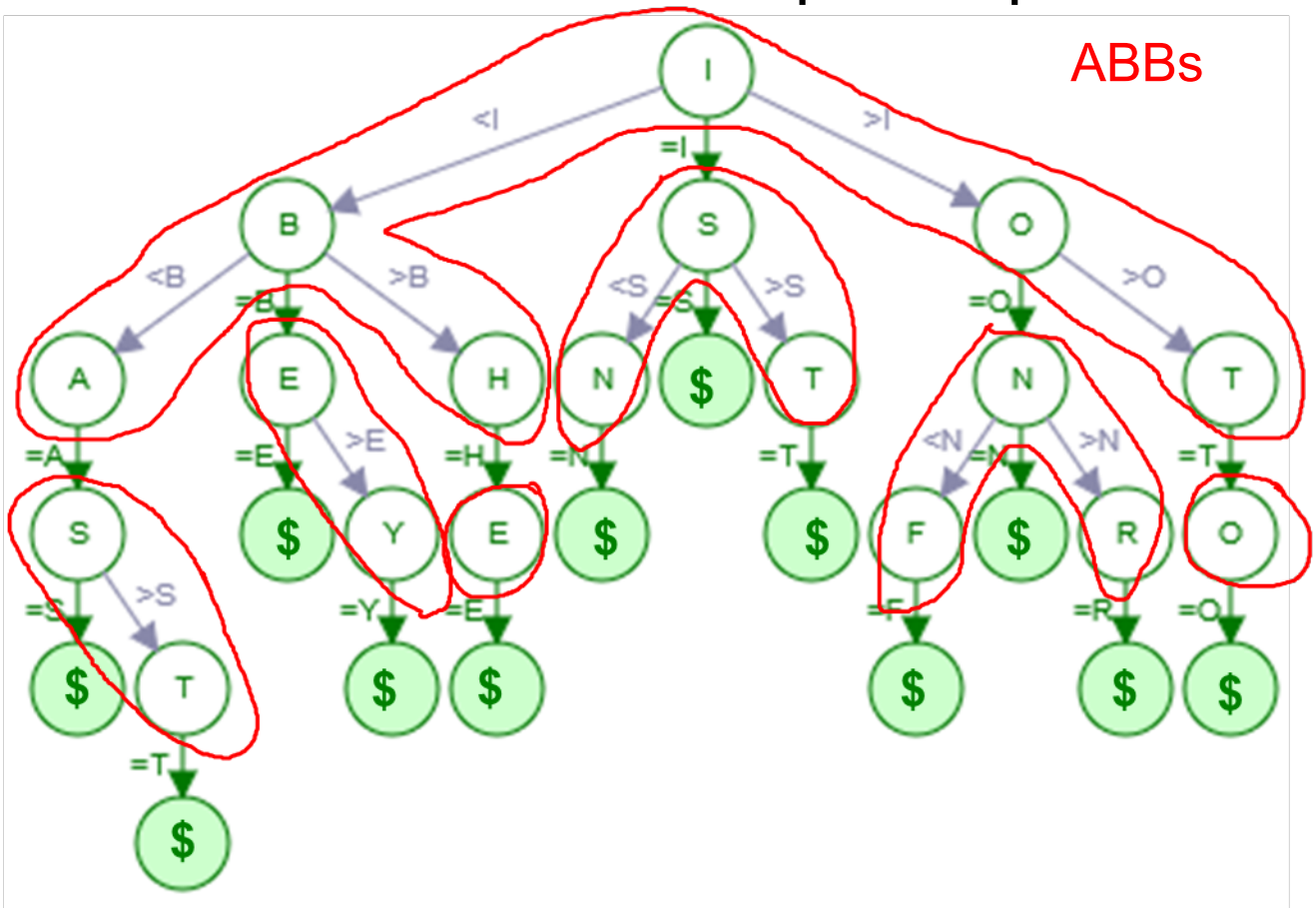


AS AT BE BY HE IN IS IT OF ON OR TO

Esta imagen es un recorte de pantalla de una ejecución de las animación disponible en:
https://www.cs.usfca.edu/~galles/visualization/TST.html
Que se ha retocado para incluir el $ en los nodos verdes que marcan el final de las palabras

Implementaciones: Nodo-abb

- Como árbol ternario de búsqueda quedan:

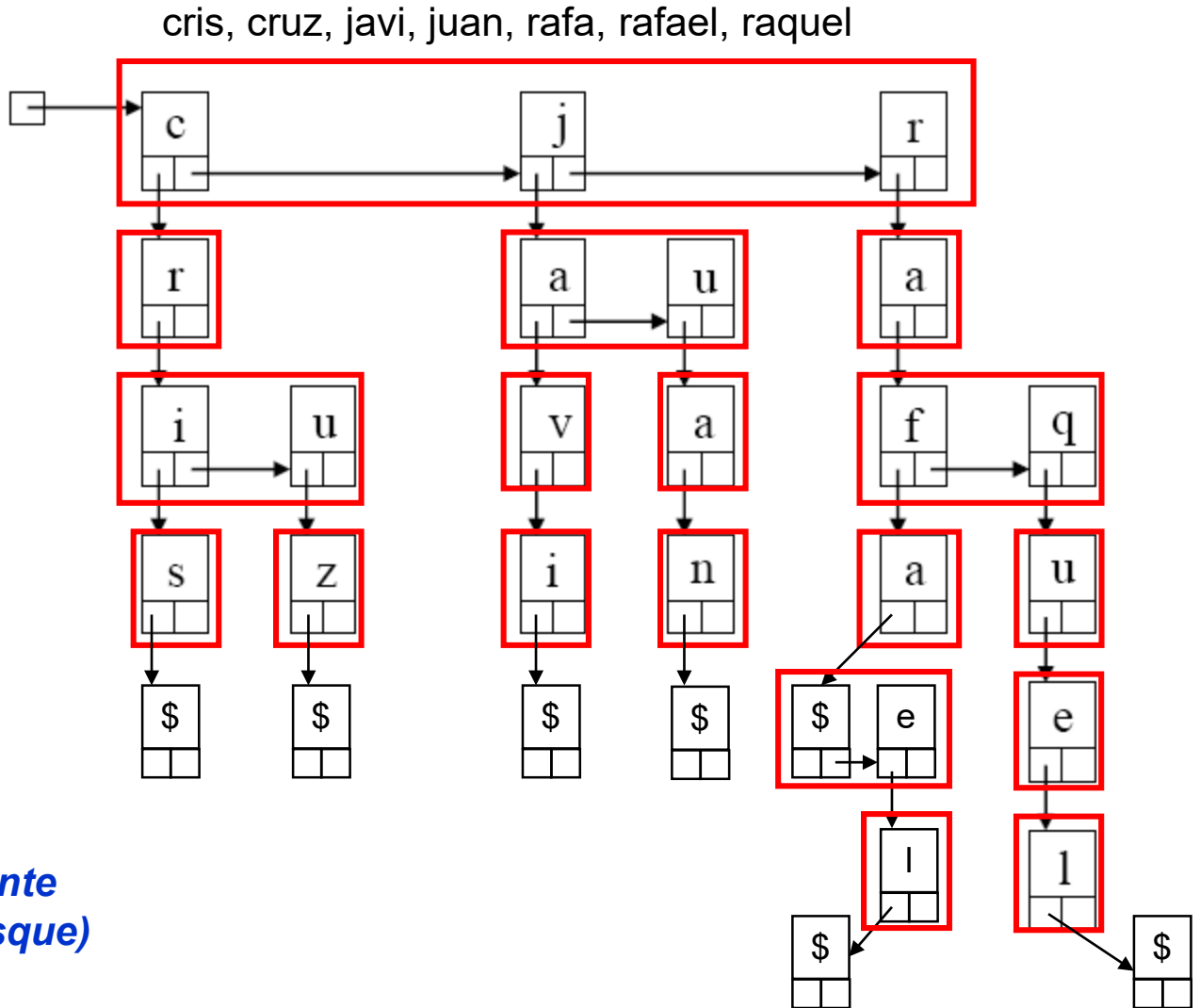


AS AT BE BY HE IN IS IT OF ON OR TO

Esta imagen es un recorte de pantalla de una ejecución de las animación disponible en:
https://www.cs.usfca.edu/~galles/visualization/TST.html
Que se ha retocado para incluir el $ en los nodos verdes que marcan el final de las palabras

Detalles implementación Nodo-lista

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



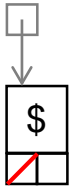
(Representación primogénito-siguiente hermano de un bosque)

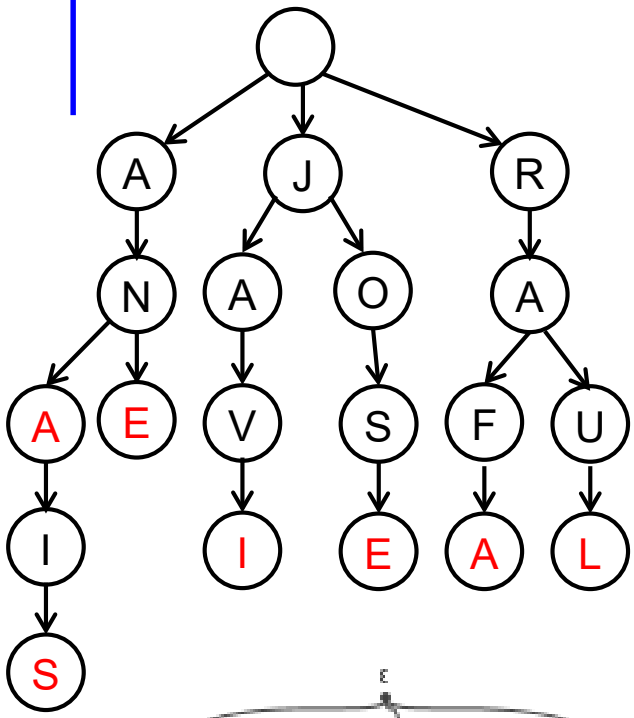
Detalles implementación Nodo-lista

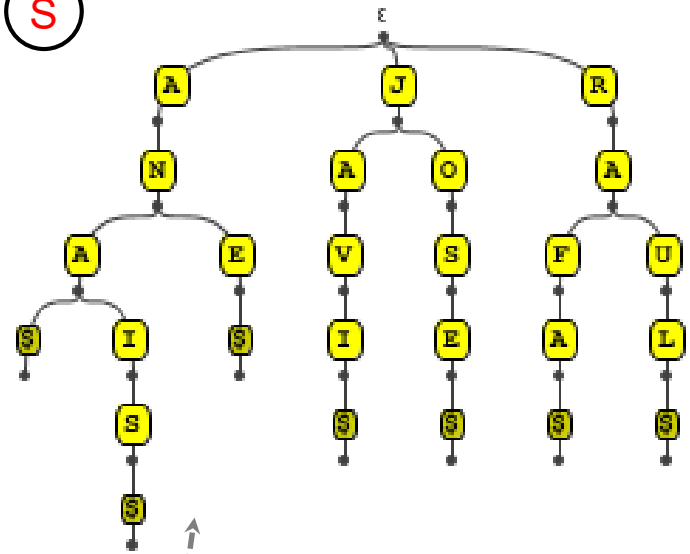
```
tipos
    trie = ↑nodo;
    nodo = registro
              dato:carácter; {como símbolo de fin de palabra se usará el '$' }
                {o definirlo como subrango de los caracteres: predecesor('a') ..'z' }
              primogenito,sigHermano:trie
           freg


procedimiento creaVacío(sal t:trie)
principio
    t:=nil
fin
```

En esta implementación:

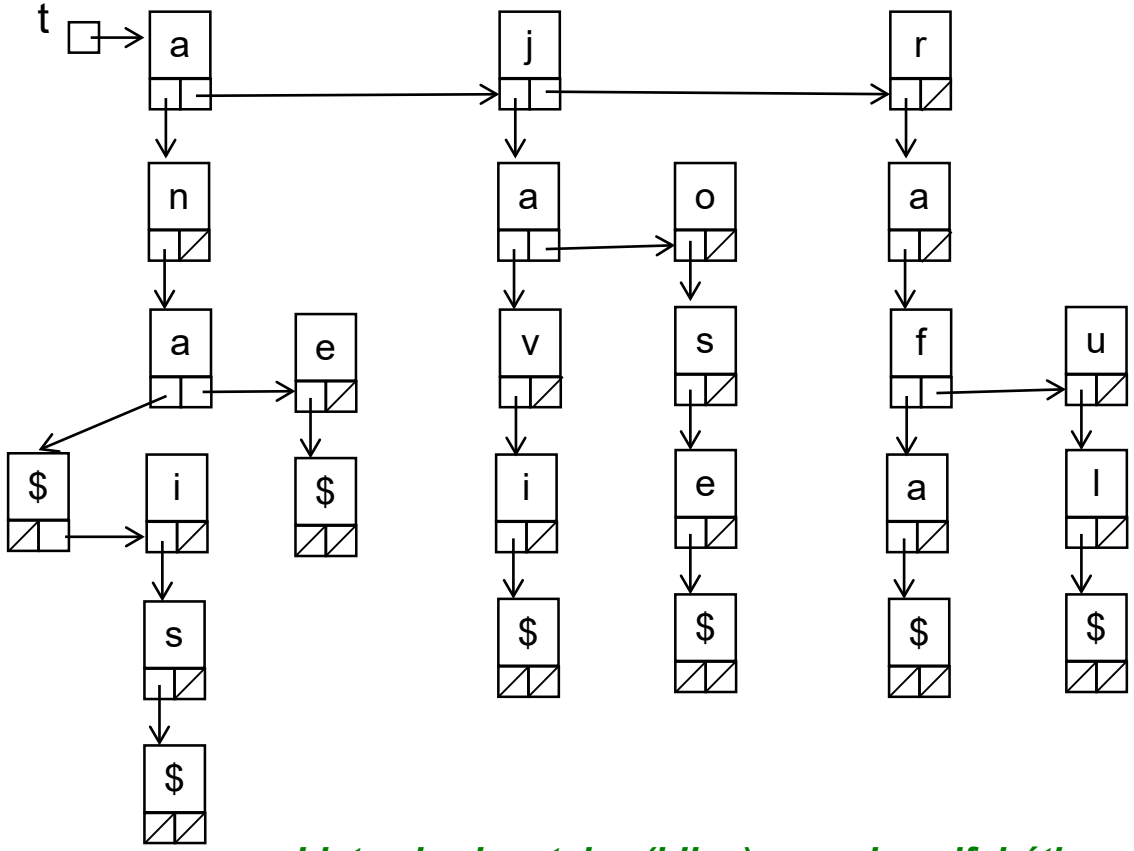
- ***únicamente detalles sobre gestión de las claves (sin detalles sobre los valores)***
- ***cada parte de las claves es de tipo carácter (letra)***
- ***Marca de terminación: '$'***
  - ***No puede aparecer formando parte de las claves***
  - ***Es menor que todos los valores posibles para las partes de la clave***







Esta imagen es un recorte de pantalla de una ejecución de la animación que se ha presentado anteriormente

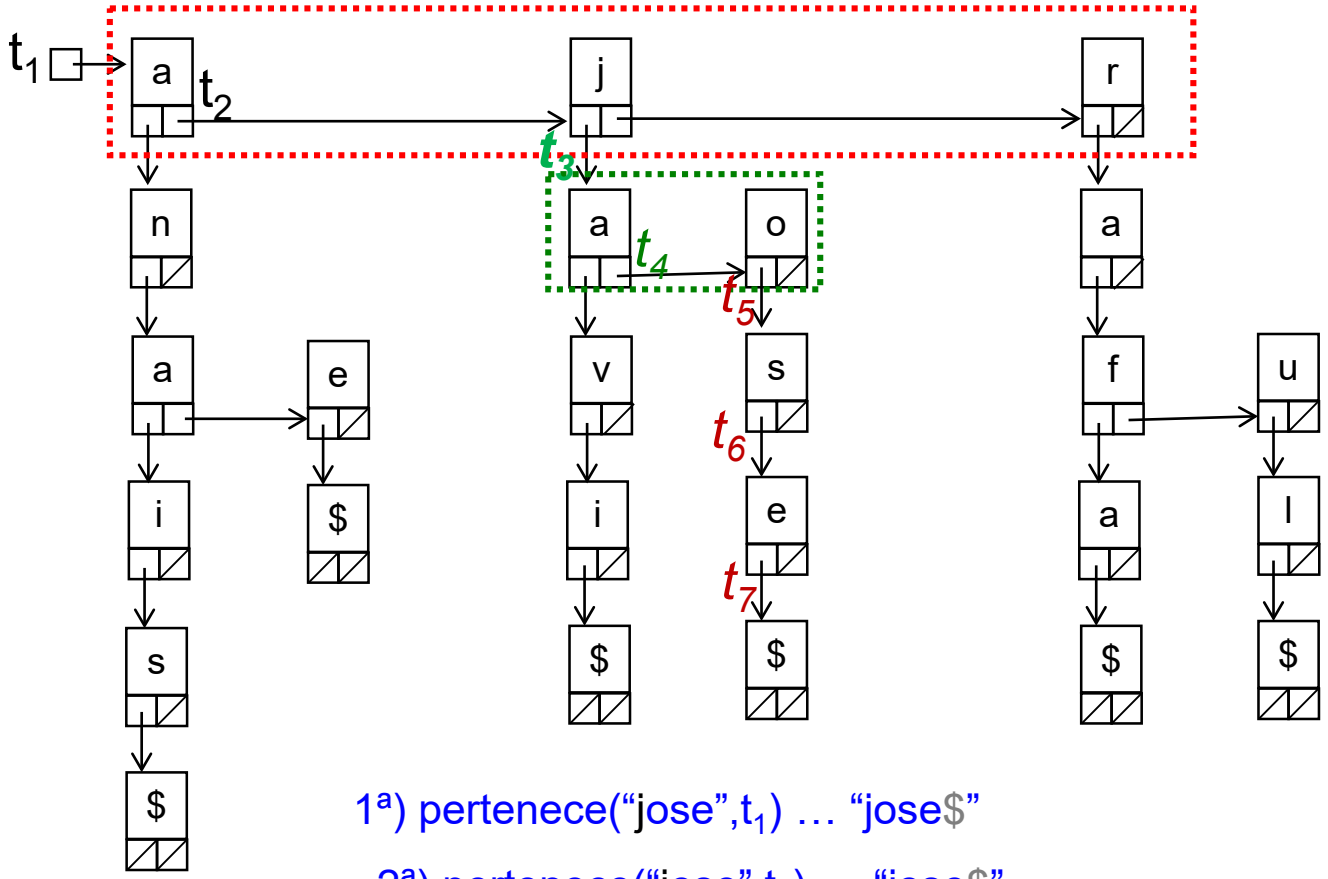


Listas horizontales (hijos) en orden alfabético

En esta implementación

todas las hojas son :

Los únicos nodos sin hijos son :



1ª) pertenece("jose",t_1) … "jose$"

2ª) pertenece("jose",t_2) … "jose$"

3ª) pertenece("ose",t_3) … "ose$"

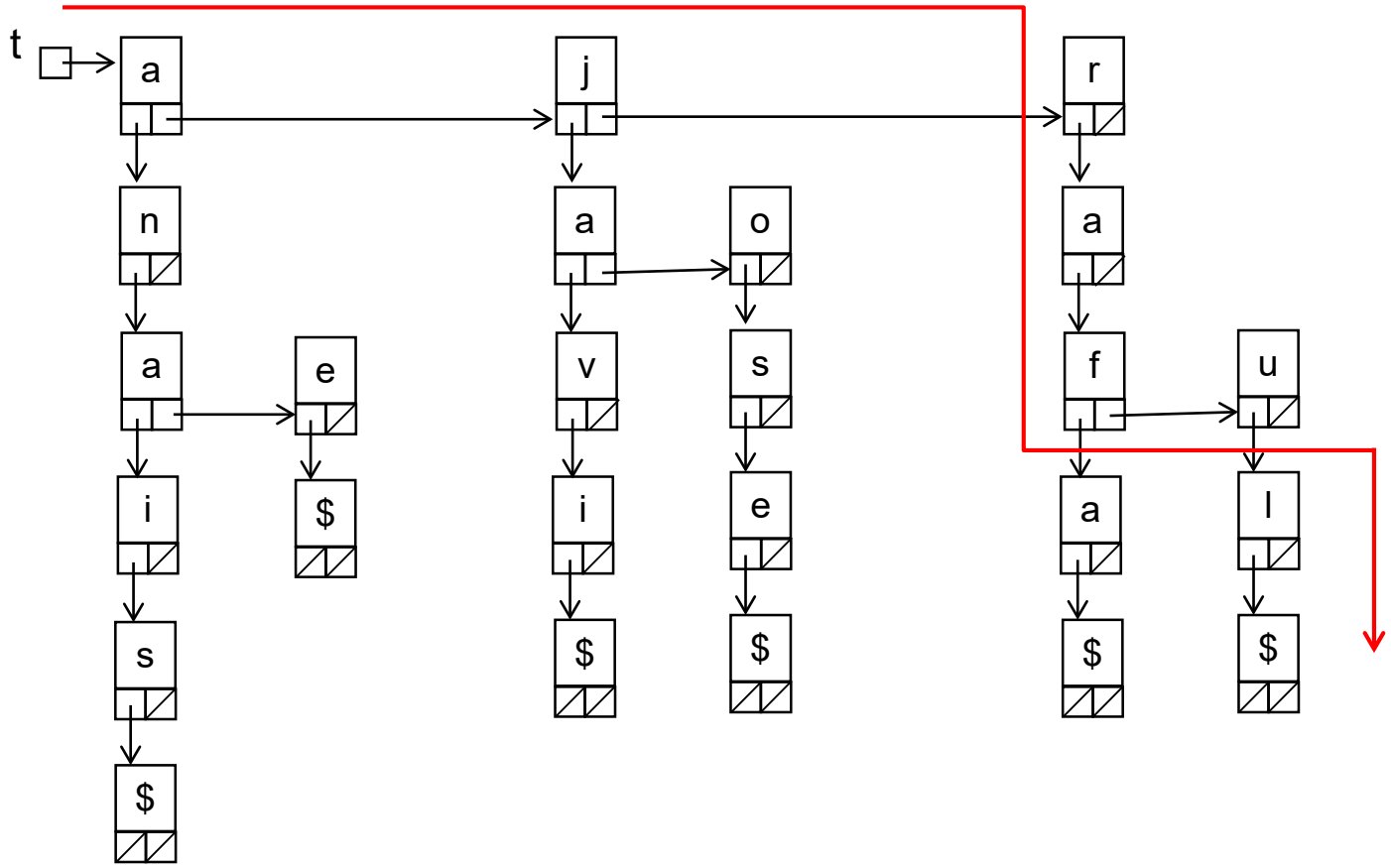
4ª) pertenece("ose",t_4) … "ose$"

5ª) pertenece("se",t_5) … "se$"

6ª) pertenece("e",t_6) … "e$"

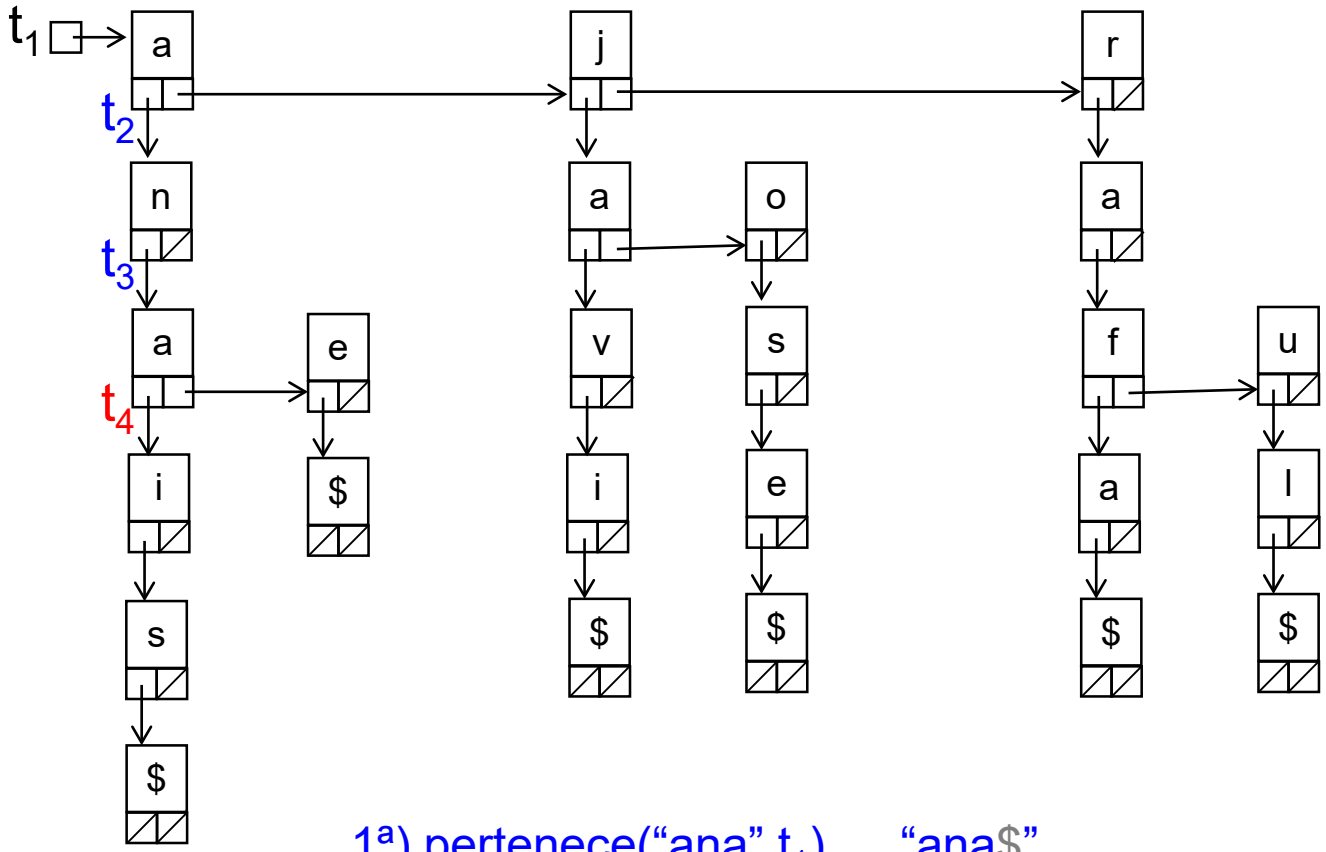
7ª) pertenece("",t_7) … "$"

→ *"jose" sí está*



1ª) pertenece(“raul”,t) … “raul$”

→ *“raul” sí está*



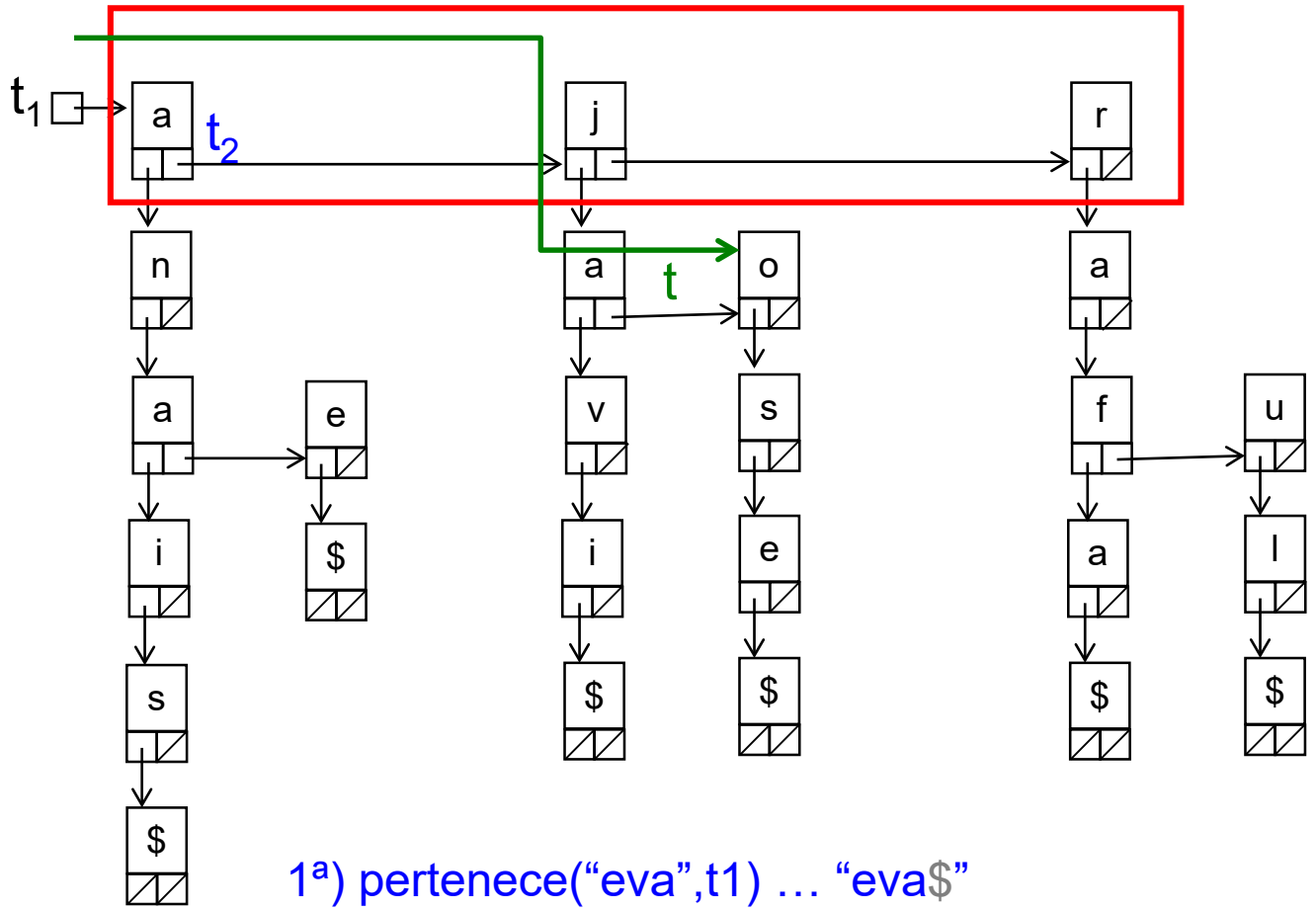
1ª) pertenece("ana",t_1) … "ana$"

2ª) pertenece("na",t_2) … "na$"

3ª) pertenece("a",t_3) … "a$"

4ª) pertenece("",t_4) … "$"

→ *"ana" no está*



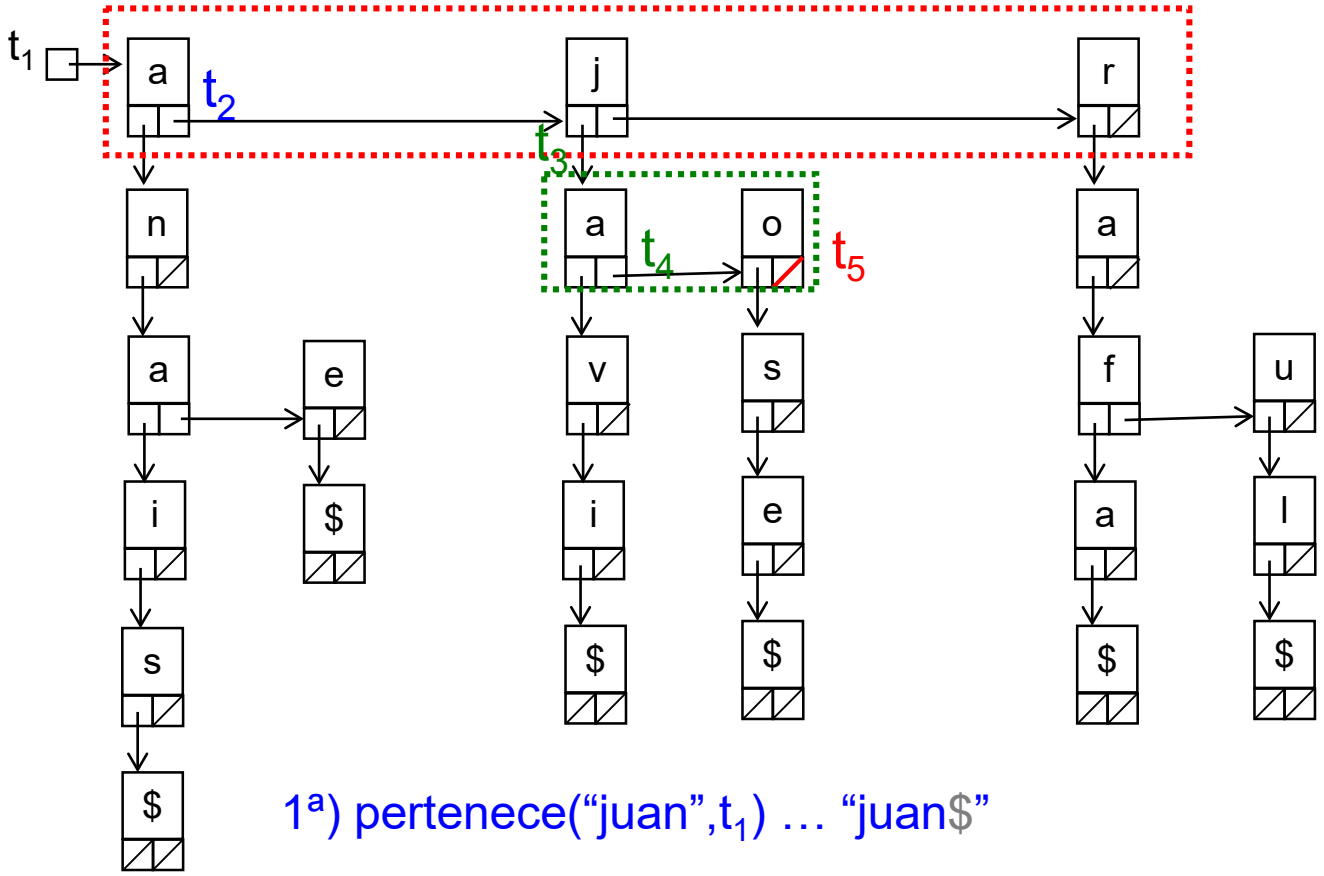
1ª) pertenece("eva",t1) … "eva$"

2ª) pertenece("eva",t2) … "eva$"

"eva" no está

1ª) pertenece("jean",t1) … "jean$"

"jean" no está



1ª) pertenece("juan",t_1) … "juan$"

2ª) pertenece("juan",t_2) … "juan$"

3ª) pertenece("uan",t_3) … "uan$"

4ª) pertenece("uan",t_4) … "uan$"

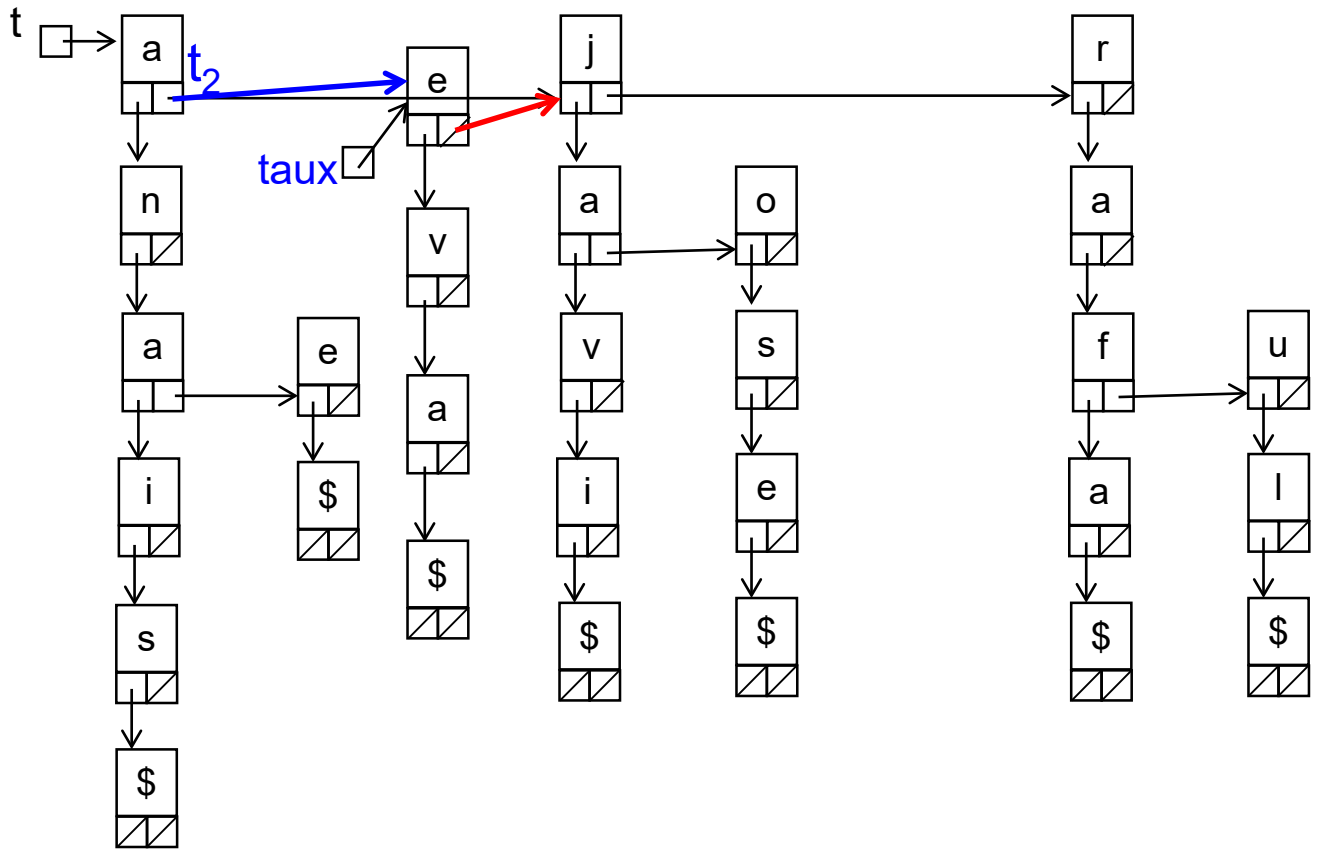
5ª) pertenece("uan",t_5) … "uan$"

→ "juan" no está

Detalles implementación Nodo-lista

```
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
```

{Búsqueda recursiva en lista horizontal ordenada}



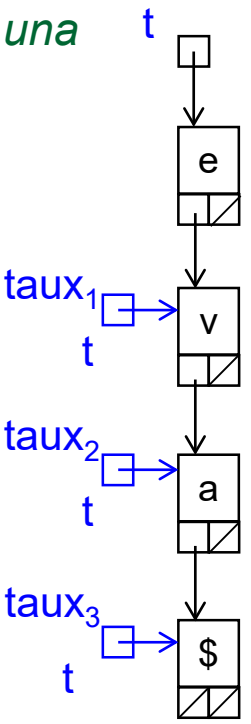
1ª) insertar(“eva”,t) … “eva$”

2ª) insertar(“eva”,t_2) … “eva$”

plantar(“eva”,taux) y encadenarlo en la lista

Detalles implementación Nodo-lista

```
procedimiento plantaPalabra(ent palabra:cadena; sal 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 palabra}
      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
```

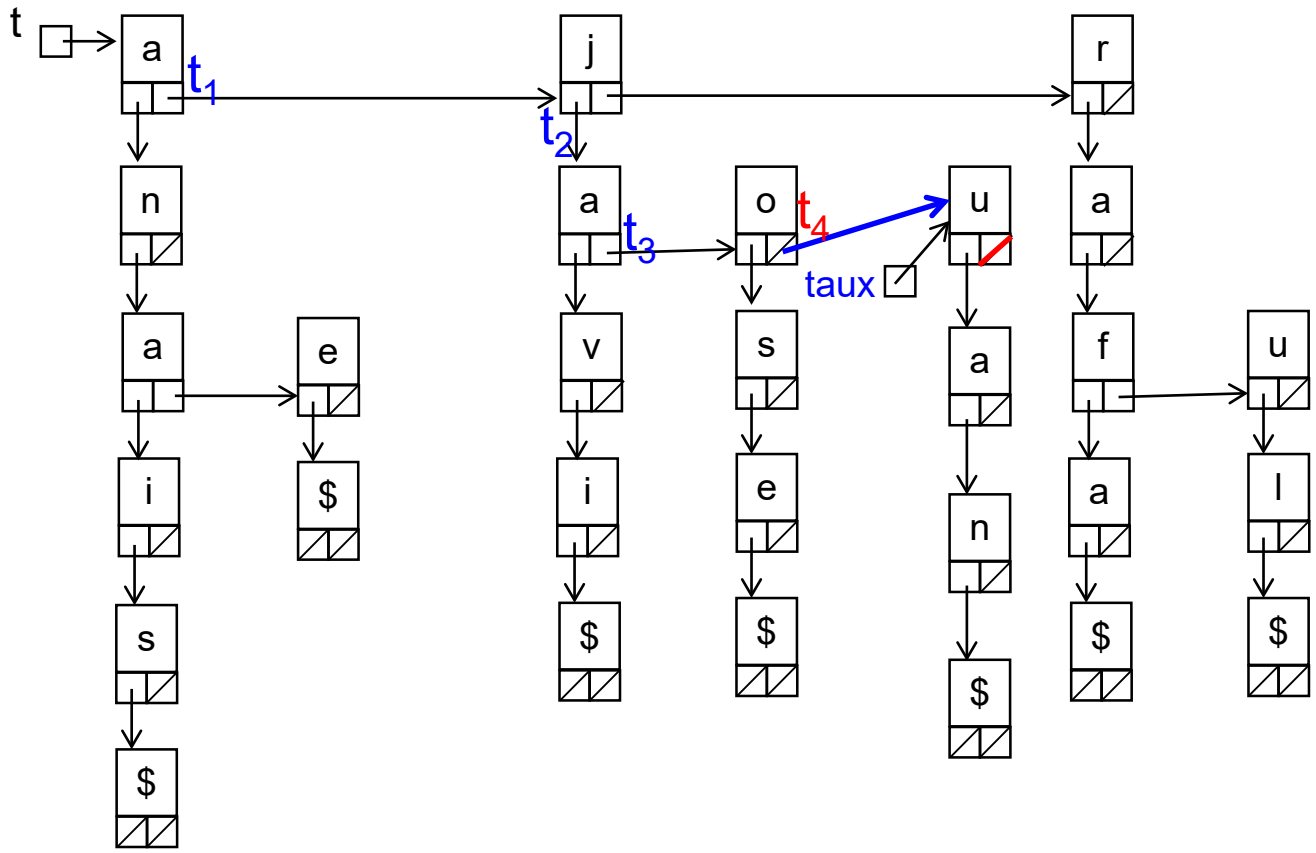


1ª) plantaPalabra("eva",t)

2ª) plantaPalabra("va",$taux_1$)

3ª) plantaPalabra("a",$taux_2$)

4ª) plantaPalabra("",$taux_3$)



1ª) insertar("juan",t) … "juan$"

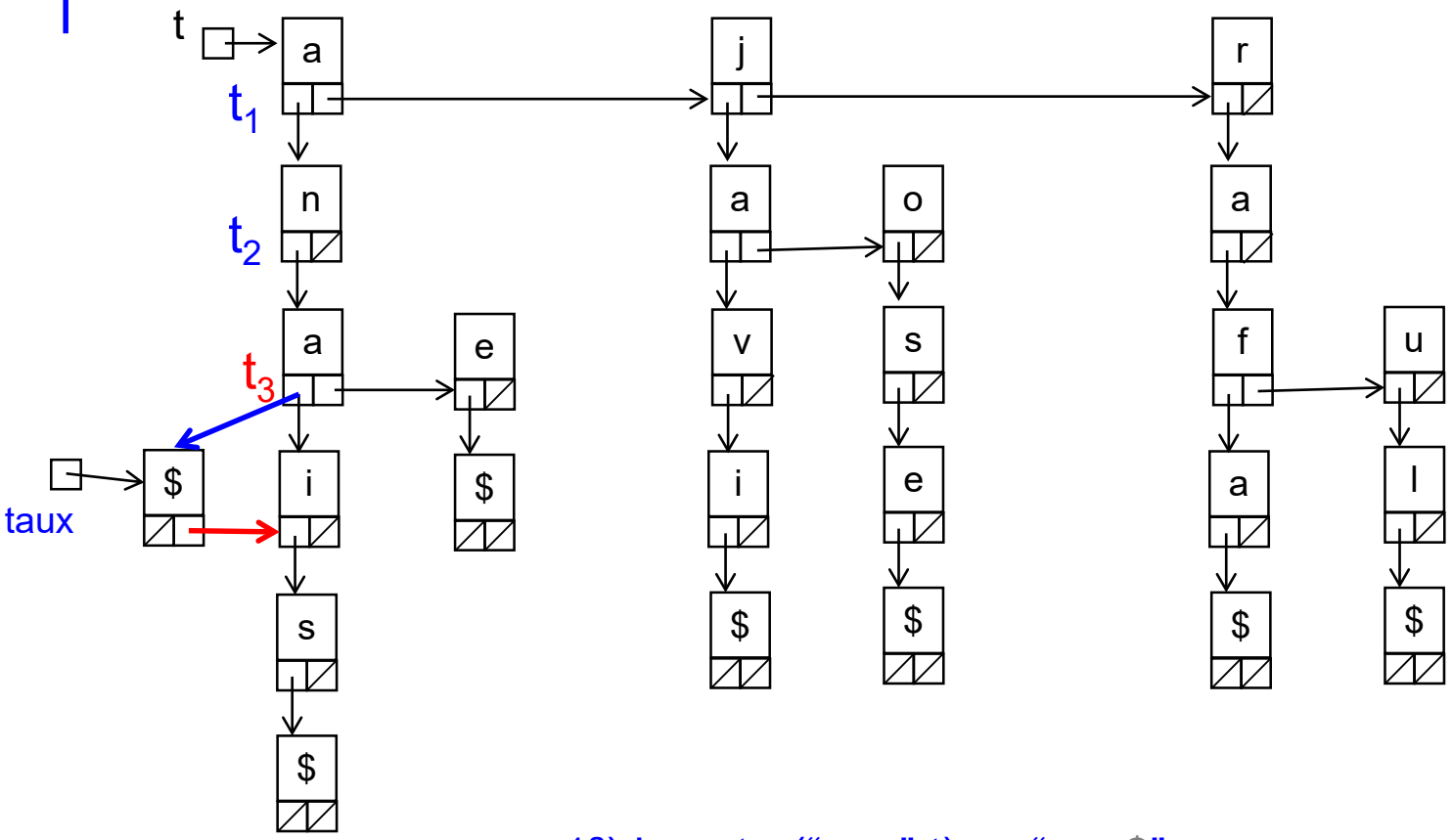
2ª) insertar("juan",t_1) … "juan$"

3ª) insertar("uan",t_2) … "uan$"

4ª) insertar("uan",t_3) … "uan$"

5ª) insertar("uan",t_4) … "uan$"

plantar("uan",taux) y encadenarlo en la lista



1ª) insertar("ana",t) … "ana$"

2ª) insertar("na",t_1) … "na$"

3ª) insertar("a",t_2) … "a$"

4ª) insertar("",t_3) … "$"

crear nodo con '$' y encadenarlo

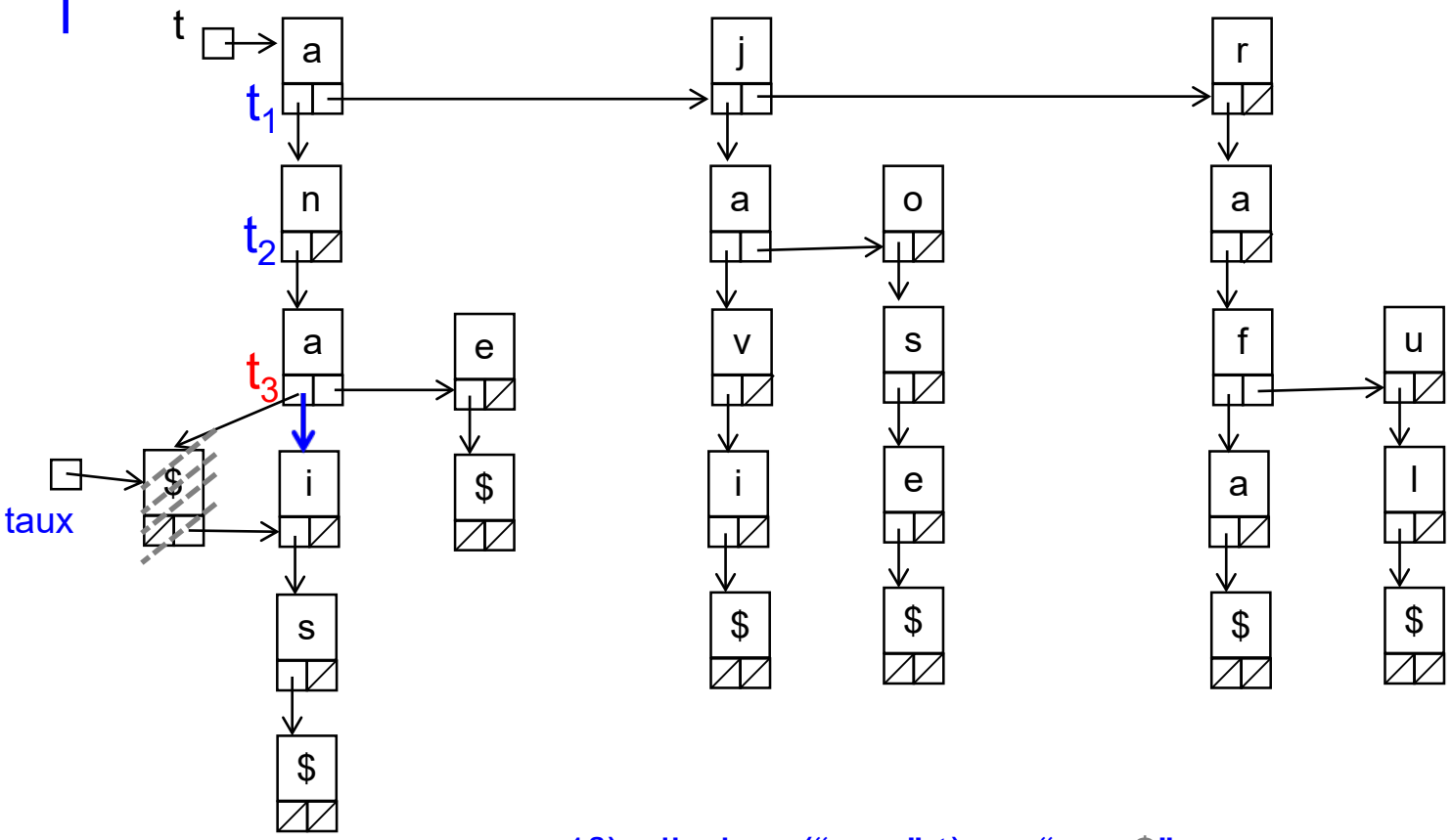
Detalles implementación Nodo-lista

```
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
          {sino…. la palabra a insertar ya estaba, no hacer nada}
          fsi
      {sino….}
…
```

{ podrían sustituirse por: plantar(“”,taux); }

Detalles implementación Nodo-lista

```
...
        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
```



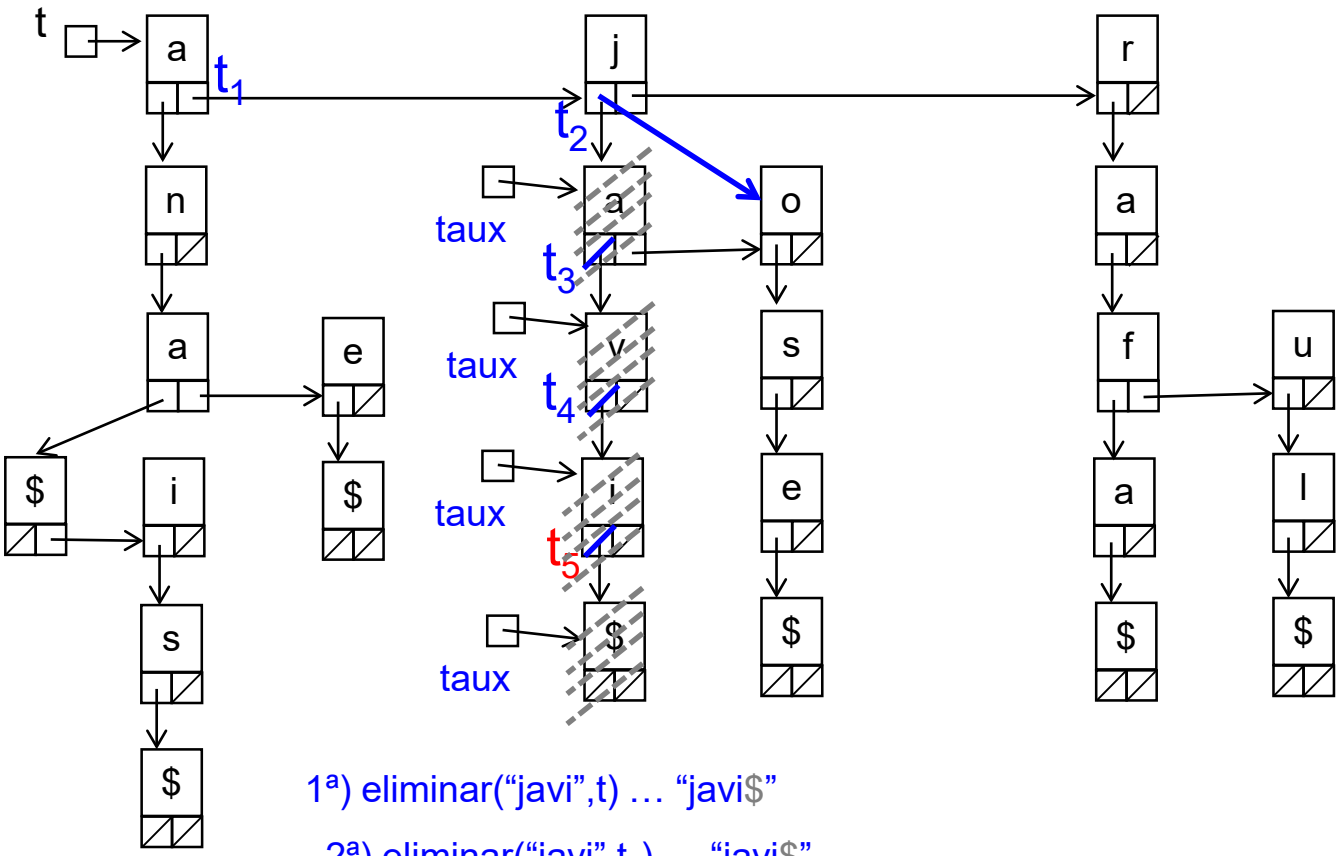
1ª) eliminar("ana",t) … "ana$"

2ª) eliminar("na",t_1) … "na$"

3ª) eliminar("a",t_2) … "a$"

4ª) eliminar("",t_3) … "$"

desencadenar nodo con '$' y liberar su memoria



1ª) eliminar("javi",t) … "javi$"

2ª) eliminar("javi",t_1) … "javi$"

3ª) eliminar("avi",t_2) … "avi$"

4ª) eliminar("vi",t_3) … "vi$"

5ª) eliminar("i",t_4) … "i$"

6ª) eliminar("",t_5) … "$"

desencadenar nodo con '$' y disponer su memoria

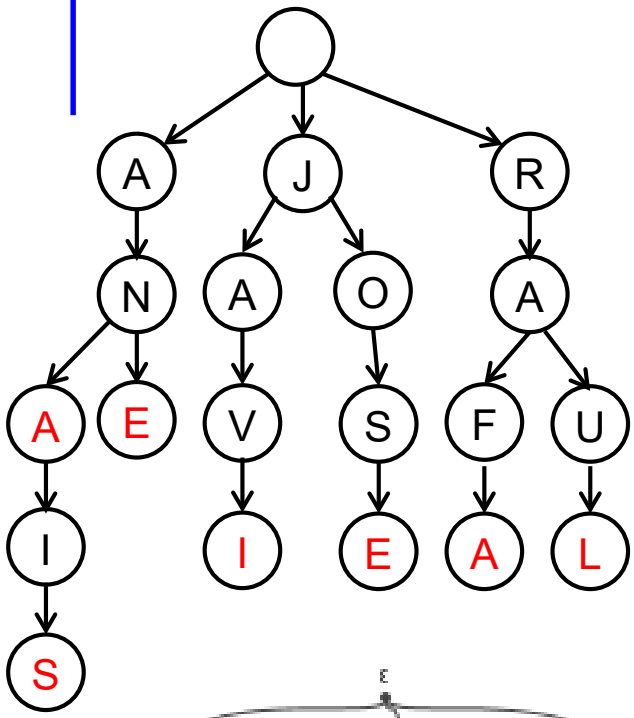
Repetir: si el padre ha quedado sin hijos: desencadenarlo y disponer

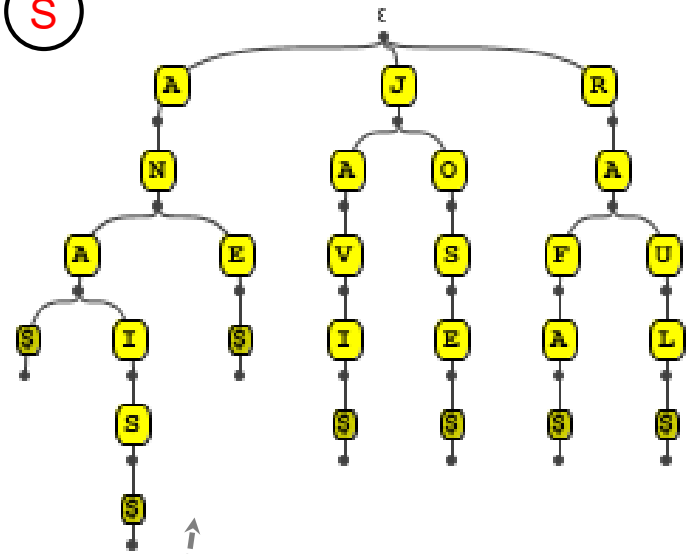
Los únicos nodos sin hijos son :

Detalles implementación Nodo-lista

```
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)
           { sino …. palabra[1]<t↑.dato …. la clave no está, y parar}
           fsi
      fsi
 fsi
fin
```

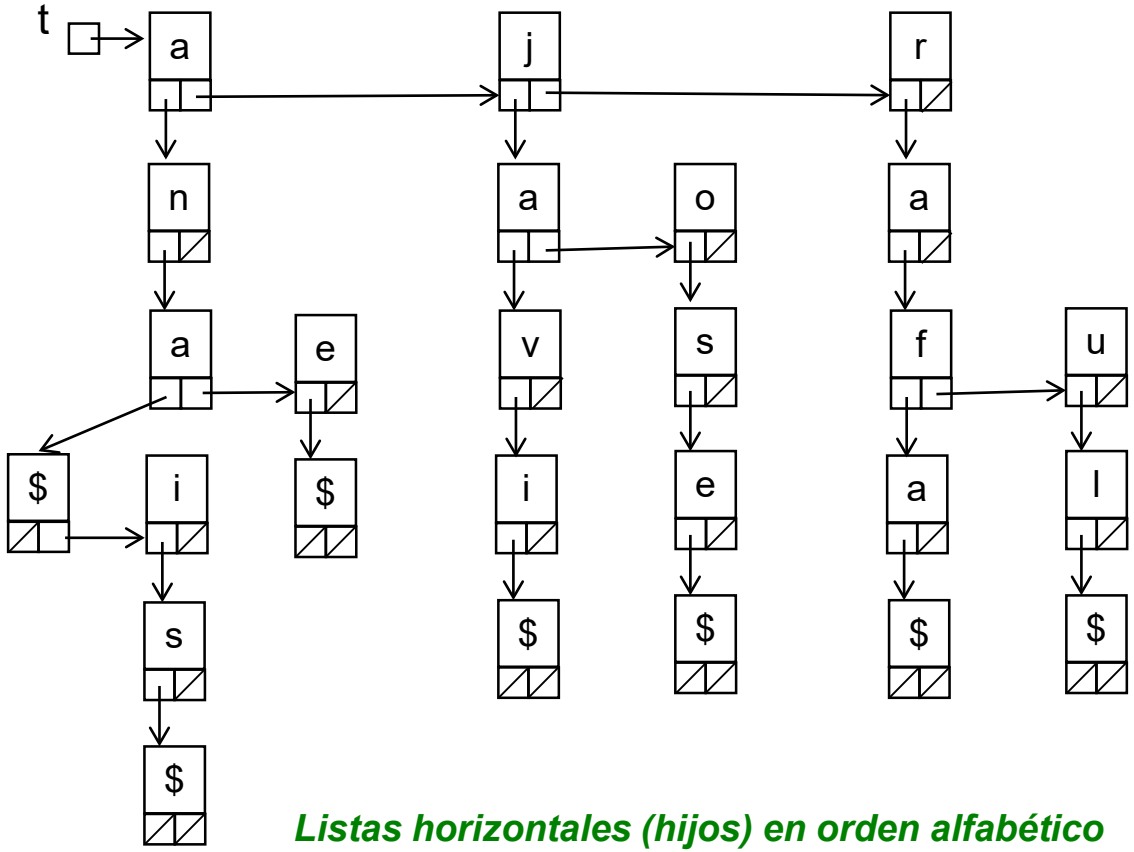
Los únicos nodos sin hijos son :





Esta imagen es un recorte de pantalla de una ejecución del applet que se ha presentado anteriormente

¿Cómo obtener en orden alfabético todas las palabras almacenadas en un *trie, por ejemplo para escribirlas en pantalla?*



Listas horizontales (hijos) en orden alfabético

En esta implementación todas las hojas son :

Los únicos nodos sin hijos son :

¿Cómo obtener en orden alfabético todas las palabras almacenadas en un *trie, por ejemplo para escribirlas en pantalla? Adaptación del recorrido en pre-orden para bosque y árboles n-ario vistos en la lección 15:*

```
procedimiento escribe(ent t:trie)
    procedimiento preTrie(ent t:trie; ent palabra:cadena) {En el parámetro 'palabra' se recibe
    la concatenación de caracteres del camino que va de la raíz del trie original hasta el 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)
    principio
      si t↑.dato='$' entonces
        escribirLínea(palabra) { el nodo con '$' no tiene hijos}
      sino
        preTrie(t↑.primogenito, palabra + t↑.dato)  {+ es la concatenación de cadenas}
      fsi
    fin
  variable palabra:cadena
principio {de escribe}
   palabra:="";        {"" representa la cadena vacía}
   preTrie(t,palabra)
fin
```

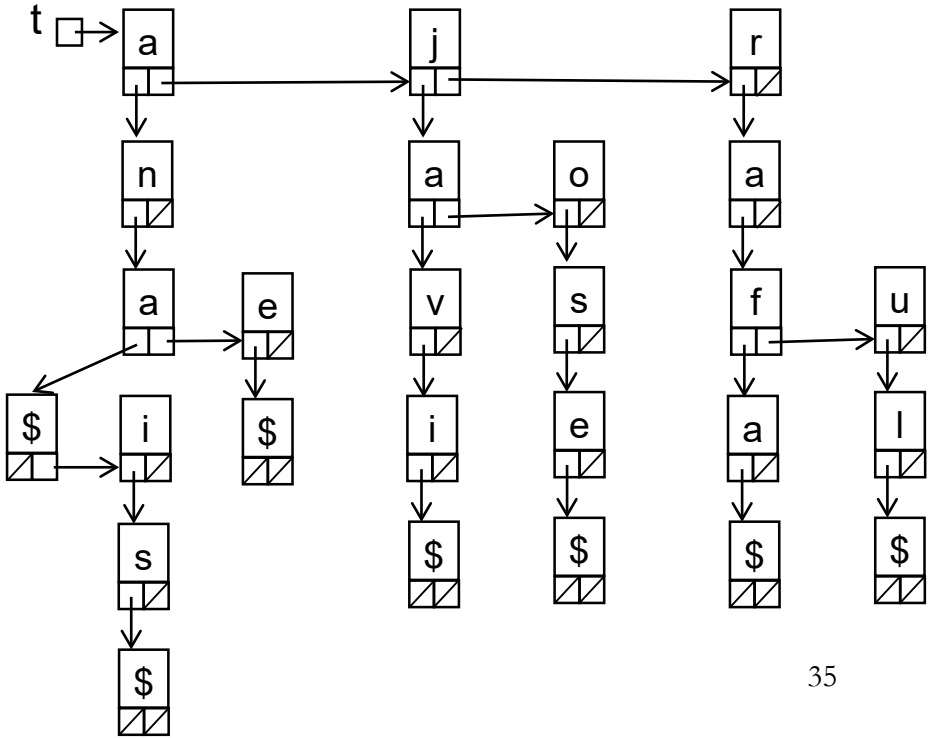
Los únicos nodos sin hijos son :

Versión concentrada de un recorrido por un bosque n-ario:

{Adaptación de la forma alternativa de hacer un recorrido en pre-orden de bosques n-arios}

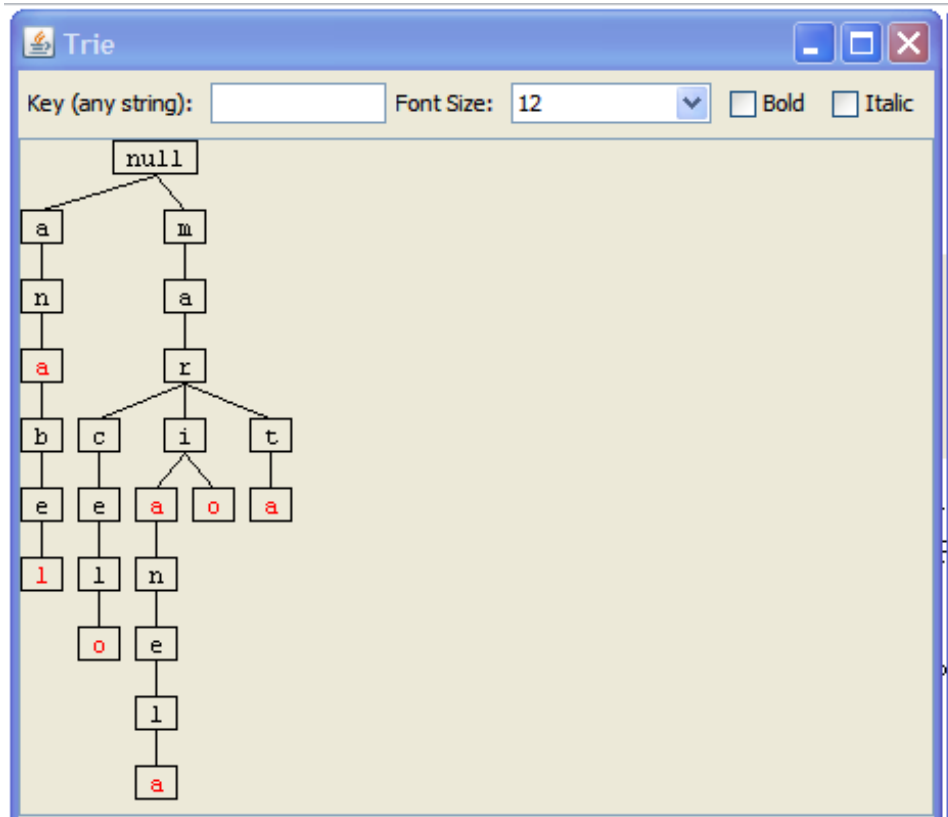
```
procedimiento pretrie(ent t:trie; ent palabra:cadena) {En el parámetro 'palabra' se recibe la
concatenación de caracteres del camino que se ha recorrido hasta llegar a t}
principio
   si t ≠ nil entonces
      si t↑.dato='$' entonces
         escribirlínea(palabra) { el nodo con '$' no tiene hijos}
      sino
         pretrie(t↑.primogénito,palabra + t↑.dato)
      fsi;
      pretrie(t↑.sigHermano,palabra)
   fsi
fin

procedimiento escribe(ent t:trie)
principio
   pretrie(t,"")
fin
```

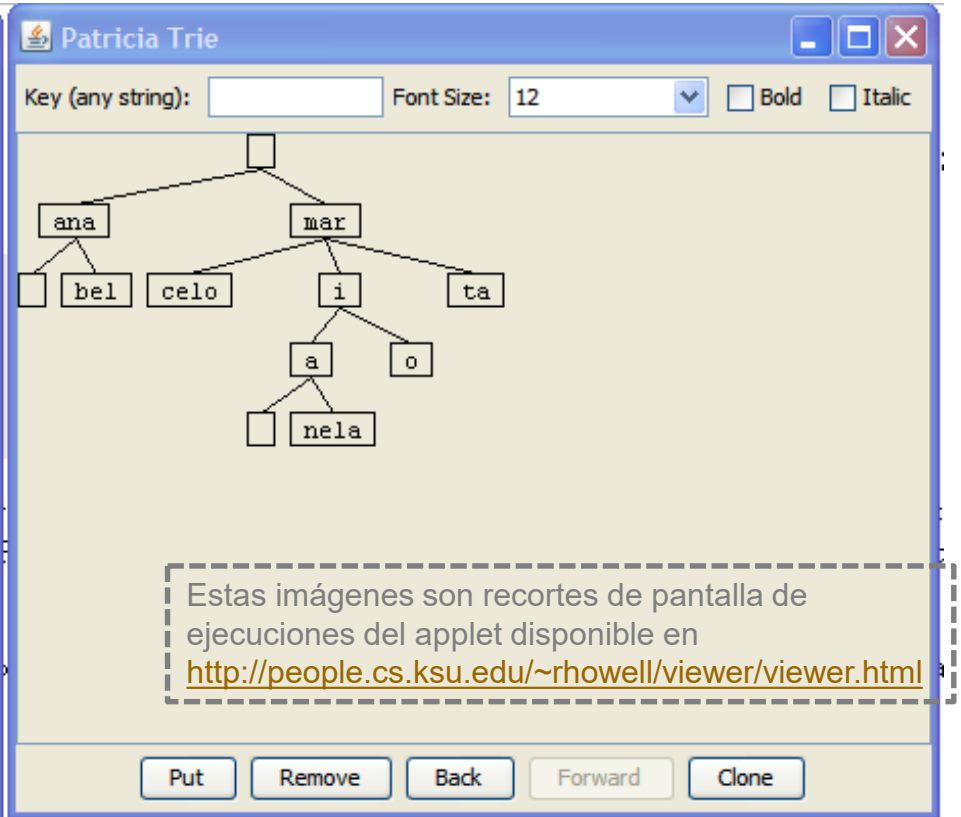
Los únicos nodos sin hijos son :



Patricia: Practical Algorithm To Retrieve Information Coded In Alphanumeric (D.R. Morrison, 1968)

- Evitan la proliferación de nodos con un único hijo en los tries, y por tanto reducen su altura y el coste de las operaciones (detalles: http://webdiis.unizar.es/asignaturas/TAP/material/2.4.digitales.pdf)
- Con la misma idea se definieron, más o menos al mismo tiempo, los **Radix Tree**, también llamados **radix trie** o **compact prefix tree.** Los PATRICIA son Radix trees con r=2
- Una animación para probar los radix tree https://www.cs.usfca.edu/~galles/visualization/RadixTree.html





Estas imágenes son recortes de pantalla de ejecuciones del applet disponible en http://people.cs.ksu.edu/~rhowell/viewer/viewer.html