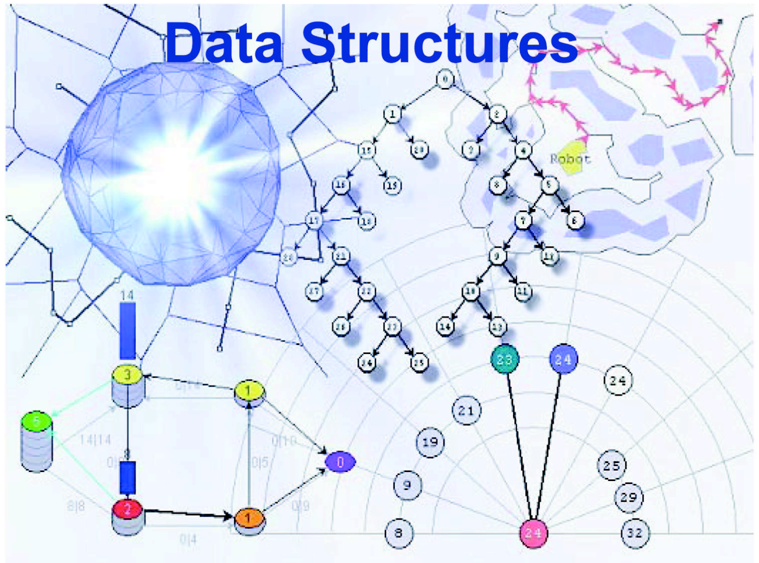
# BBM 201
# DATA STRUCTURES

## Lecture 11:
## Data Structures for Strings



# Outline

- Tries
- R-way tries
- Patricia Trees
- Suffix Trees (details will be discussed in BBM202)
- Suffix Arrays (details will be discussed in BBM202)

# Introduction

Numbers as key values: are data items of constant size and can be compared in constant time.

In real applications, text processing is more important than the processing of numbers

We need different structures for strings than for numeric keys.

# Motivating Example

Example: 112 < 467 , Numerical comparison in O(1).

Compare Strings lexicographically does not reflect the similarity of strings.

- Western > Eastern , Strings comparison in O(min(|s1|,|s2|)). where |s| denotes the length of the string s

Text fragments have a length; they are not elementary objects that the computer can process in a single step.

- Pneumonoultramicroscopicsilicovolcanoconiosis !!!

# Applications

Bioinformatics (DNA/RNA or protein sequence data).

Search Engines

Spell checkers

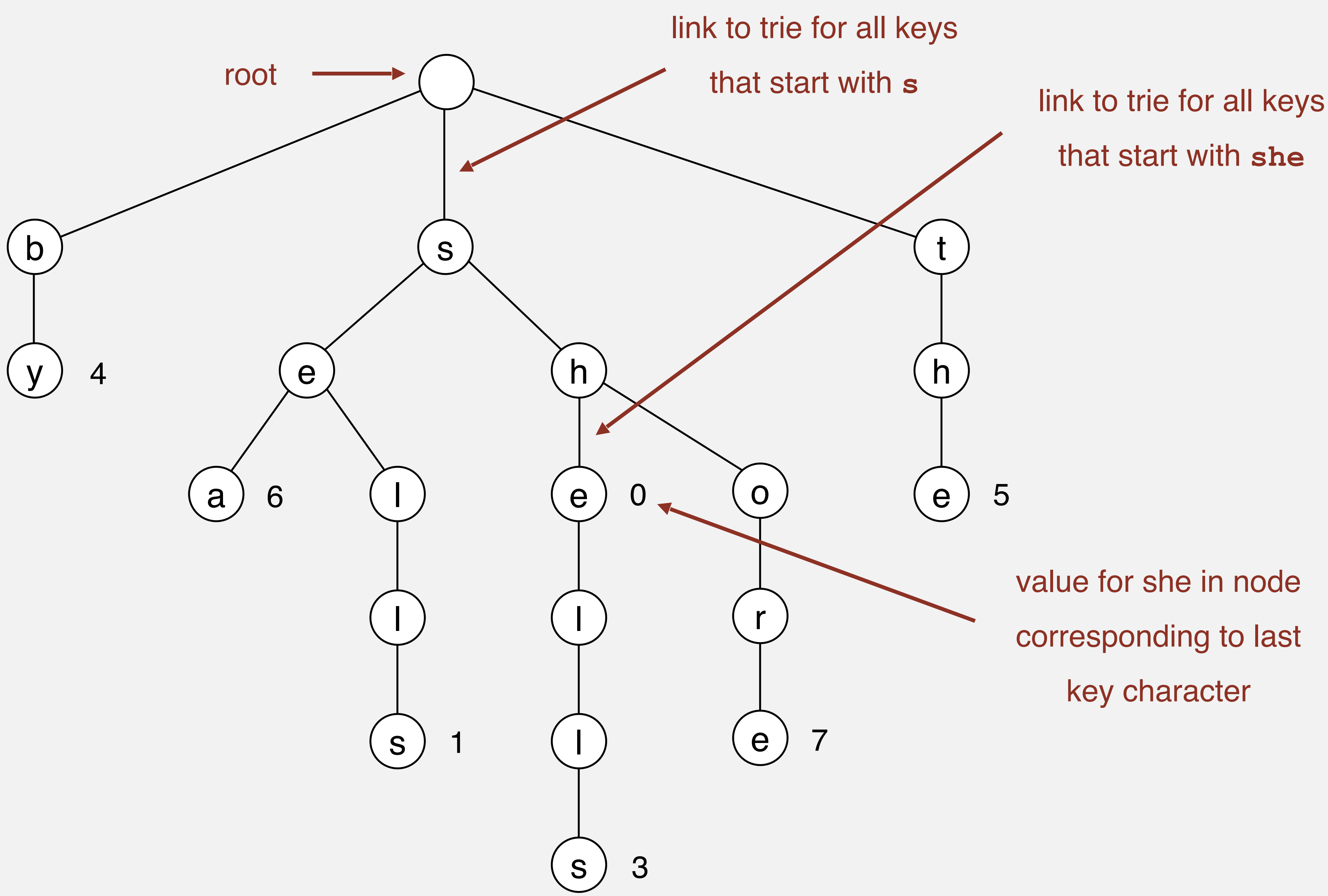
# Tries

**Tries.** [from re**trie**val, but pronounced "try"]

- Store characters in nodes (not keys).
- Each node has $R$ children, one for each possible character.
- Store values in nodes corresponding to last characters in keys.

for now, we do not draw null links

| key | value |
|---|---|
| by | 4 |
| sea | 6 |
| sells | 1 |
| she | 0 |
| shells | 3 |
| shore | 7 |
| the | 5 |

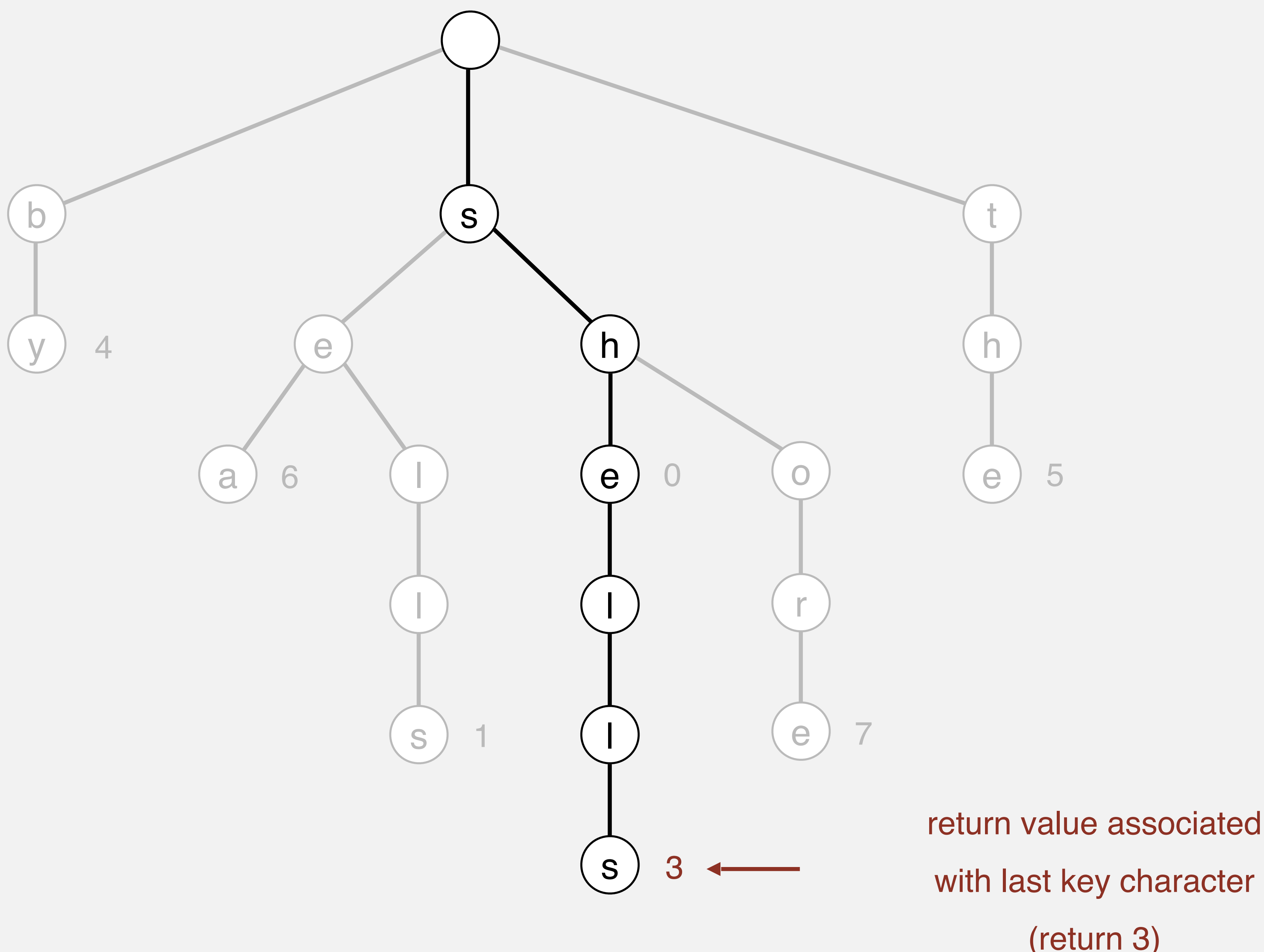


# Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.

get("shells")

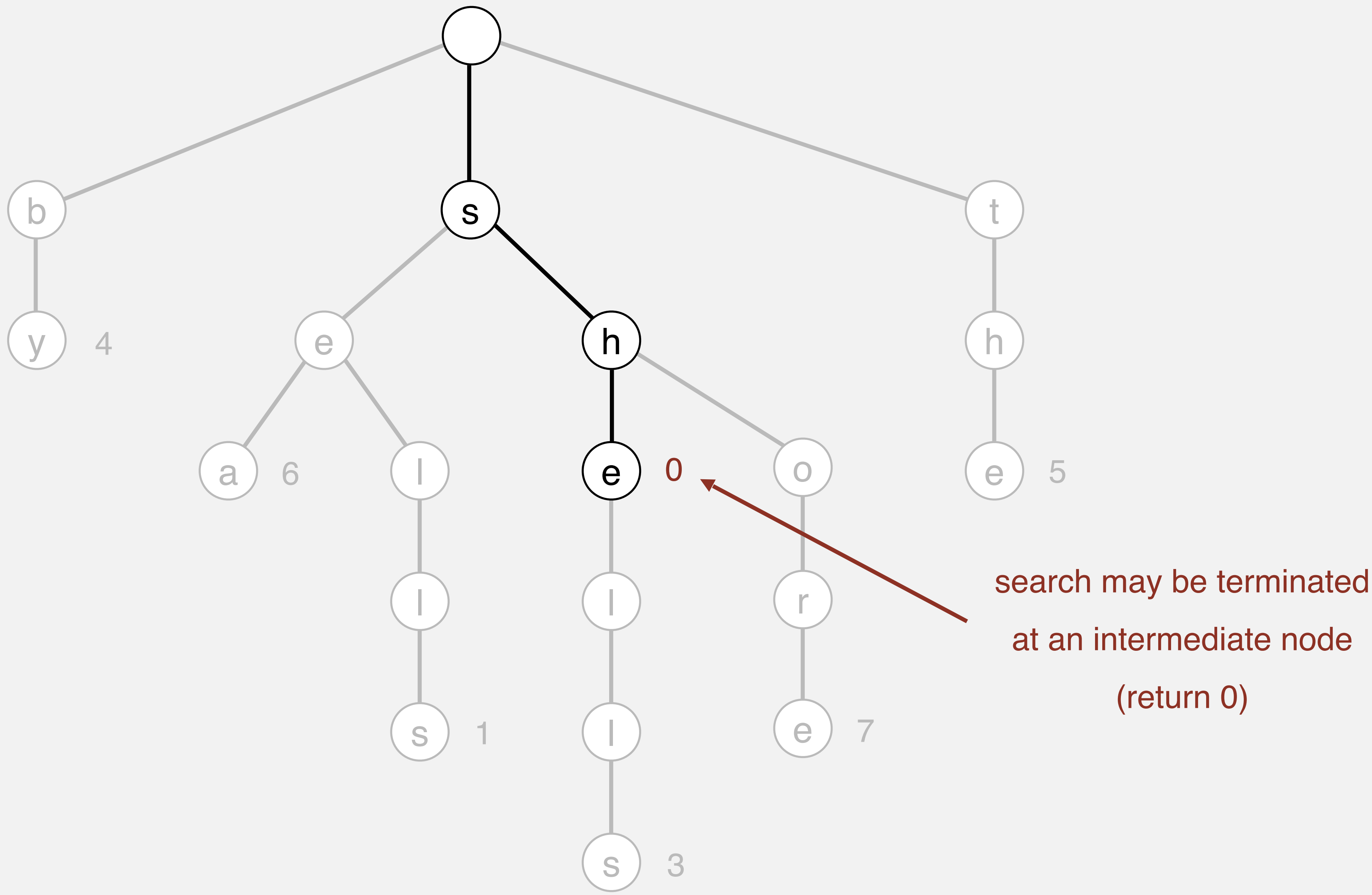


# Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.

get("she")

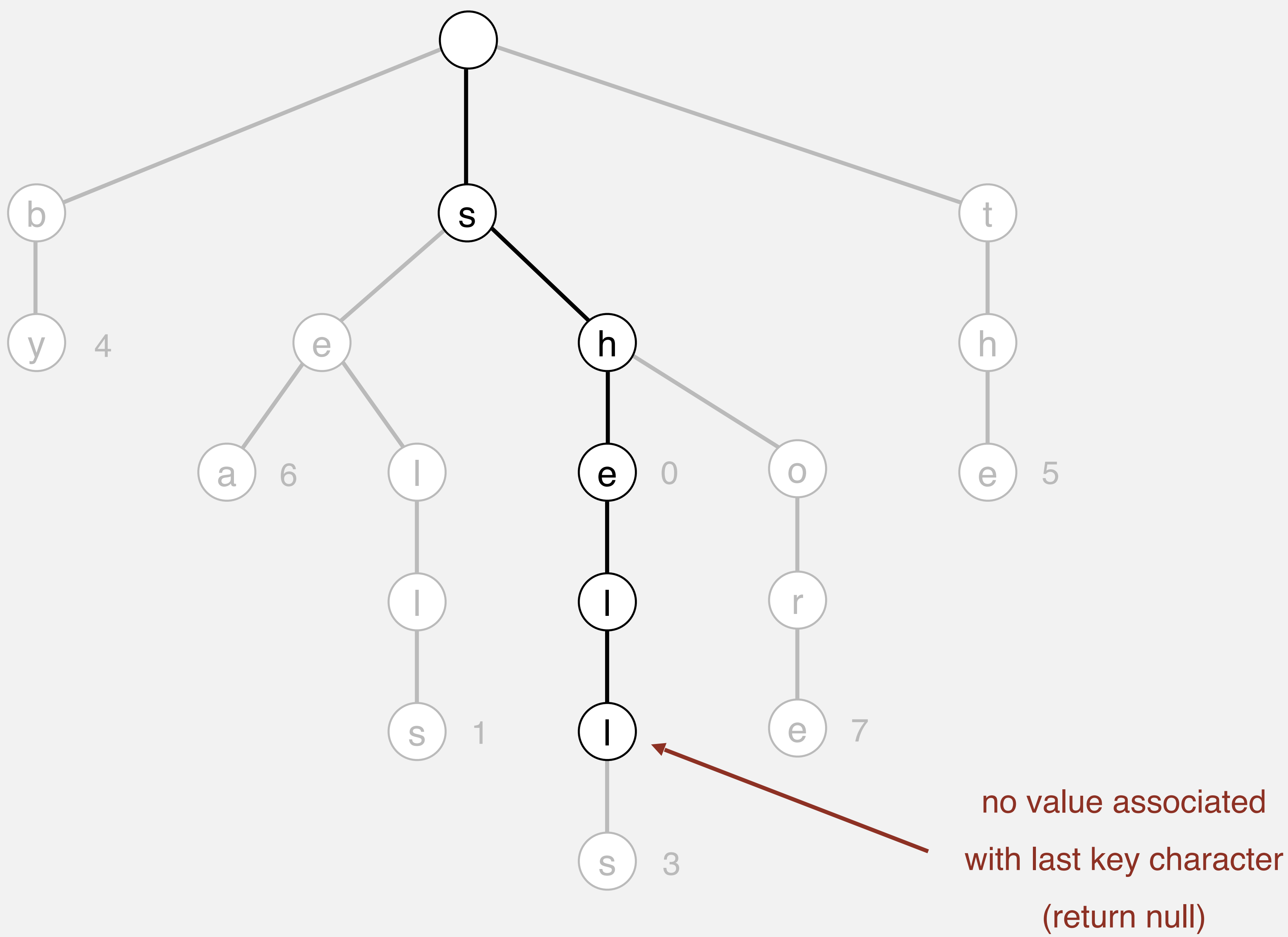


# Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.

get("shell")

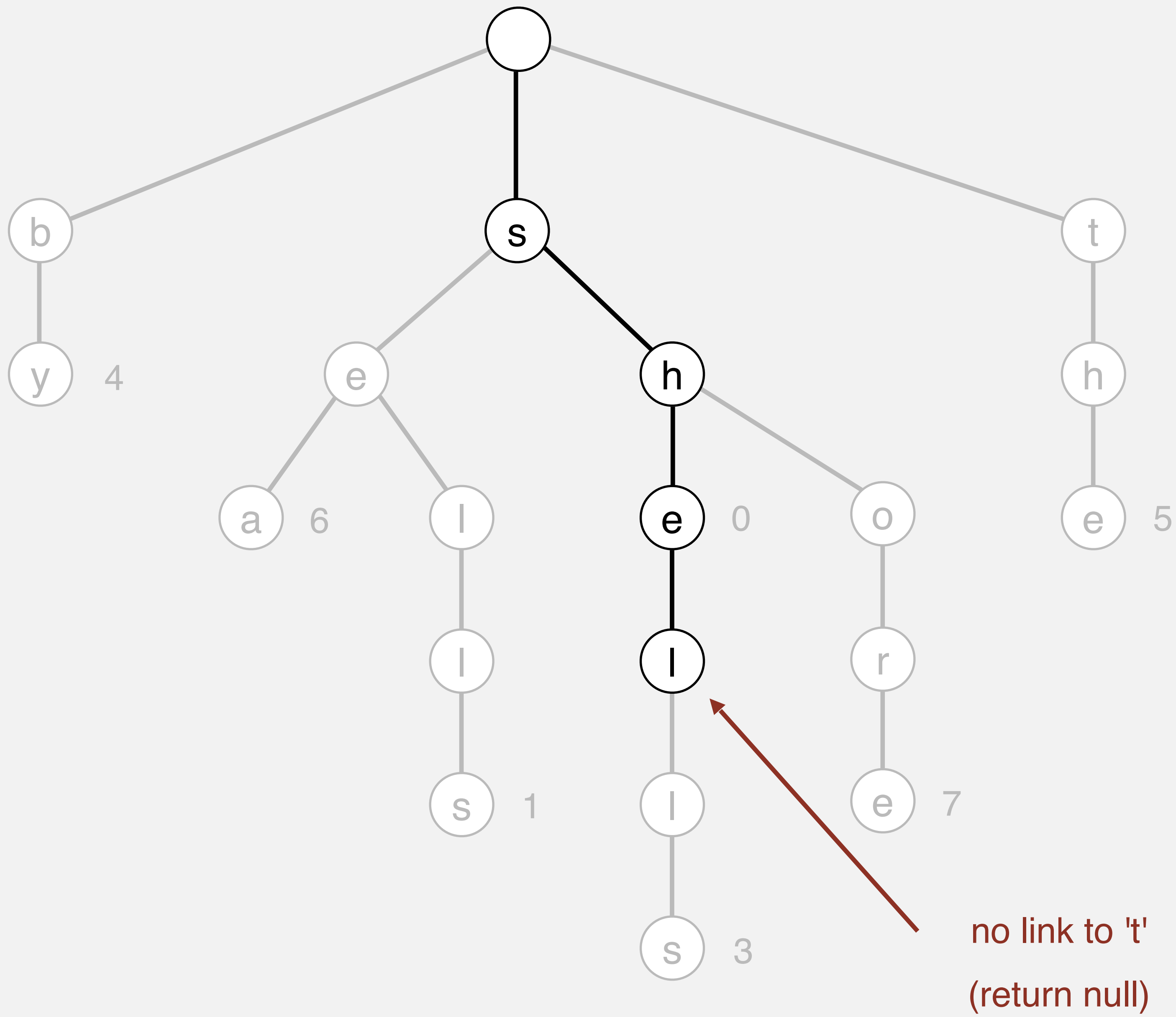


# Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach a null link or node where search ends has null value.

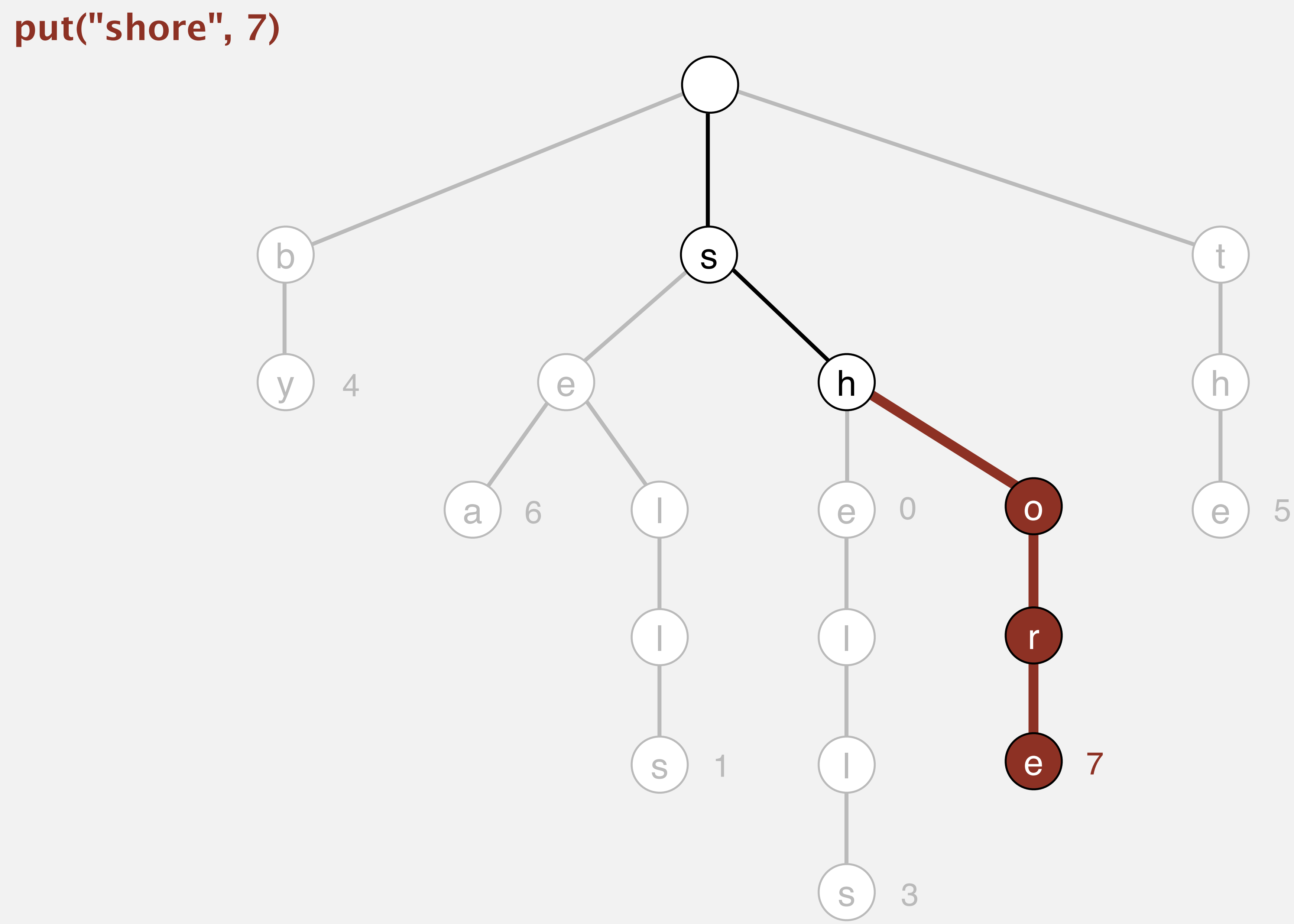
get("shelter")



# Insertion into a trie

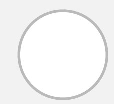
Follow links corresponding to each character in the key.

- Encounter a null link: create new node.
- Encounter the last character of the key: set value in that node.



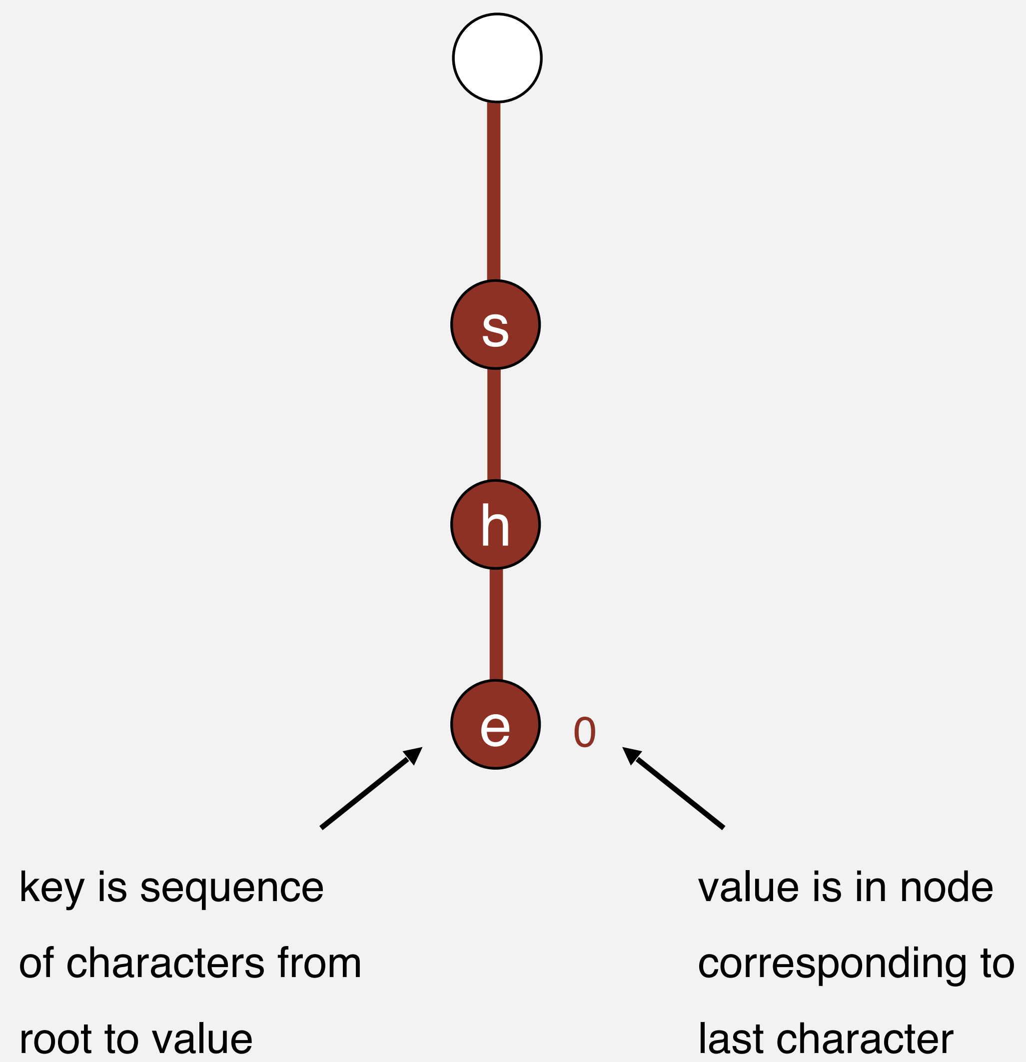
# Trie construction demo

trie



# Trie construction demo

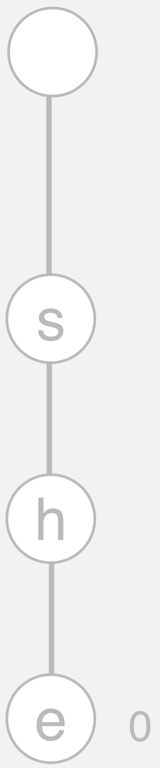
put("she", 0)



# Trie construction demo

she

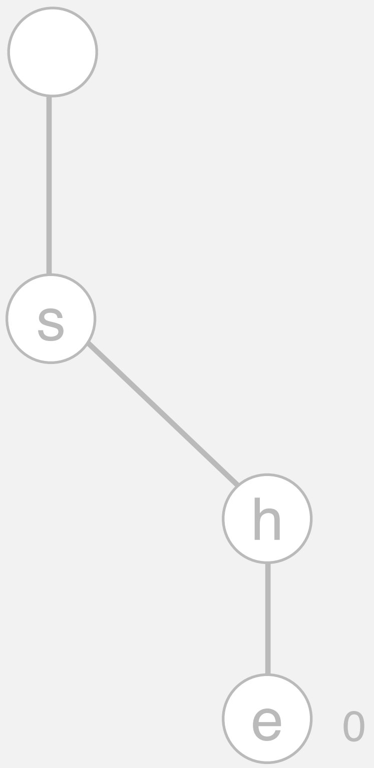
trie



# Trie construction demo

she

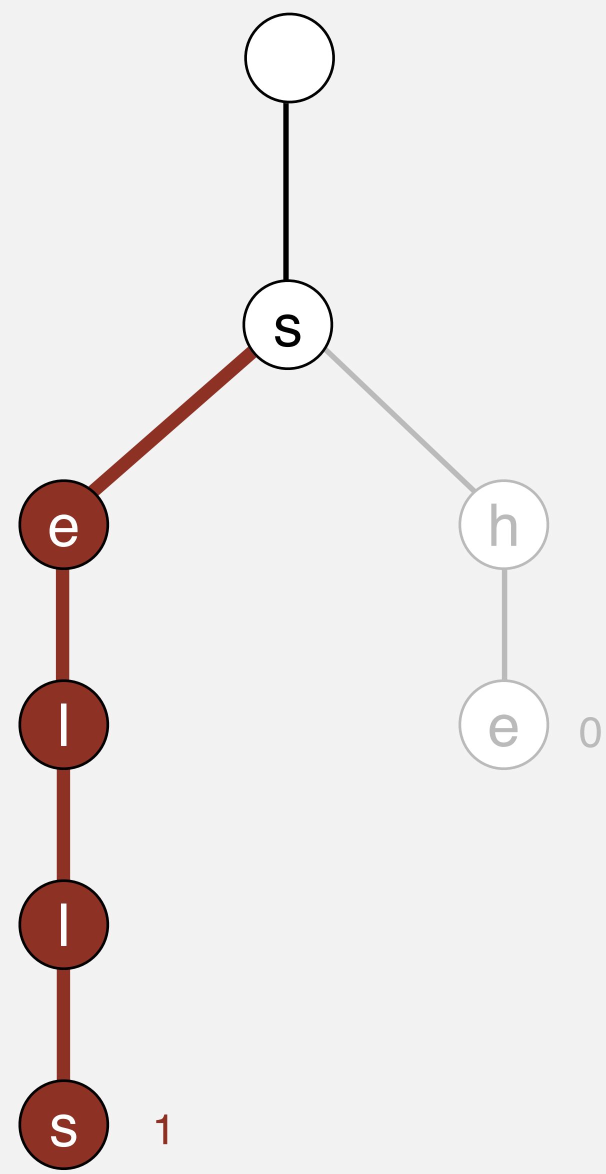
trie



# Trie construction demo

she

put("sells", 1)

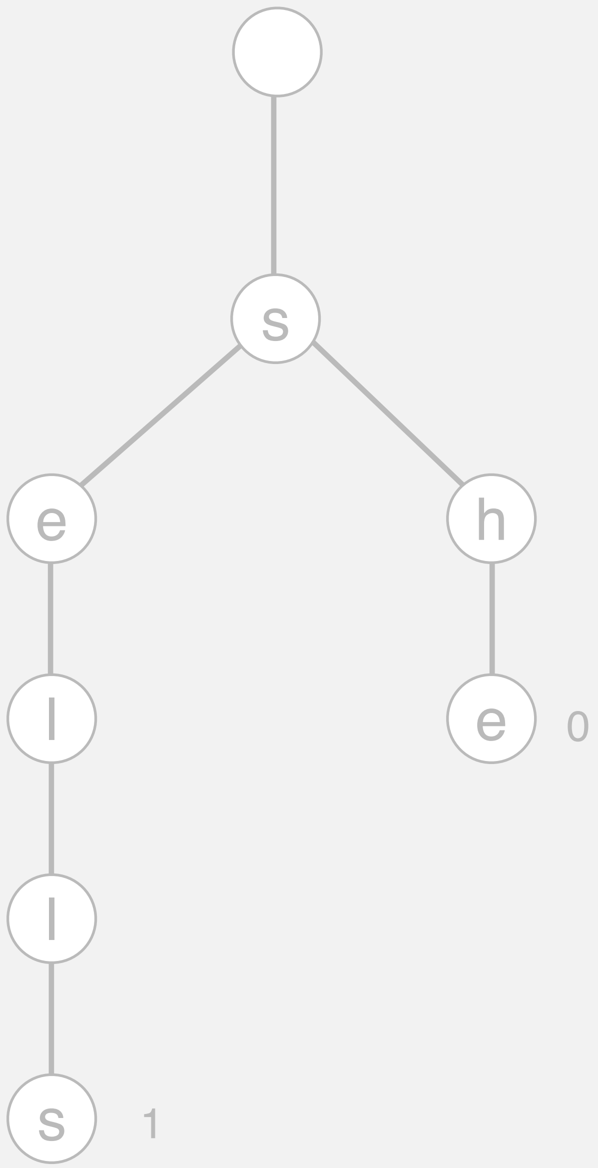


# Trie construction demo

she

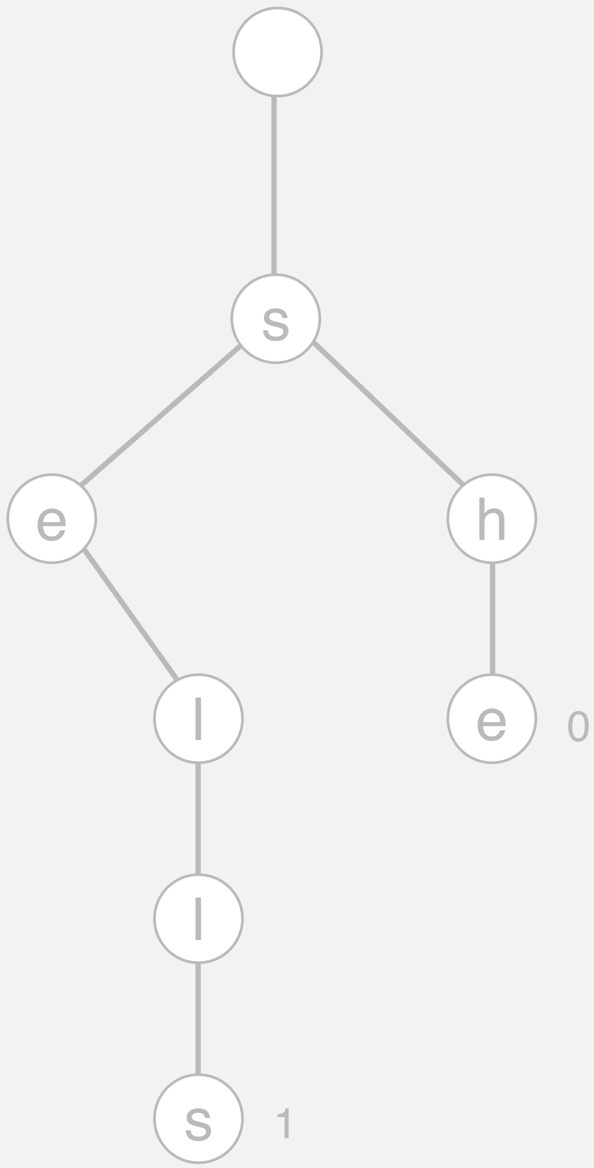
sells

trie



# Trie construction demo

she
sells
trie

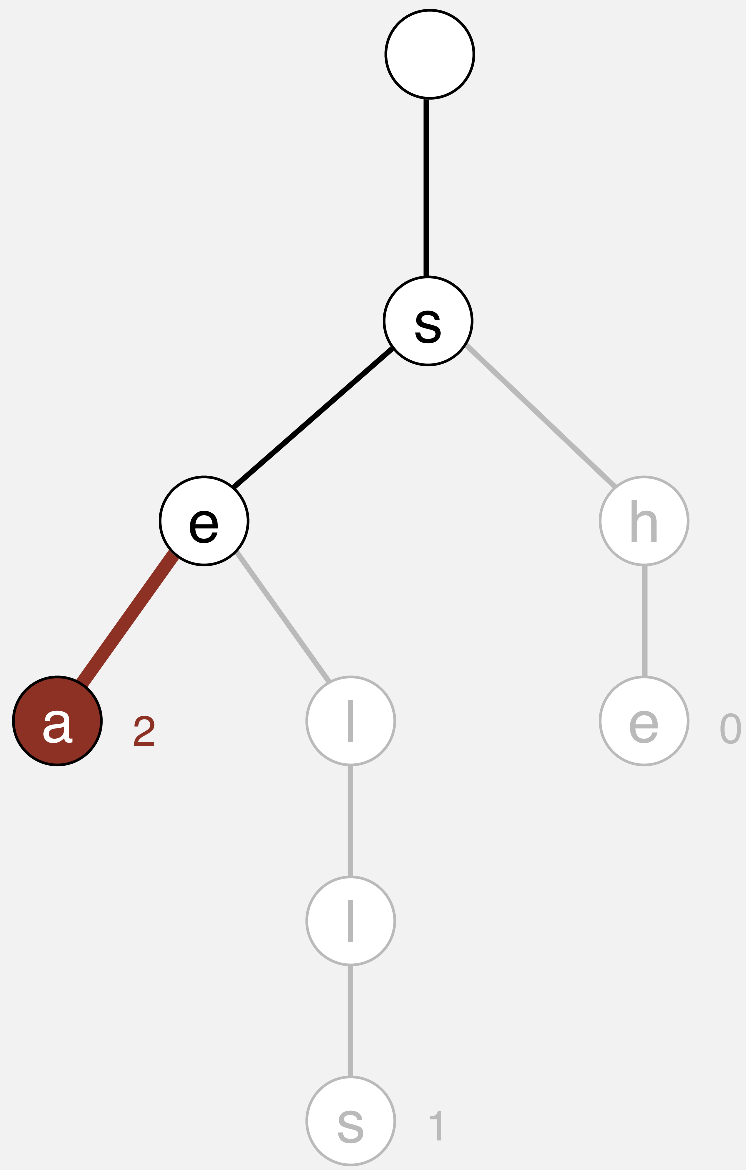


# Trie construction demo

she

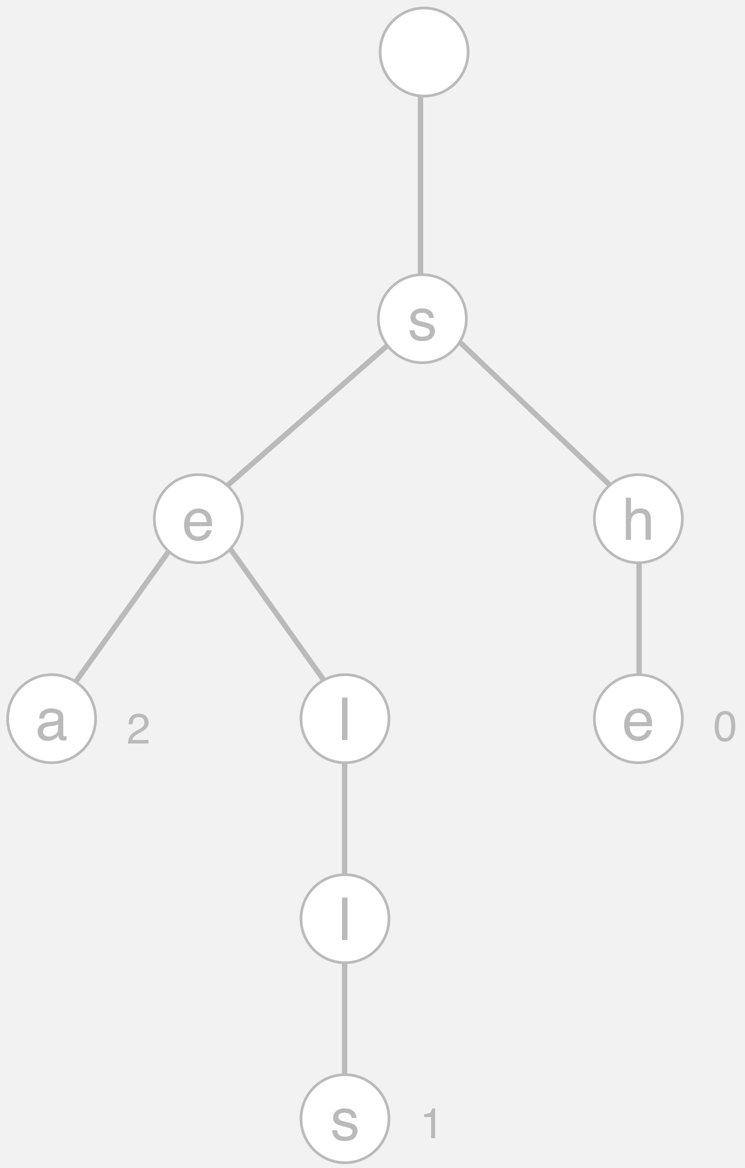
sells

put("sea", 2)



# Trie construction demo

she
sells
sea
trie



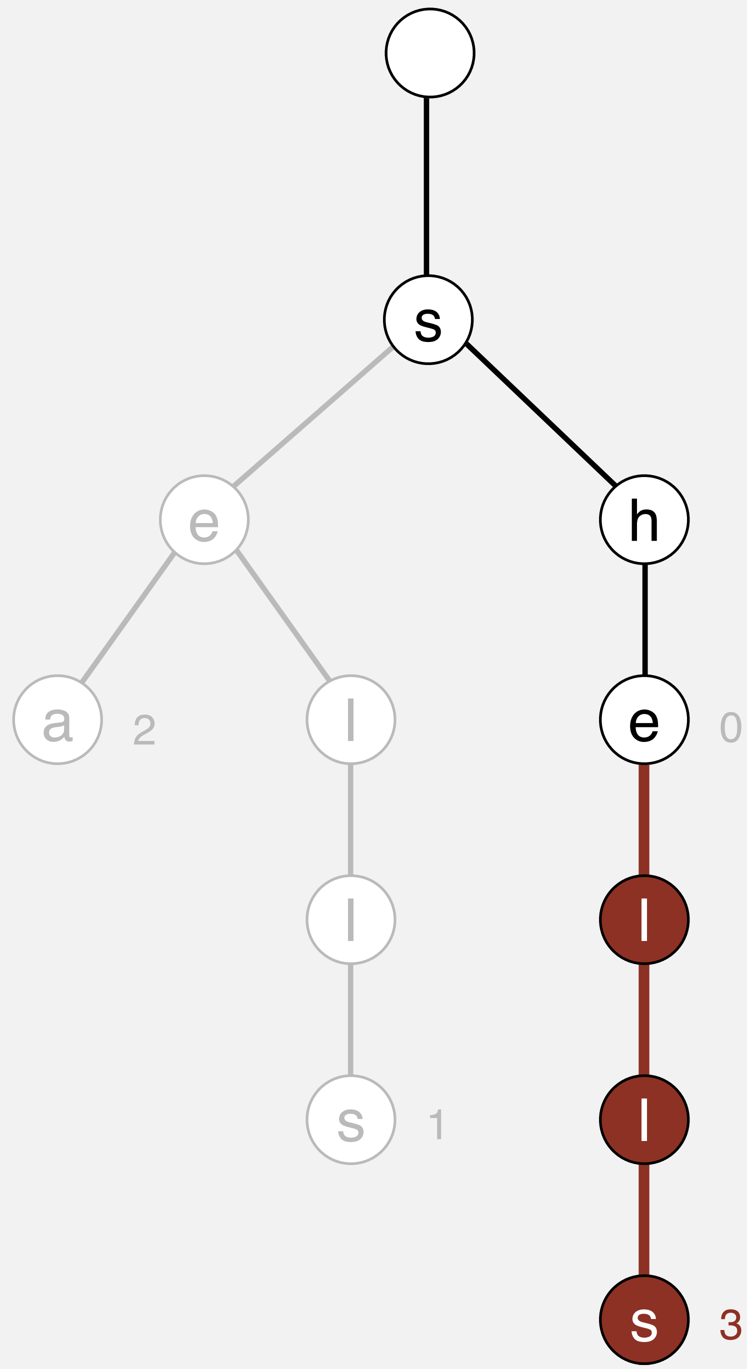
# Trie construction demo

she

sells

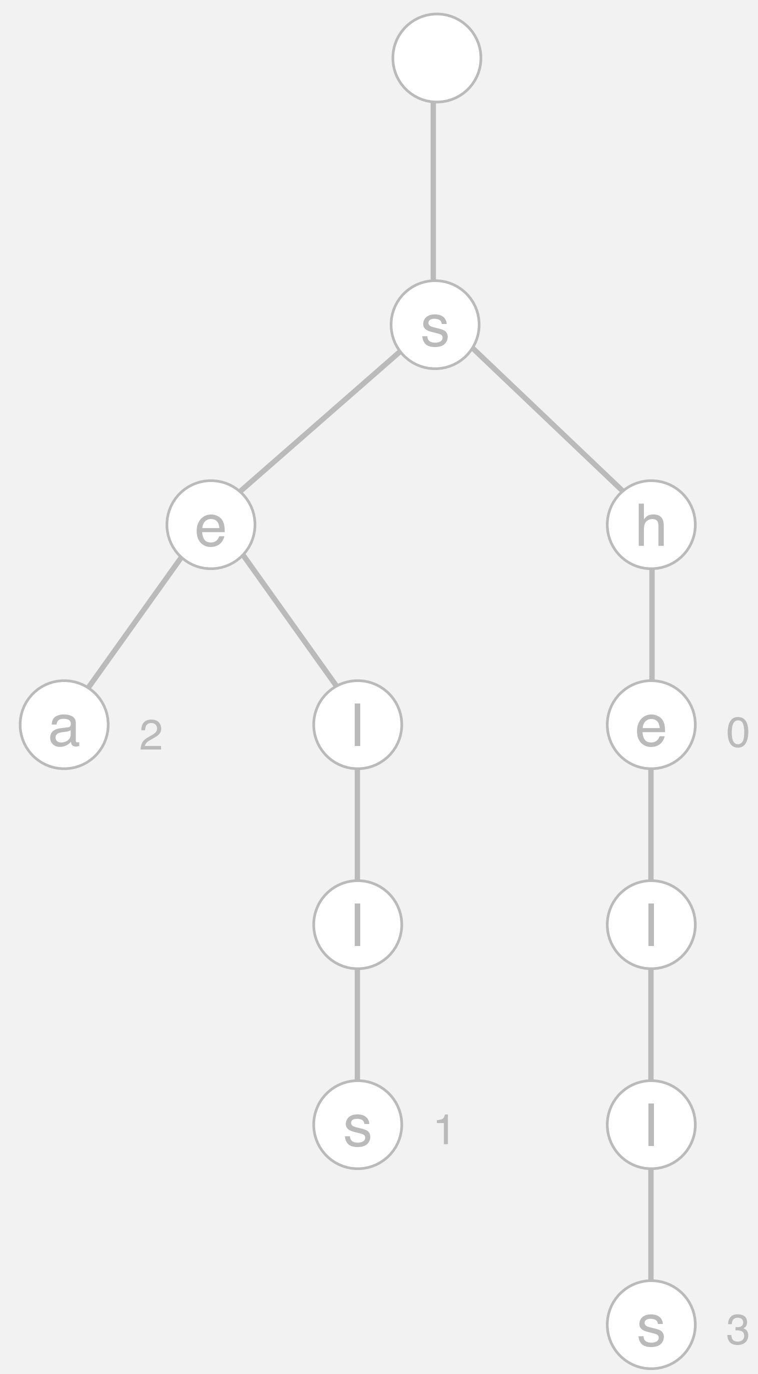
sea

put("shells", 3)



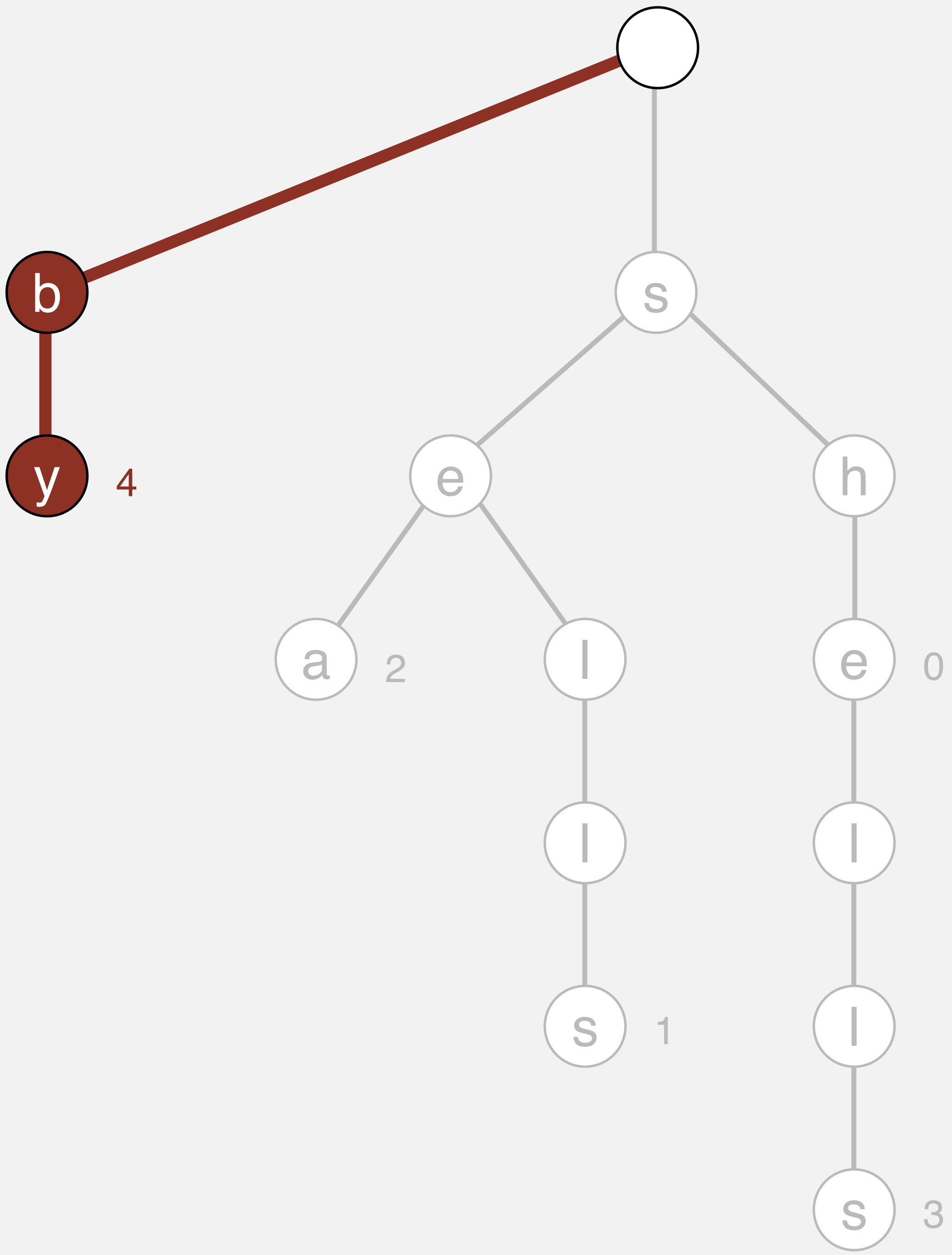
# Trie construction demo

she
sells
sea
trie



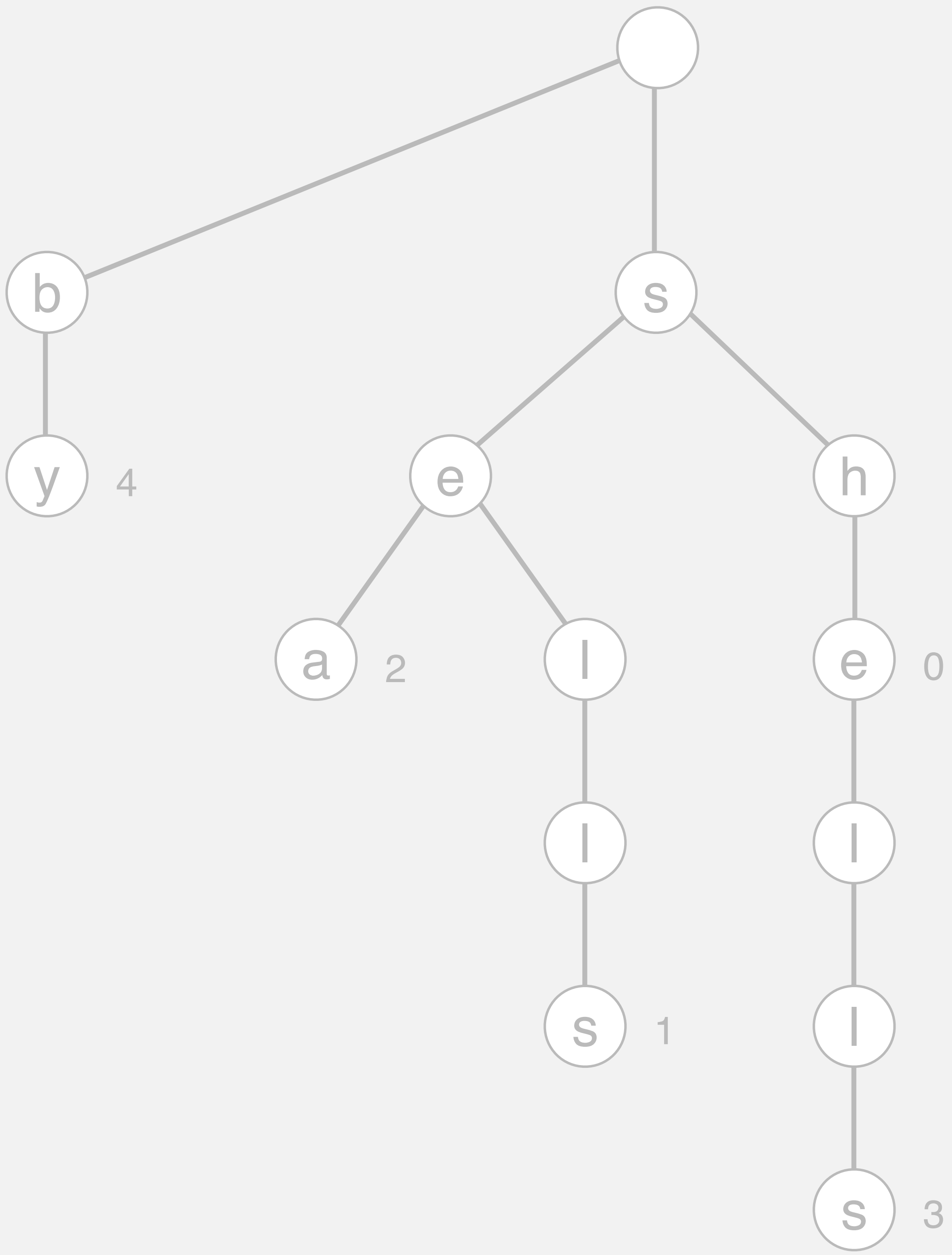
# Trie construction demo

she
sells
sea
put("by", 4)



# Trie construction demo

she
sells
sea
by
trie



# Trie construction demo

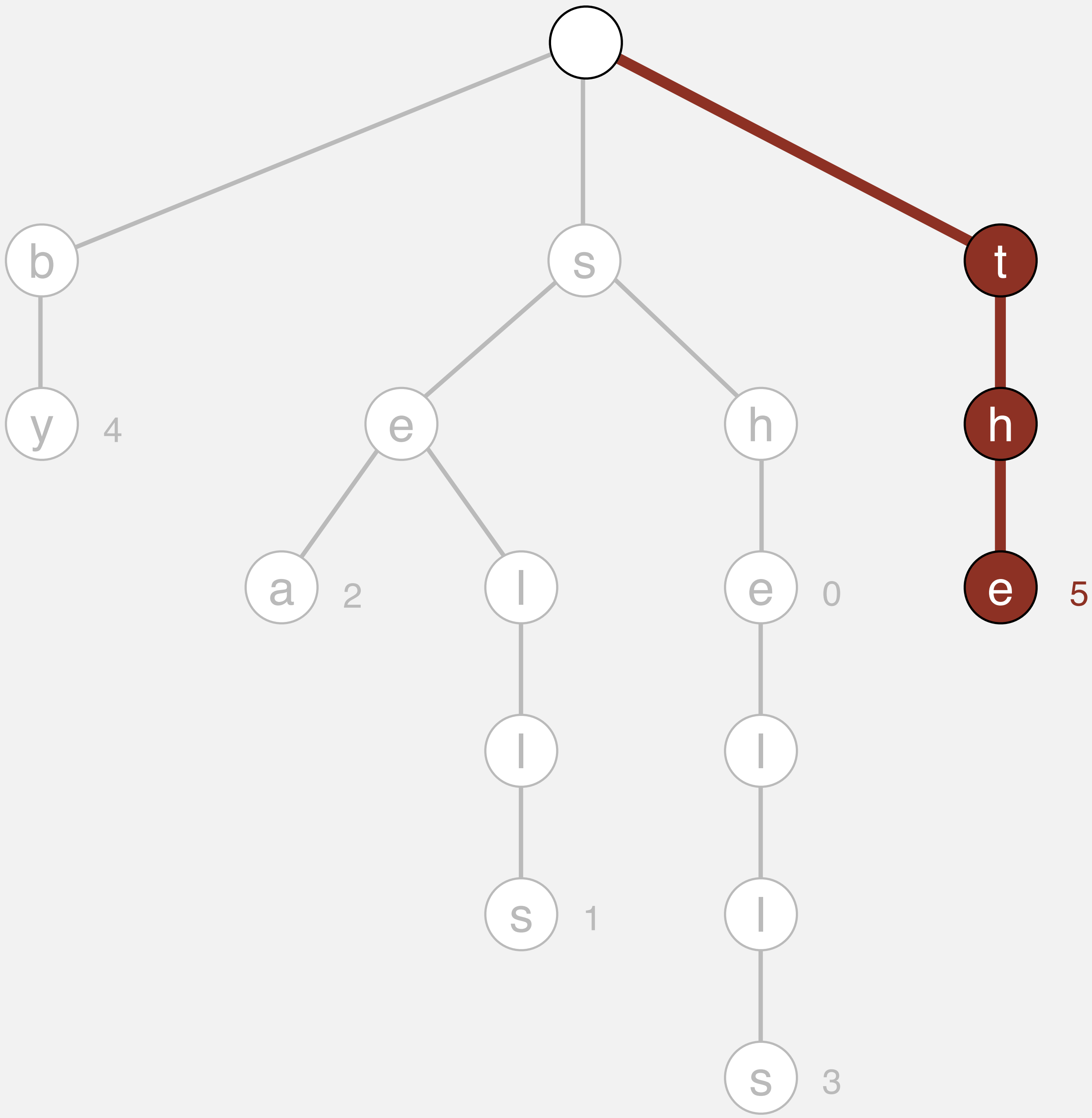
she

sells

sea

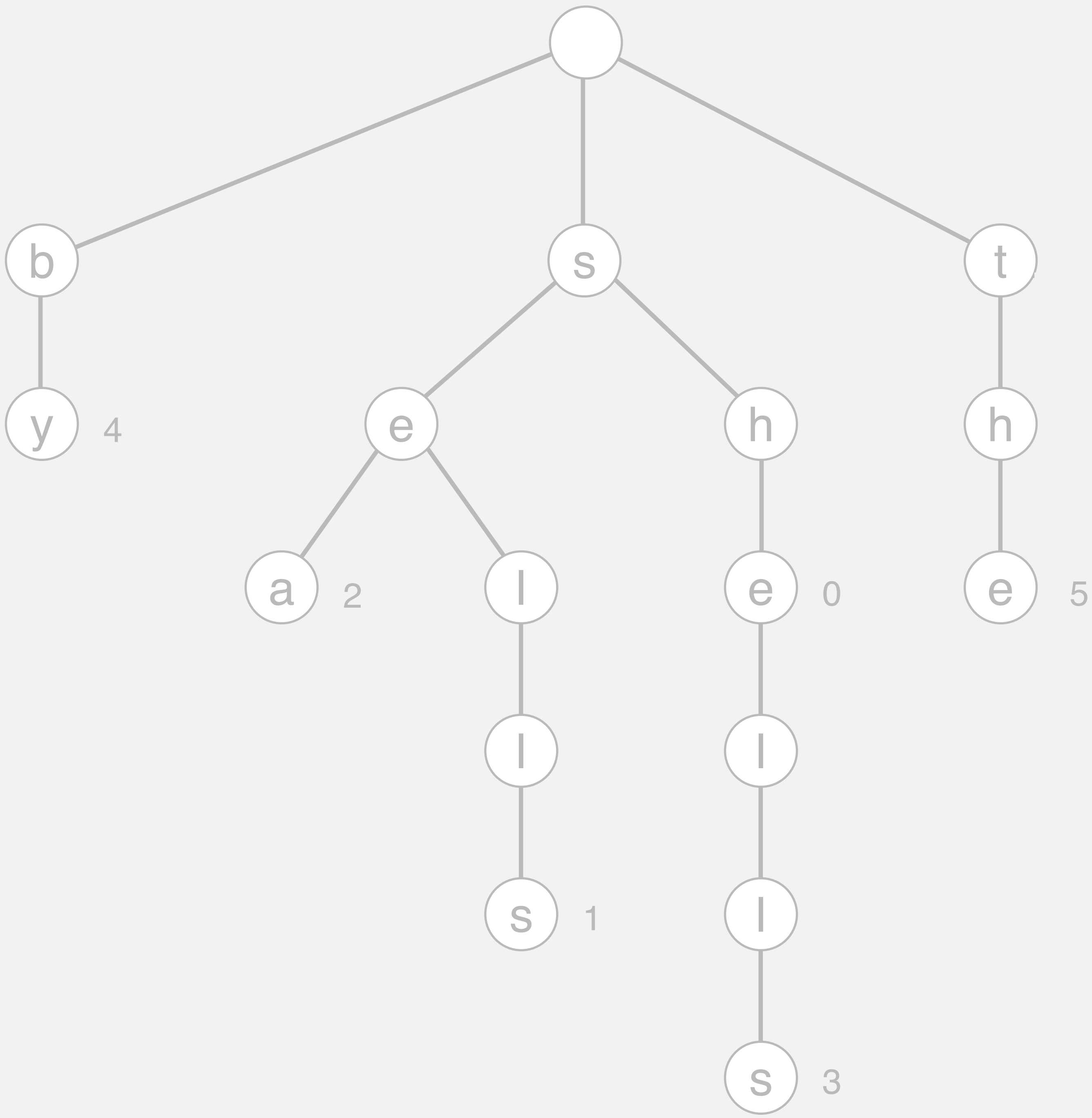
by

put("the", 5)



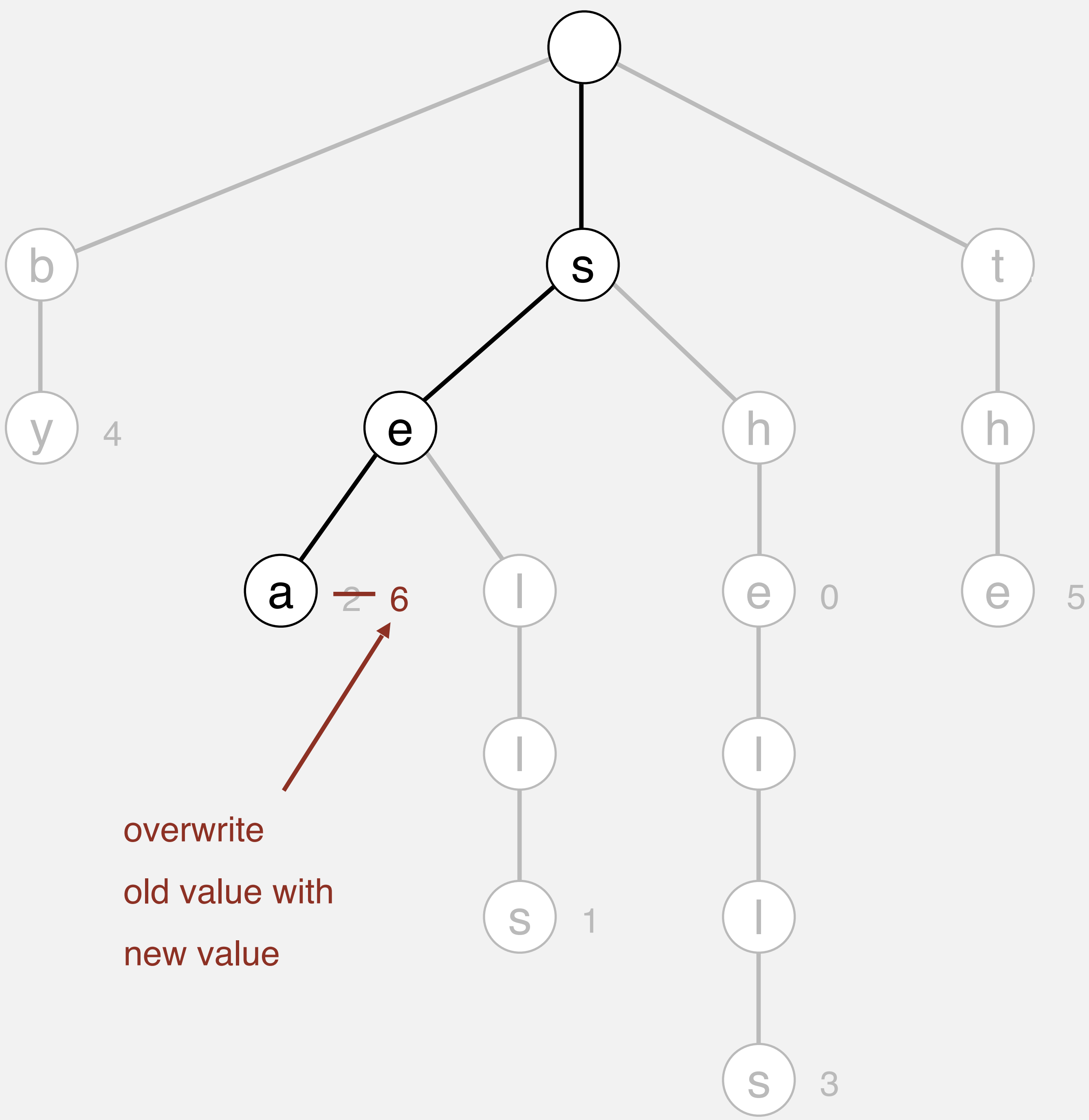
# Trie construction demo

she
sells
sea
by
the
trie



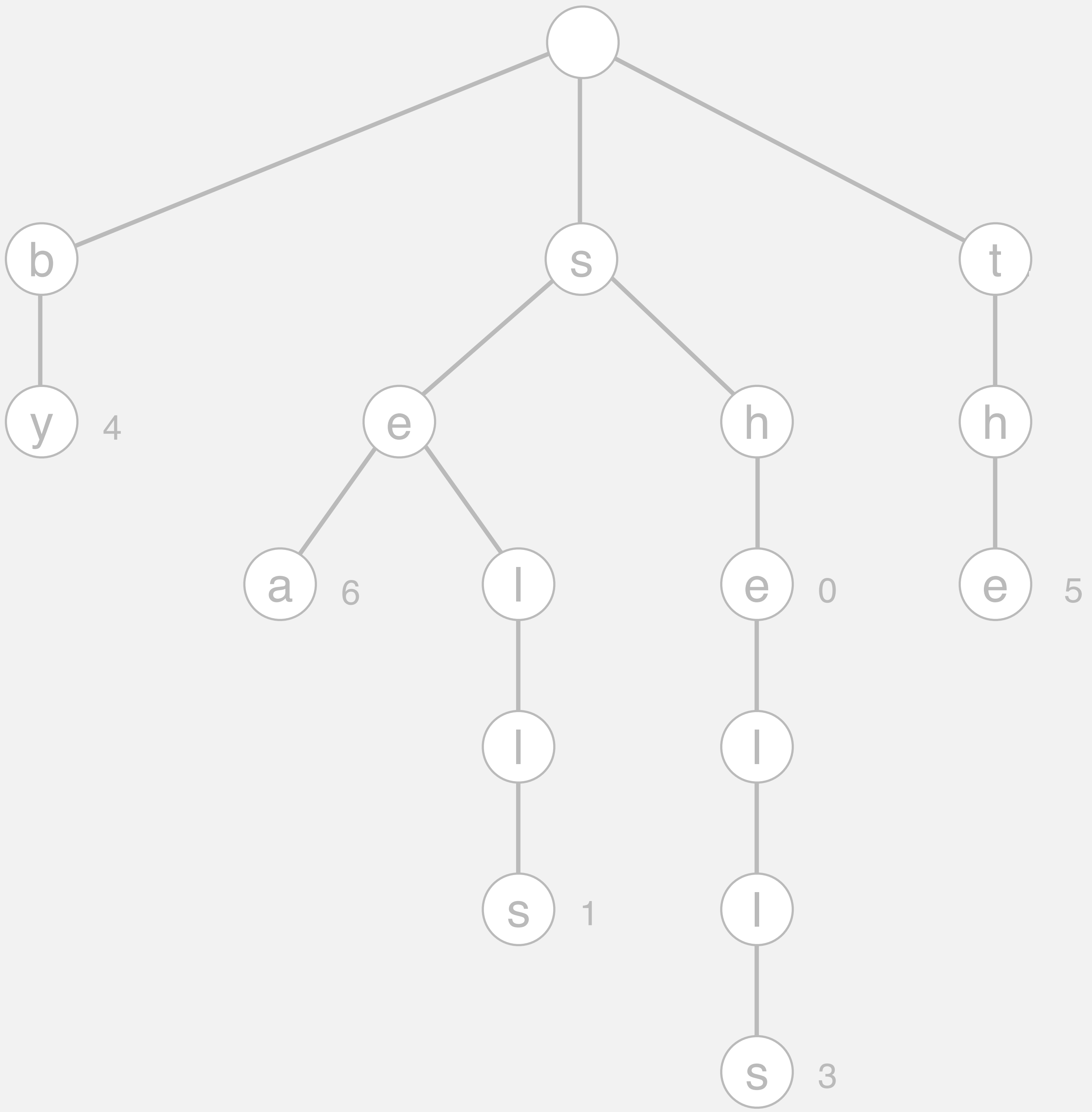
# Trie construction demo

**put("sea", 6)**



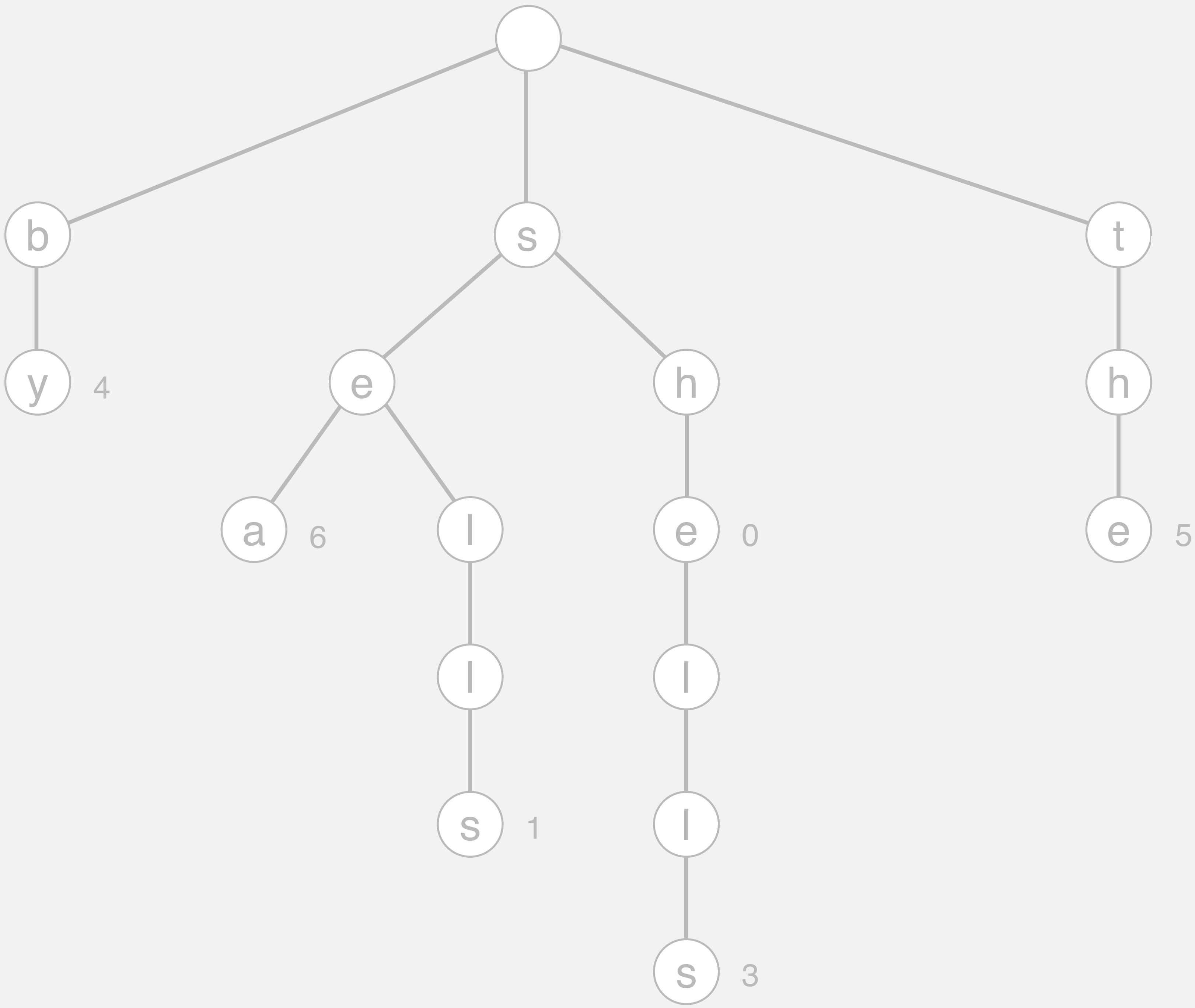
# Trie construction demo

**trie**



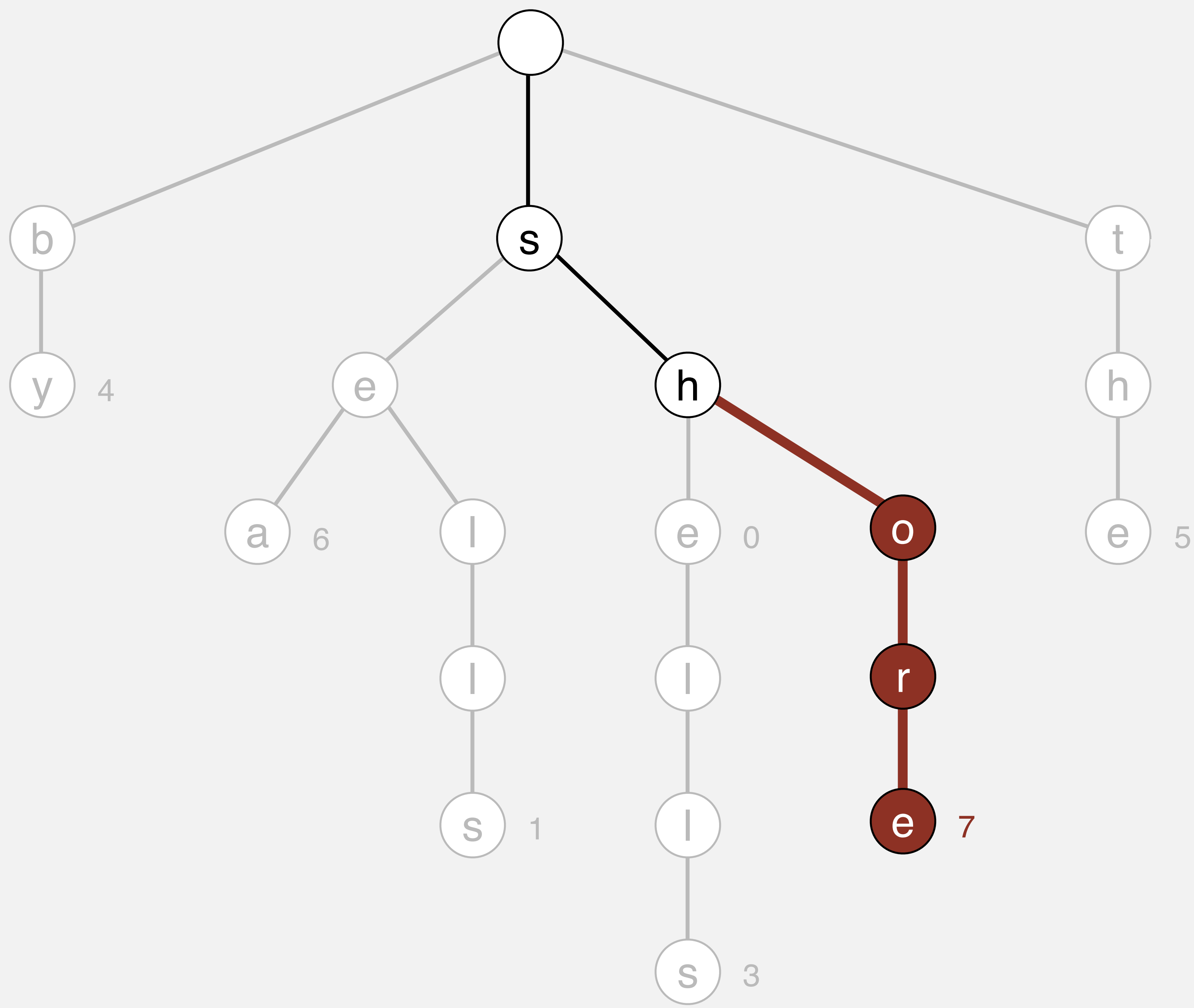
# Trie construction demo

trie



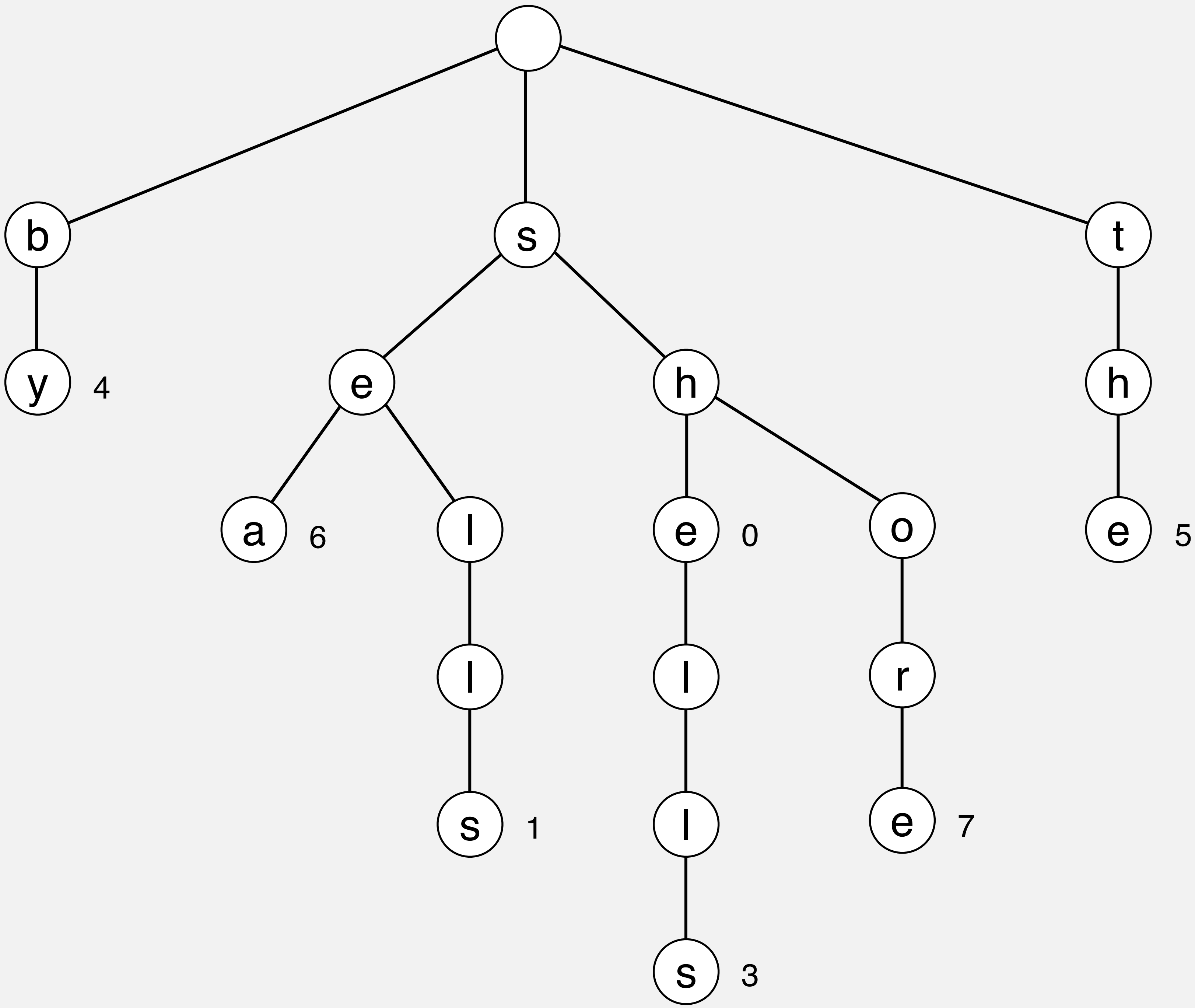
# Trie construction demo

she
sells
sea
by
the
put("shore", 7)



# Trie construction demo

she
sells
sea
by
the
shore
trie

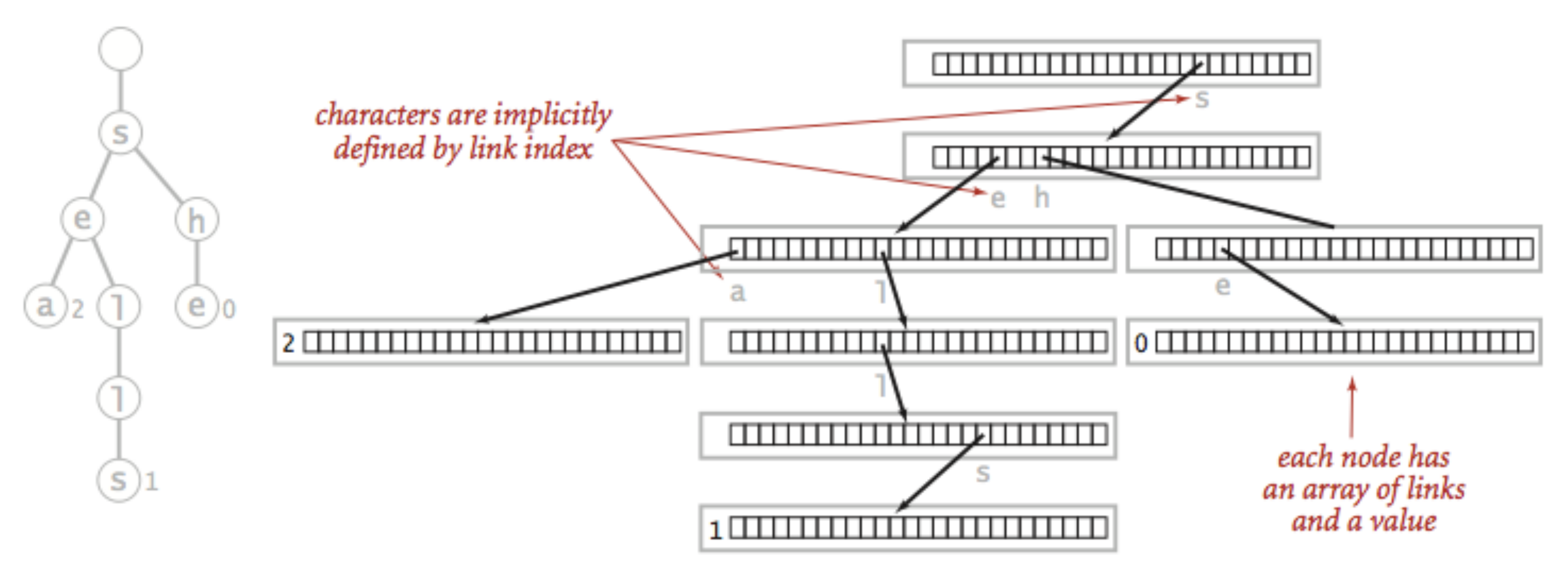


# Trie representation: implementation

Node. A value, plus pointers to $R$ nodes.

```
class Node
{
    int value;
    Node* next[R];
}
```

A child node for each character in Alphabet.
No need to search for character, but a pointer reserved for each character in memory




Trie representation (R = 26)

# R-way trie: implementation

```
#define R 256          ← extended ASCII

Node root;

put(&root, key, val, 0);

void put(Node*& x, char* key, int val, int d)
{
   if (x == NULL)
         x = getNode();
   if (d ==strlen(key)) {x->value = val; return;}
   char c = key[d];
   put(x->next[c], key, val, d+1);
}

⋮
```

# R-way trie: implementation (continued)

```
Node* getNode(){
    Node* pNode = NULL;
    pNode = new Node;
    if (pNode){
        for (int i = 0; i < R; i++)
            pNode->next[i] = NULL;
    }
    return pNode;
}
```

# R-way trie: implementation (continued)

```
int get(Node* x, char key, int d)
   {
      if (x == NULL) return -1; //-1 refers no match
      if (d == strlen(key))
           return x->value;
      char c = key[d];
      return get(x->next[c], key, d+1);
   }

}
```

# Trie performance

Search hit. Need to examine all $L$ characters for equality.

Search miss.

- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

Space. R links at each node; $R$ null links at each leaf.
(but sublinear space possible if many short strings share common prefixes)

Bottom line. Fast search hit and even faster search miss, but wastes space.

# Tries cont.

- Prefix Vs. Suffix.

Ex. “computer”.

  - Prefix:(c, co, com).
  - Suffix: (r, er, ter)
- Each node in this tree structure corresponds to a prefix of some strings of the set.
- If the same prefix occurs several times, there is only one node to represent it.
- The root of the tree structure is the node corresponding to the empty prefix.

# String Termination

Strings are sequences of characters from some alphabet. But for use in the computer, we need an important further information: how to recognize where the string ends.

There are two solutions for this:

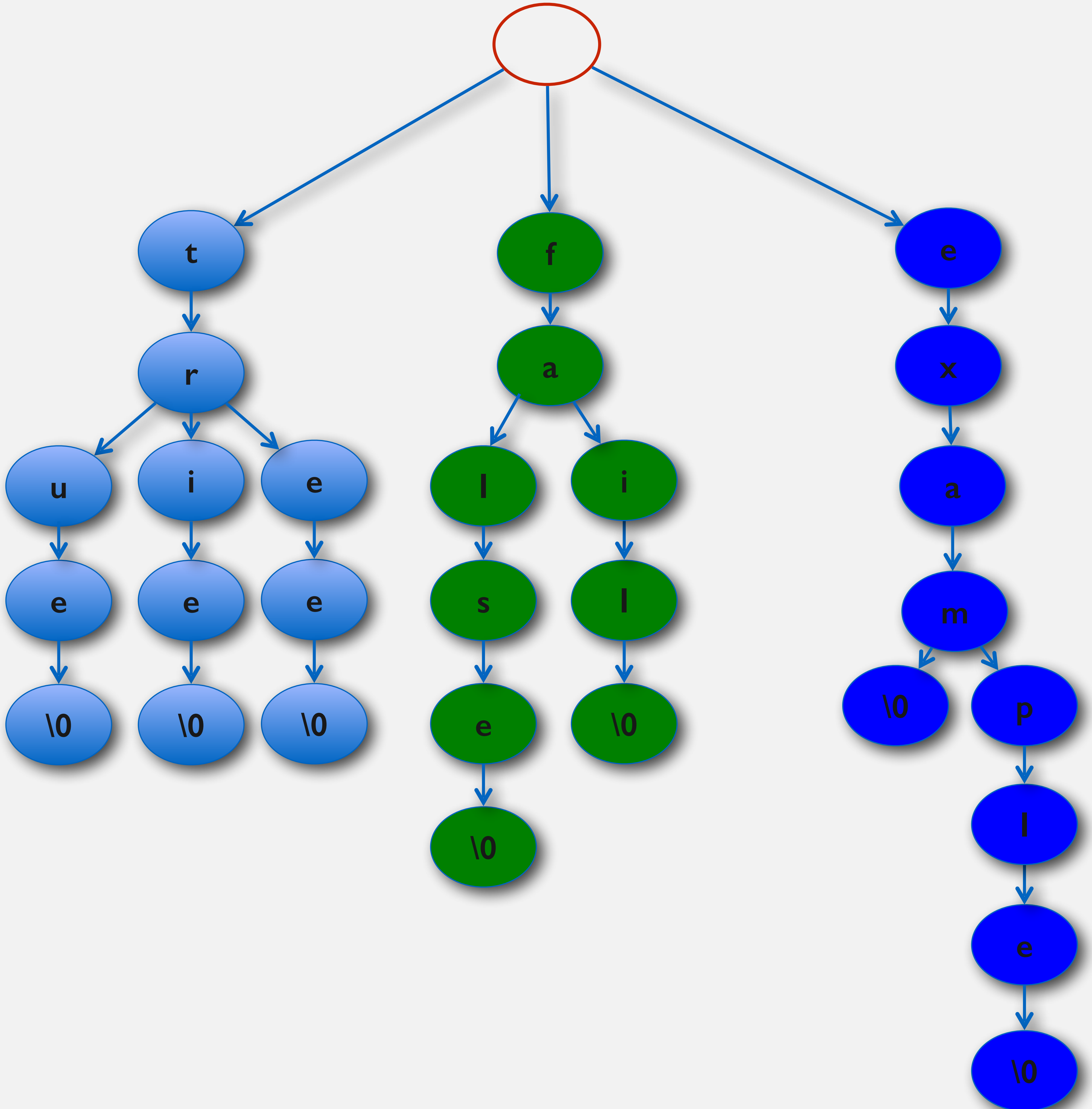
1. We can have an explicit termination character, which is added at the end of each string, but may not occur within the string “\0” (ASCII code 0) , or
2. We can store together with each string its length.

# String Termination

- The use of the special termination character '\0' has a number of advantages in simplifying code.
- It has the disadvantage of having one reserved character in the alphabet that may not occur in strings.
- There are many nonprintable ASCII codes that should never occur in a text and '\0' is just one of them.
- There are also many applications in which the strings do not represent text, but, for example, machine instructions.

Strings:

- exam
- example
- fail
- false
- tree
- trie
- true



# Find, Insert and Delete

To perform a *find* operation in this structure:

1. Start in the node corresponding to the empty prefix.
2. Read the query string, following for each read character the outgoing pointer corresponding to that character to the next node.
3. After we read the query string, we arrived at a node corresponding to that string as prefix.
4. If the query string is contained in the set of strings stored in the trie, and that set is prefix-free, then this node belongs to that unique string.

# Find, Insert and Delete

To perform a *find* operation in this structure:

1. Start in the node corresponding to the empty prefix.
2. Read the query string, following for each read character the outgoing pointer corresponding to that character to the next node.
3. After we read the query string, we arrived at a node corresponding to that string as prefix.
4. If the query string is contained in the set of strings stored in the trie, and that set is prefix-free, then this node belongs to that unique string.

To perform an *insert* operation in this structure:

1. Perform *find*
2. Any time we encounter a nil pointer we create a new node

# Find, Insert and Delete

To perform a *find* operation in this structure:

1. Start in the node corresponding to the empty prefix.
2. Read the query string, following for each read character the outgoing pointer corresponding to that character to the next node.
3. After we read the query string, we arrived at a node corresponding to that string as prefix.
4. If the query string is contained in the set of strings stored in the trie, and that set is prefix-free, then this node belongs to that unique string.

To perform an *insert* operation in this structure:

1. Perform *find*
2. Any time we encounter a nil pointer we create a new node

To perform a *delete* operation in this structure:

1. Perform *find*
2. Delete all nodes on the path from '\0' to the root of the tree unless we reach a node with more than 1 child

# String symbol table implementations cost summary

| | character accesses (worst case) | | | |
|---|---|---|---|---|
| implementation | Search hit | Search miss | insert | space (references) |
| hashing (separate chaining) | NL | NL | 1 | N |
| R-way trie | L | L | L | RNw |

N = number of entries, L= key length,
R= alphabet size, w= average key length

## R-way trie.

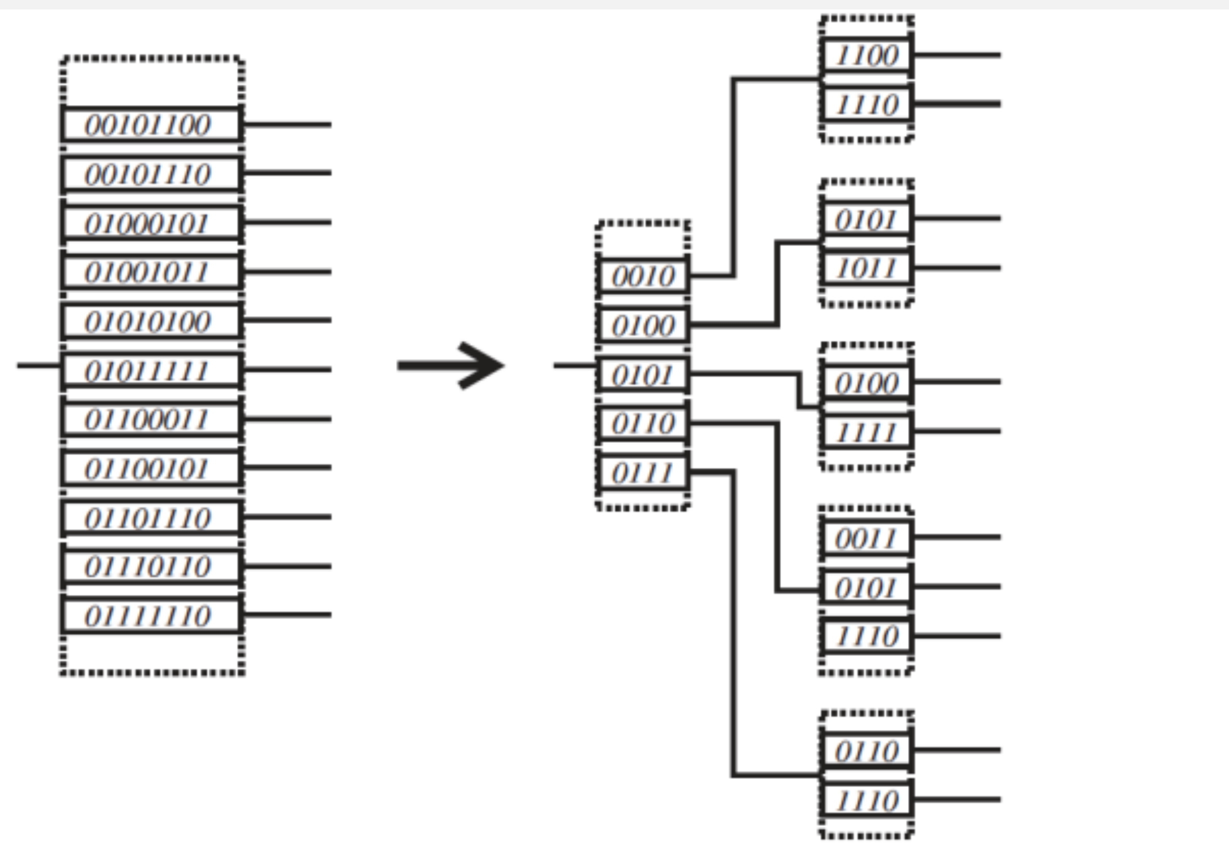
- Method of choice for small $R$.
- Too much memory for large $R$.

Challenge. Use less memory, e.g., 65,536-way trie for Unicode!

# Alphabet Size

- The problem here is the dependence on the size of the alphabet which determines the size of the nodes.
- There are several ways to reduce or avoid the problem of the alphabet size.
  - A simple method, is to replace the big nodes by linked lists of all the entries that are really used.
  - Another way to avoid the problem with the alphabet size R is alphabet reduction. We can represent the alphabet R as set of k -tuples from some direct product $R_1 x R_2 \ldots x R_k$

For the standard ASCII codes, we can break each 8-bit character by two 4-bit characters, which reduces the node size from 256 pointers to 16 pointers




Alphabet Reduction: Instead of One Node with 256 Entries, of Which Only 11 Are Used, We Have Five Nodes with 16 Entries Each

# Other Reduction Techniques

- The trie structure with balanced search trees as nodes
- The ternary trie structure: nodes are arranged in a manner similar to a binary search tree, but with up to three children. each node contains one character as key and one pointer each for query characters that are smaller, larger, or equal

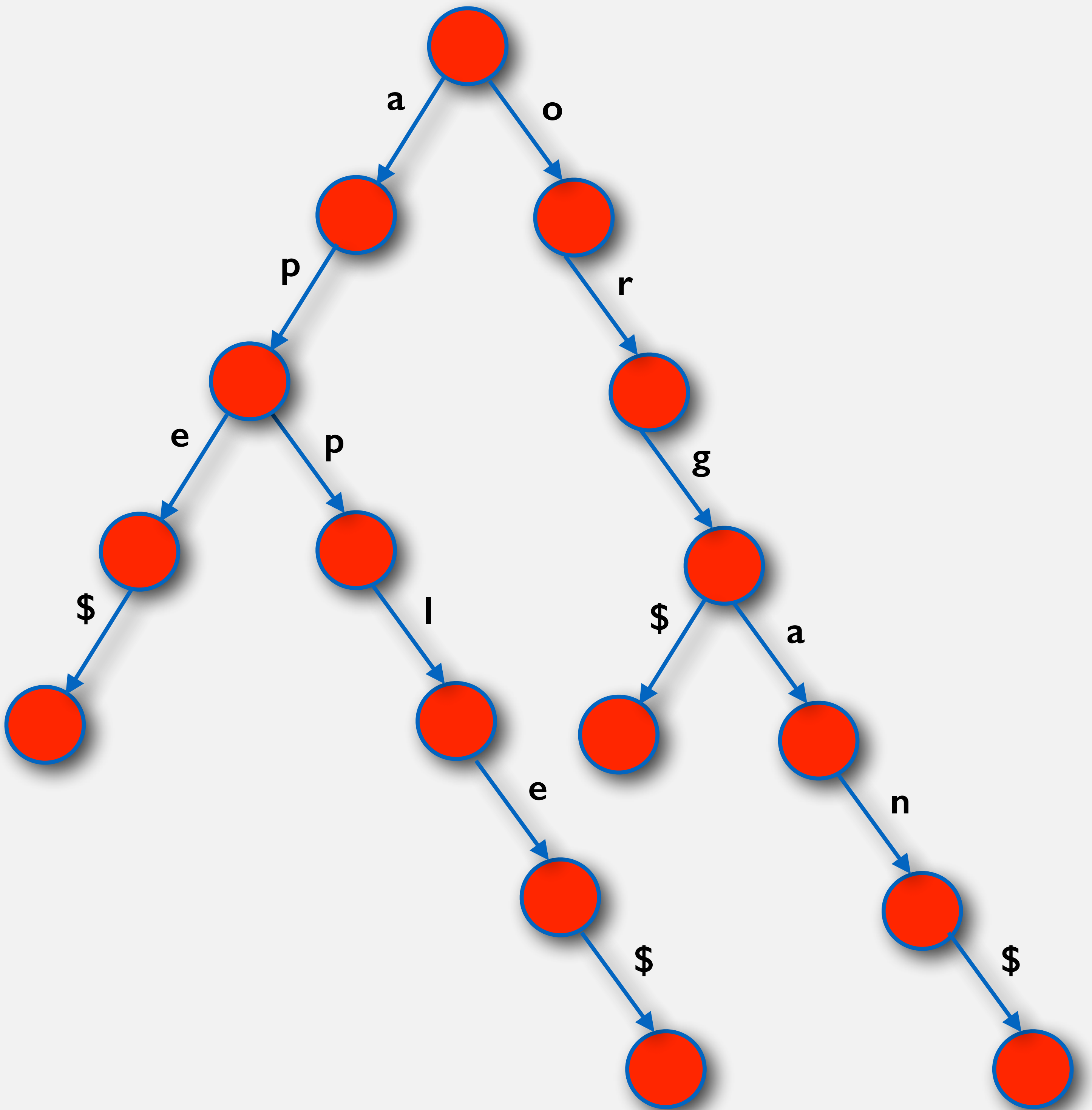
# Patricia Tree *(a.k.a. Compressed Trie)*

- “Practical Algorithm To Retrieve information Coded in Alphanumeric.”
- A path compression trie.
- The path compressed trie contains only nodes with at least two outgoing edges.
  - All internal nodes have >=2 child
- Edges may be labeled with strings instead of single characters.
- The edge labels are represented using the pointer/length string representation. (Again null terminated strings)

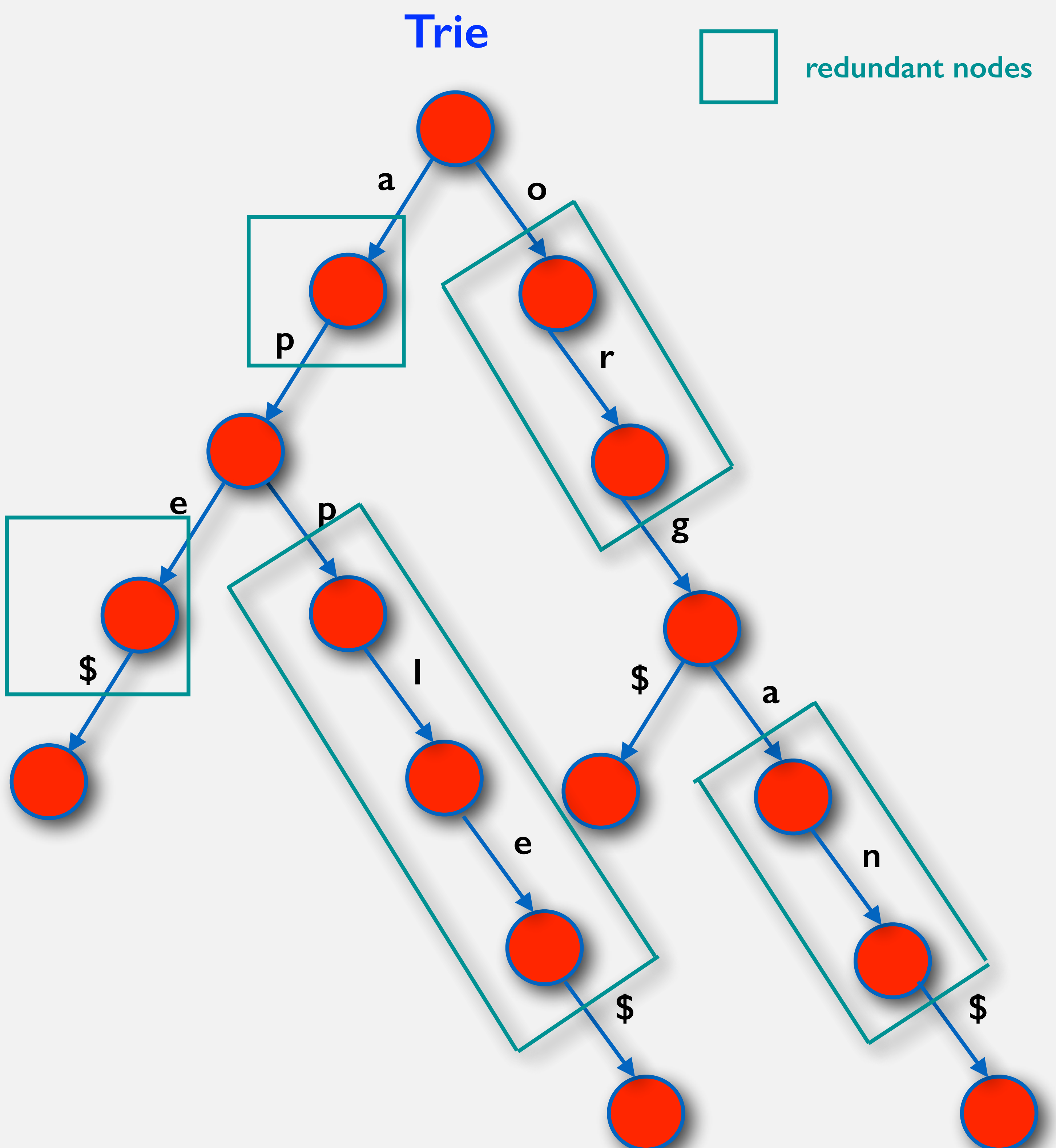
$: end of string symbol

S = {ape, apple, org, organ}

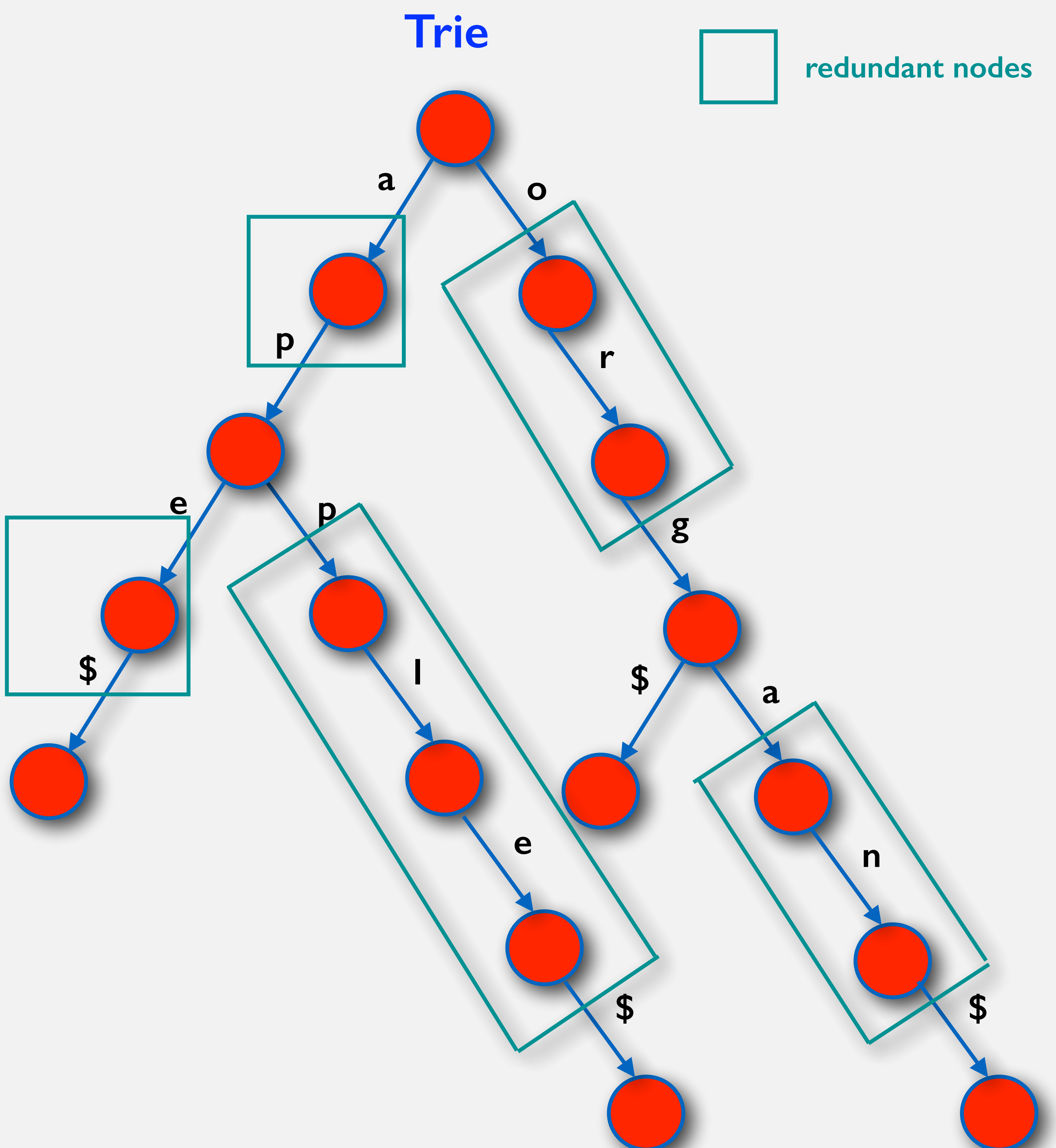
## Trie

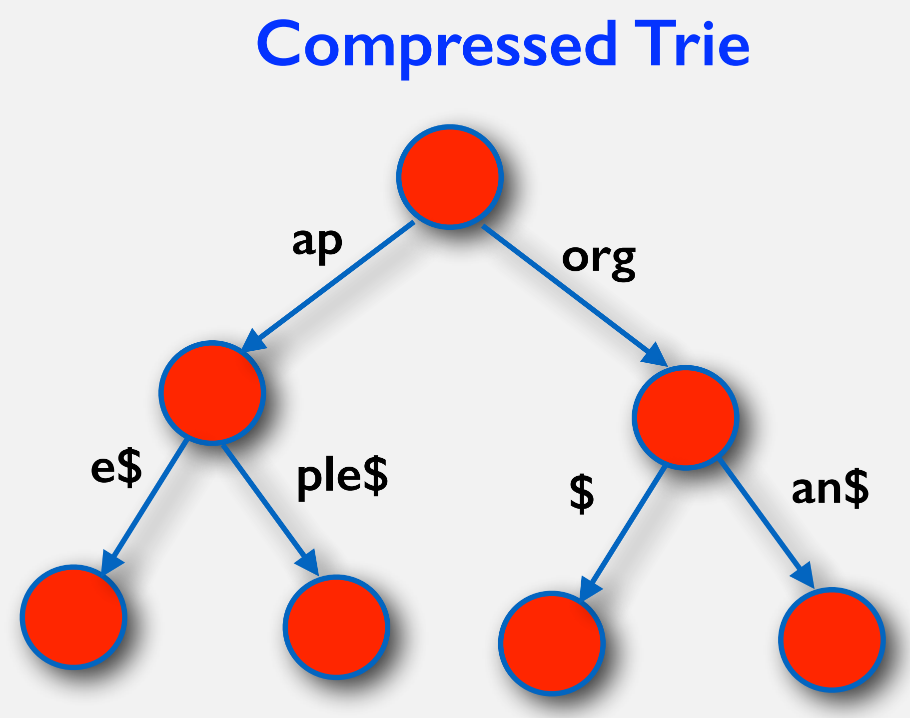


S = {ape, apple, org, organ}



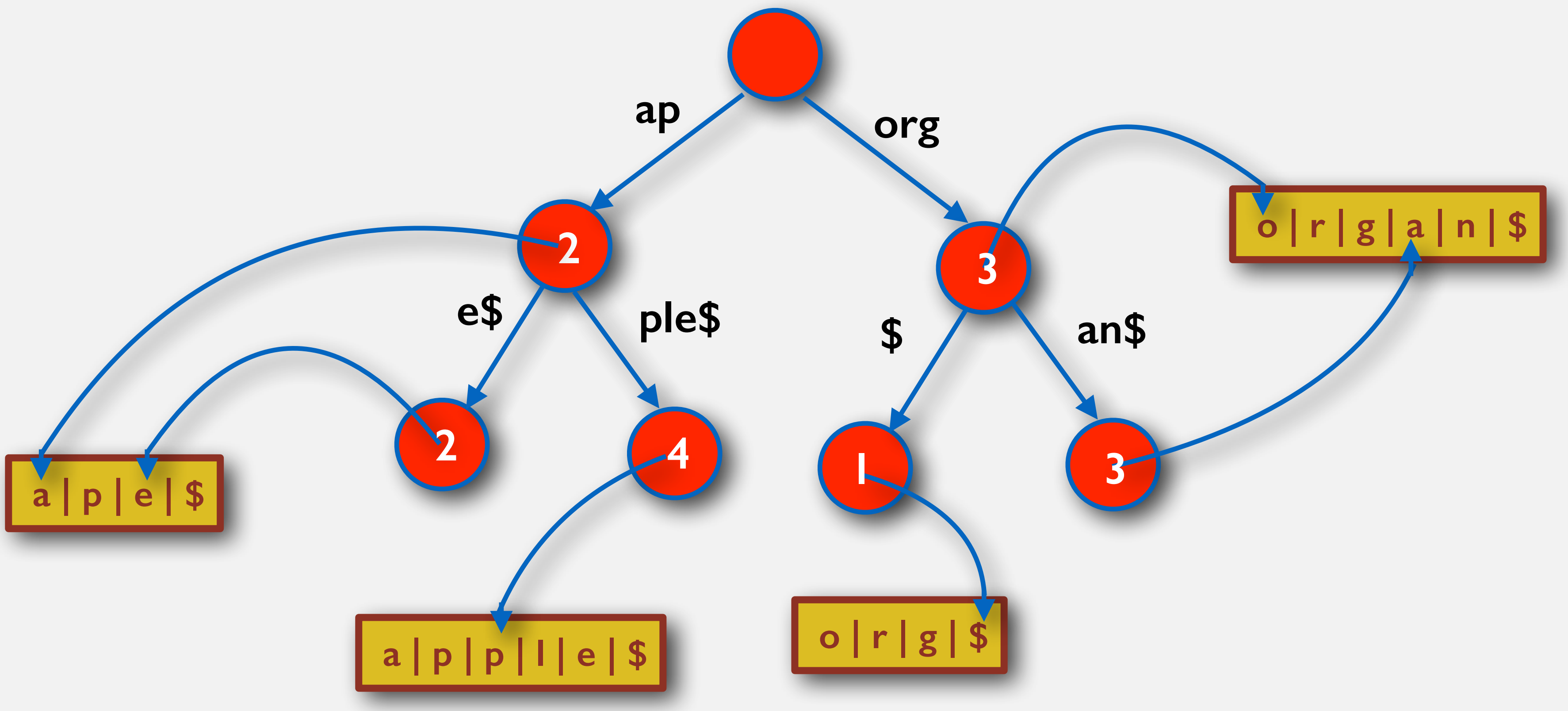
S = {ape, apple, org, organ}



S = {ape, apple, org, organ}

## Compressed Trie

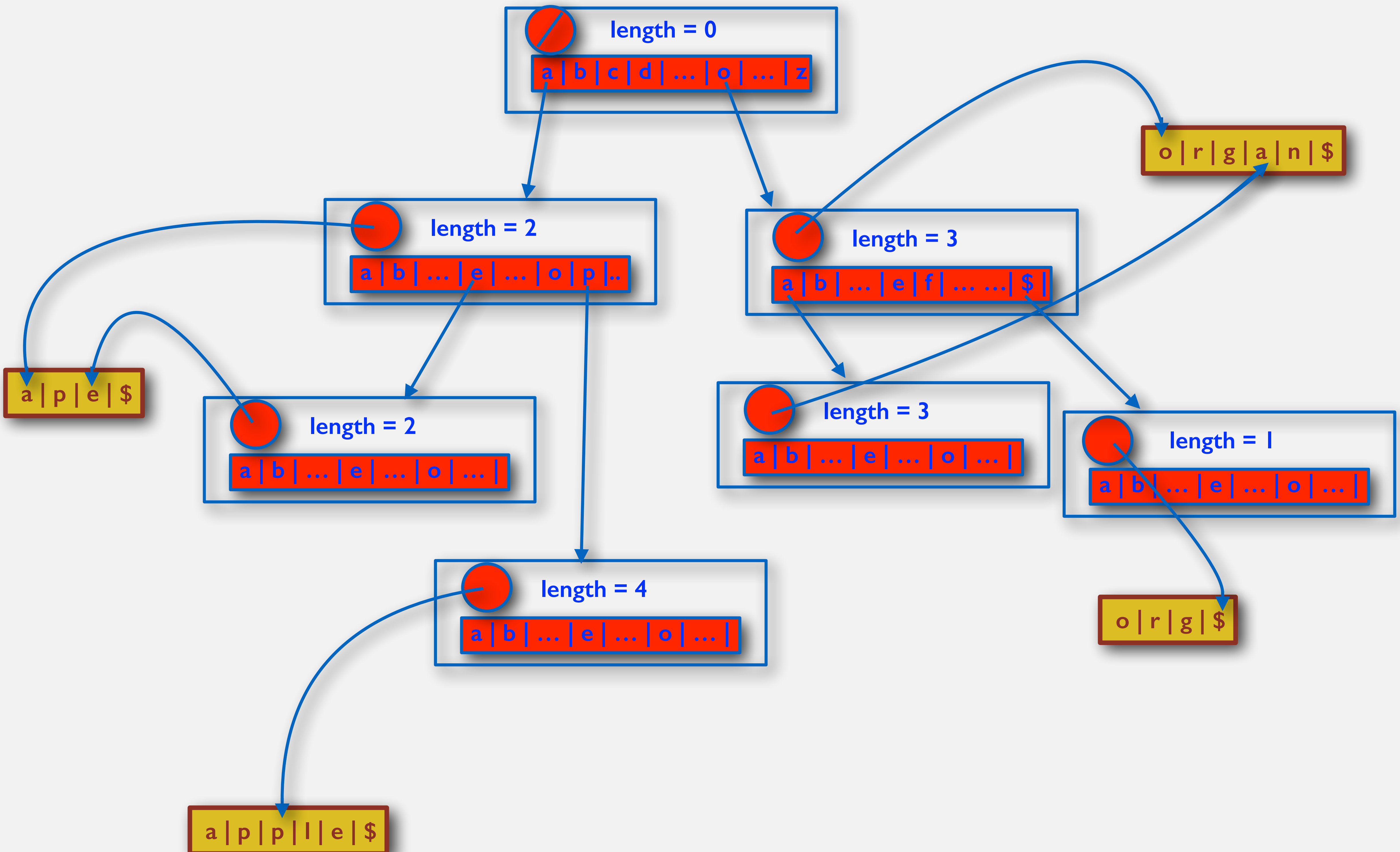



Pointer and length representation of strings is used

S = {ape, apple, org, organ}

Pointer and length representation of strings

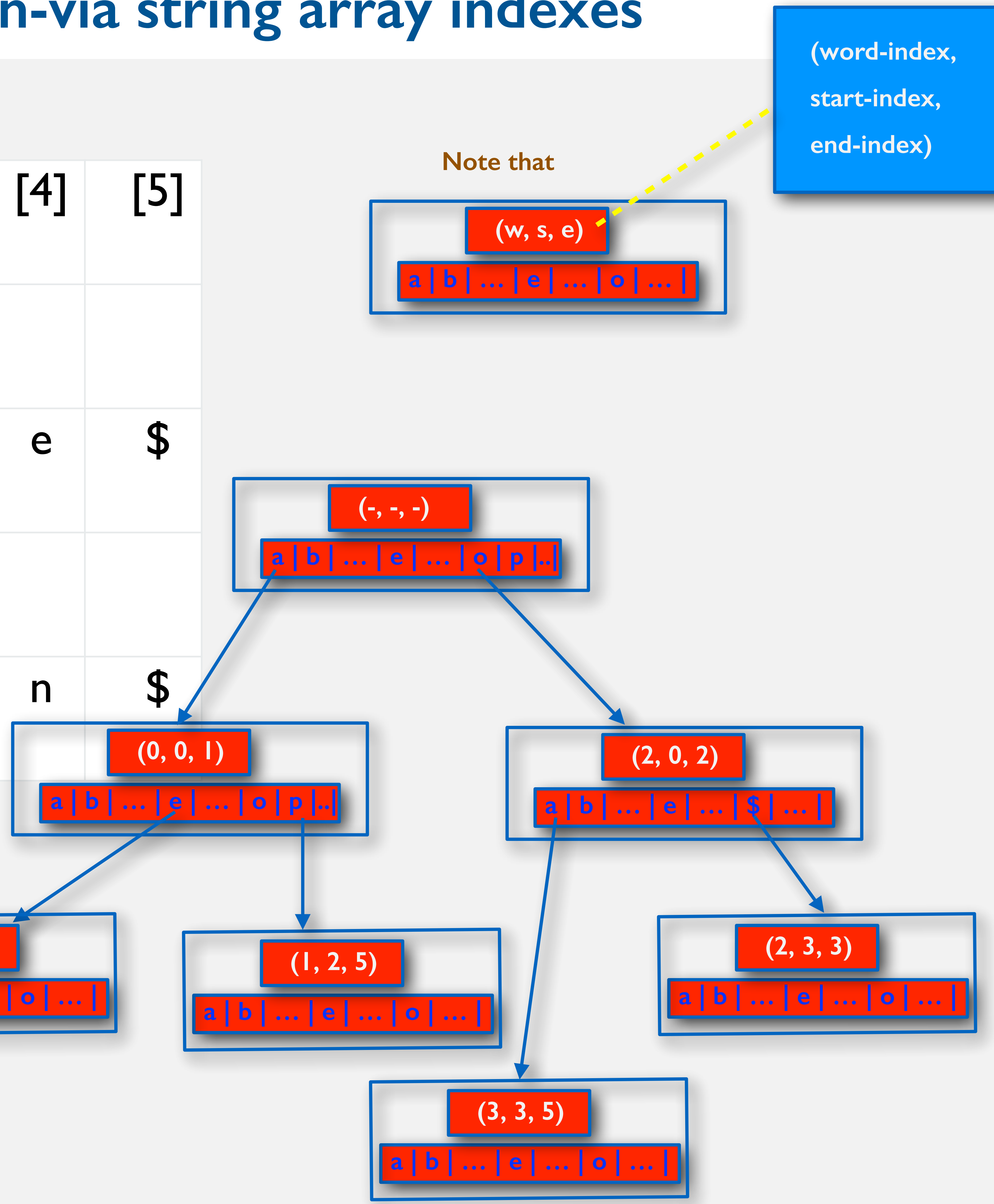
## Compressed Trie



# Alternative Representation-via string array indexes

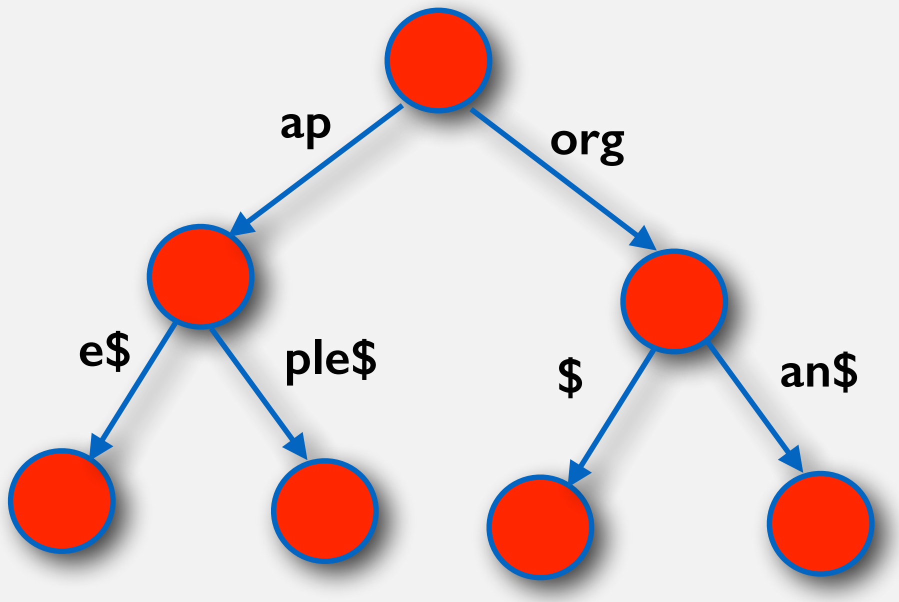
String Array

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |



# Patricia Tree *(a.k.a. Compressed Trie)*

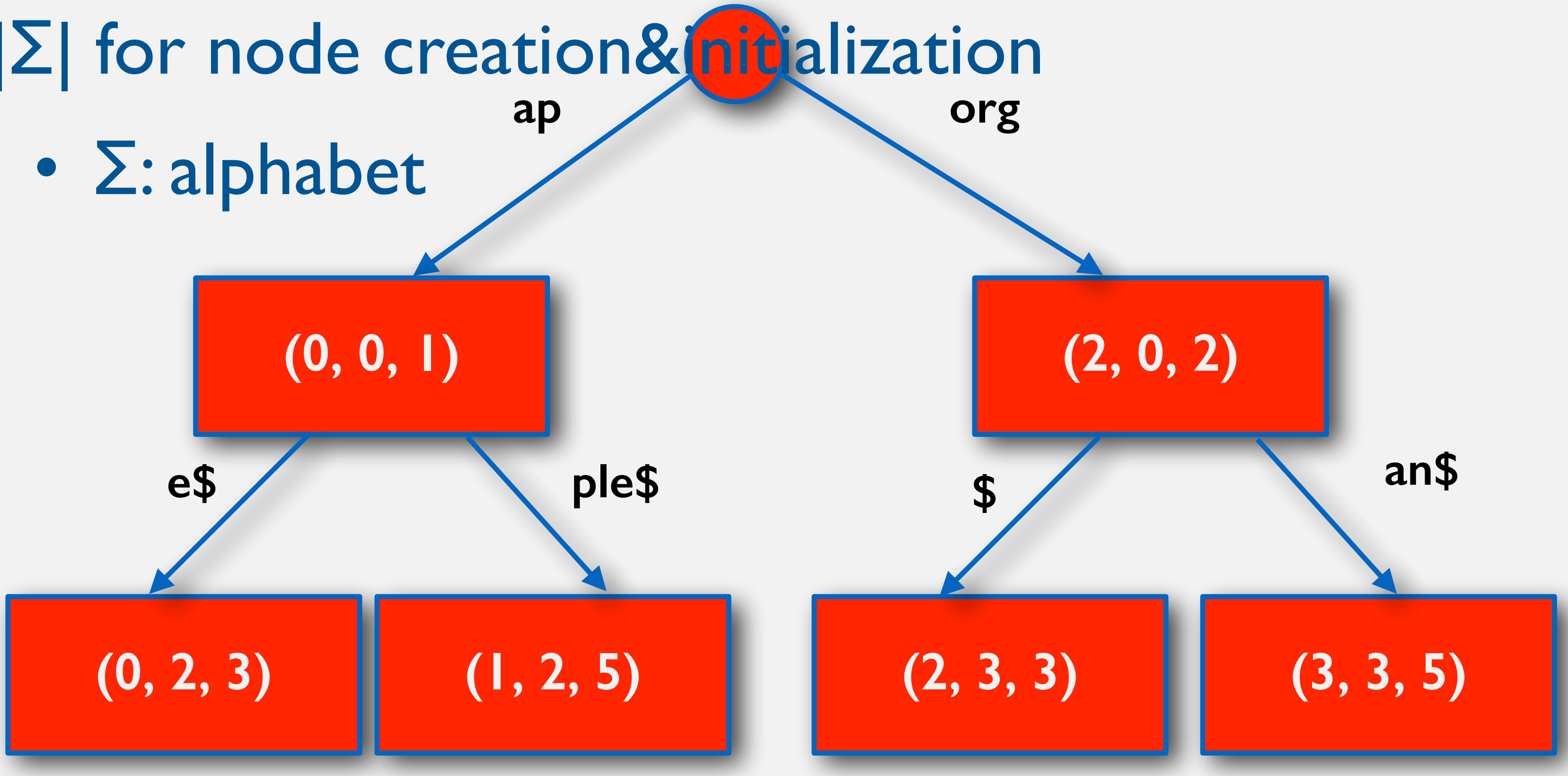
- Searching for a string s in a Patricia Tree:
  - similar to searching in a trie, except that when the search traverses an edge it checks the *edge label* against a substring of s (instead of a single char)
  - if the substring matches, the edge is traversed.
  - if there is a mismatch, the search fails without finding s.
  - if the search uses up all the characters of s, then it is a hit.
    - the leaf reached contains s.
- O(|s|)



# Patricia Tree (*a.k.a. Compressed Trie*)

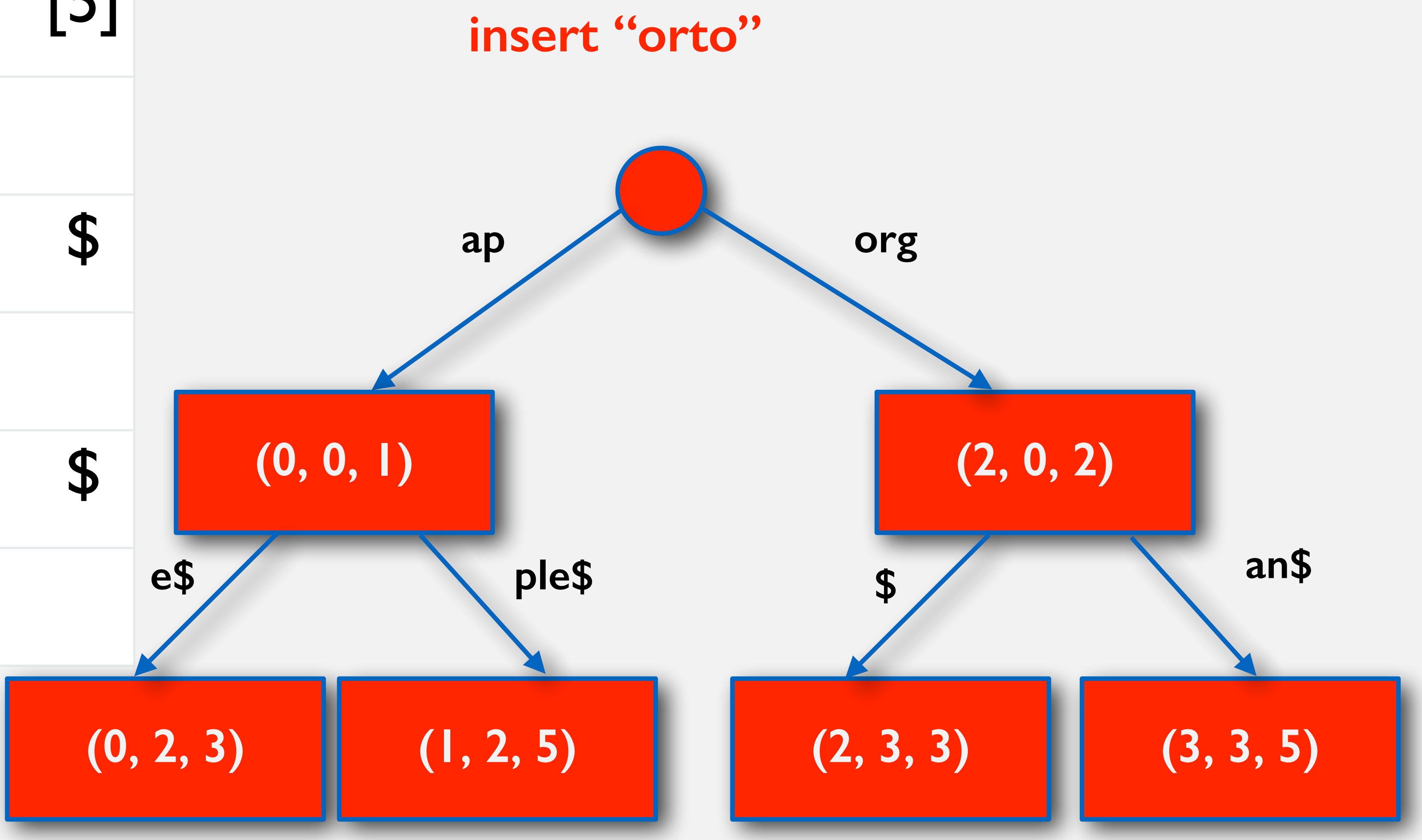
- Inserting a string s in a Patricia Tree:
  - similar to searching up until the point where the search gets stuck
    - since s is not in the tree
  - if the search is over in the middle of an edge, e, then e is split into two new edges, joined by a new node u
    - the remainder of s becomes new edge label, which connects u to the new leaf node
  - if the search is over at a node u, the remainder of s becomes new edge label, which connects u to the new leaf node
  - O(|s|+|Σ|), |s| for search+|Σ| for node creation&initialization
    - Σ: alphabet

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |



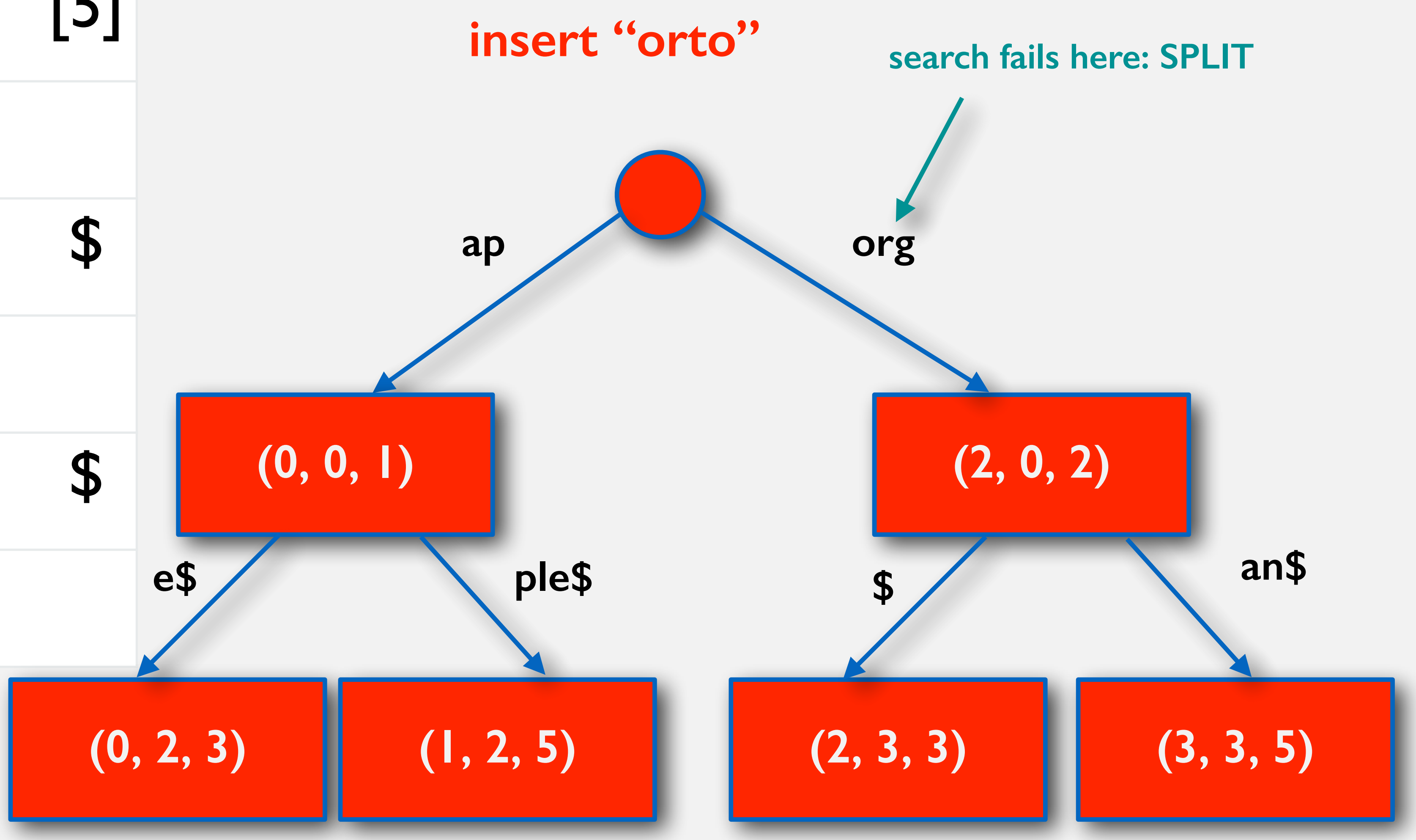
# Patricia Tree *(a.k.a. Compressed Trie)*

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |



# Patricia Tree *(a.k.a. Compressed Trie)*

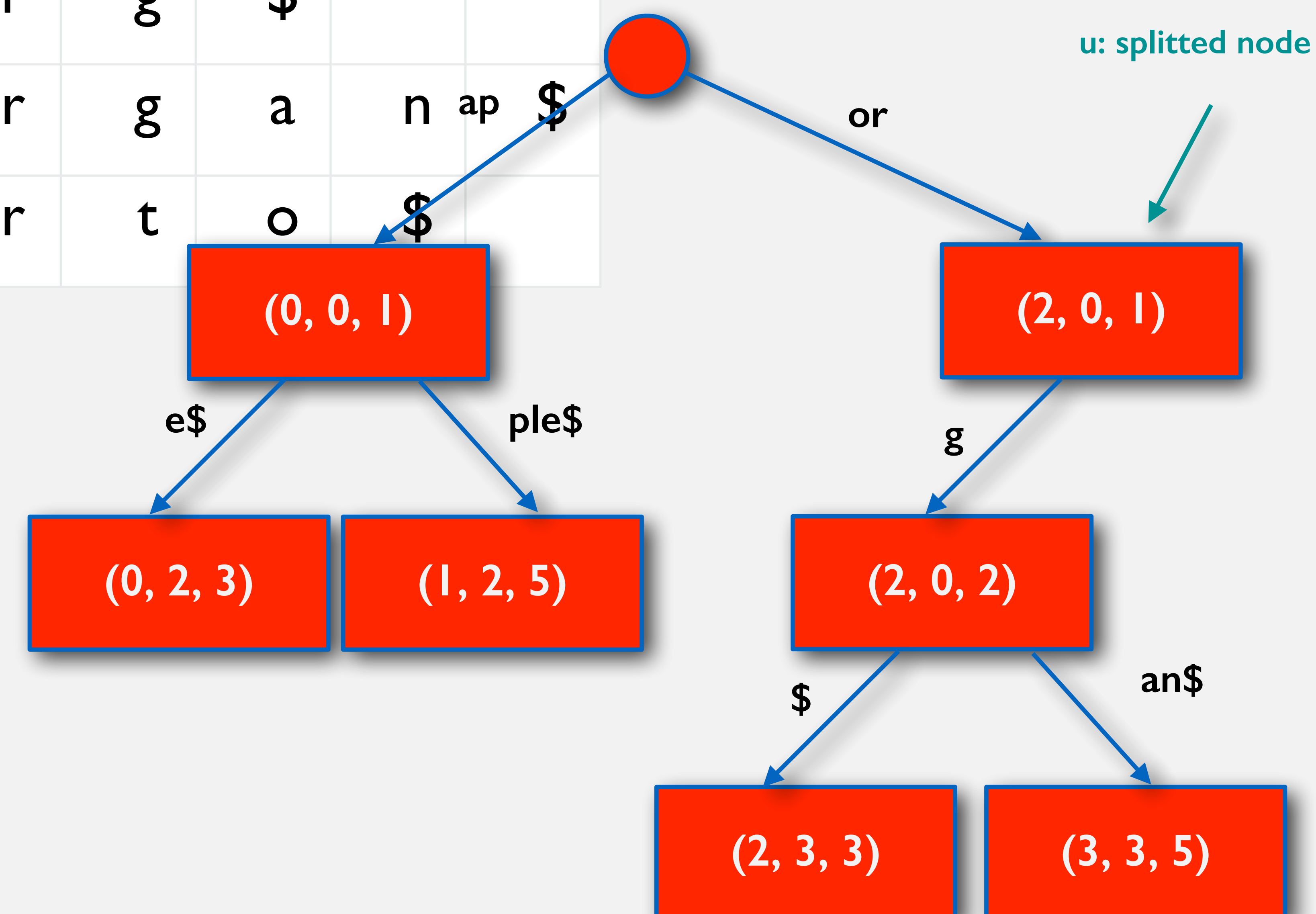
| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |



# Patricia Tree *(a.k.a. Compressed Trie)*

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |

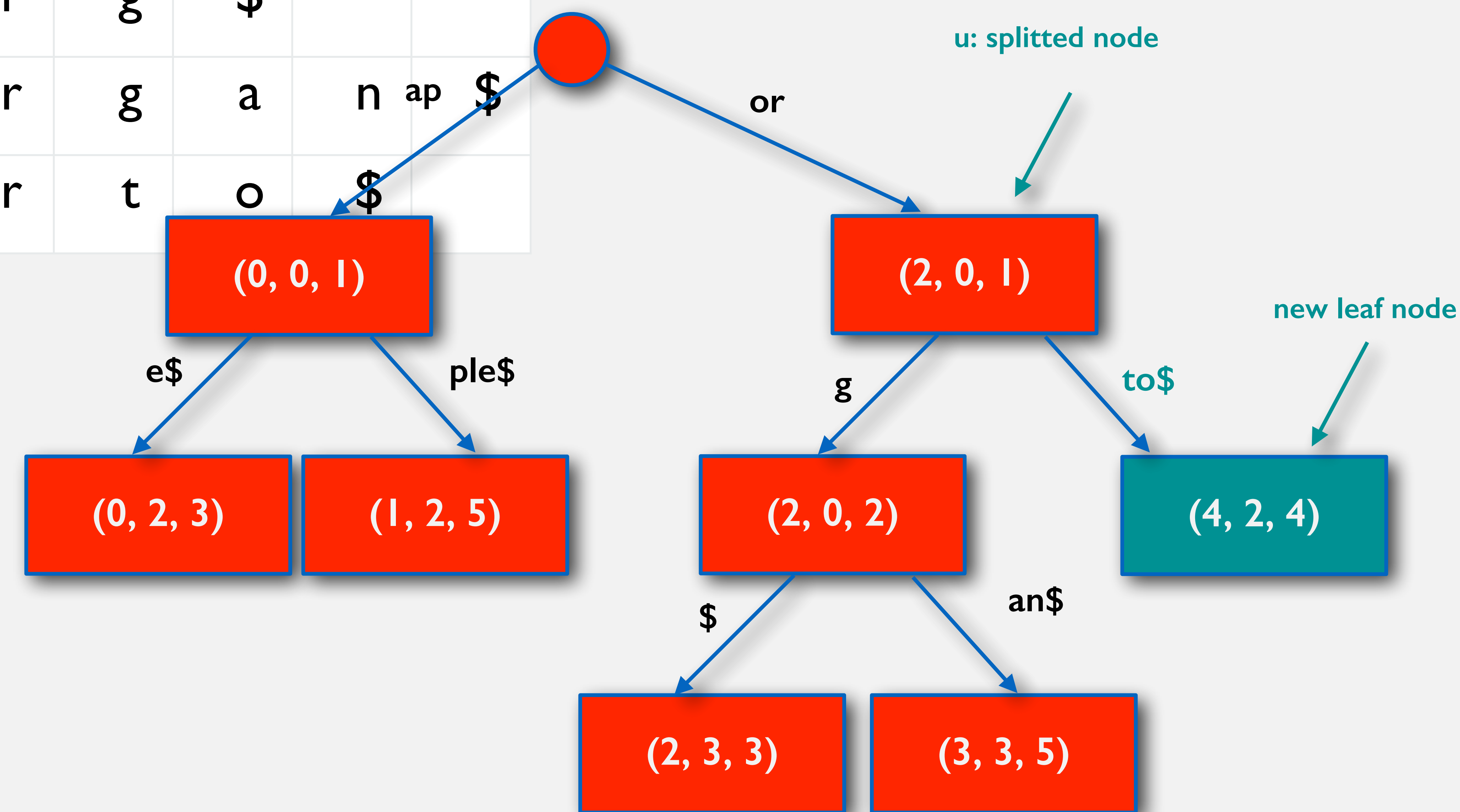
insert "orto"



# Patricia Tree *(a.k.a. Compressed Trie)*

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |

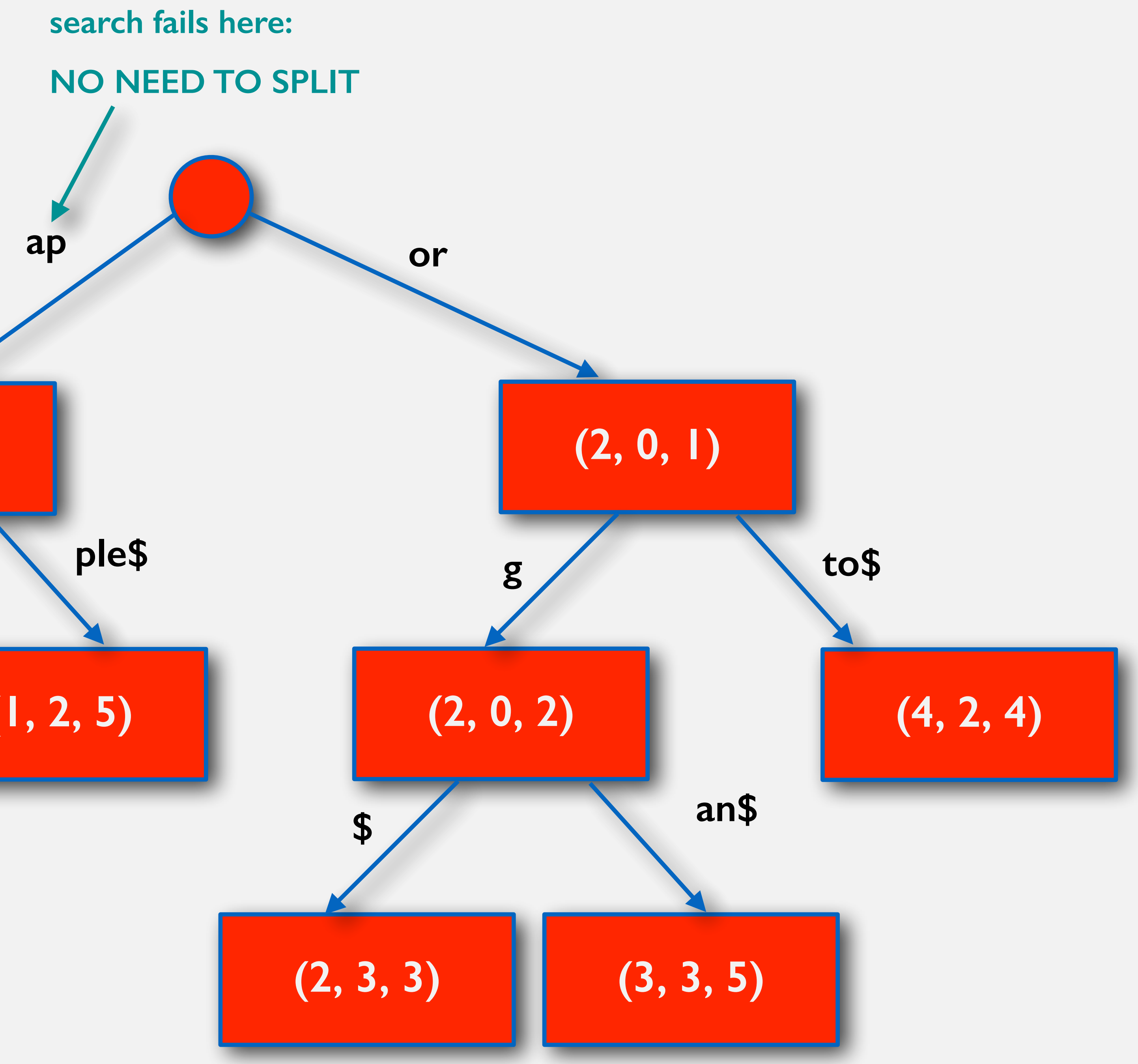
insert “orto”



# Patricia Tree *(a.k.a. Compressed Trie)*

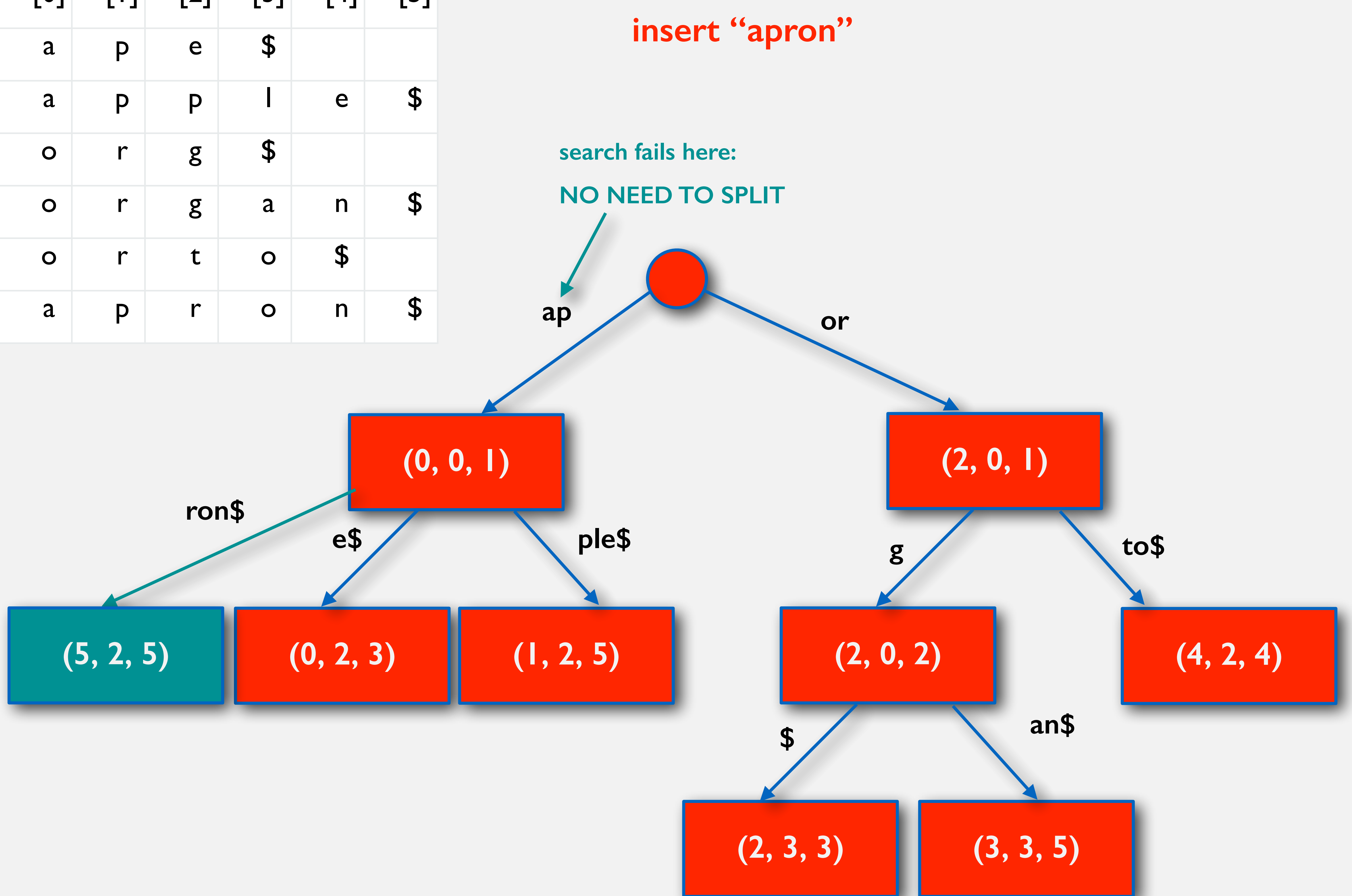
| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |
| [5] | a | p | r | o | n | $ |

insert "apron"



# Patricia Tree *(a.k.a. Compressed Trie)*

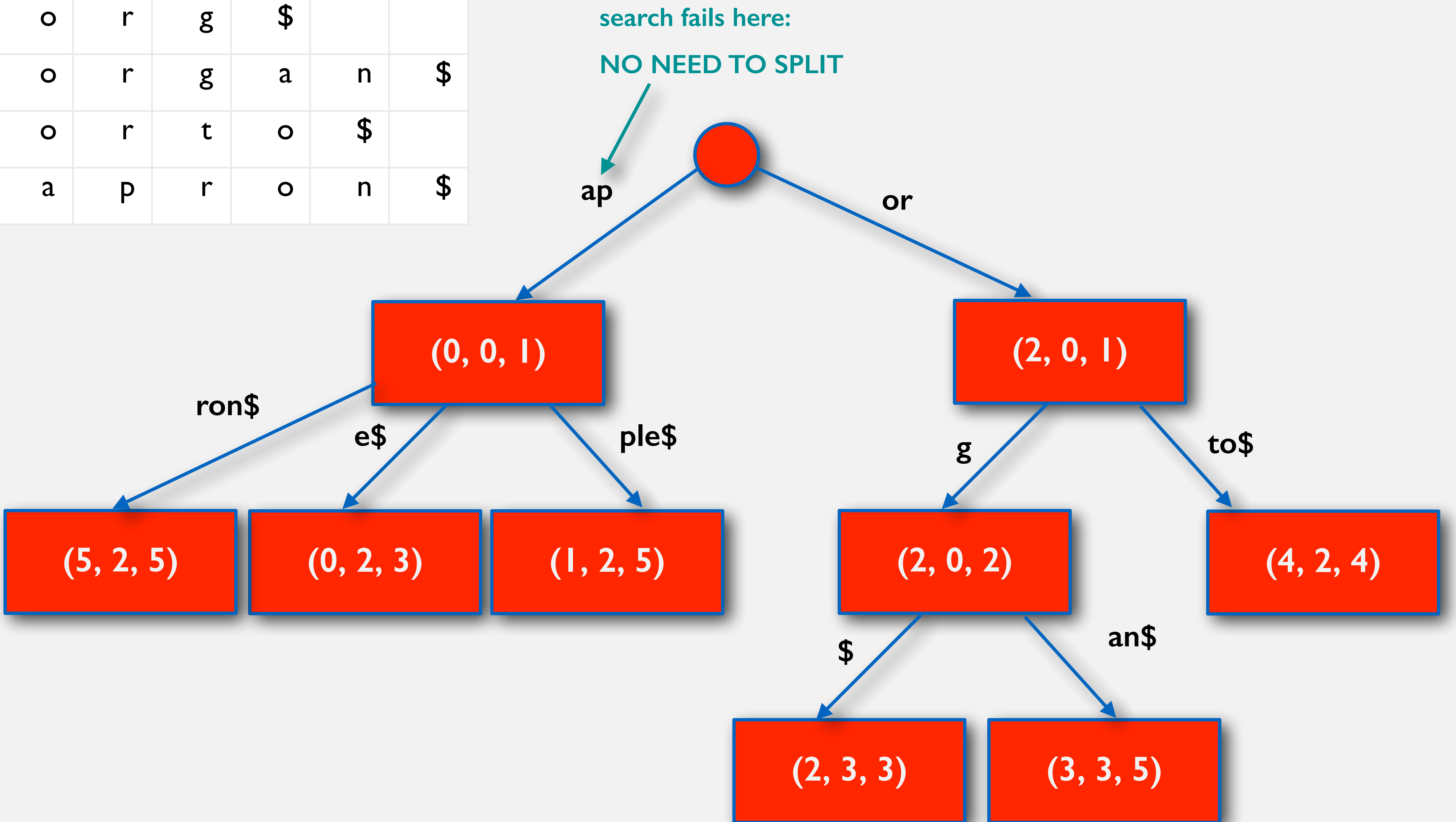
| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |
| [5] | a | p | r | o | n | $ |



# Patricia Tree *(a.k.a. Compressed Trie)*

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |
| [5] | a | p | r | o | n | $ |

insert "apron"



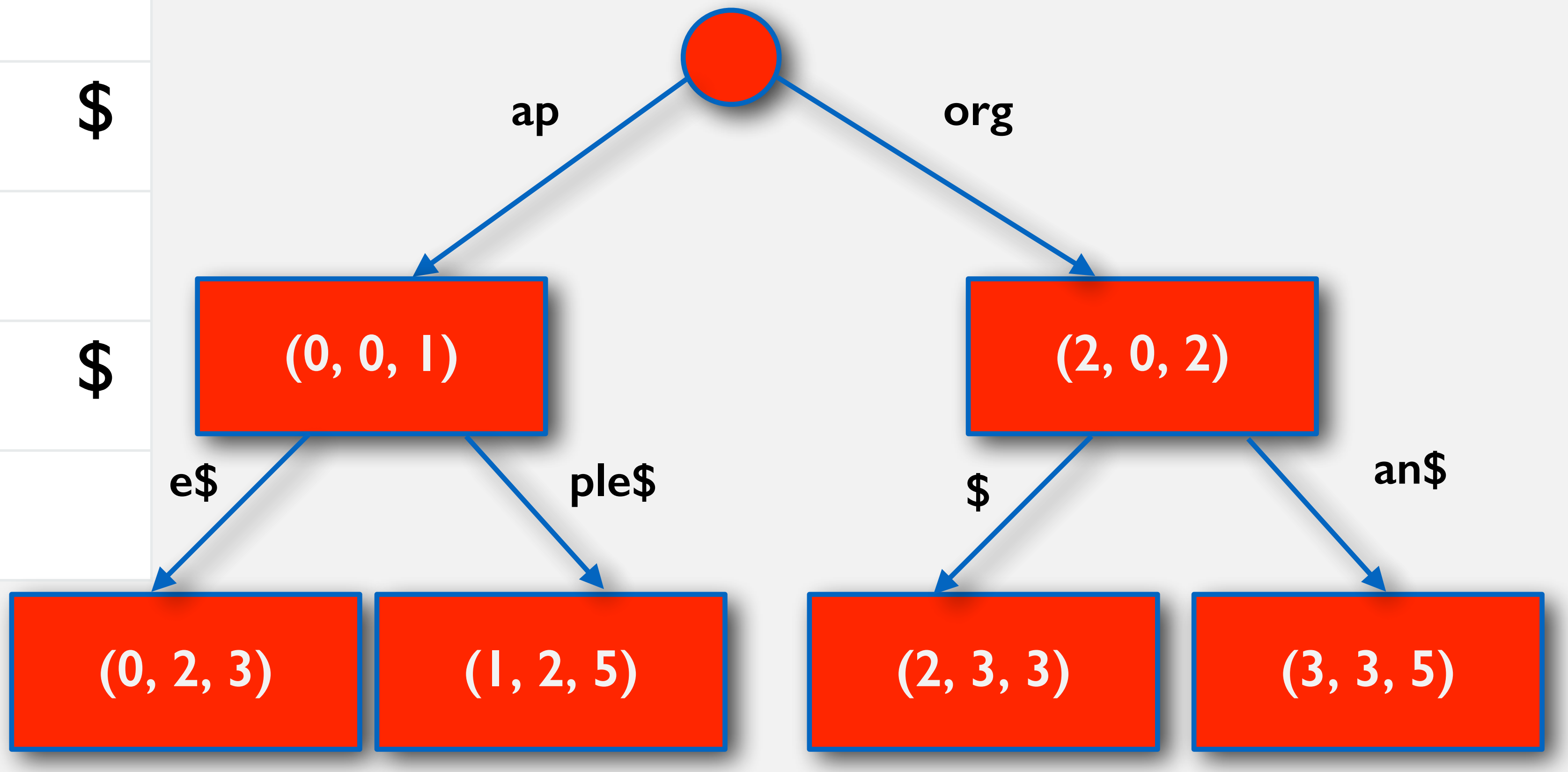
# Patricia Tree *(a.k.a. Compressed Trie)*

- Removing a string s from a Patricia Tree:
  - opposite of insertion
  - locate the leaf corresponding to s and remove it from the tree
    - if the parent node u is left with only one child, w, then we also remove u and replace it with a single edge, e, joining u's parent to w.
  - O(|s|+|Σ|), |s| for search+|Σ| for node creation

# Patricia Tree *(a.k.a. Compressed Trie)*

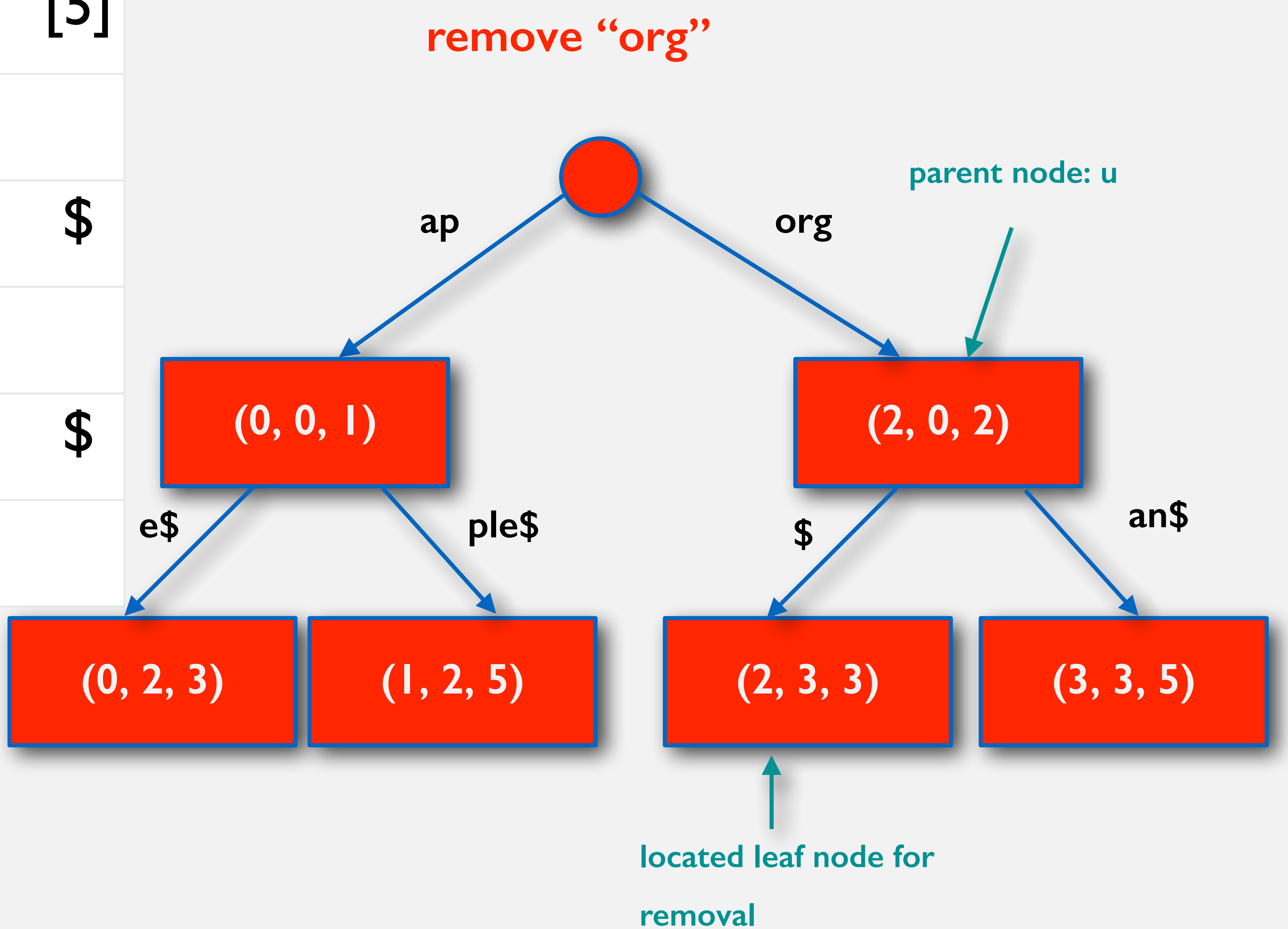
| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |

remove "org"



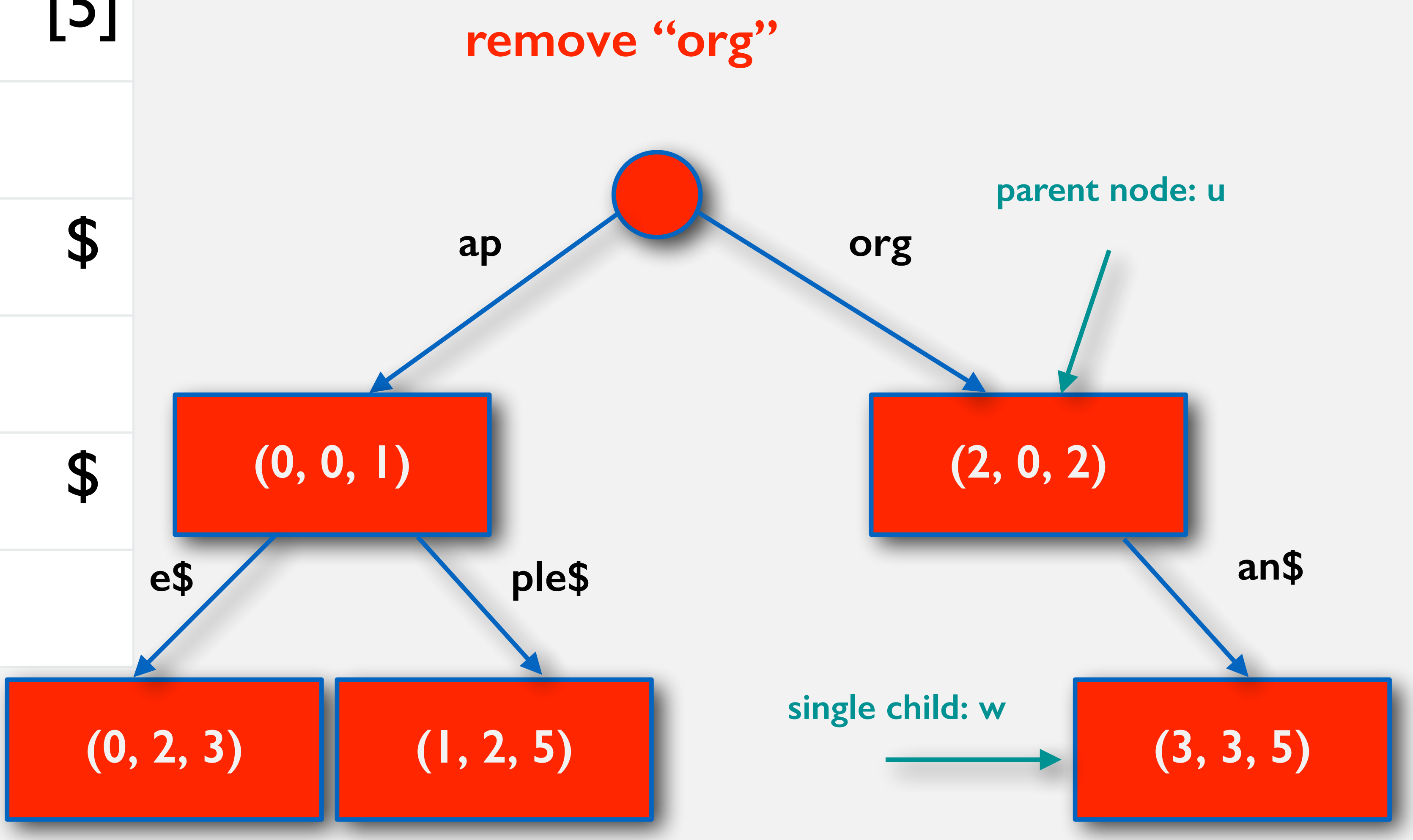
# Patricia Tree *(a.k.a. Compressed Trie)*

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |



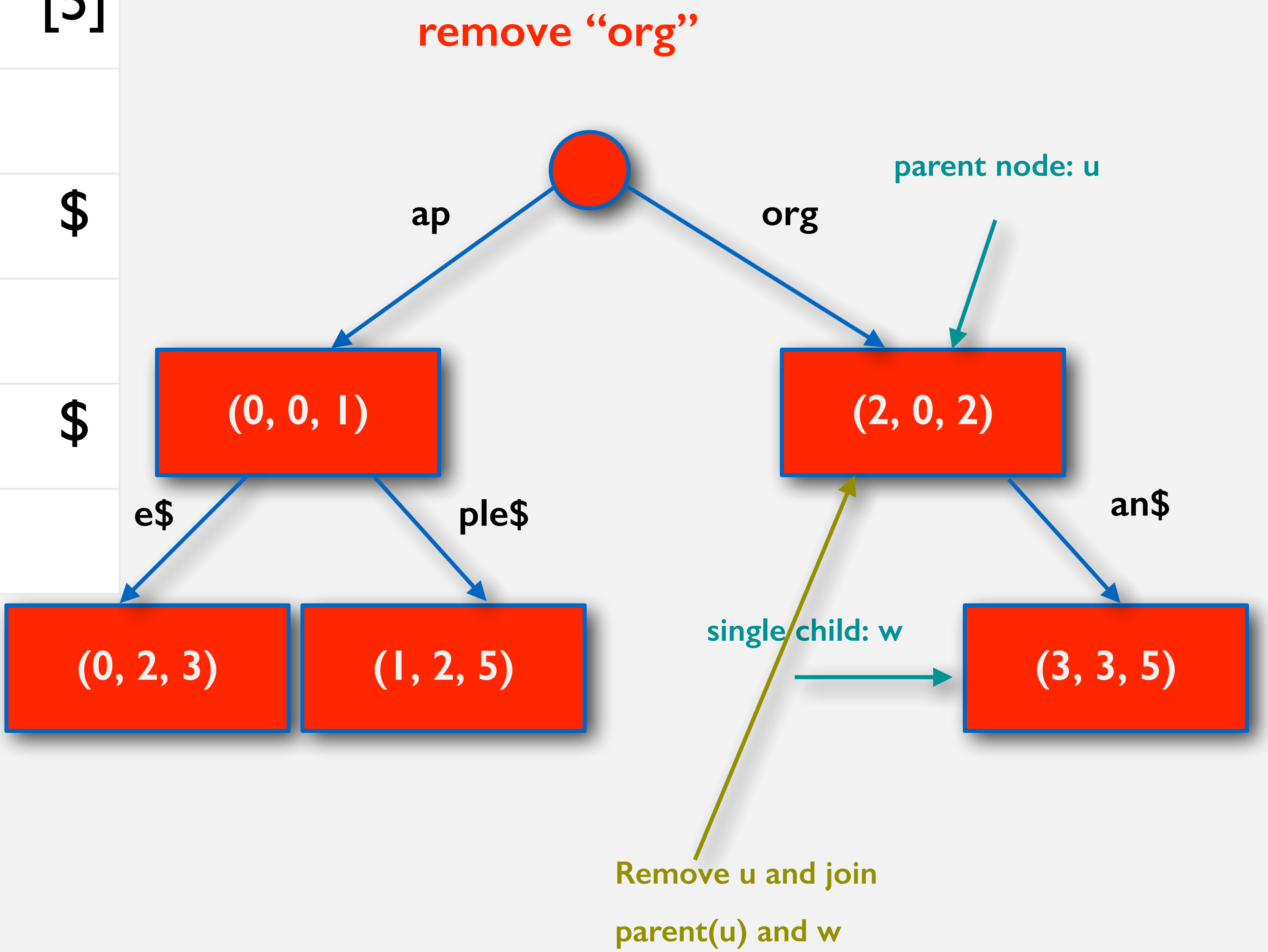
# Patricia Tree *(a.k.a. Compressed Trie)*

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |



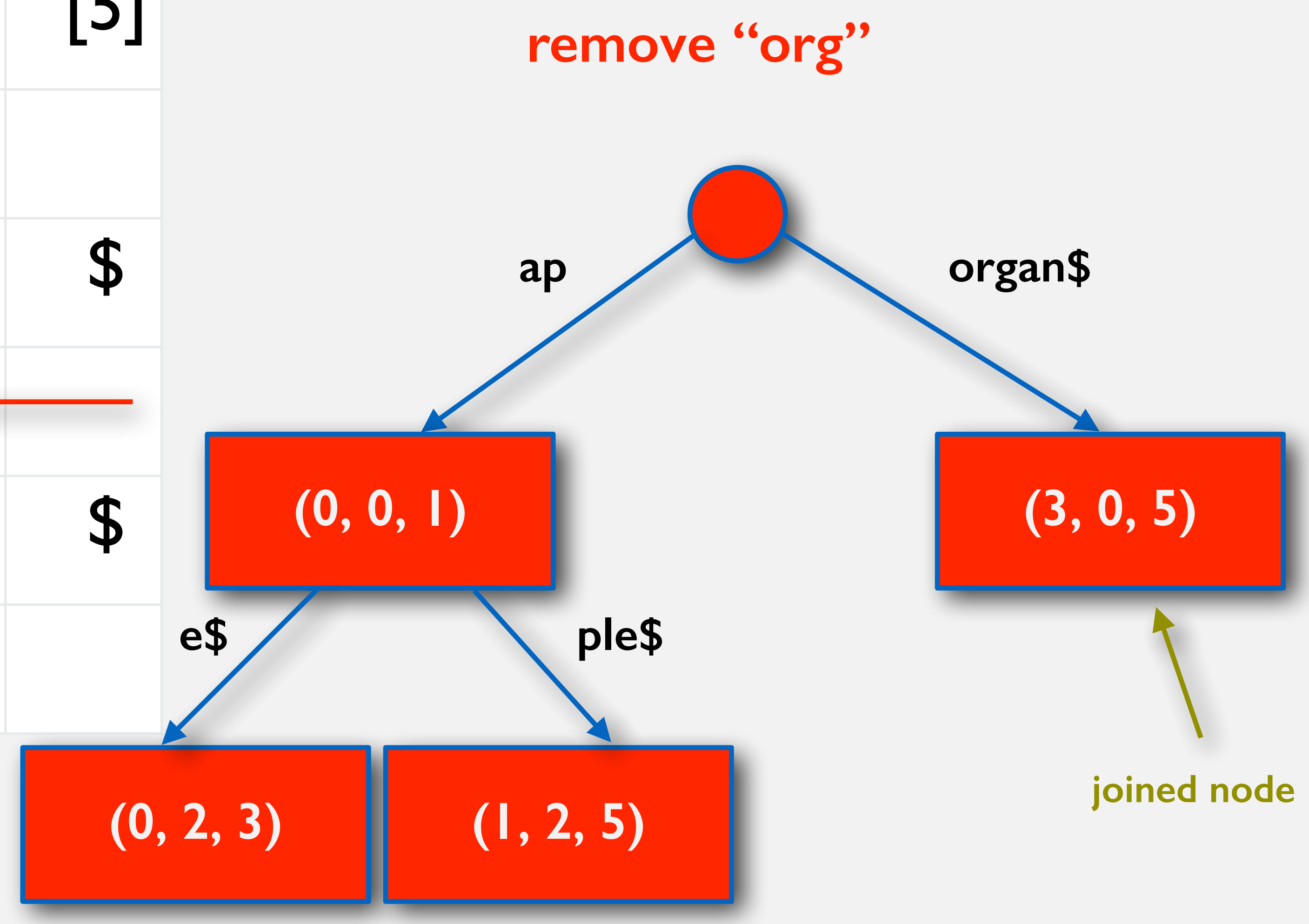
# Patricia Tree *(a.k.a. Compressed Trie)*

| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| [2] | o | r | g | $ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |



# Patricia Tree *(a.k.a. Compressed Trie)*

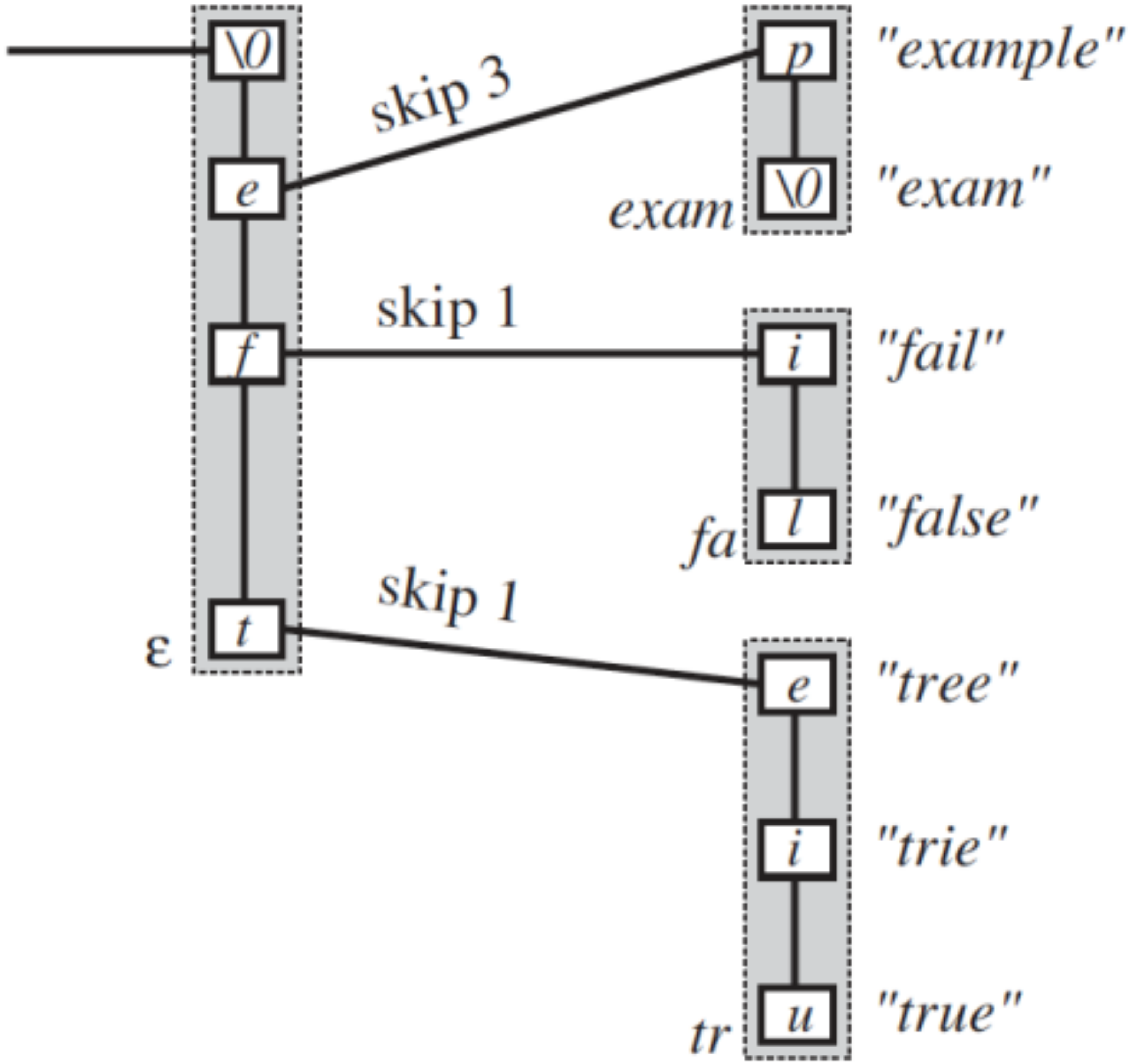
| | [0] | [1] | [2] | [3] | [4] | [5] |
|---|---|---|---|---|---|---|
| [0] | a | p | e | $ | | |
| [1] | a | p | p | l | e | $ |
| ~~[2]~~ | ~~o~~ | ~~r~~ | ~~g~~ | ~~$~~ | | |
| [3] | o | r | g | a | n | $ |
| [4] | o | r | t | o | $ | |



# Patricia Tree-Alternative representations

- We skip these nodes and keep track of the number of skipped characters.
- It contains a number, which is the number of characters that should be skipped before the next relevant character is looked at.
- This reduces the required number of nodes from the total length of all strings to the number of words in our structure.
- We need in each access a second pass over the string to check all those skipped characters of the found string against the query string.
- This technique to reduce the number of nodes is justified only if the alphabet is large.

# Patricia Tree Example




PATRICIA TREE FOR THE STRINGS *exam, example, fail, false, tree, trie, true:*
NODES IMPLEMENTED AS LISTS; EACH LEAF CONTAINS ENTIRE STRING

# Patricia Tree: *Insert & Delete*

- The insertion and deletion operations create significant difficulties.
- We need to find where to insert a new branching node, but this requires that we know the skipped characters.
- One (clumsy) solution would be a pointer to one of the strings in the subtrie reached through that node, for there we have that skipped substring already available.

# Suffix Trees

The suffix tree is a static structure that preprocesses a long string *s* and answers for a query string *q*, if and where it occurs in the long string.

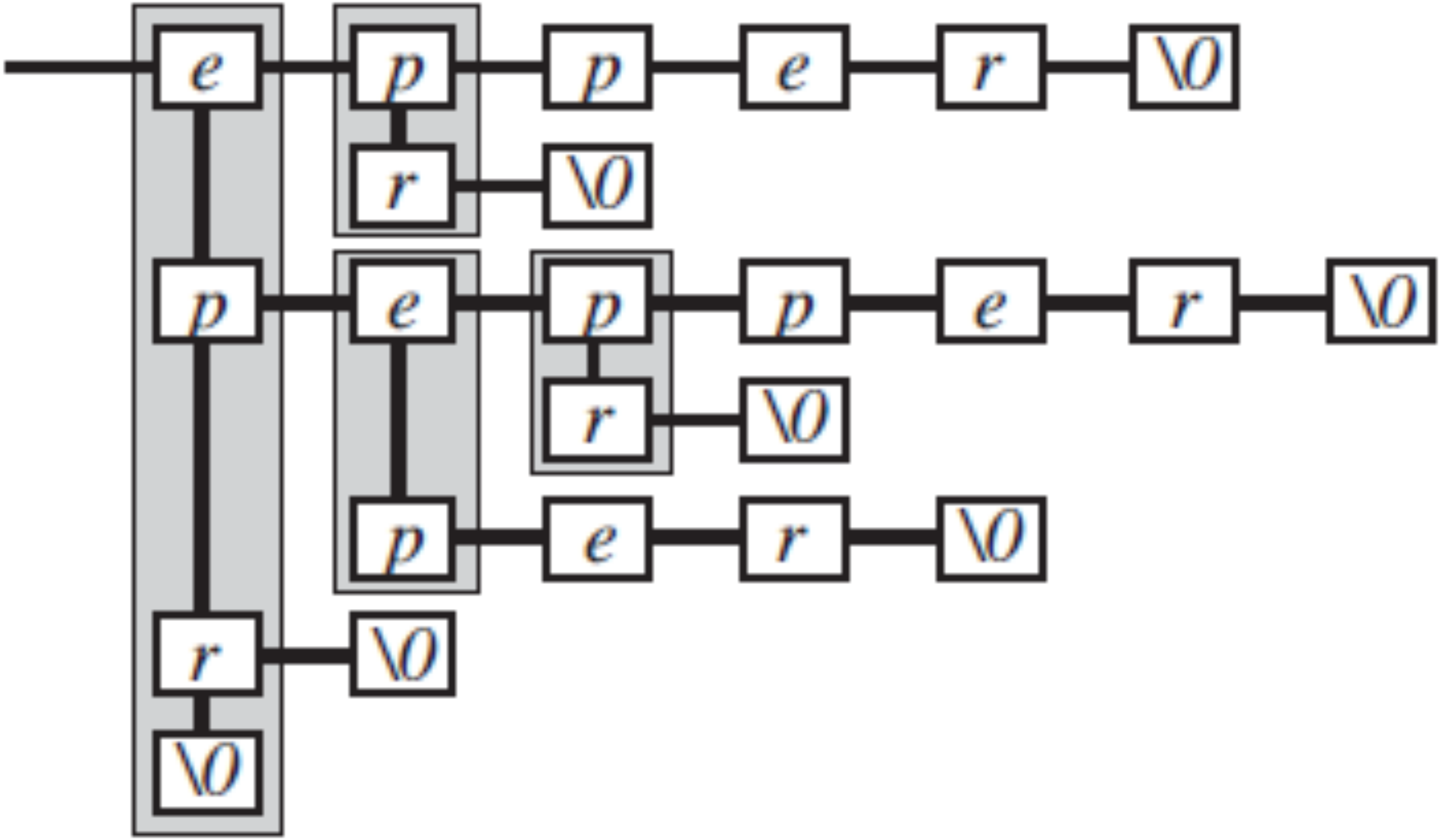
- Each substring of *s* is prefix of a suffix of *s*.
- If we construct a trie that stores all suffixes of the long string *s*, then its nodes correspond to the substrings of *s*, and we can decide for any query *q* in $O(\text{length}(q))$ whether it is a substring of *s*.
- This structure would use $O(\text{length}(s)^2)$ nodes.

Build a **trie** containing all **suffixes** of a text S

```
S : GTTATAGCTGATCGCGGCGTAGCGG
    GTTATAGCTGATCGCGGCGTAGCGG
     TTATAGCTGATCGCGGCGTAGCGG
      TATAGCTGATCGCGGCGTAGCGG
       ATAGCTGATCGCGGCGTAGCGG
        TAGCTGATCGCGGCGTAGCGG
         AGCTGATCGCGGCGTAGCGG
          GCTGATCGCGGCGTAGCGG
           CTGATCGCGGCGTAGCGG
            TGATCGCGGCGTAGCGG
             GATCGCGGCGTAGCGG
              ATCGCGGCGTAGCGG
               TCGCGGCGTAGCGG
                CGCGGCGTAGCGG
                 GCGGCGTAGCGG
                  CGGCGTAGCGG
                   GGCGTAGCGG
                    GCGTAGCGG
                     CGTAGCGG
                      GTAGCGG
                       TAGCGG
                        AGCGG
                         GCGG
                          CGG
                           GG
                            G
```

$m(m+1)/2$ chars

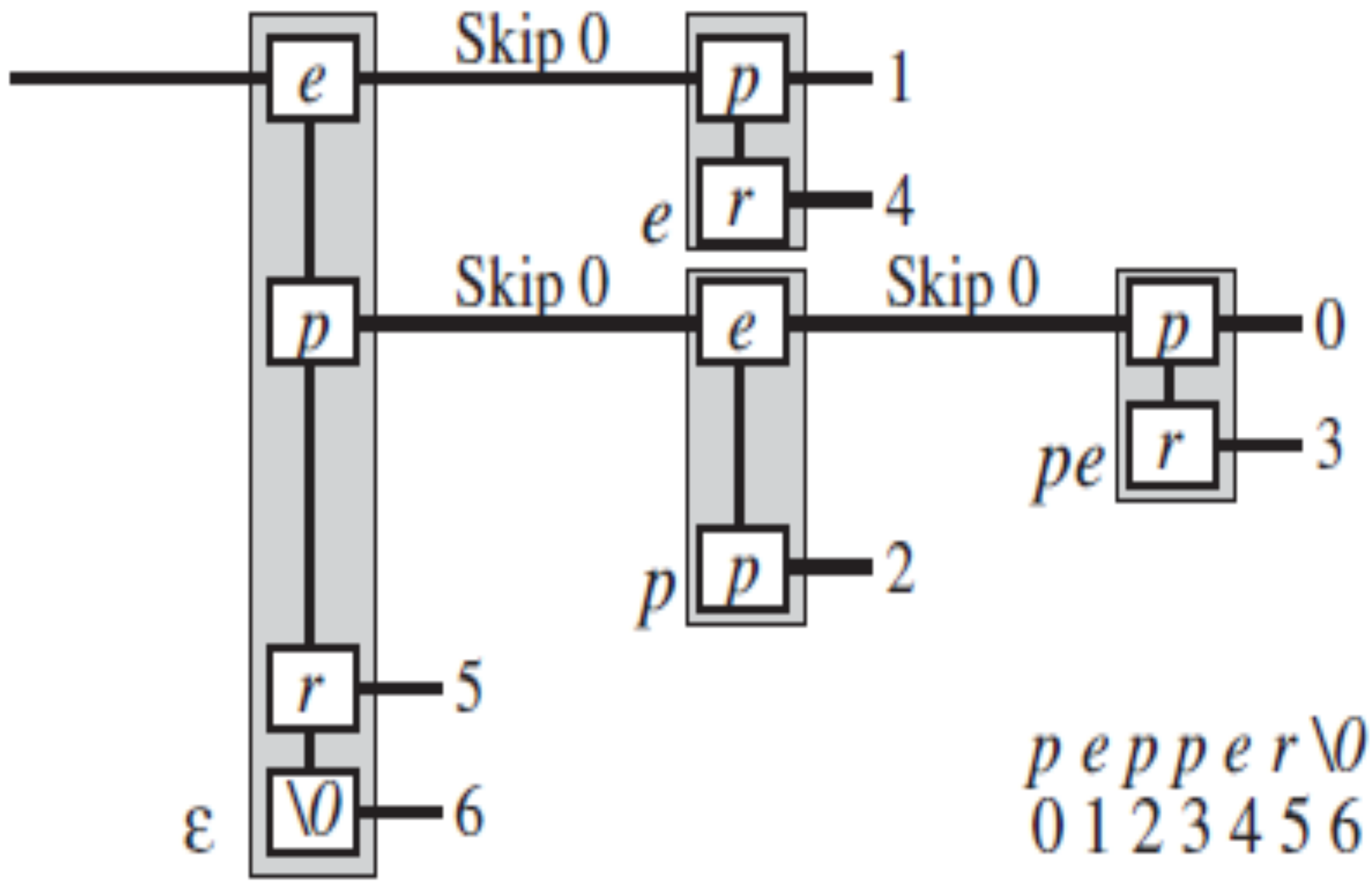
# Suffix tree




TRIE OF THE SUFFIXES OF *pepper*

# A more Compact Representation

No need to store all suffixes explicitly, but can encode each by a beginning and end address in the long string S. ➔ O(length(s)) nodes representation.




PATRICIA TREE OF THE SUFFIXES OF *pepper*:

THE LEAF NUMBERS GIVE THE STARTING POSITIONS OF THE SUFFIXES

# Suffix Arrays

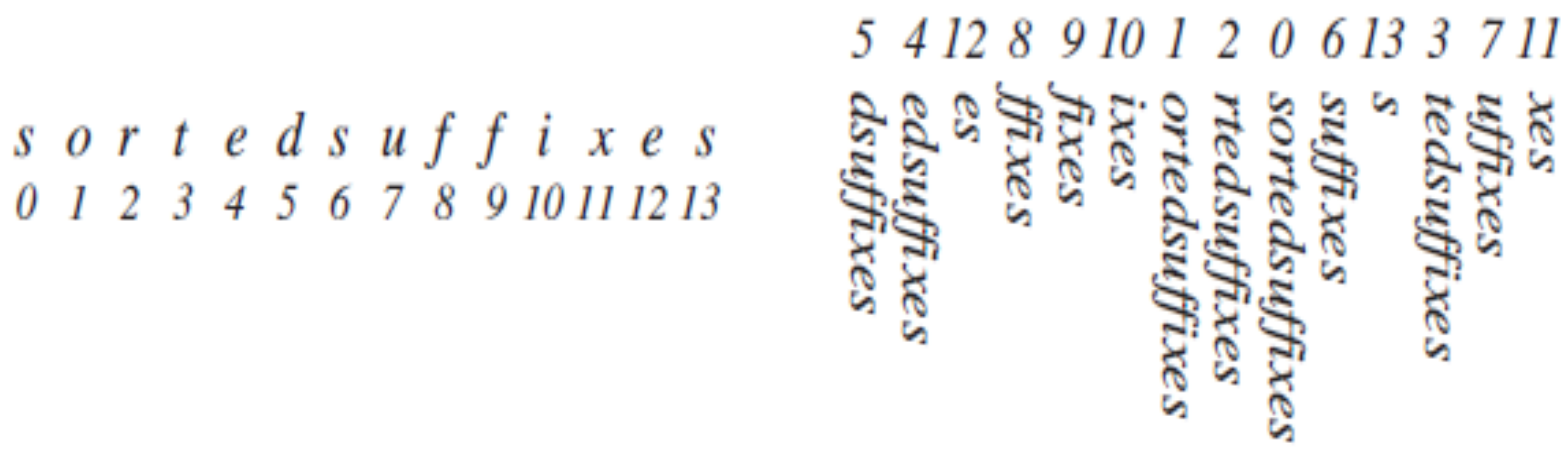
The suffix array is an alternative structure to the suffix tree that was developed by Manber and Myers (1993). It preprocesses a long string and then answers for a query string whether it occurs as substring in the preprocessed string.

Possible Advantages:

- Its size does not depend on the size of the alphabet.
- It offers a quite different tool to attack the same type of string problems.
- Straightforward implementation and it is said to be smaller than suffix trees

# The Underlying Idea

To consider all suffixes of the preprocessed string *s* in lexicographic order and perform binary search on them to find a given query string.




THE SUFFIXES OF *sortedsuffixes* IN LEXICOGRAPHIC ORDER WITH THEIR STARTING INDICES IN THE STRING

**O(|s|logN)**

**|s|: length of the query string**

**N: suffix array size**