Tries

❖ Tries

A trie ...

- is a data structure for representing a set of strings
    - e.g. all the distinct words in a document, a dictionary etc.
- supports string matching queries in *O(L)* time
    - *L* is the length of the string being searched for

Note: generally assume "string" = character string; could be bit-string

Note: Trie comes from *retrieval* ; but pronounced as "try" not "tree"

❖ ... Tries

Each node in a trie ...

- contains one part of a key (typically one character)
- may have up to 26 children
- may be tagged as a "finishing" node
- but even "finishing" nodes may have children
- may contain other data for application (e.g. word frequency)

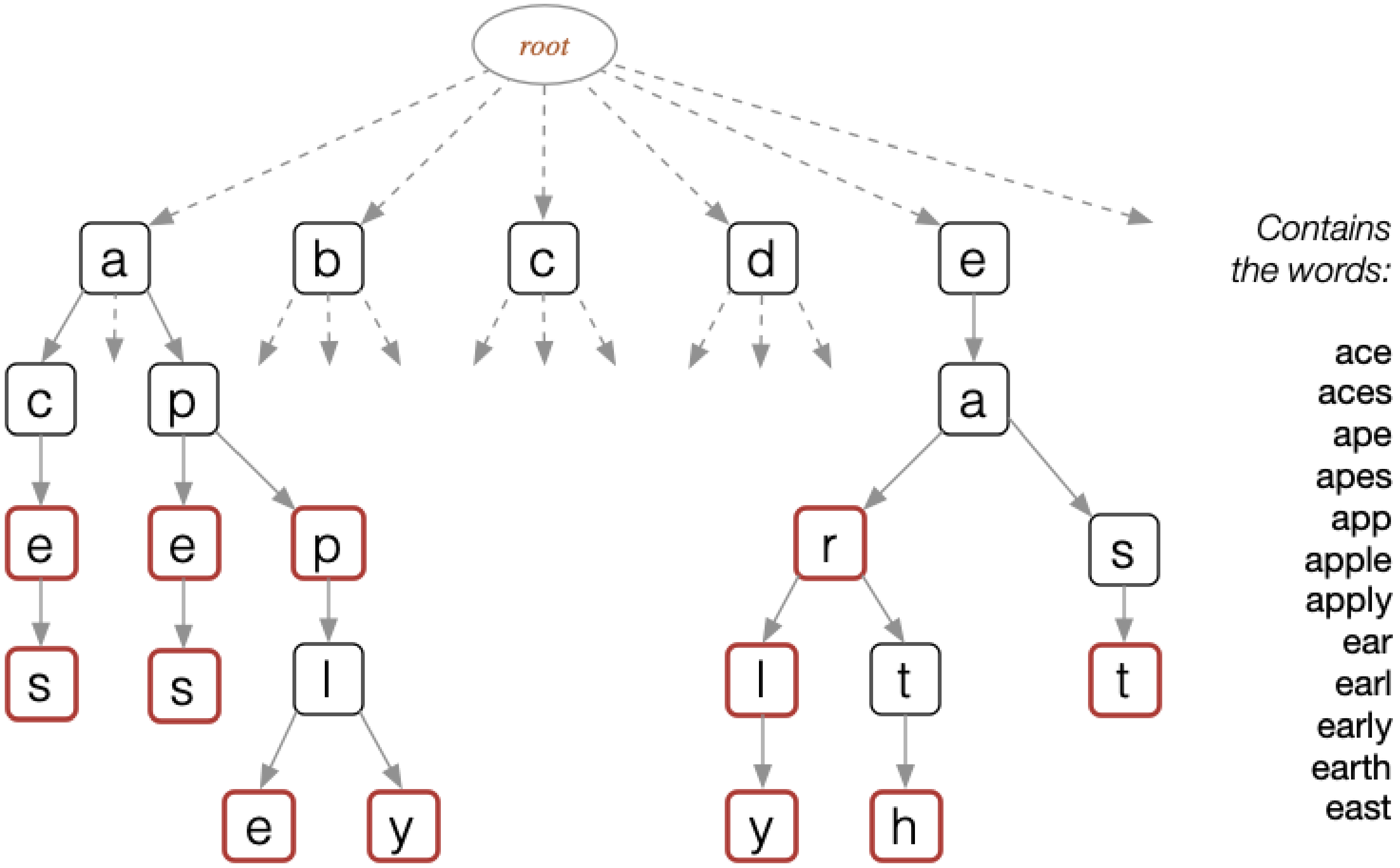
A "finishing" node marks the end of one key

- this key may be a prefix of another key stored in trie

Depth *d* of trie = length of longest key value

❖ ... Tries

Trie example:



❖ ... Tries

Possible trie representation:

```
#define ALPHABET_SIZE 26

typedef struct Node *Trie;

typedef struct Node {
   char onechar;      // current char in key
   Trie child[ALPHABET_SIZE];
   bool finish;       // last char in key?
   Item data;         // no Item if !finish
} Node;

typedef char *Key;    // just lower-case letters
```

❖ ... Tries

Above representation is space inefficient

- each node has 26 possible children
- even with very many keys, most child links are unused

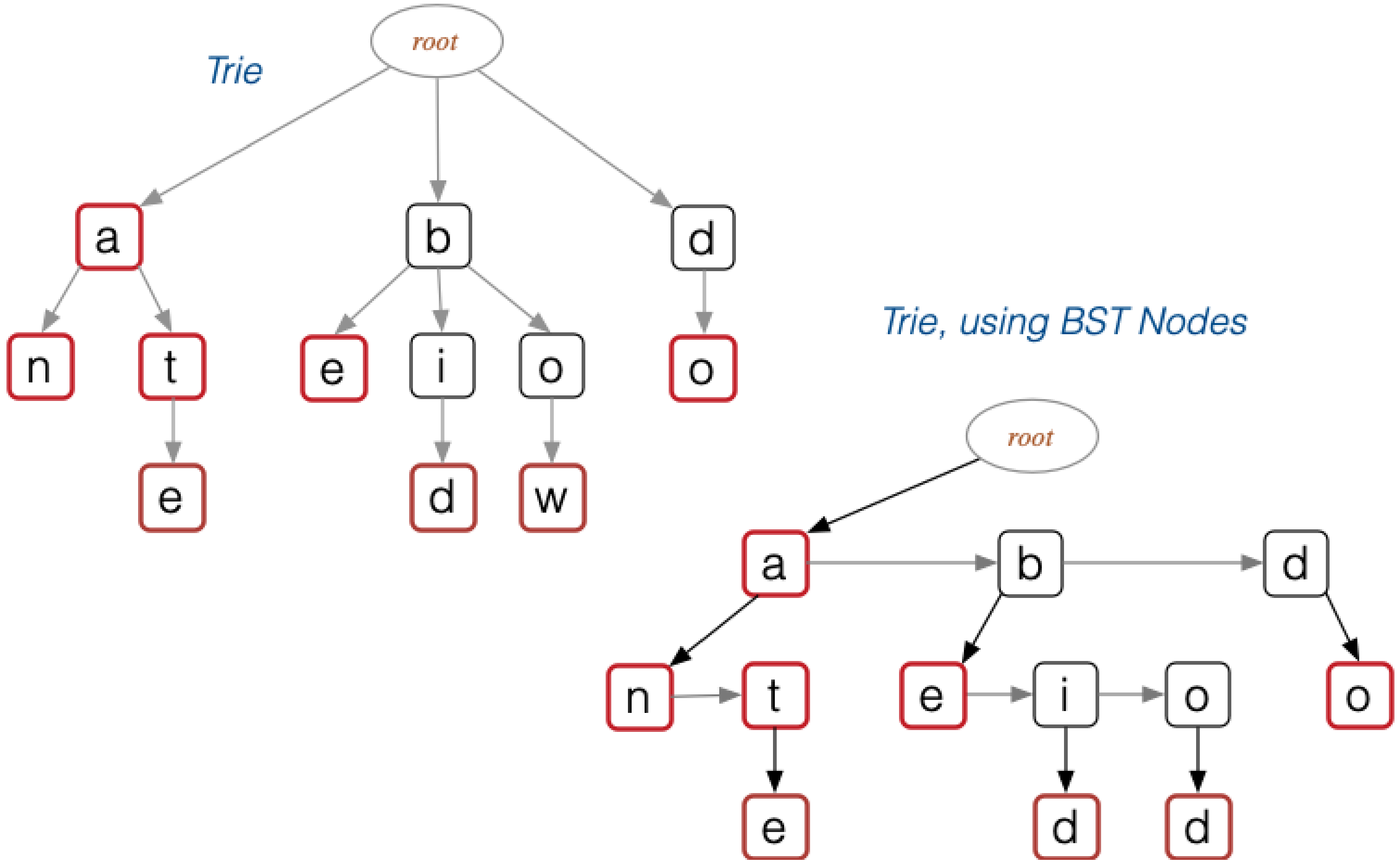
If we allowed all ascii chars in alphabet, 128 children

Could reduce branching factor by reducing "alphabet"

- break each 8-bit char into two 4-bit "nybbles"
- branching factor is 16, even for full ascii char set
- but each branch is twice as long

❖ ... Tries

Note: Can also use BST-like nodes (cf. red-black trees) ...

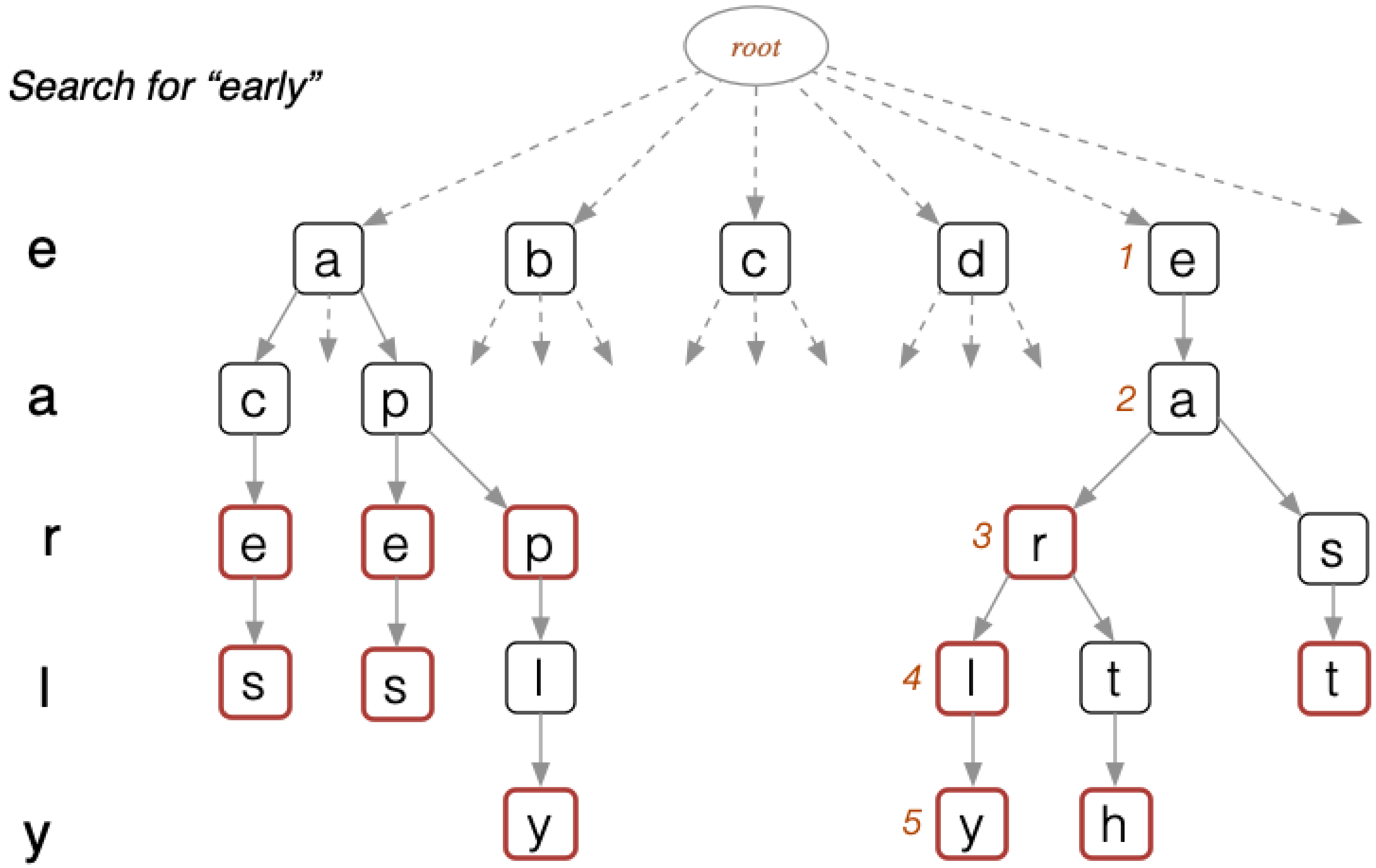


❖ Searching in Tries

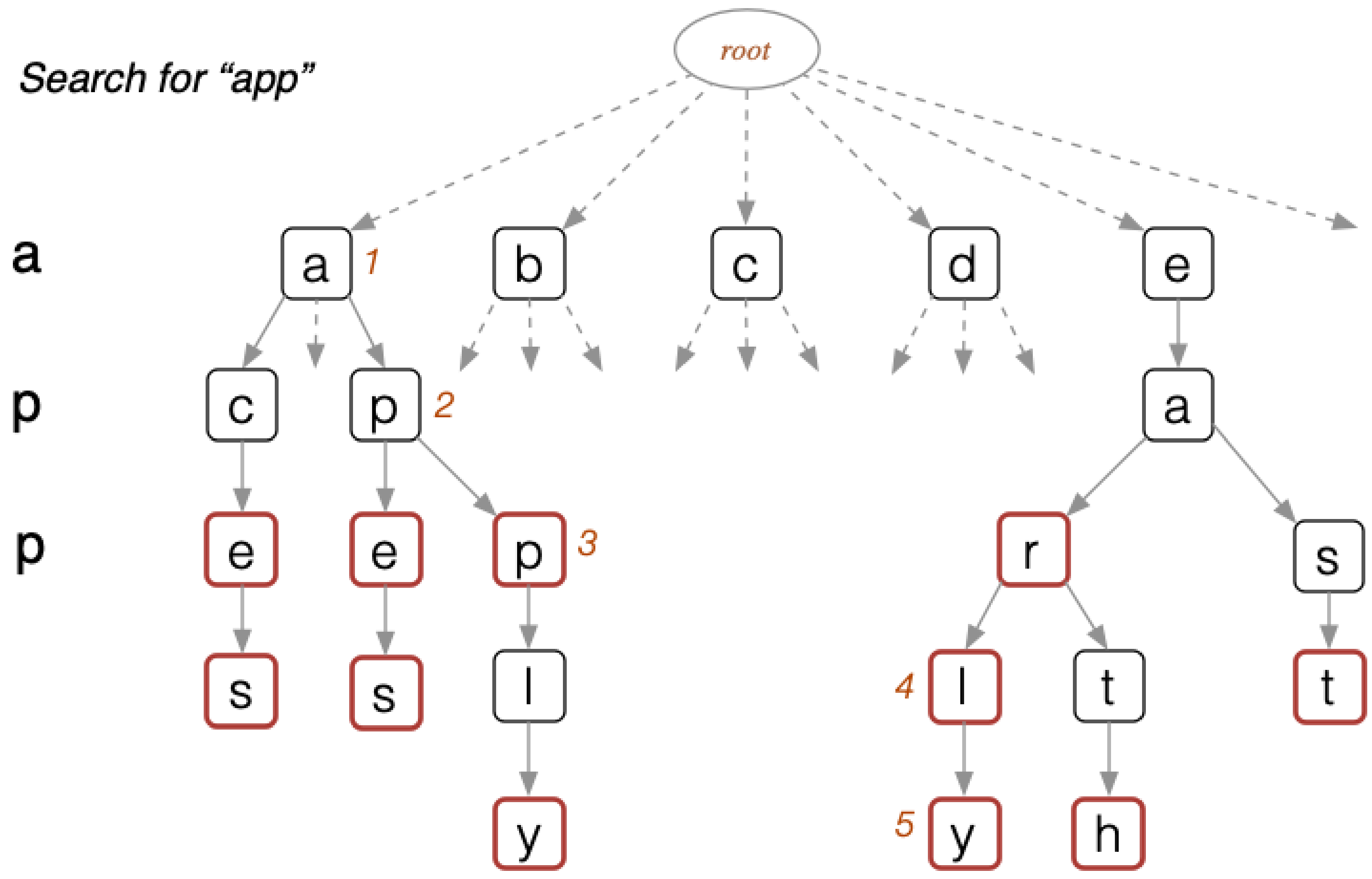
Search requires traversing a path, char-by-char from Key:

```
find(trie,key):
|  Input  trie, key
|  Output pointer to element in trie if key found
|         NULL otherwise
|
|  node=trie
|  for each char c in key do
|  |  if node.child[c] exists then
|  |     node=node.child[c]  // move down one level
|  |  else
|  |     return NULL
|  |  end if
|  end for
|  if node.finish then  // "finishing" node reached?
|     return node
|  else
|     return NULL
|  end if
```

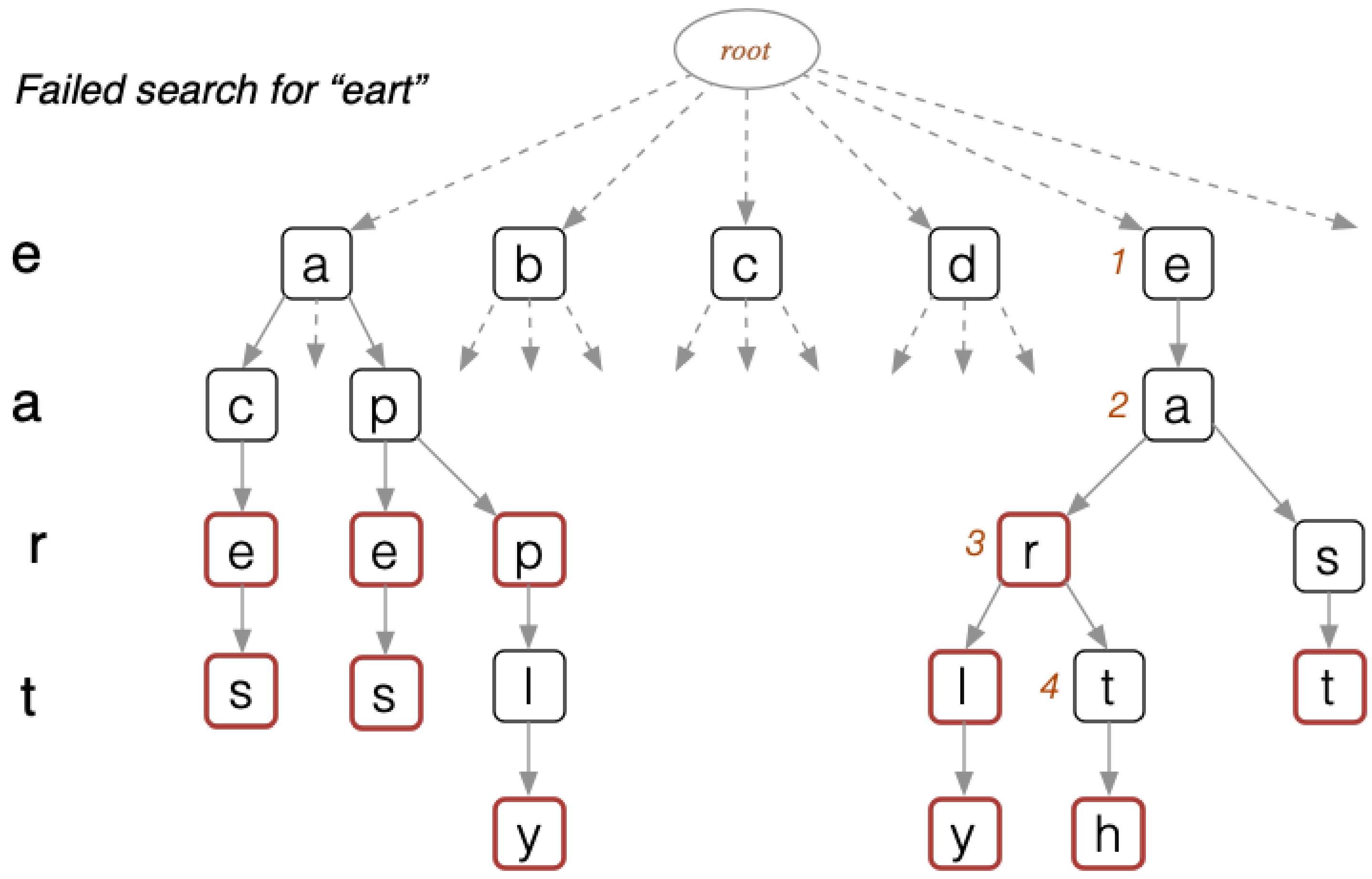
❖ ... Searching in Tries



❖ ... Searching in Tries



❖ ... Searching in Tries



❖ Insertion into Tries

Insertion into a Trie ...

```
Trie insert(trie,item,key):
|  Input  trie, item with key of length m
|  Output trie with item inserted
|
|  if trie is empty then
|  |  t=new trie node
|  end if
|  if m=0 then  // end of key
|  |  t.finish=true, t.data=item
|  else
|  |  first=key[0],  rest=key[1..m-1]
|  |  t.child[first]=insert(t.child[first],item,rest)
|  end if
|  return t
```

❖ Cost Analysis

Analysis of standard trie:

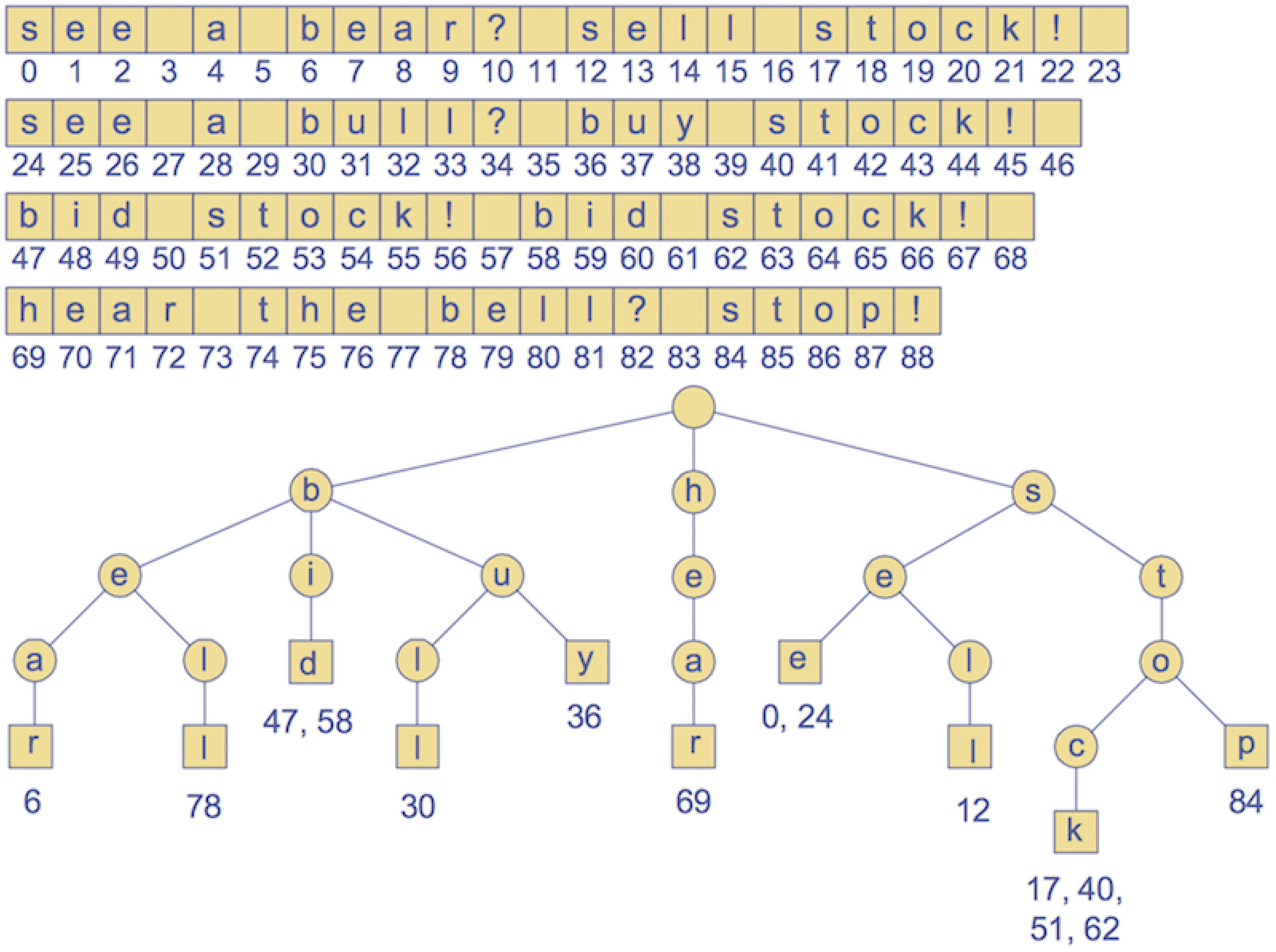
- *O(n)* space
- *O(m)* insertion and search

where

- *n* ... total size of text (e.g. sum of lengths of all strings)
- *m* ... length of the key string
- *d* ... size of the underlying alphabet (e.g. 26)

❖ Example Trie

Example text and corresponding trie of searchable words:



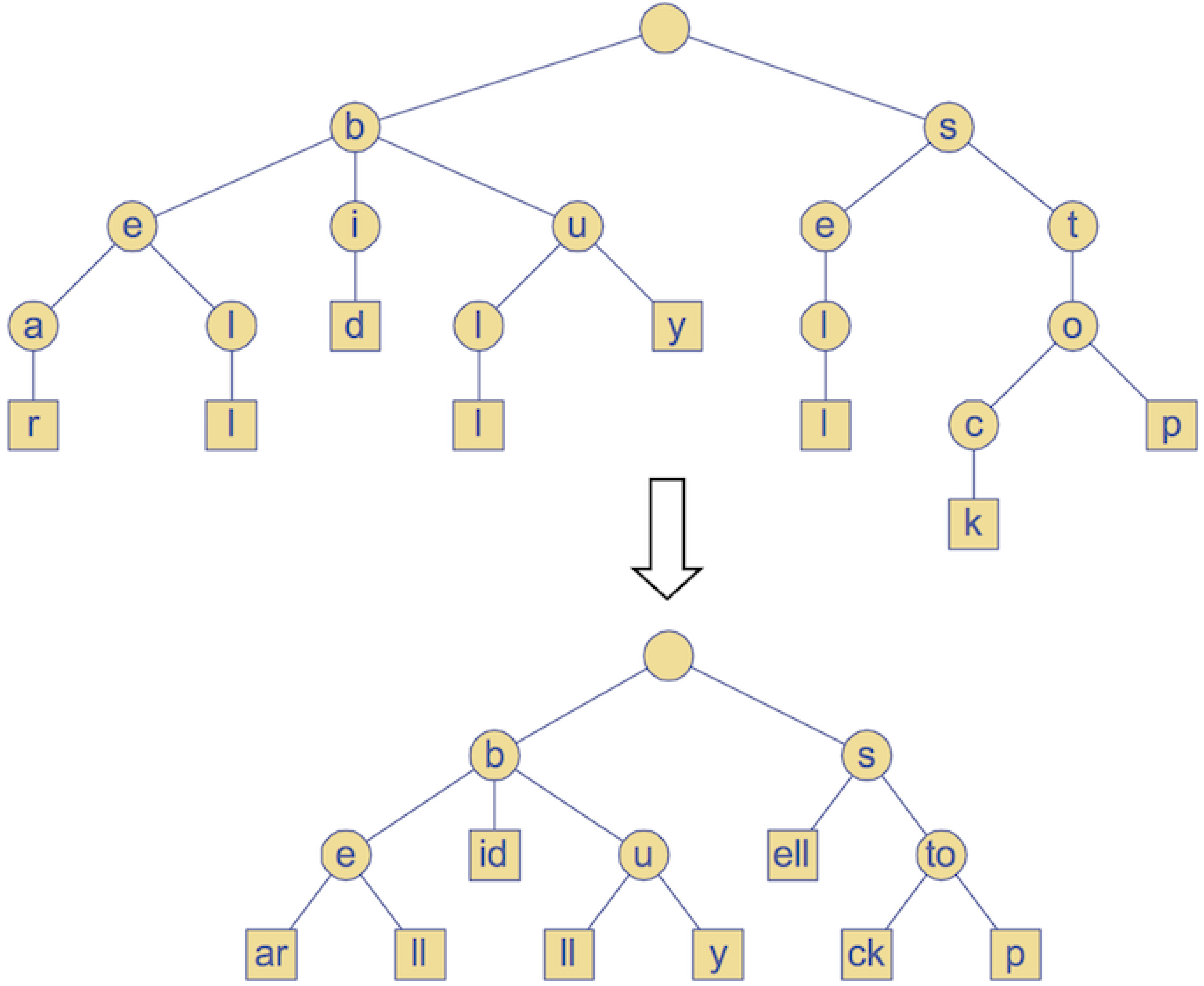
Note: trie has no prefixes $\Rightarrow$ all finishing nodes are leaves

❖ Compressed Tries

Compressed tries ...

- have internal nodes of degree ≥ 2;   each node contains ≥ 1 char
- obtained by compressing non-branching chains of nodes

Example:

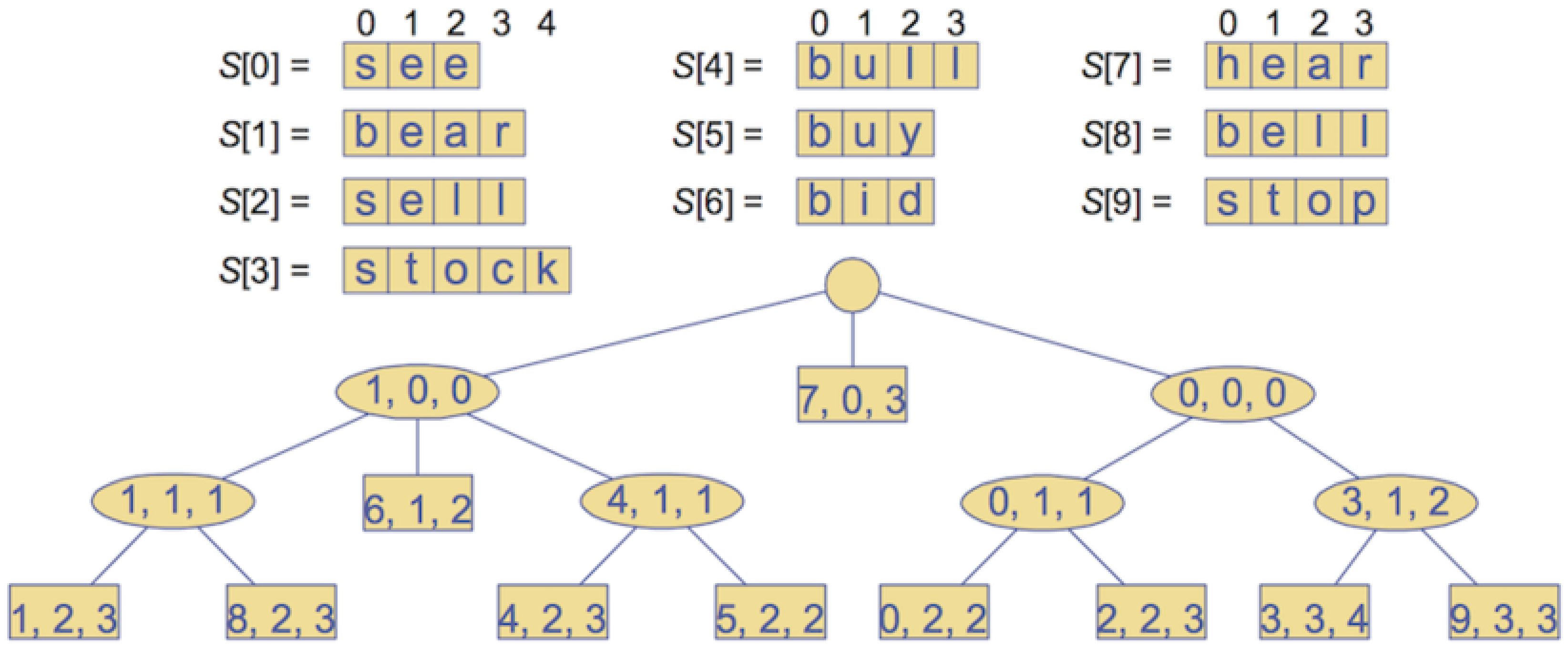

b
s
e
i
u
e
t
a
l
d
l
y
l
o
r
l
l
l
c
p
k
b
s
e
id
u
ell
to
ar
ll
ll
y
ck
p

❖ ... Compressed Tries

Compact representation of compressed trie to encode array *S* of strings:

- nodes store ranges of indices instead of substrings
  - use triple *(i,j,k)* to represent substring *S[i][j..k]*
- requires *O(s)* space (*s* = #strings in array *S*)

Example:



Produced: 11 Jul 2020