Tries, radix trees, suffix trees

Tries

A trie (pronounced *try*) is a data structure for representing a set of strings

- It can also be used for a map where the keys are strings
- Or where the key is a list of some kind

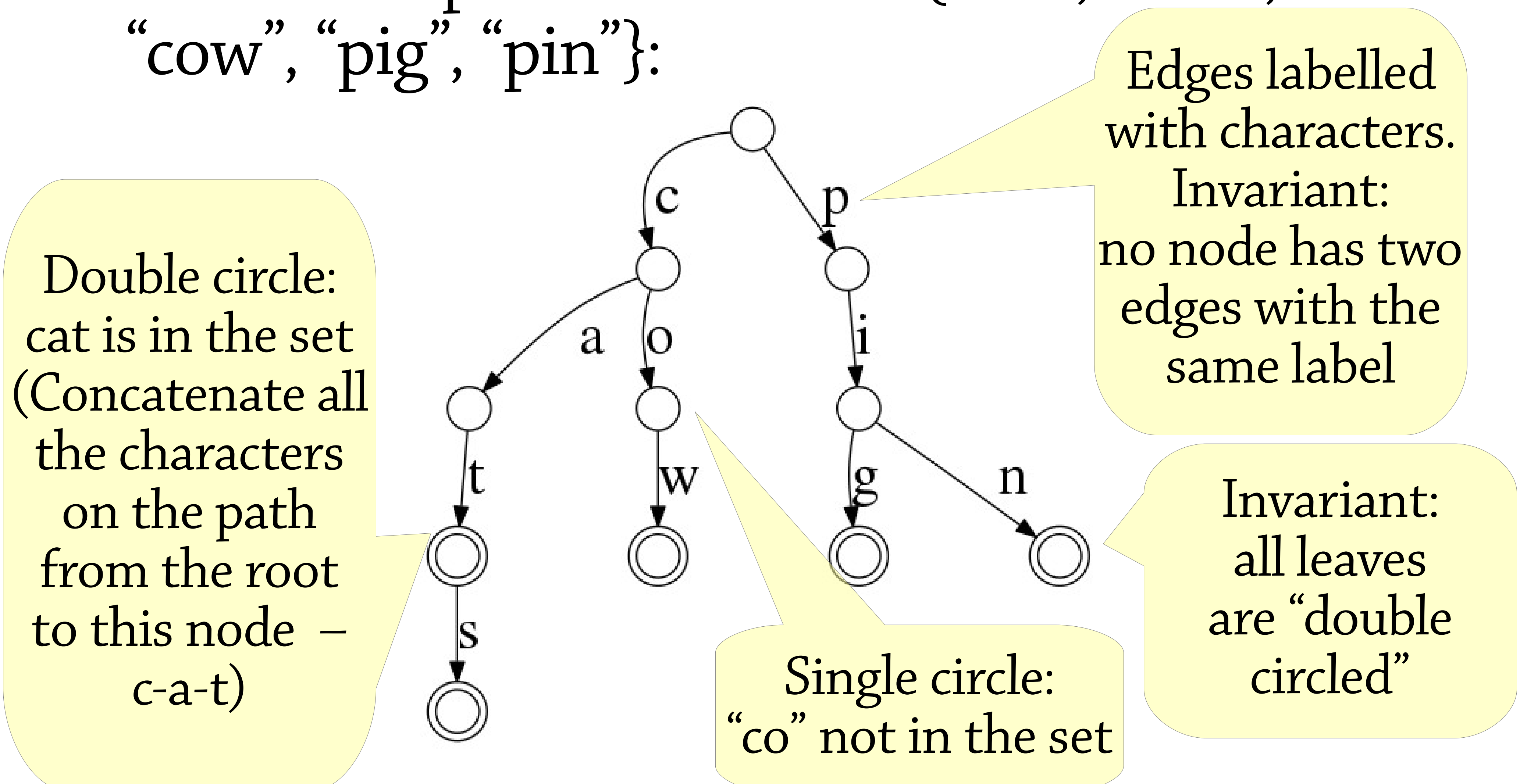
It is a kind of tree, but not based on comparisons

Name: pun on *re**trie**val* and *tree*

- Originally pronounced “tree”, but now pronounced “try” to avoid confusion with trees…

Tries

This trie represents the set {“cat”, “cats”, “cow”, “pig”, “pin”}:



Tries, more formally

A trie is a tree where edges are labelled with characters

- Represents a set or map of strings
- More generally, keys can be lists; the edges are labelled with single elements

Each node in the tree represents a string

- Which string? Follow the path from the root to the node and concatenate the characters on those edges

Some nodes are marked as corresponding to an element of the set

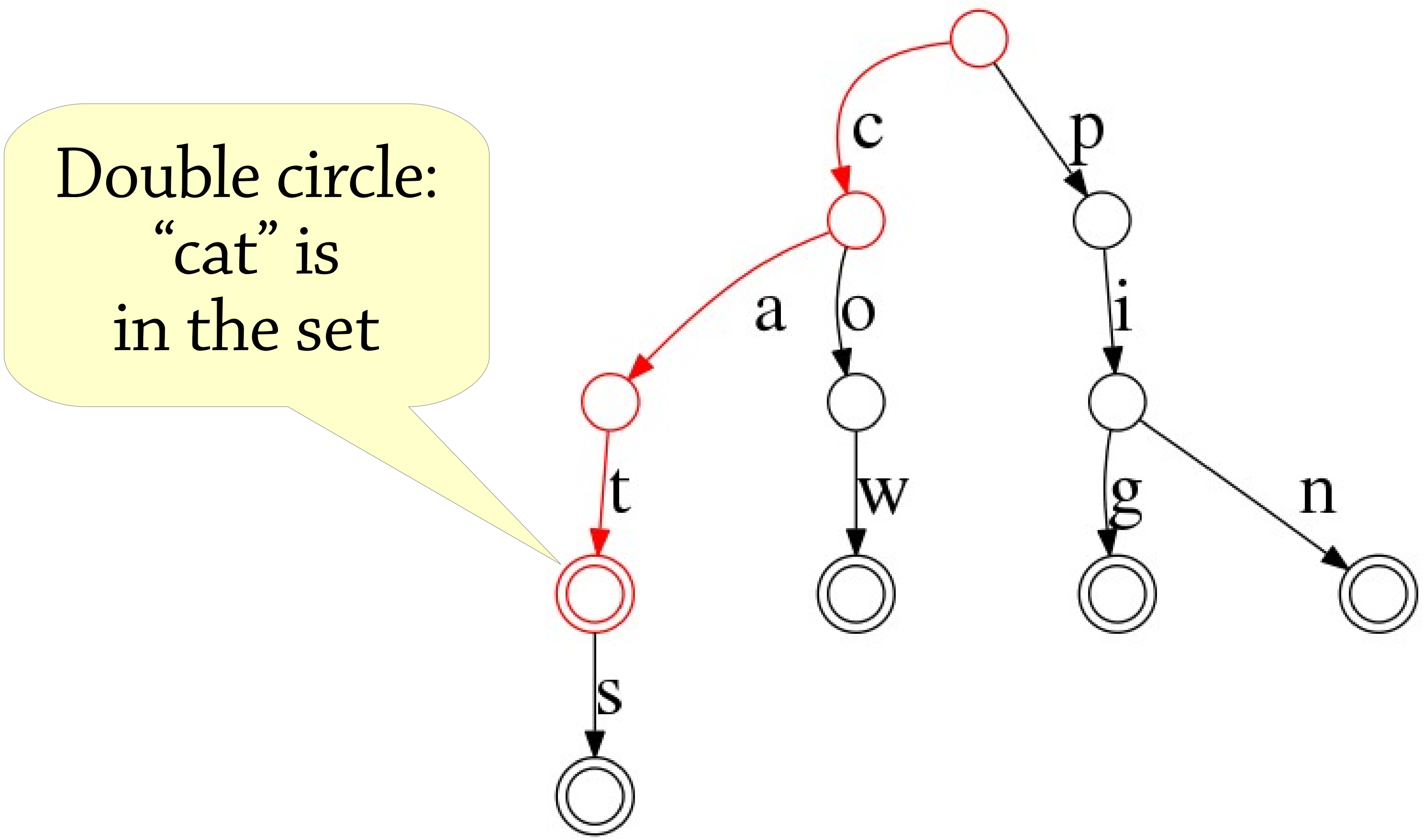
- In diagrams: a double circle

Invariant:

- Each node has at most one child labelled with a given edge
- All leaf nodes are "double circles" (they represent elements of the set)

Tries

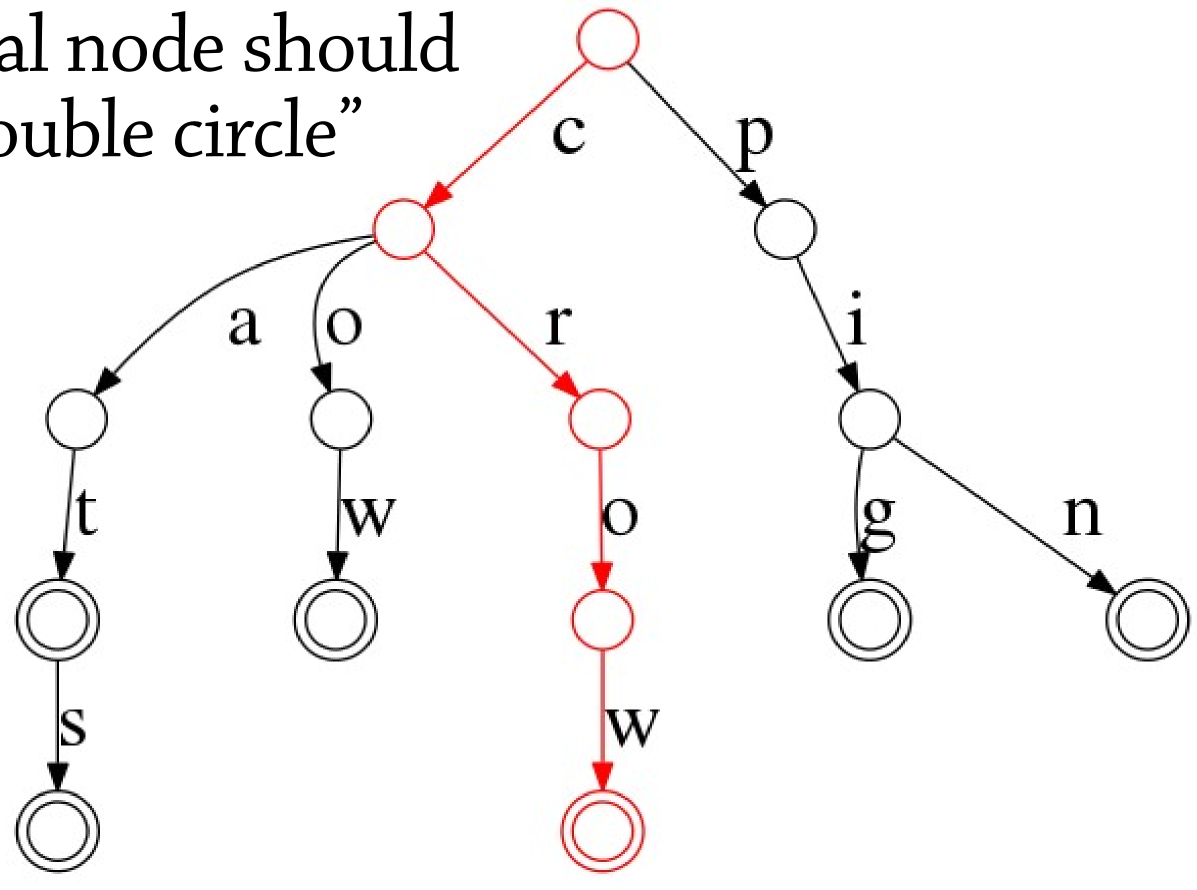
To check if a string is in the set, just follow the edges, starting from the root!



Tries

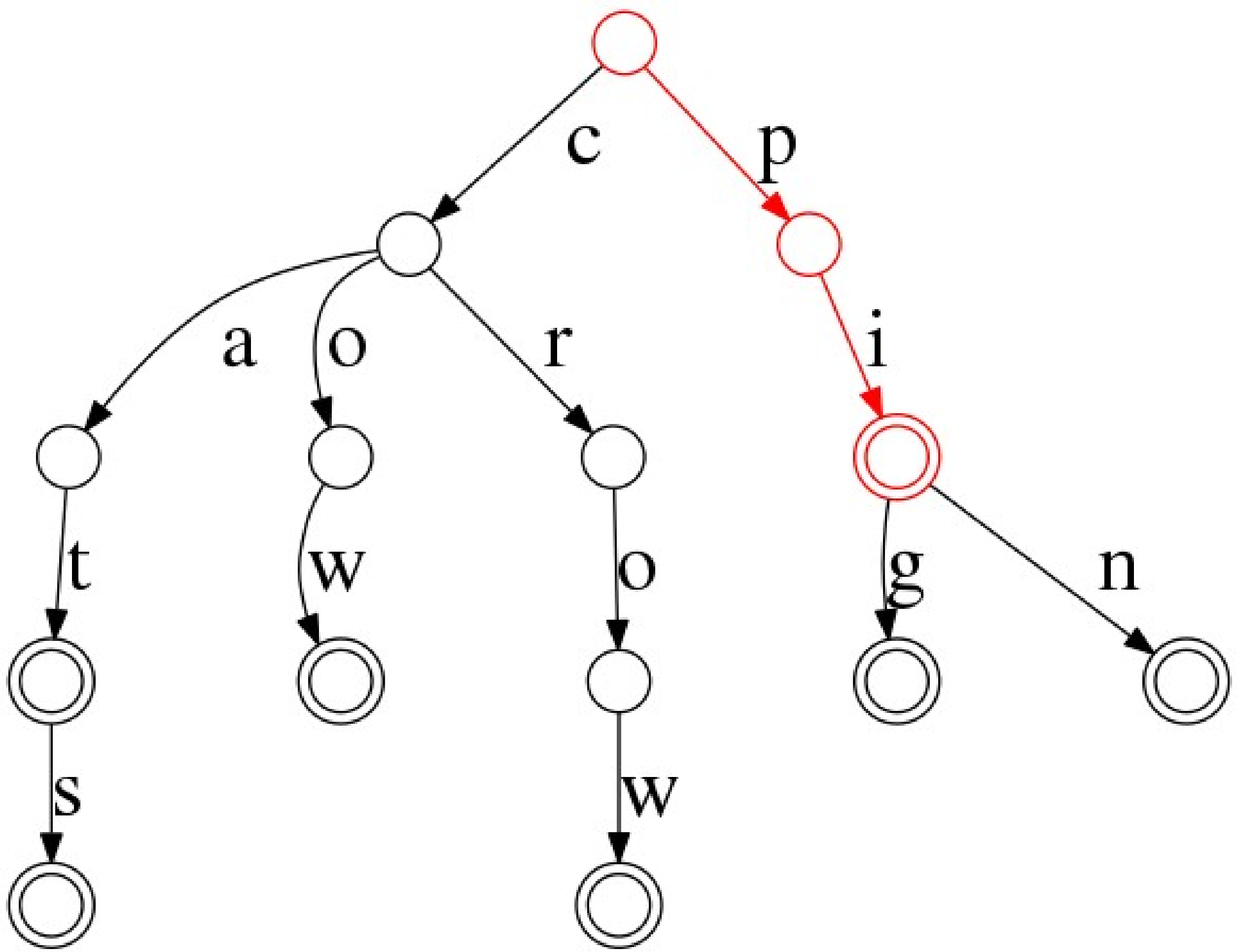
To insert a new string, also follow the edges, making new nodes as you go

- The final node should be a "double circle"



Tries

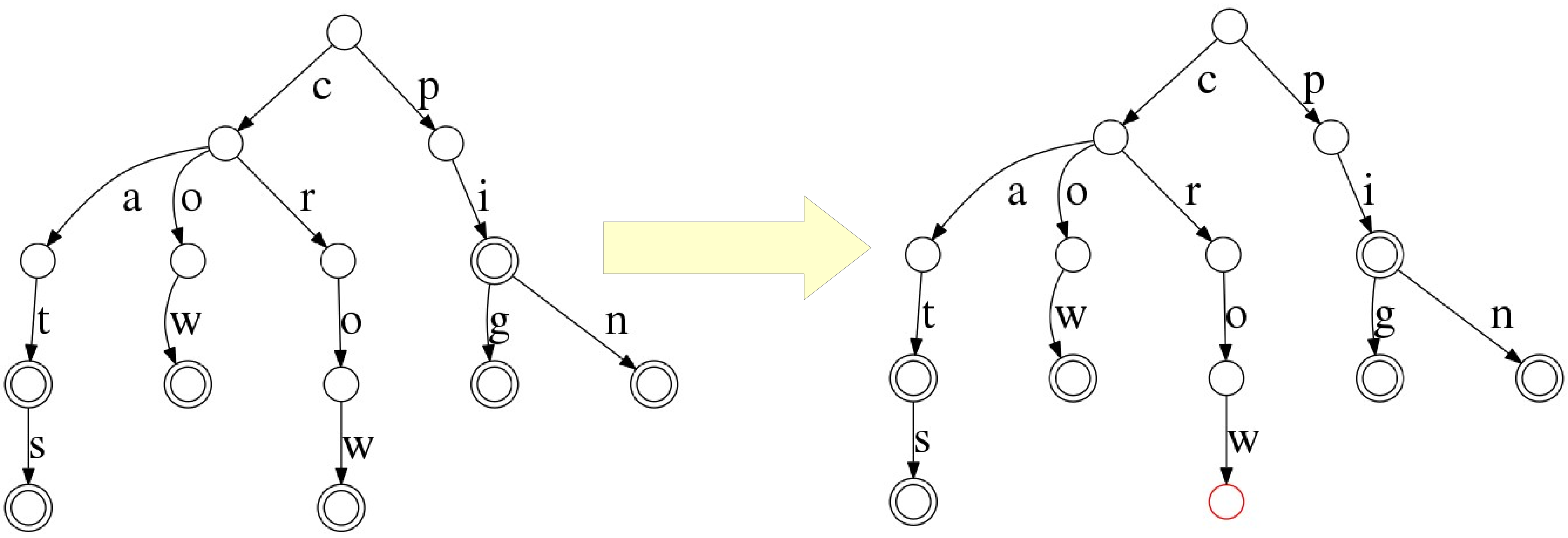
Inserting “pi” creates no new nodes, but we mark the final node as a “double circle”



Tries

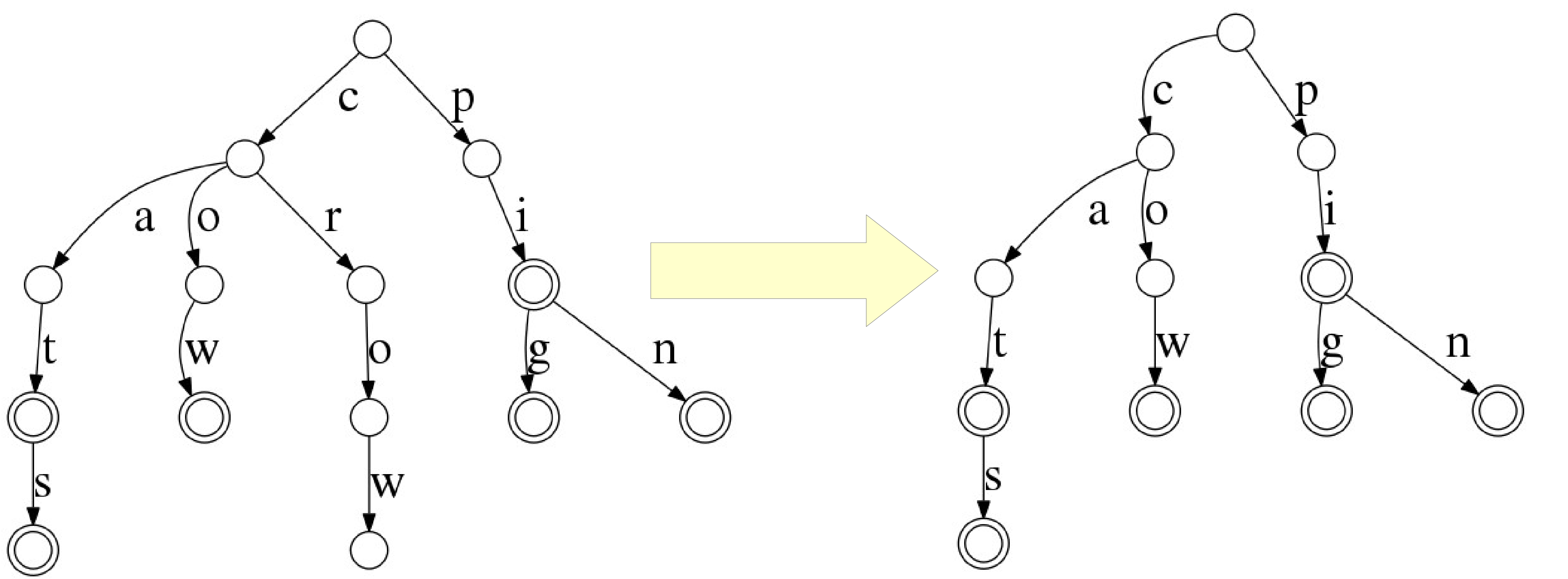
To delete a string, we first turn the node into a “single circle”...

Example: deleting “crow”



Tries

If the node is a leaf, we should remove it. We go up the tree removing any single-circled leaves, which restores the invariant:



Tries – other neat things we can do

Given a set of strings stored as a trie, we can:

- Find all strings starting with a given prefix
  (for this reason a trie is often called a *prefix tree*)

We can take the union or intersection of two tries

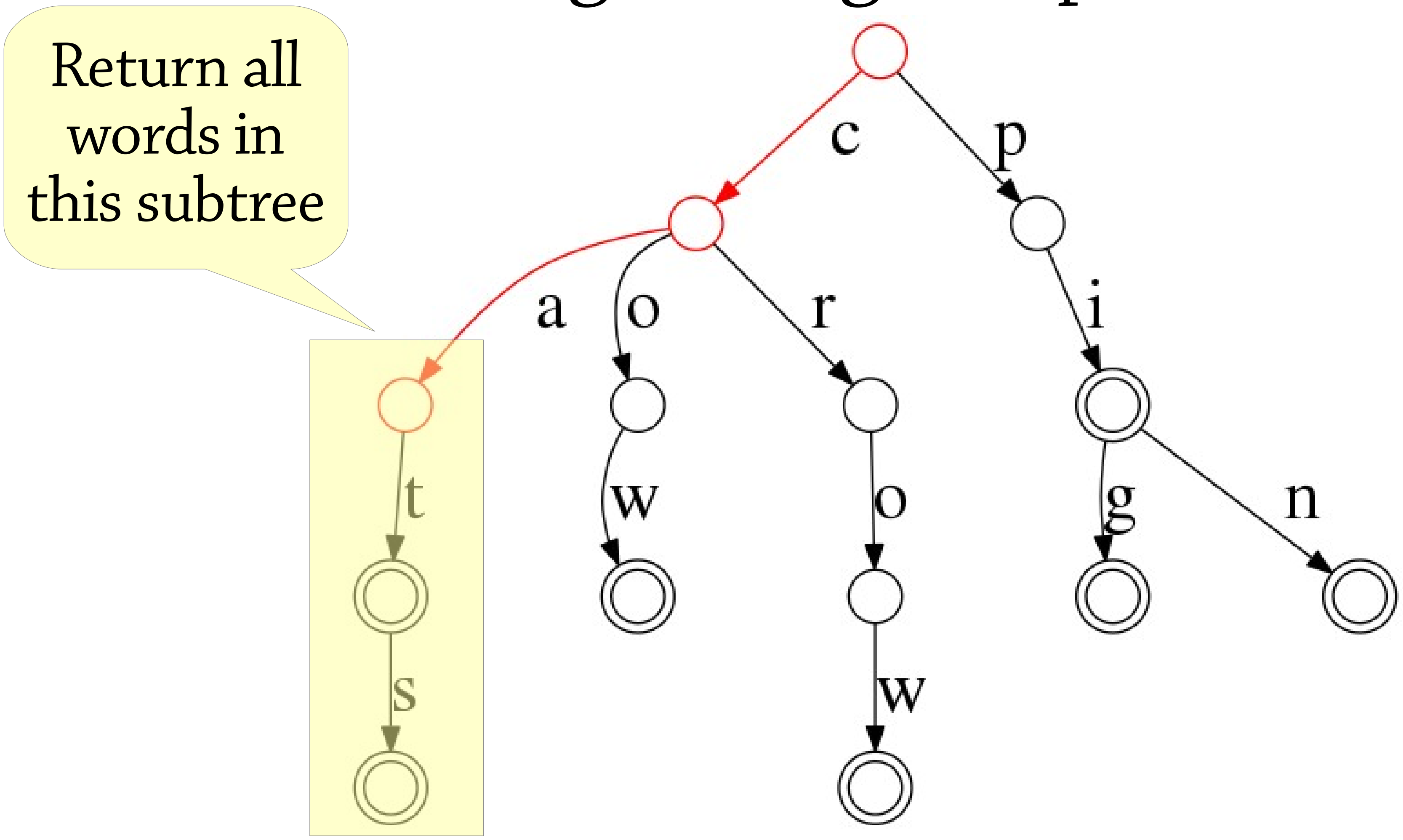
- Linear time, but much faster if the two tries are mostly disjoint

If we can iterate over all edges of a node in alphabetical order, we can also:

- Generate a list of strings in dictionary order (i.e., we can use a trie for sorting)
- Find all strings lying between two words in dictionary order
  (e.g., all words in the set that are after "chicken" but before "pickle" in the dictionary)

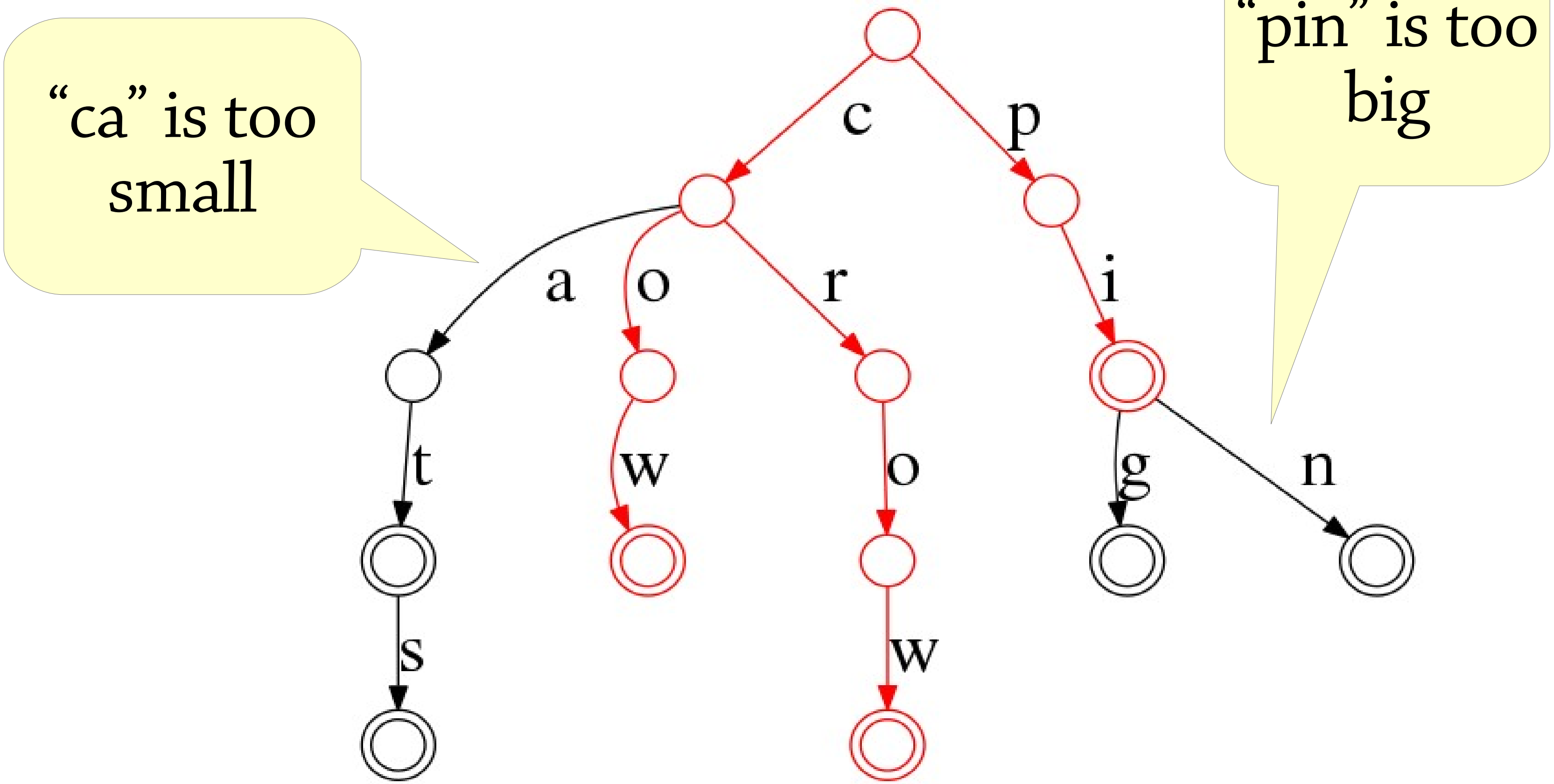
Tries

To find all strings starting with a prefix, just follow the edges along that prefix:



Tries

To find all strings between “chicken” and “pickle”, just follow all edges that lie between them in dictionary order:



Tries – implementation

How to represent a trie? Not obvious:

- Edges are labelled
- Each node can have many edges

One reasonable choice: each node carries a *map* from label to child node

- e.g., using a hash table means that following an edge will take O(1) time

"Double circles" are recorded by having a Boolean field in the node object

- If the trie is used as a map, each node object can contain a value

Just as with any tree data structure, the trie itself is represented as a reference to the root node

Fairly simple to implement!

Tries – performance

Trie operations take O(w) time, where w is the length of the string to be inserted

- Independent of the number of strings stored in the trie!

Is this better or worse than BSTs?

- Better if the trie consists of many short strings, because each node will have many children
- Worse if the trie consists of few long strings, because many nodes will only have one child

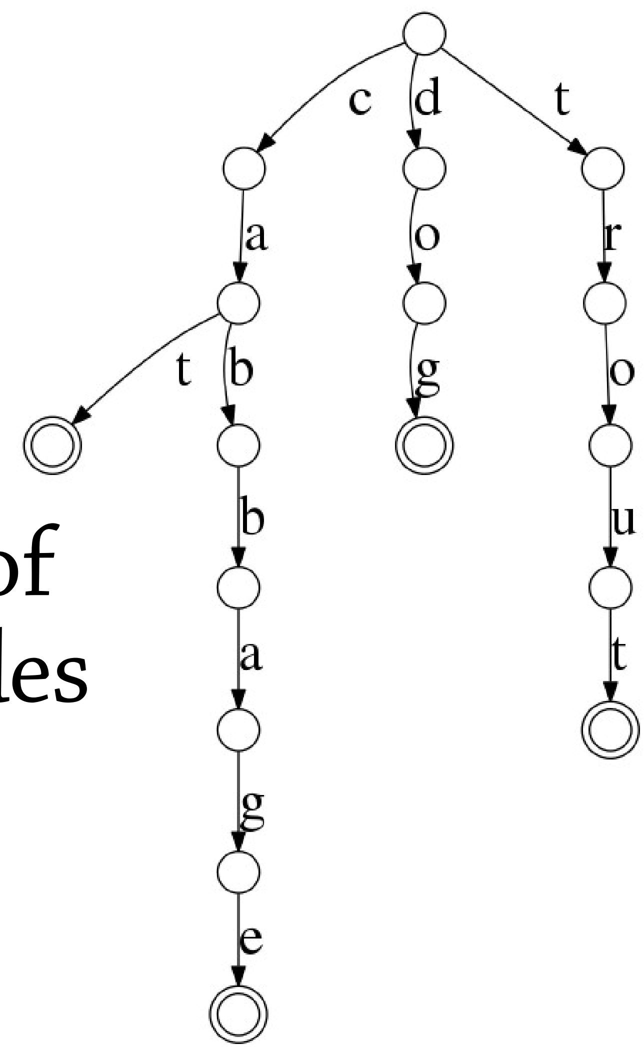
Tries – a bad case for performance

Tries containing few long strings perform worse than BSTs

Many nodes have one child!

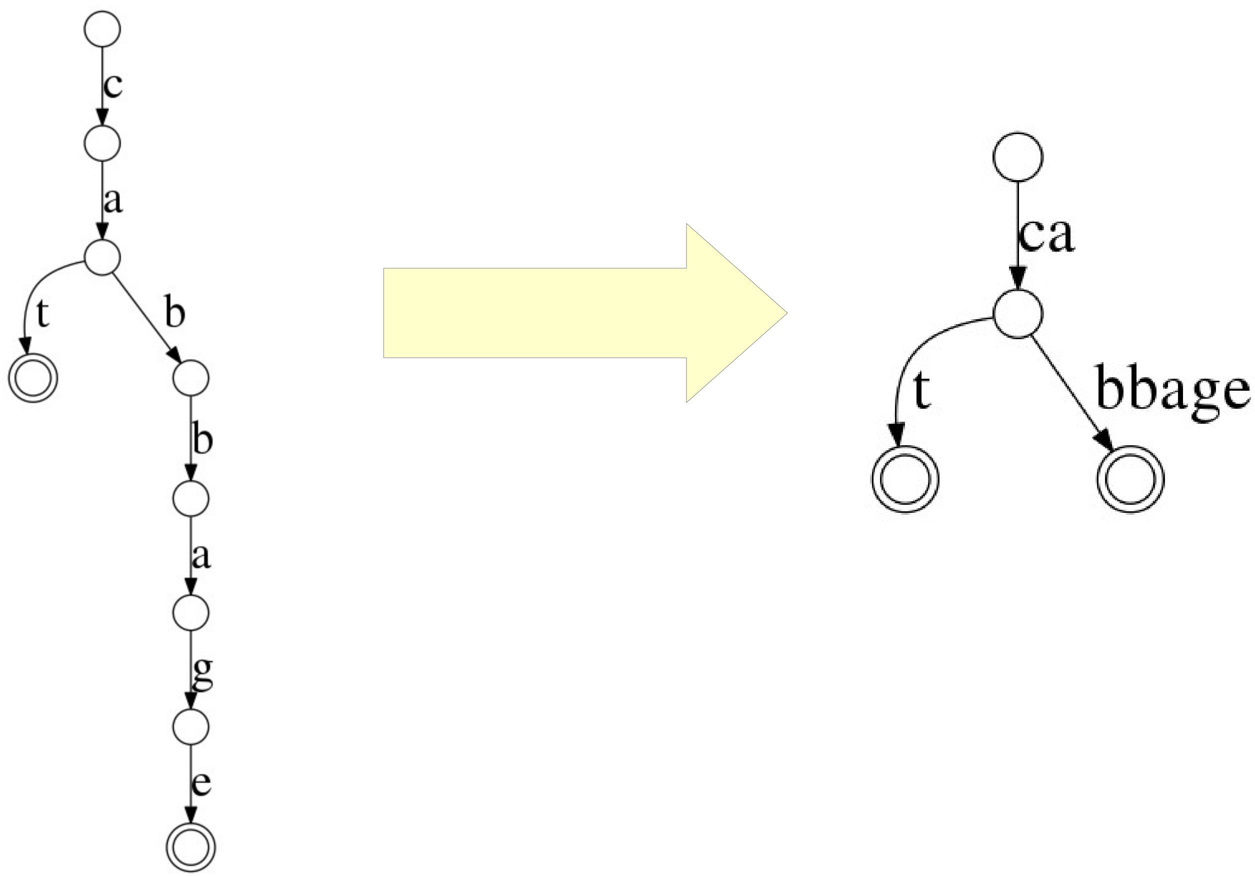
Long chains of nodes without any branching

Radix trees are a refinement of tries that only introduce nodes when branching is needed



Radix trees

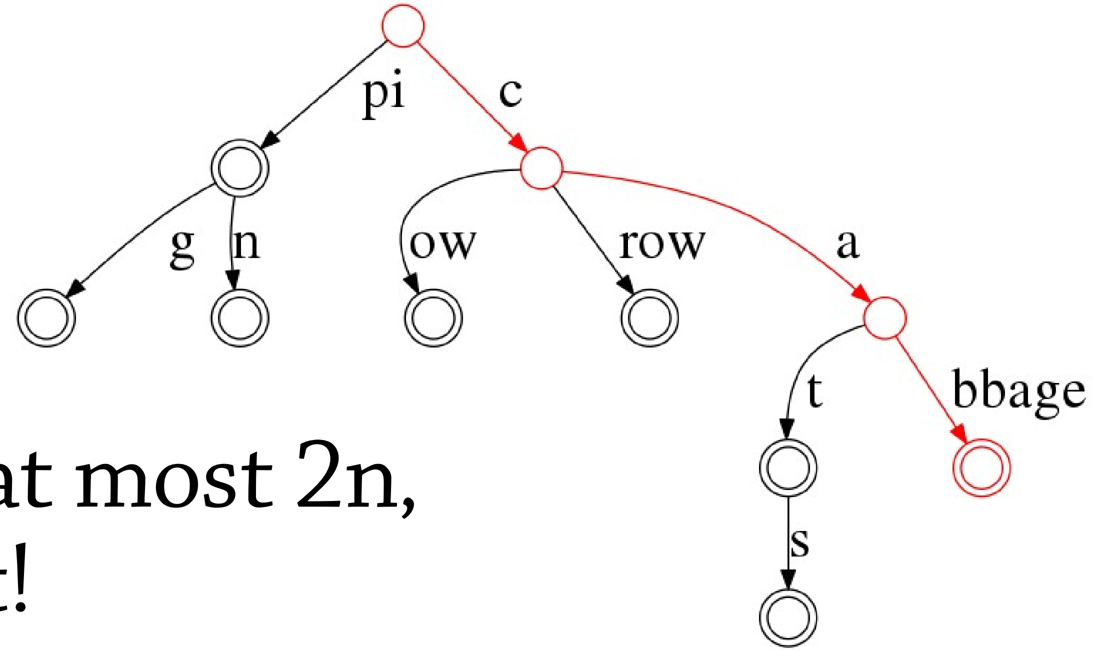
Idea: label edges with *strings* rather than characters, and compress chains of nodes into a single string



Radix trees

Finding values in a radix tree works the same as in a trie

- Important invariant: each node only has one outgoing edge starting with each letter!
- Can also maintain: each non-double-circled node has at least 2 children

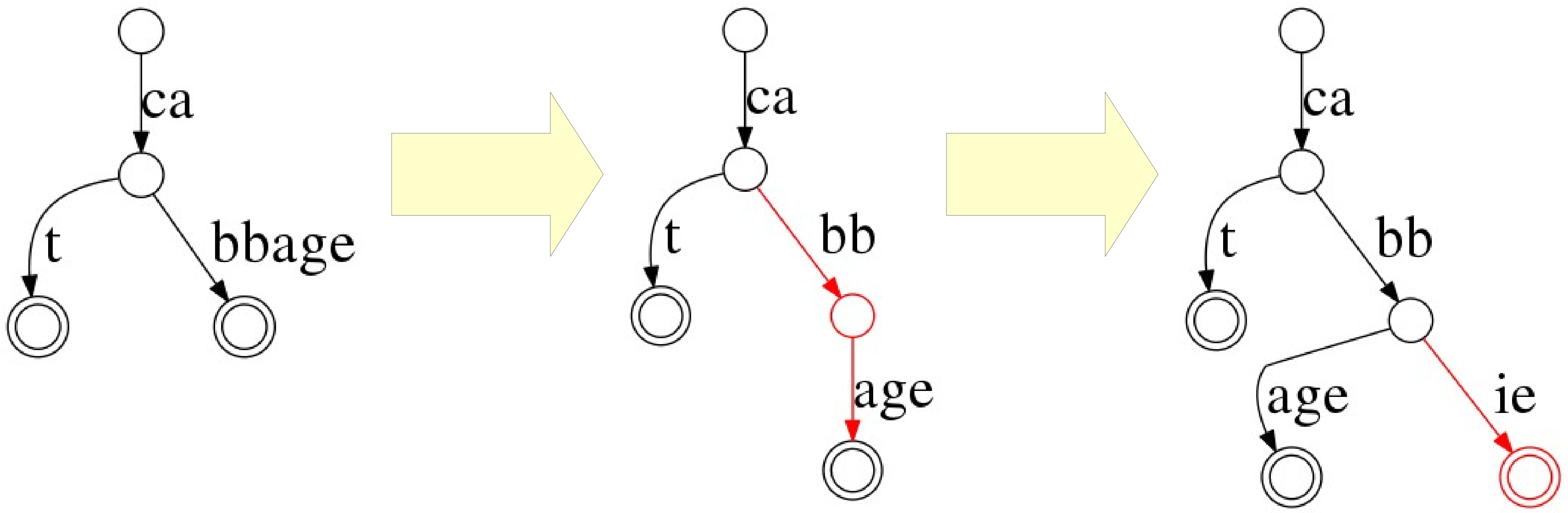


Theorem:
number of nodes is at most 2n,
where n is size of set!

Radix trees

Insertion works like in a trie, except that you sometimes have to split an edge into two

E.g. to insert "cabbie", we have to split "bbage" into "bb" and "age":



Radix trees – implementation

To navigate in a radix tree we need to be able to look up a *character* and get the *outgoing edge starting with that character*

So, each node stores its outgoing edges as a map:

- the key is the first character of the label
- the value is a pair (rest of the label, target node)

Apart from that, implementation is similar to tries

- Main other difference: splitting an edge in two

Suffix trees

A suffix tree is a radix tree that stores *all suffixes of a given string*

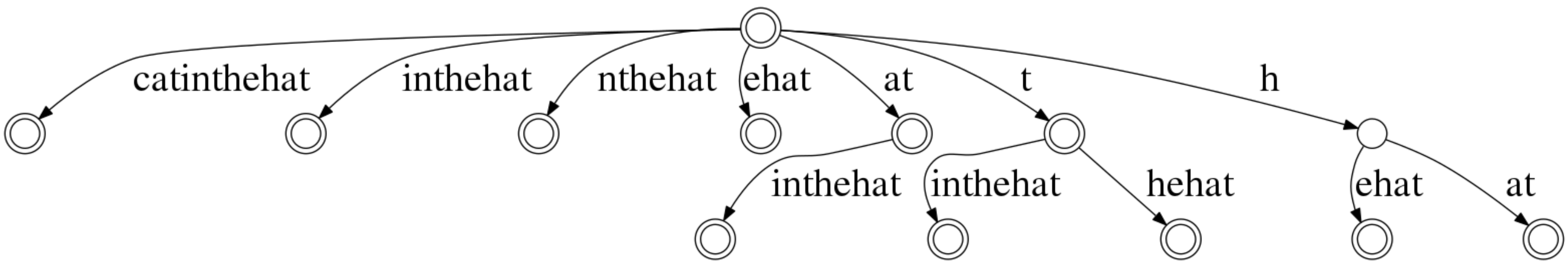
- Example: suffixes of "catinthehat" are: "catinthehat", "atinthehat", "tinthehat", etc.

Why? Can be used to search for all occurrences of given substring in a string

- In a radix tree, you can find all strings that start with a given prefix
- In a suffix tree, you can find all suffixes of a string that start with a given prefix
- This is the same as finding all occurrences of the prefix
- "at" is a substring of "catinthehat" if and only if some suffix of "catinthehat" starts with "at"

Suffix trees

A suffix tree for “catinthehat”:



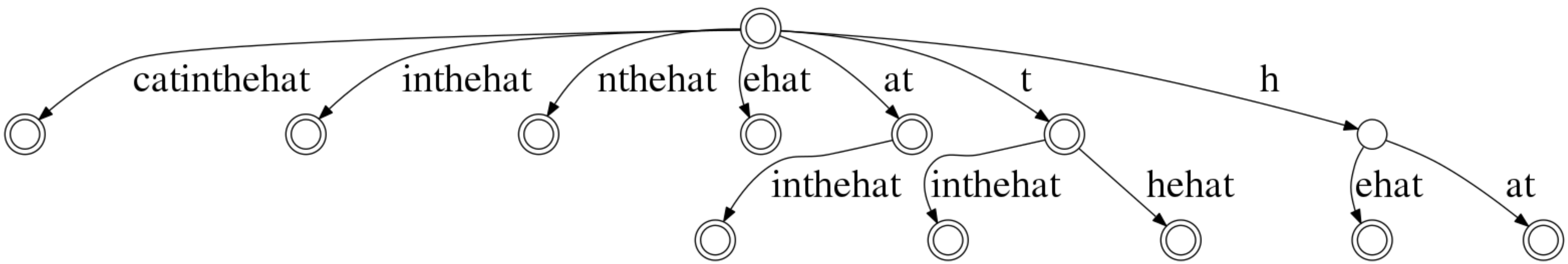
To find “at” in “catinthehat”, let’s check which suffixes start with “at”. There are two:

- “atinthehat” → at occurs followed by “inthehat”
- “at” → at occurs followed by the end of the string

From the length of the suffix we can tell what positions “at” occurs at!

- “catinthehat”.length - “atinthehat”.length = 1
- “catinthehat”.length - “at”.length = 9

Suffix trees, implementation



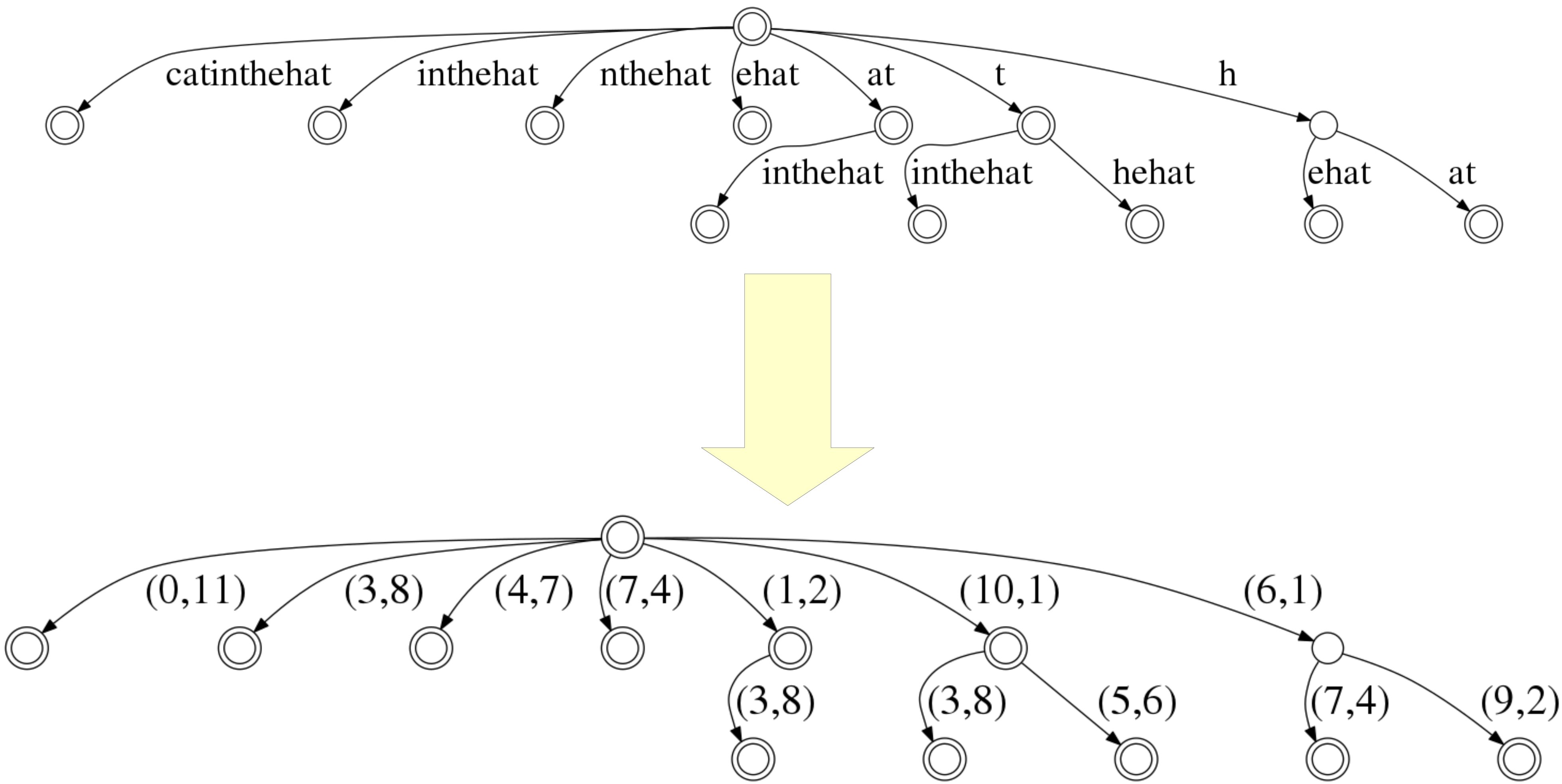
If implemented carelessly, this takes $O(n^2)$ memory!

- Each edge is labelled with a substring of the input string, which may take O(n) memory to store

The trick:

- Remember the original input string
- Label each edge with data that records *which substring* of the input string it is
  One way: a pair (position in input string, length)
  e.g. "intheh" would become (3, 6) – starts at index 3 of "catinthehat" and goes on for 6 characters

Suffix trees, implementation



(need to also remember that the input string is catinthehat)

This takes O(n) memory! (Recall that a radix tree containing n values has at most 2n nodes)

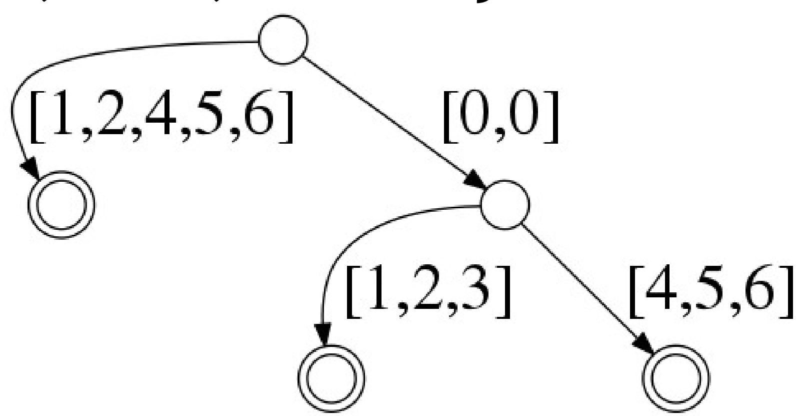
Radix trees for numbers (not on exam)

You can view an integer as a sequence of digits

- e.g. 12345 → [1, 2, 3, 4, 5]

So you can use radix trees to store sets of numbers!

Example: {123, 456, 12456}



Note: we pad the numbers with leading zeroes when necessary, so that we can do range queries like “find all the numbers between 100 and 500”

Radix trees for numbers, implementation (not on exam)

We can use several tricks to implement radix trees for numbers super-efficiently!

- Instead of storing the children of each node in a map, store them in an array of size 10 (one for each digit)
- Store strings of digits as a pair
(number, length of number)
e.g. [0,1,3] becomes (13, 3)
- Don't use base 10 but (e.g.) base 16, so that we can use e.g. bit-shifting instead of division – in other words, we view an integer as a list of 4-bit numbers ("hexadecimal digits")

Radix trees for strings, using radix trees for numbers

If we have a radix tree for lists of 4-bit numbers we can use it to store strings!

We can view a string as a series of 4-bit numbers:

- Each character has a character code, which is an 8 to 32-bit number (depending on the encoding)
- Chop up the string into a list of character codes
  e.g. "hello" → [104, 101, 108, 108, 111]
- Chop up each character code into 4-bit pieces
  e.g. 108 = (binary) 0110 1100 = [0110, 1100]
- Now you have a series of 4-bit numbers

For efficiency, radix trees for strings are often implemented this way!

Summary

Radix trees can be used to implement sets or maps, where the keys are lists

- e.g. strings, or numbers treated as strings of base-16 "digits"
- Requires a map data structure for list elements, often implemented as an array

Time taken by each operation is low!

- O(min(w, log n)) where w is length of string, n is size of set

Tries are simpler to implement, but have O(w) performance

Both also support: finding strings starting with a given prefix, range queries

- Also union and intersection, which we didn't see

Suffix trees are radix trees which store all suffixes of a string, and can be used to find all occurrences of a given substring

- A suffix tree can be built in O(n) time (which we didn't see)
- Then searching takes O(log n) worst-case time