TRIE

Motivation:

- hashmaps are very efficient so far: we can achieve **O(1)** running time for the most important operations

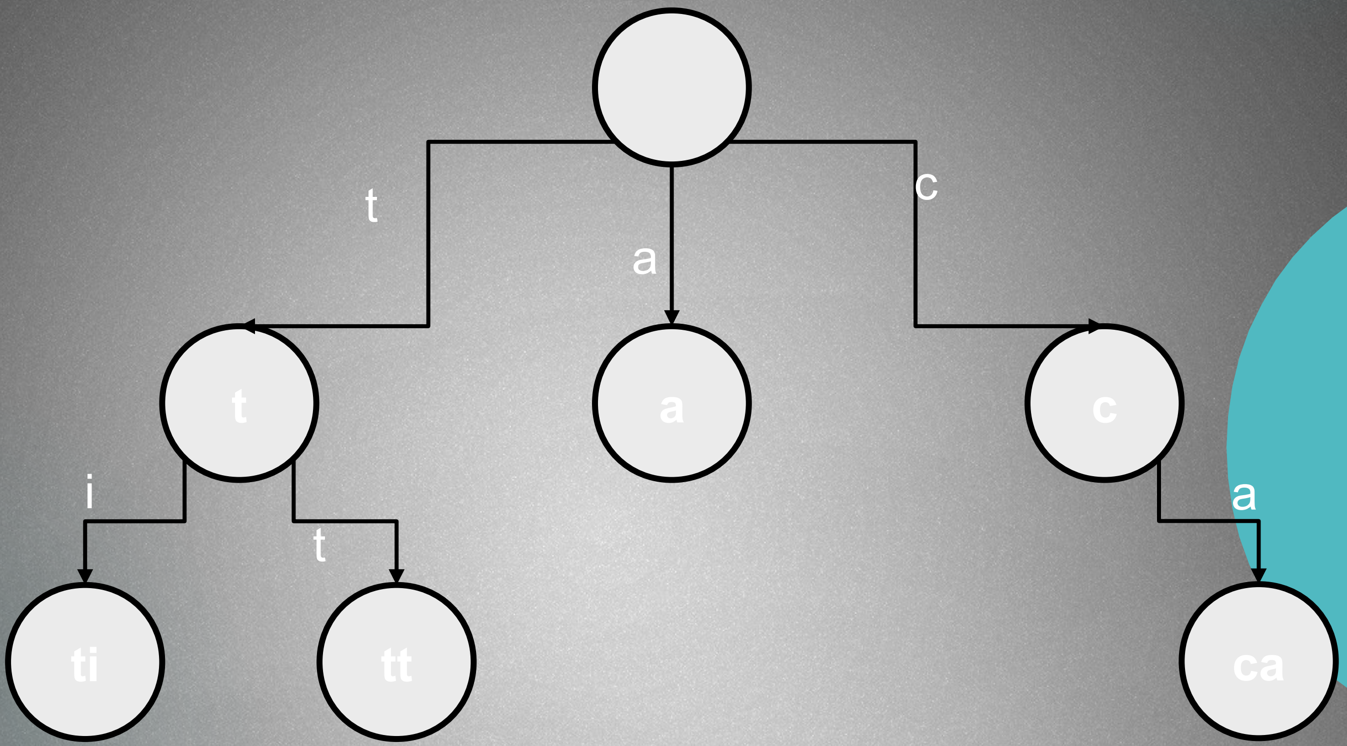
- does not support sorting + hashfunction is usually not perfect: we would like to construct a data structure where search and insert operations have running time proportional to the length of the key !!!

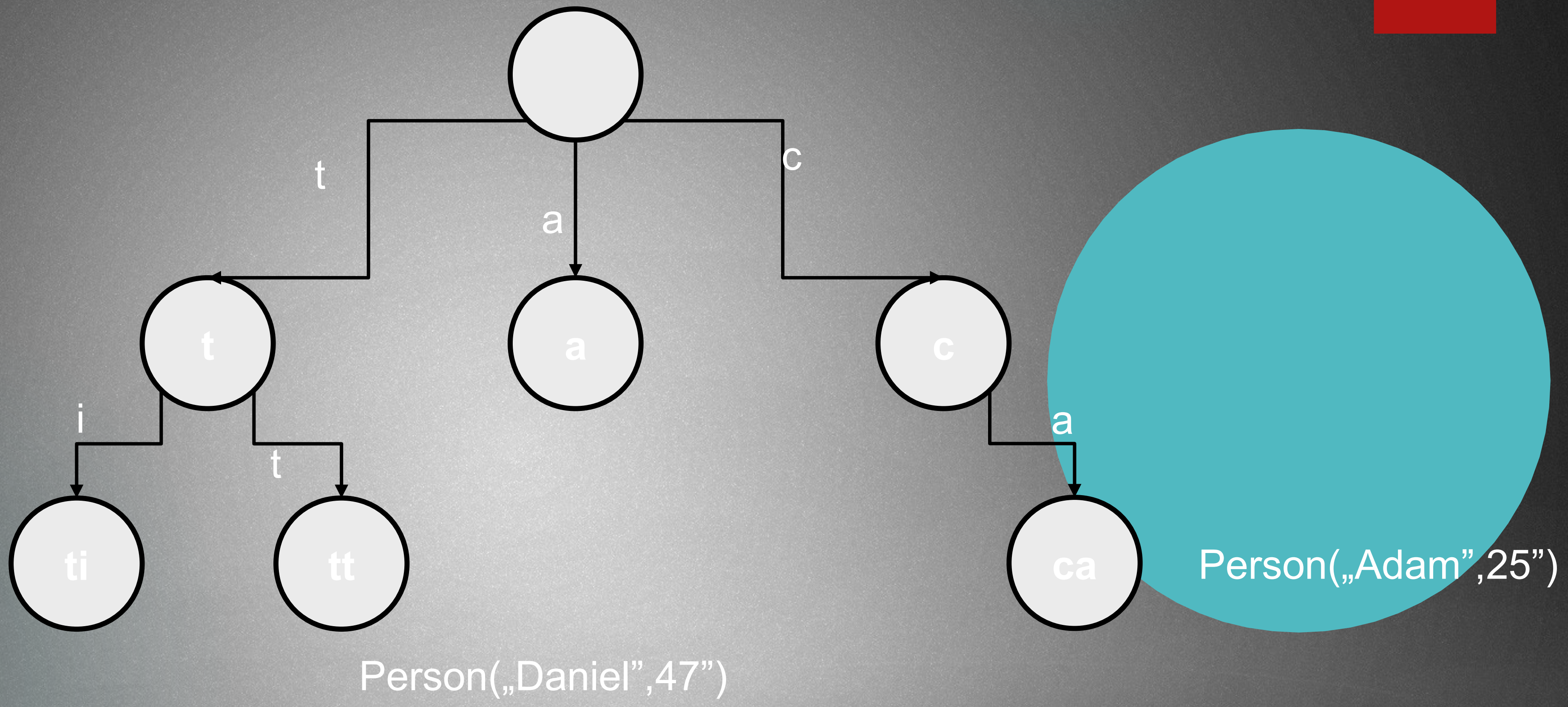
  ~ hashmap worst case search: **O(N)**

- we would like to get rid of collisions: this can be solved with tries + add another feature: sorting !!!

Tries

- Trie / radix tree / prefix tree
- It is a data structure to implement associative arrays
- The keys are usually strings
- Unlike BST no node in the tree stores the key associated with that given node → its position in the tree defines the key with which it is associated
- All the descendants of a node have a common prefix of the string associated with that node, and the root is associated with the empty string
- Values are not necessarily associated with every node // usually leaf nodes only


t
a
c
t
a
c
i
t
a
ti
tt
ca

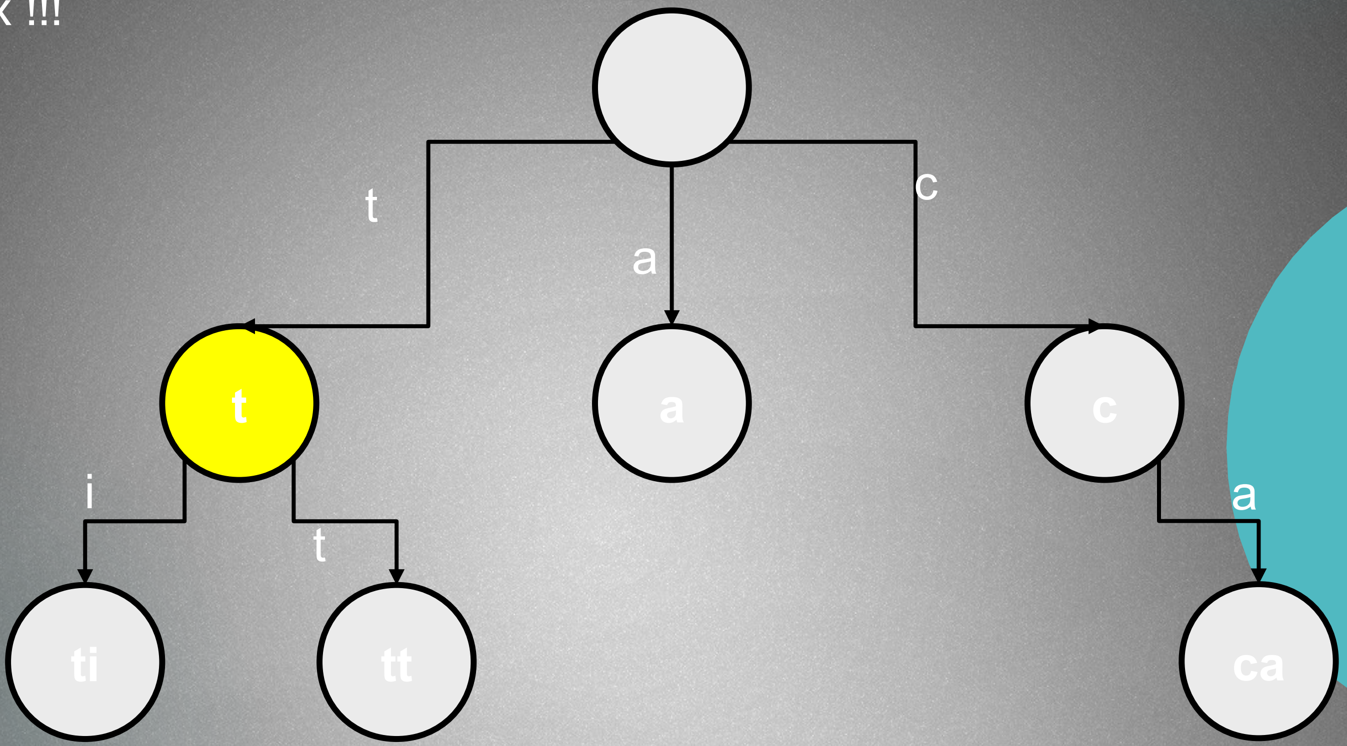


Like hashmaps: we have key-value pairs

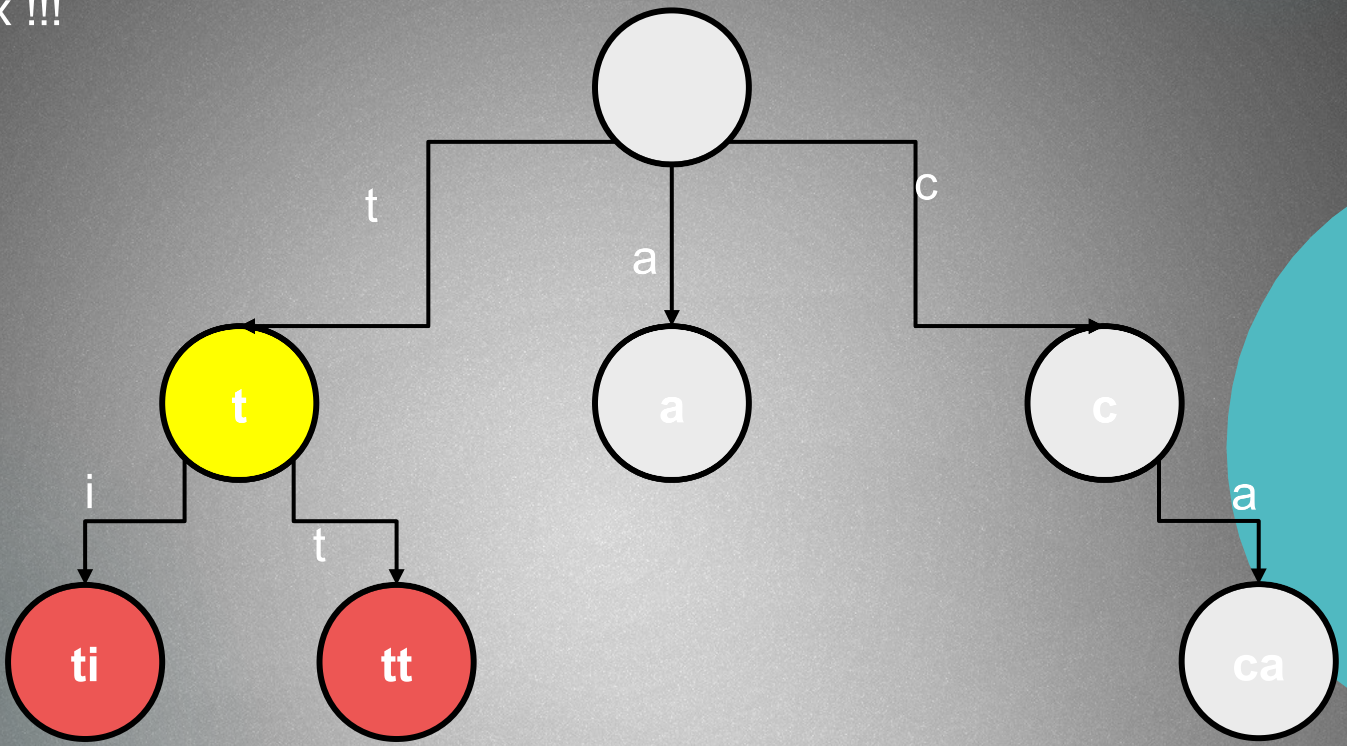
Key: tt → value: a person with name Daniel, age 47

Key: ca → value: a person with name Adam, age 25

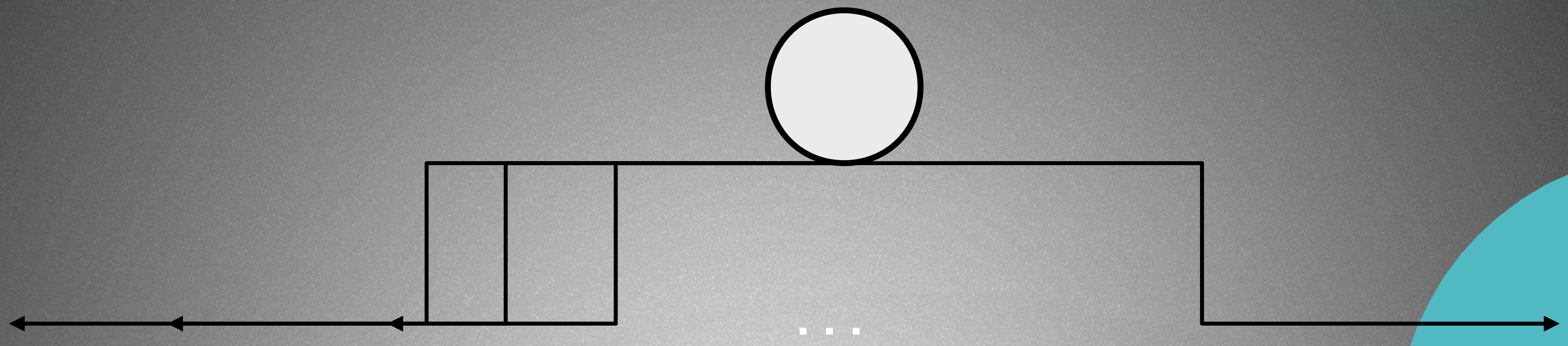
All the descendants of a node has a common prefix !!!



All the descendants of a node has a common prefix !!!



In general



In general we have as many pointers / edges from every node as the number of characters in the alphabet

We have to define an alphabet in advance + ALPHABET_SIZE

For example: in english alphabet there are 26 characters, so

ALPHABET_SIZE = 26 → 26 pointers from every node !!!

In general

```
class Node {
	value
	children Node[ALPHABET_SIZE]
}
```

In general we have as many pointers / edges from every node as the number of characters in the alphabet

We have to define an alphabet in advance + ALPHABET_SIZE

For example: in english alphabet there are 26 characters, so

ALPHABET_SIZE = 26 → 26 pointers from every node !!!

MOST OF THE TIME, WE DO NOT NEED 26 CHILD NODES

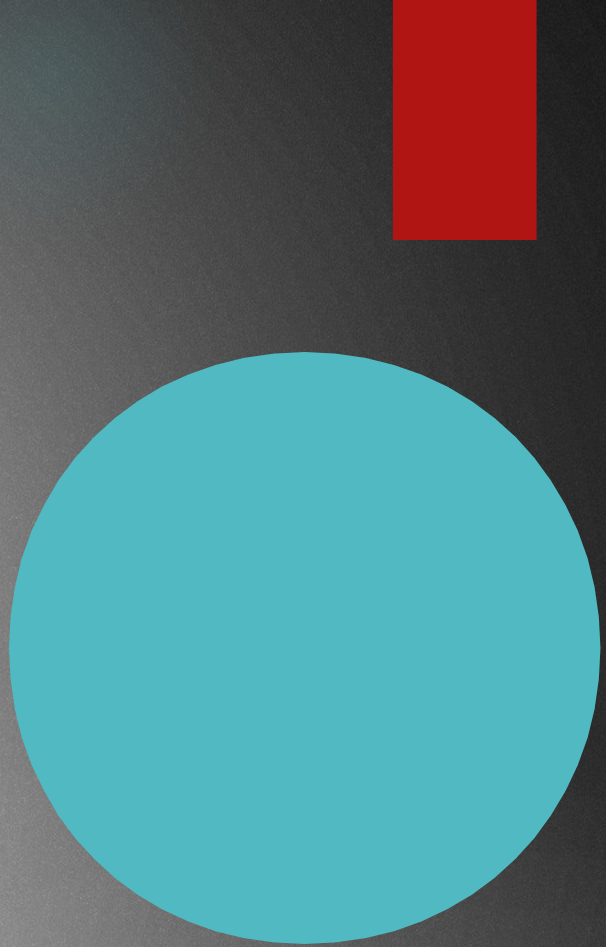
MEMORY INEFFICIENT !!!

Running time – memory tradeoff: it is fast but needs lots of memory (slow, but memory friendly)

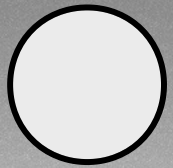
- With the help of tries we can search and sort strings very very efficiently
- The problem is that tries consume a lot of memory, so we should use ternary search trees instead which stores less references and null objects

INSERTION

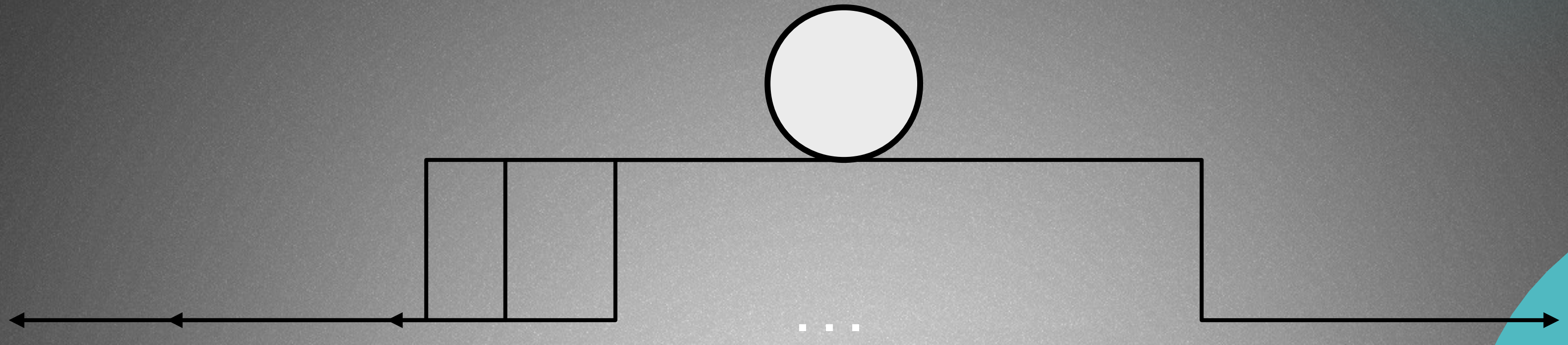
<u>Insertion</u> insert(„apple”)



Insertion insert(„apple”)

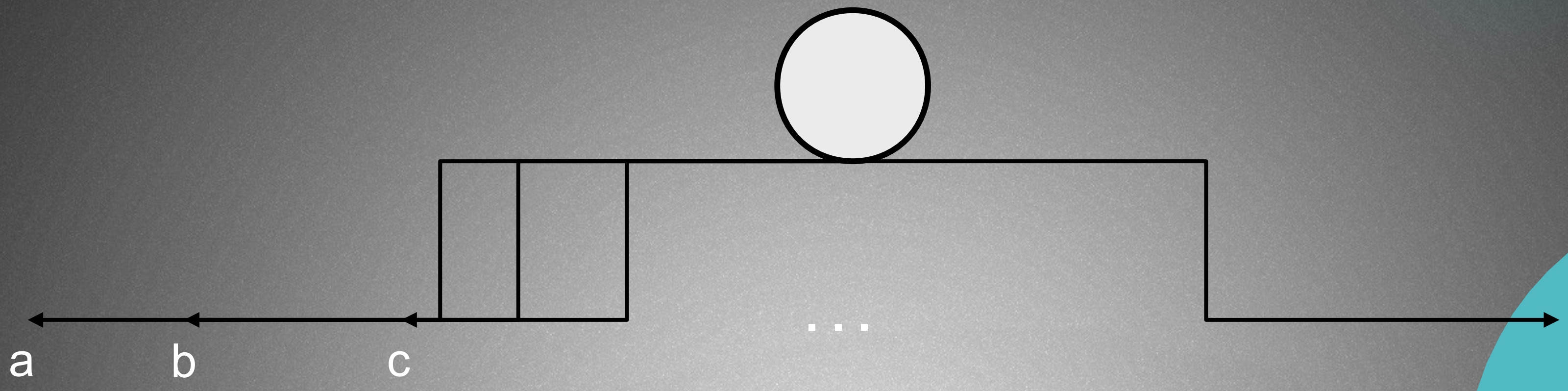


Insertion insert(„apple”)



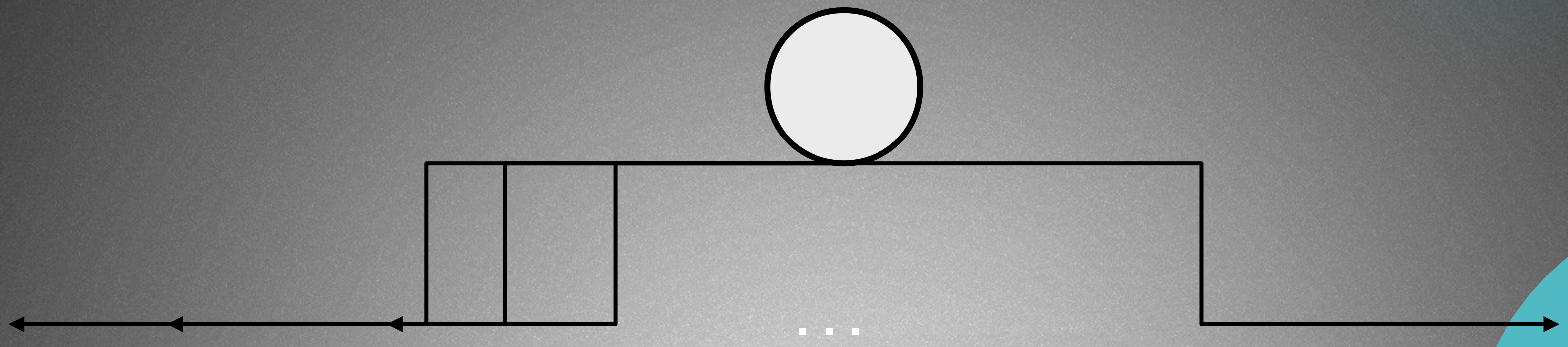
Basically it has 26 children pointing to all the characters in the alphabet → a, b, c …

Insertion insert(„apple”)



Basically it has 26 children pointing to all the characters in the alphabet → a, b, c …

Insertion insert(„apple”)

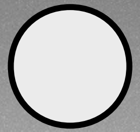


Basically it has 26 children pointing to all the characters in the alphabet → a, b, c …

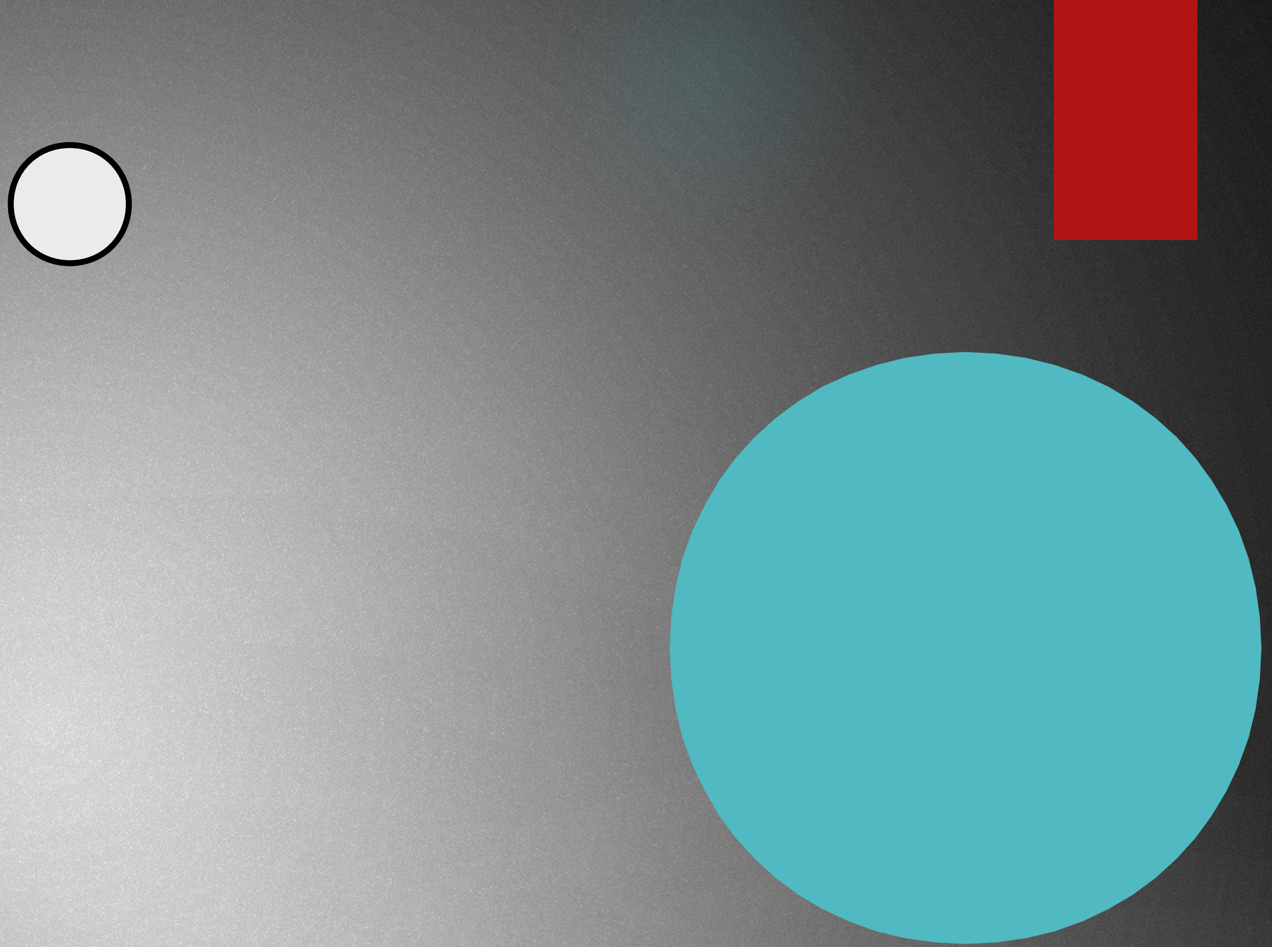
BUT ther are null values at the beginning

When we insert a character → we insert to the right place

Insertion insert(„apple")

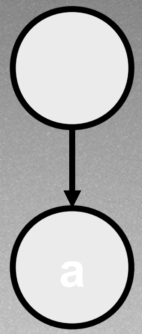


Insertion insert(„apple”)

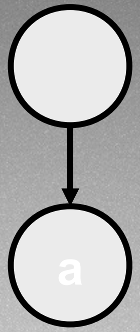


Insertion

insert(„apple”)

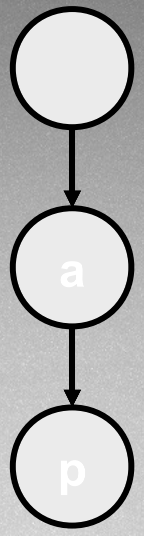


Insertion insert(„**ap**ple”)

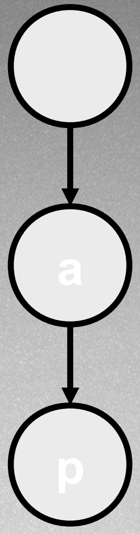


Insertion

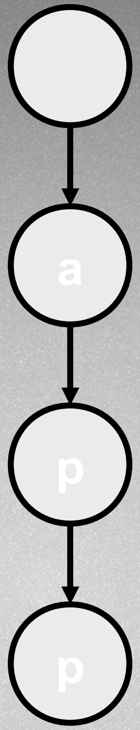
insert(„**ap**ple”)



Insertion insert(„**ap**ple”)

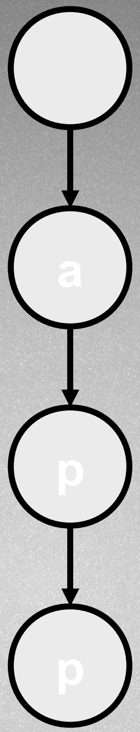


Insertion insert(„**app**le”)

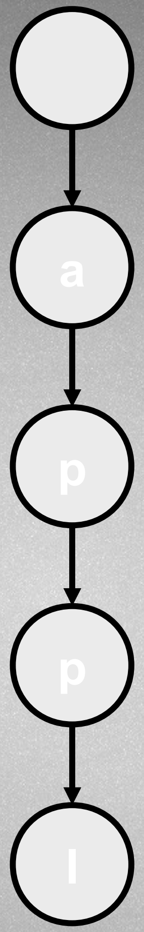


Insertion

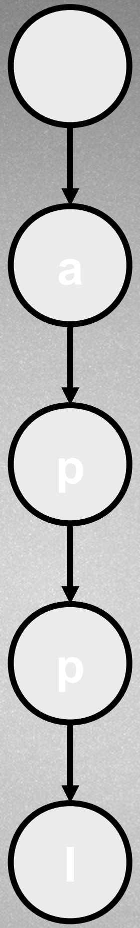
insert(„**apple**”)



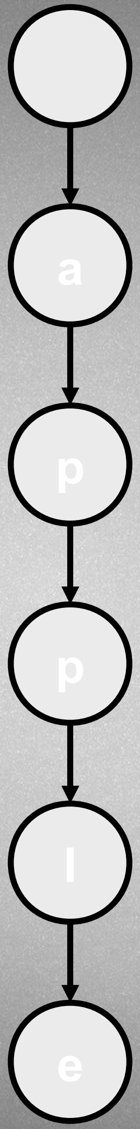
Insertion insert(„**apple**”)



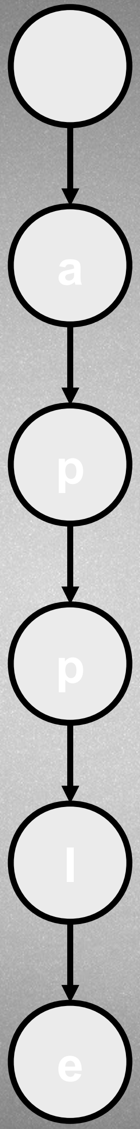
Insertion insert(„**apple**”)



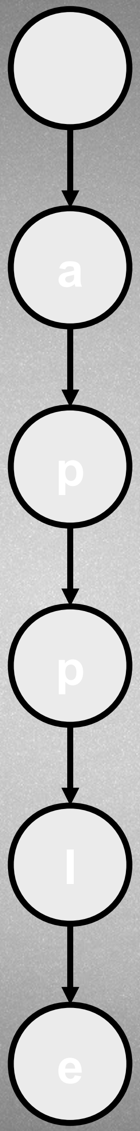
Insertion insert(„**apple**”)



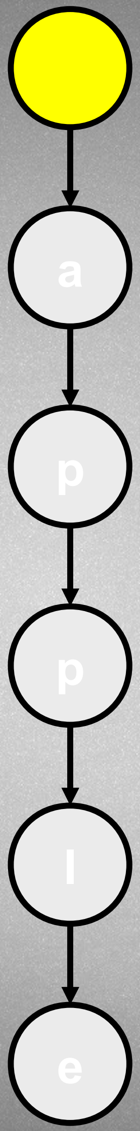
Insertion



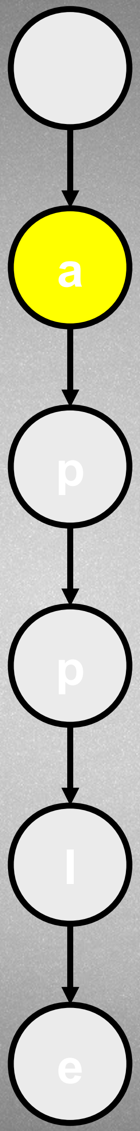
Insertion insert(„air”)



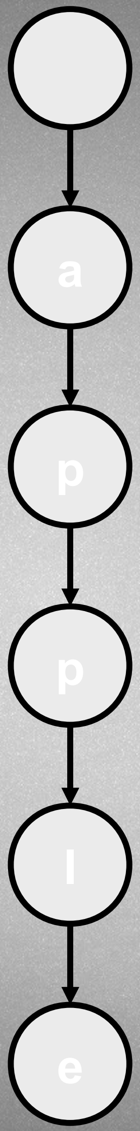
Insertion insert(„air”)



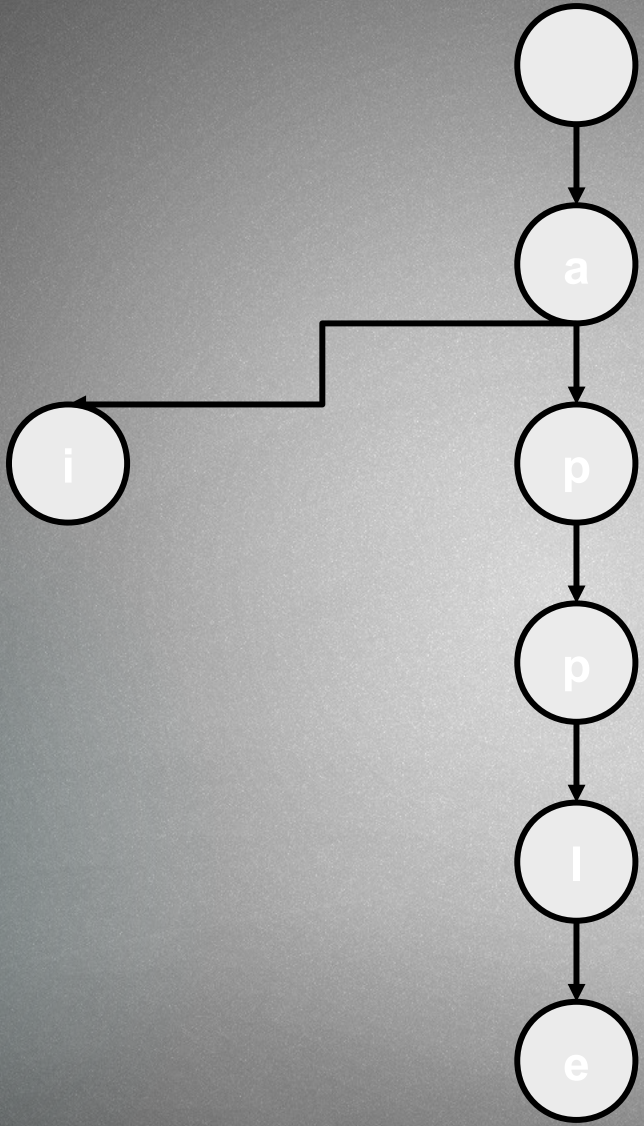
Insertion insert(„air”)



Insertion insert(„air”)

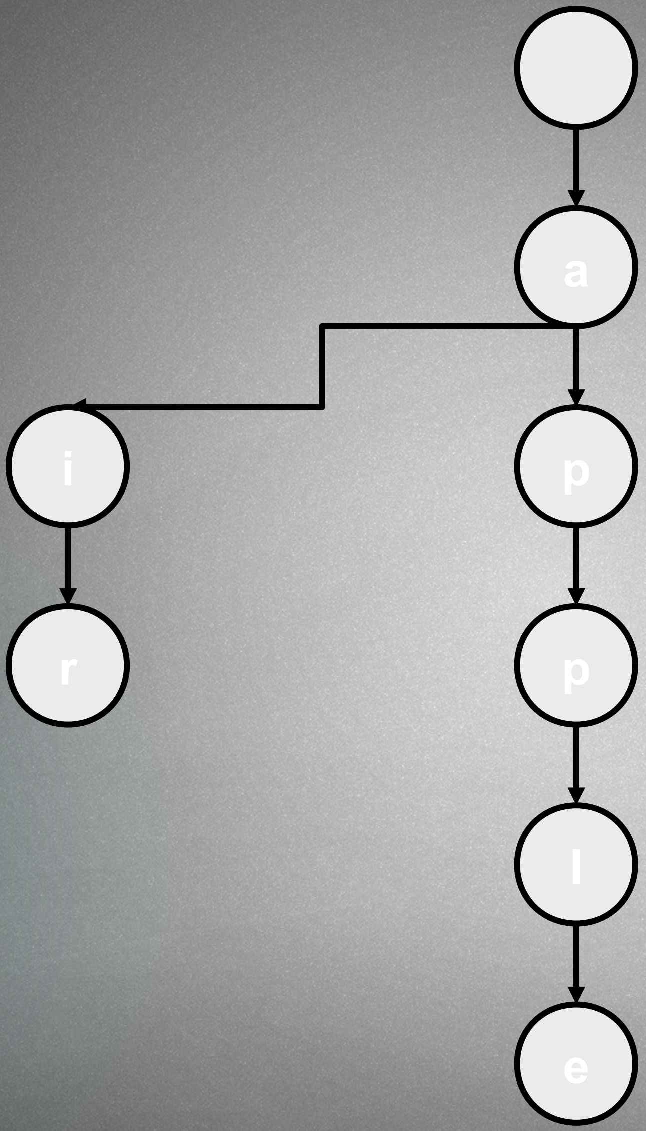


Insertion insert(„air”)

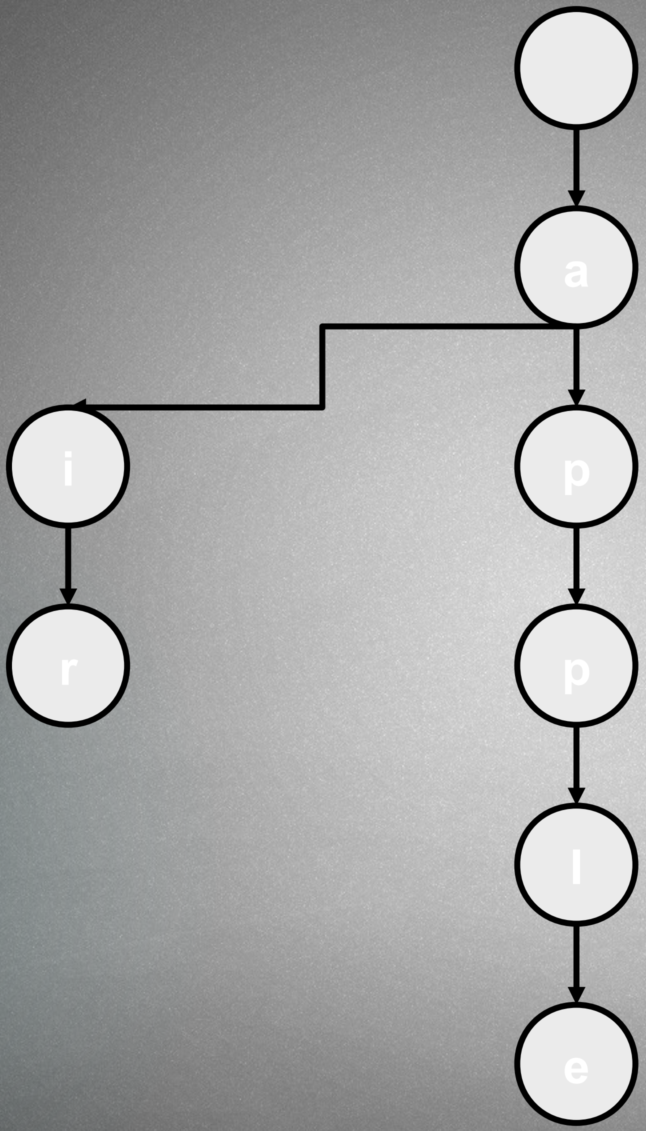


Insertion

insert(„**air**”)

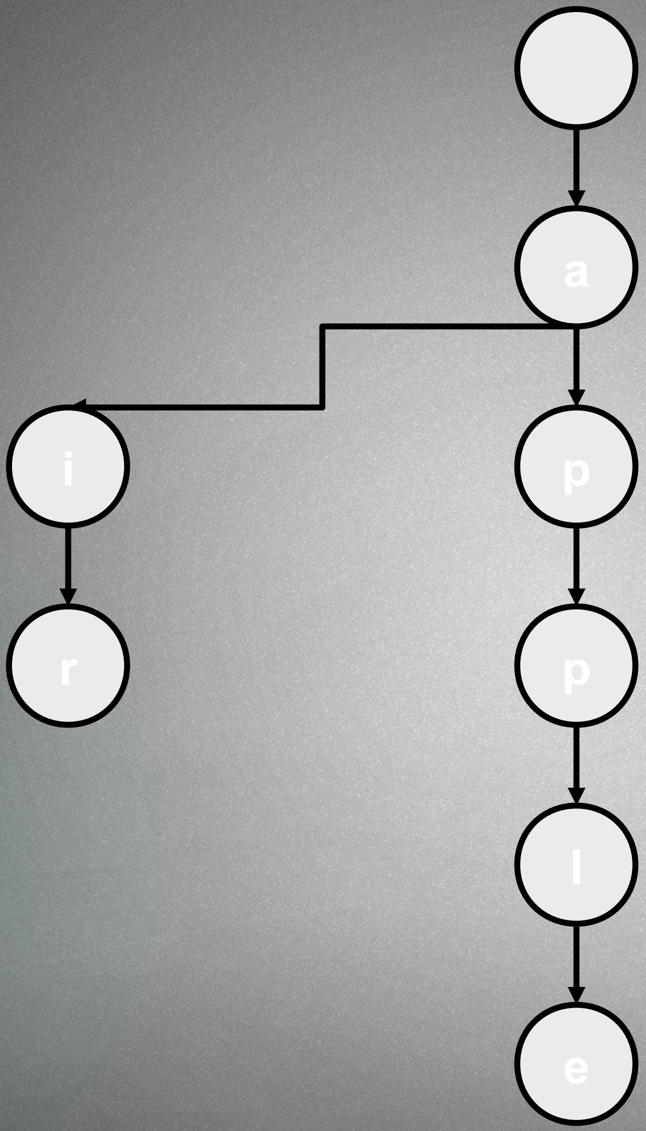


Insertion

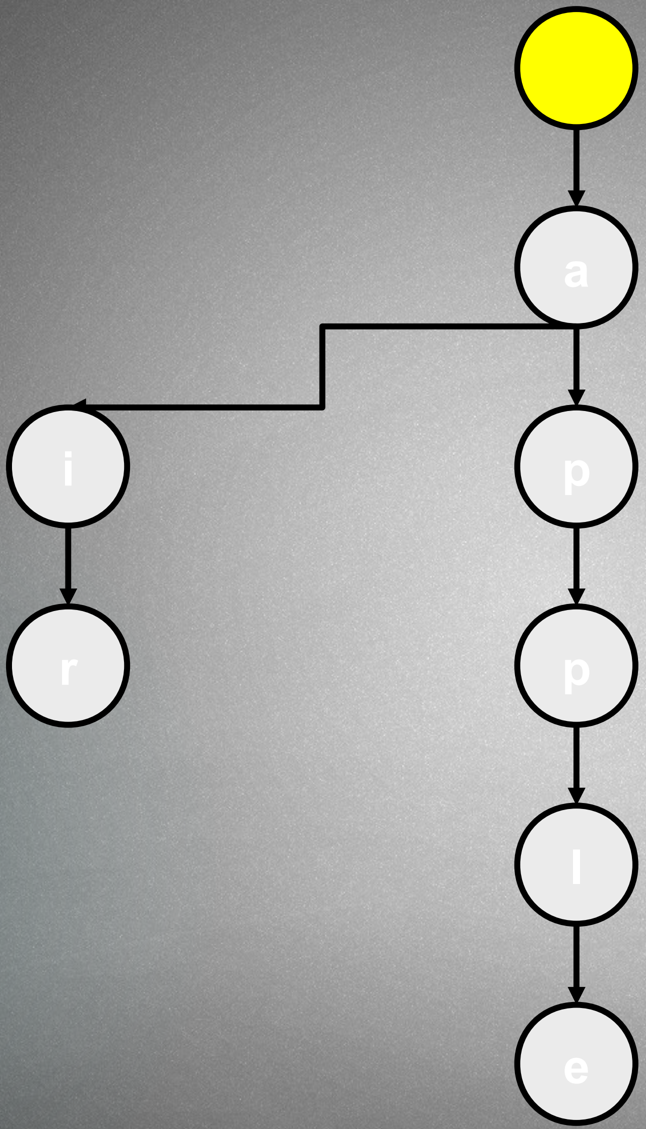


Insertion

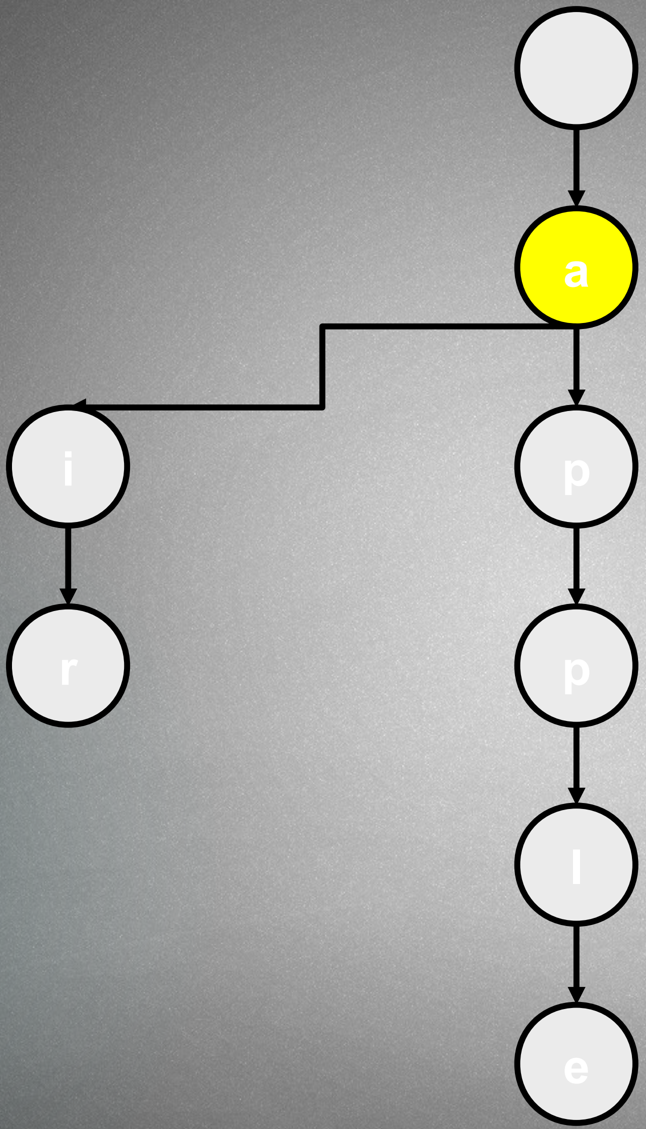
insert(„approve”)



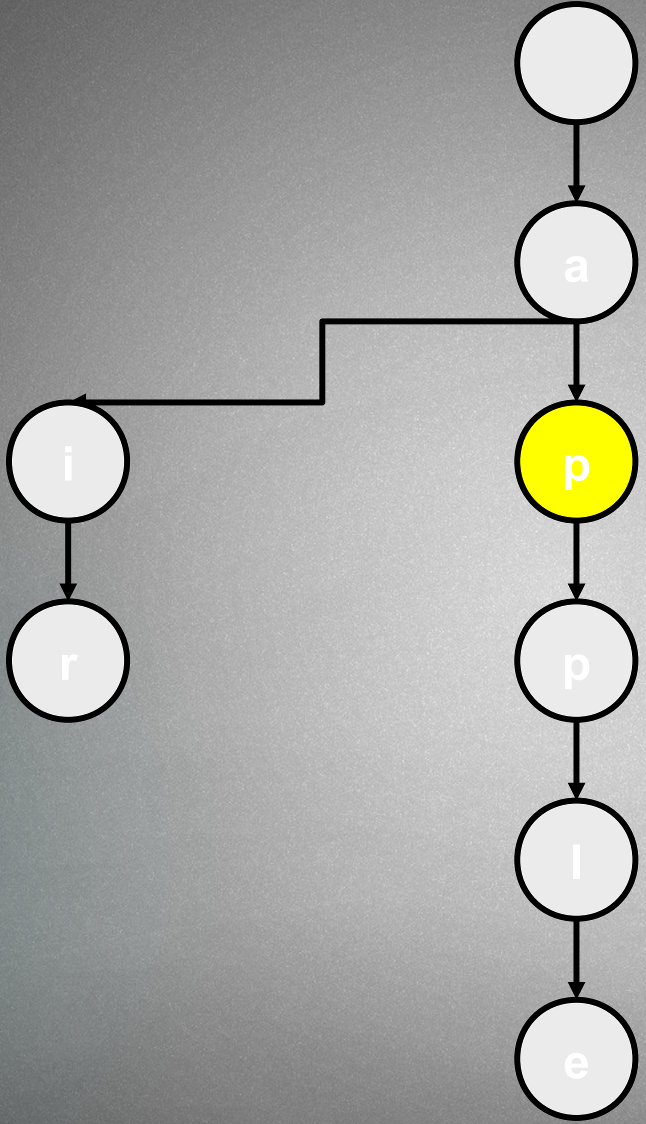
Insertion insert(„approve”)



Insertion insert(„approve”)

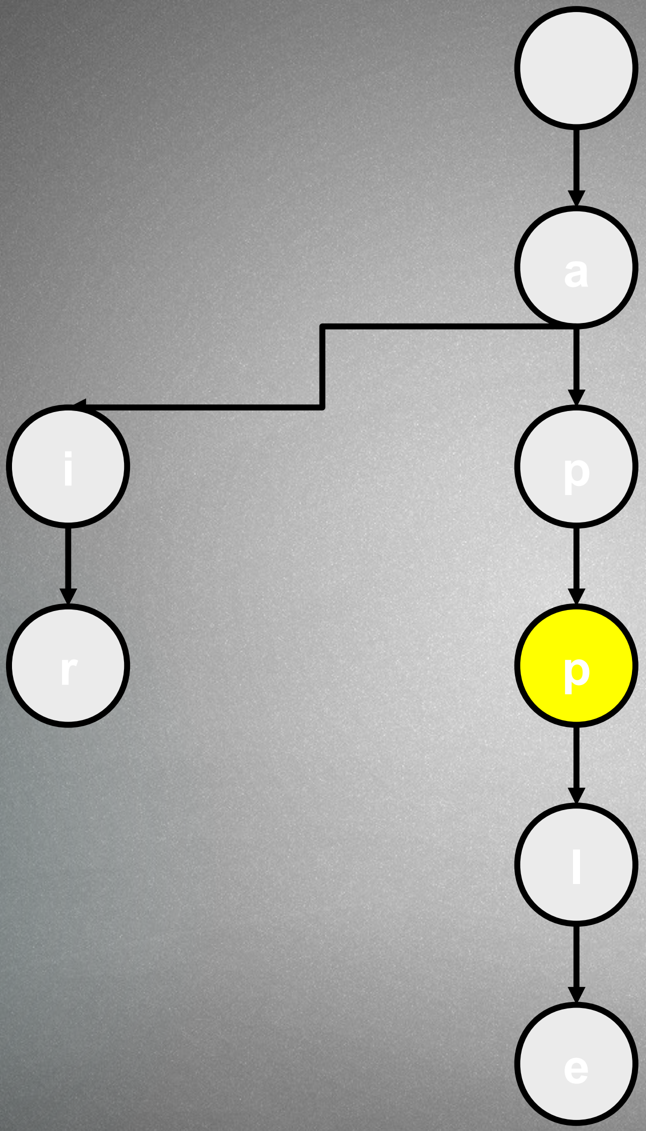


Insertion insert(„**ap**prove”)

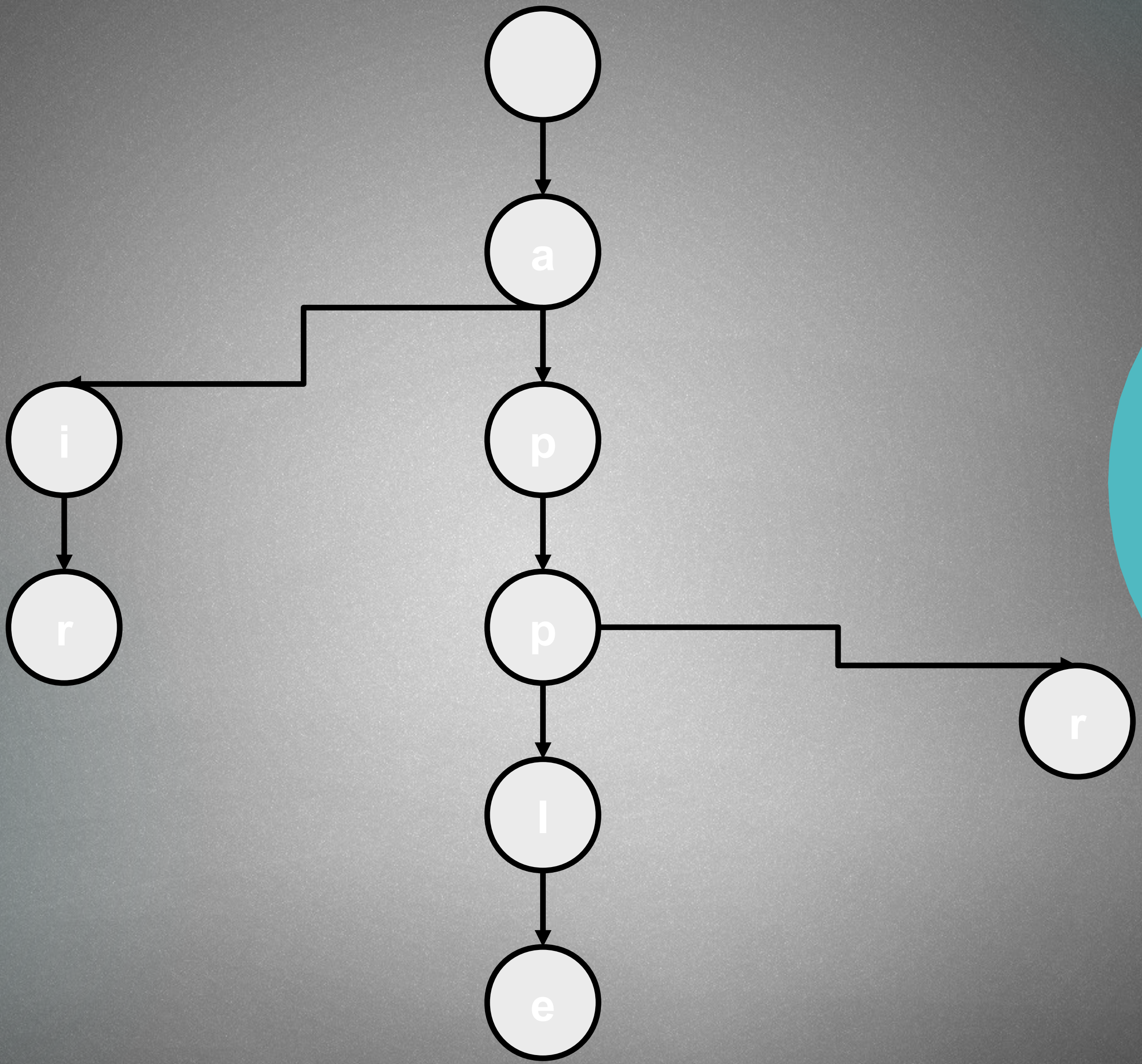


Insertion

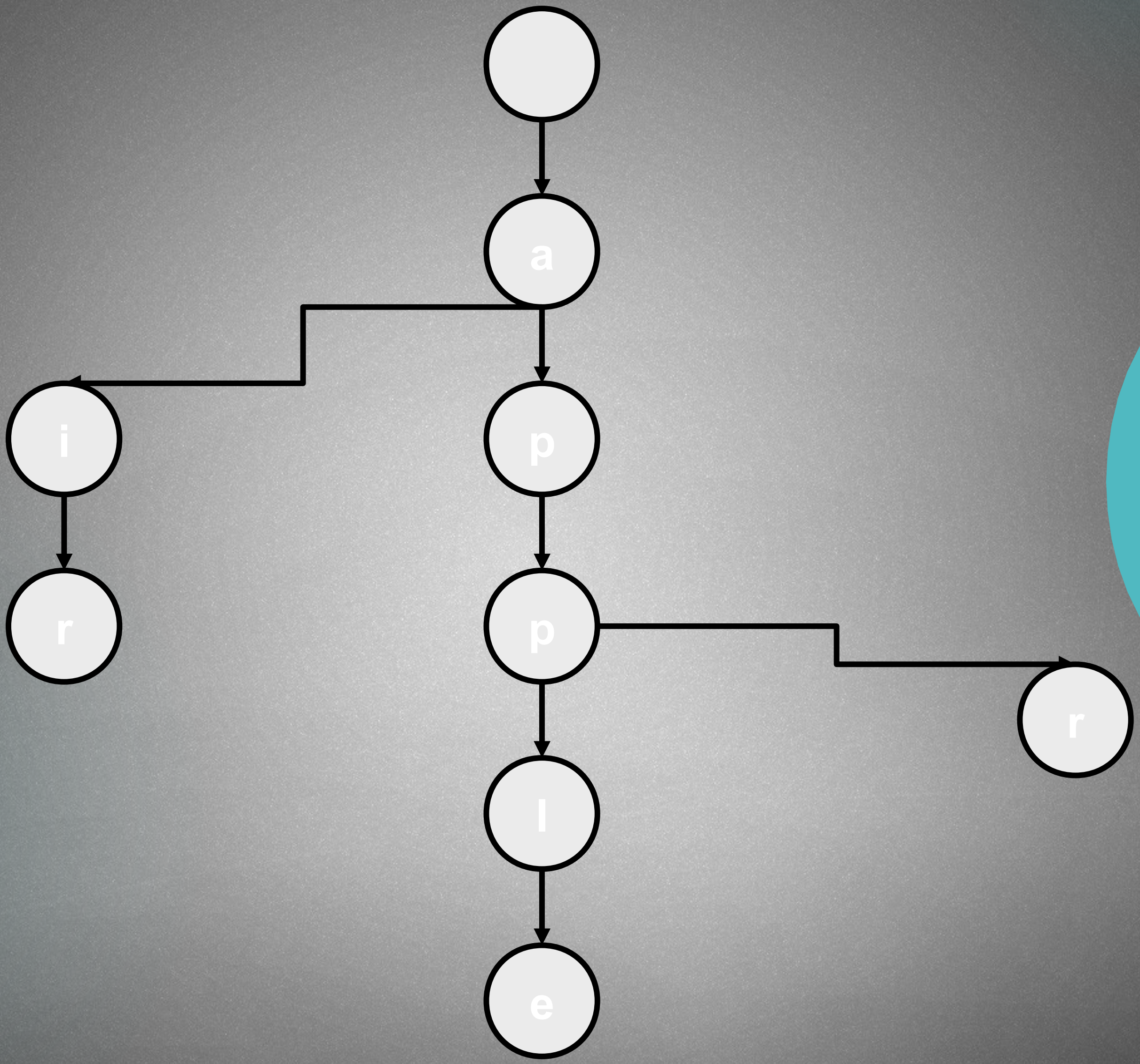
insert(„**app**rove”)



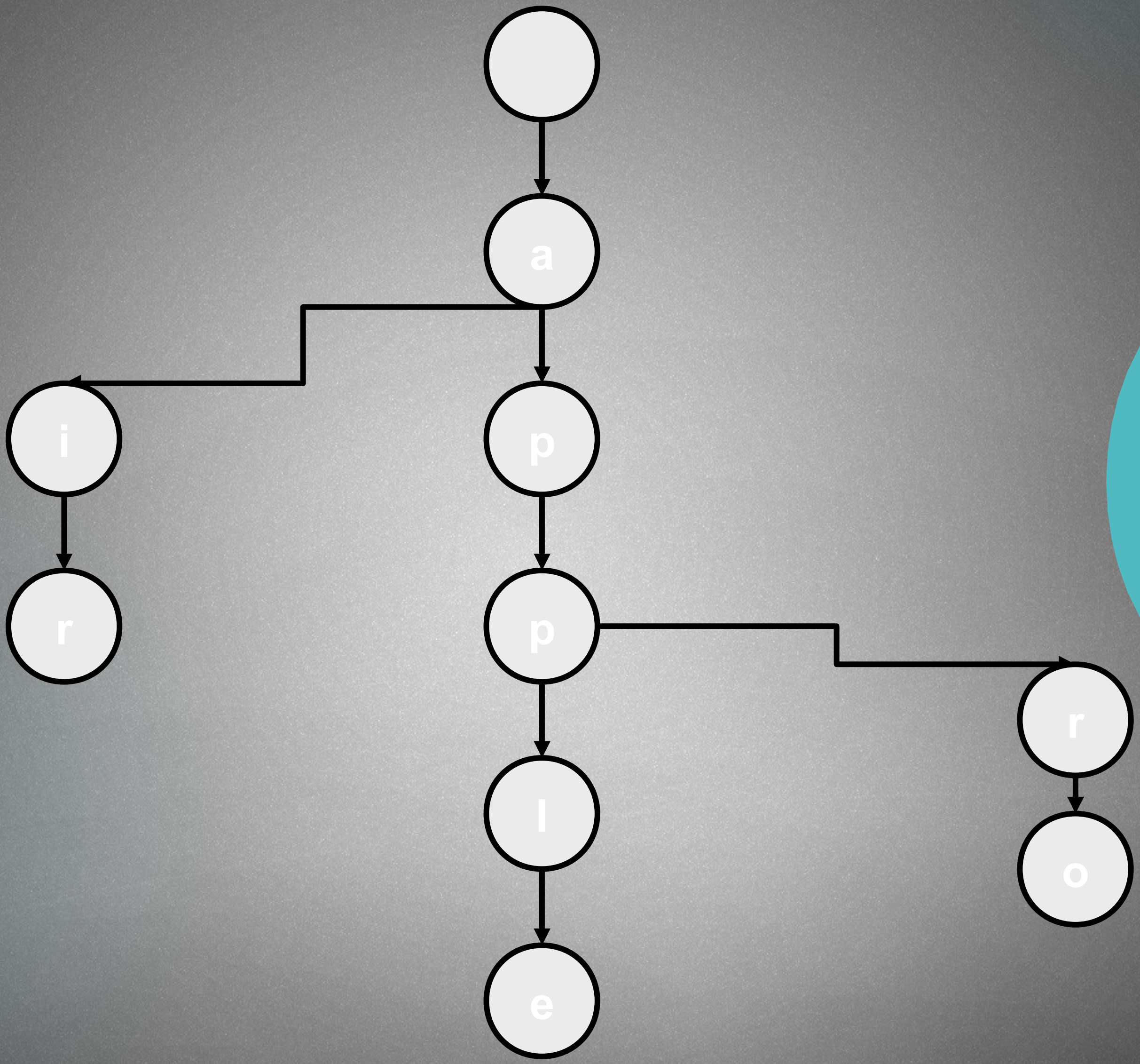
Insertion insert(„**appr**ove”)



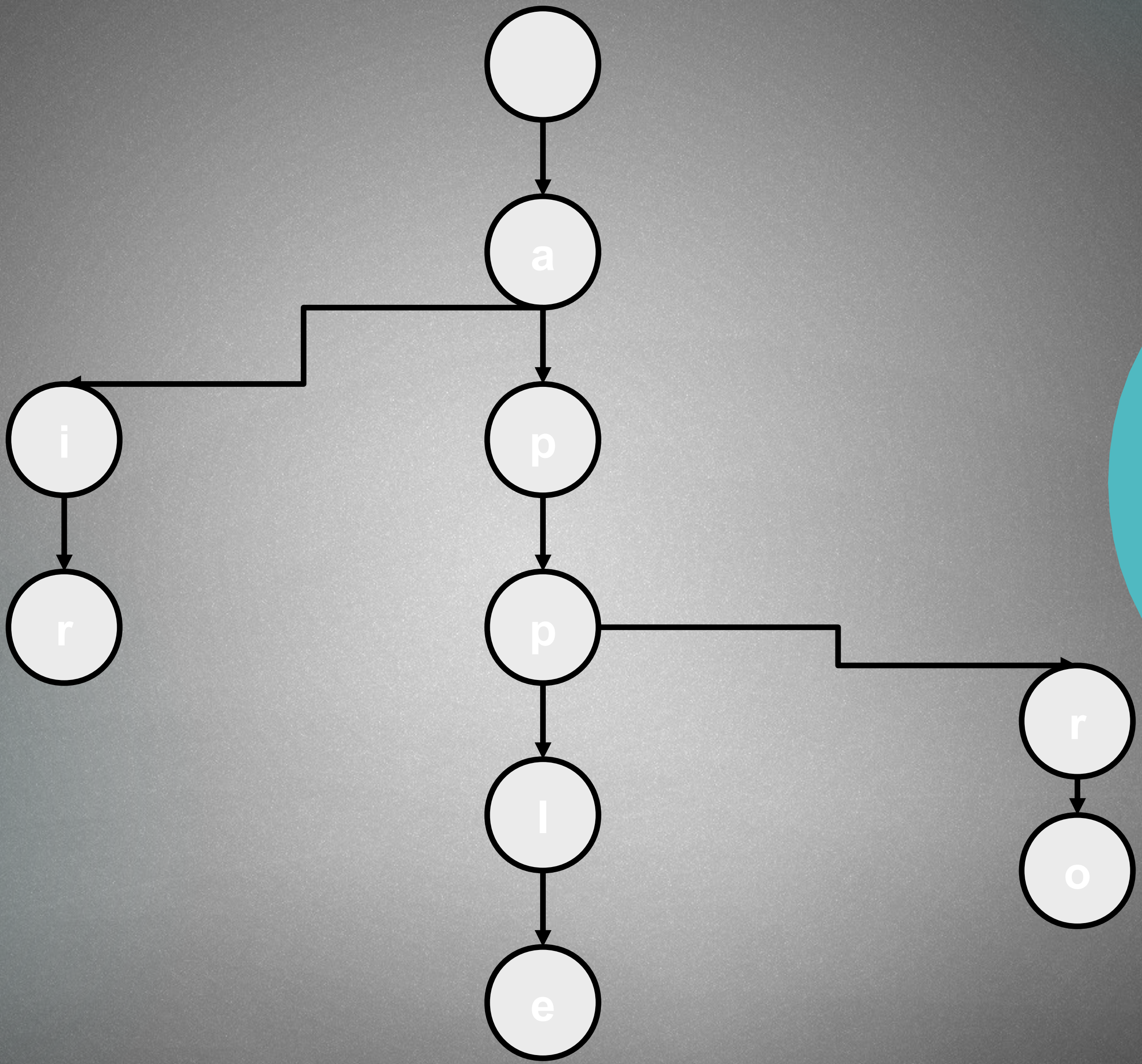
Insertion insert(„**appro**ve")



Insertion insert(„**appro**ve”)

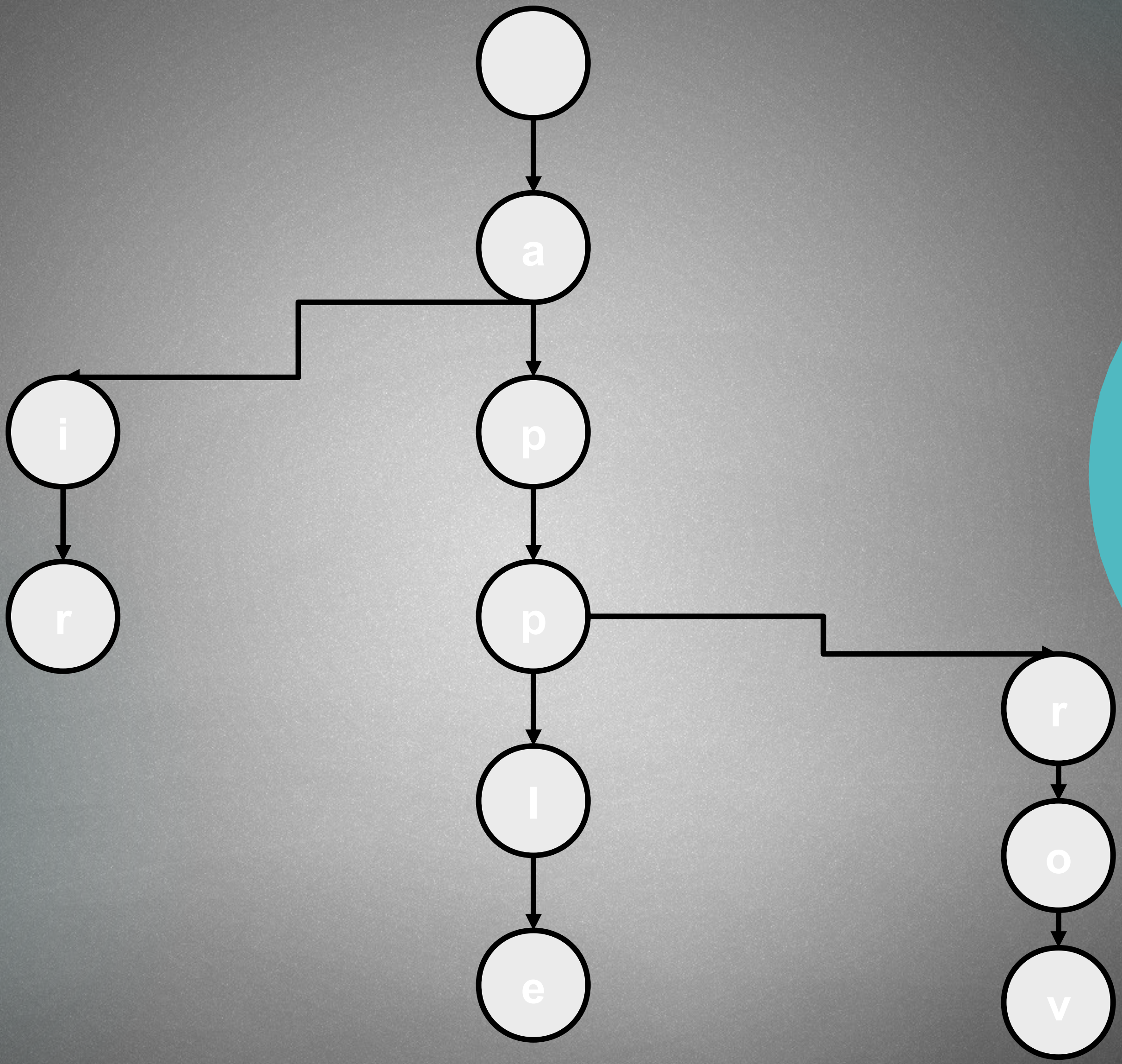


Insertion insert(„**approve**”)



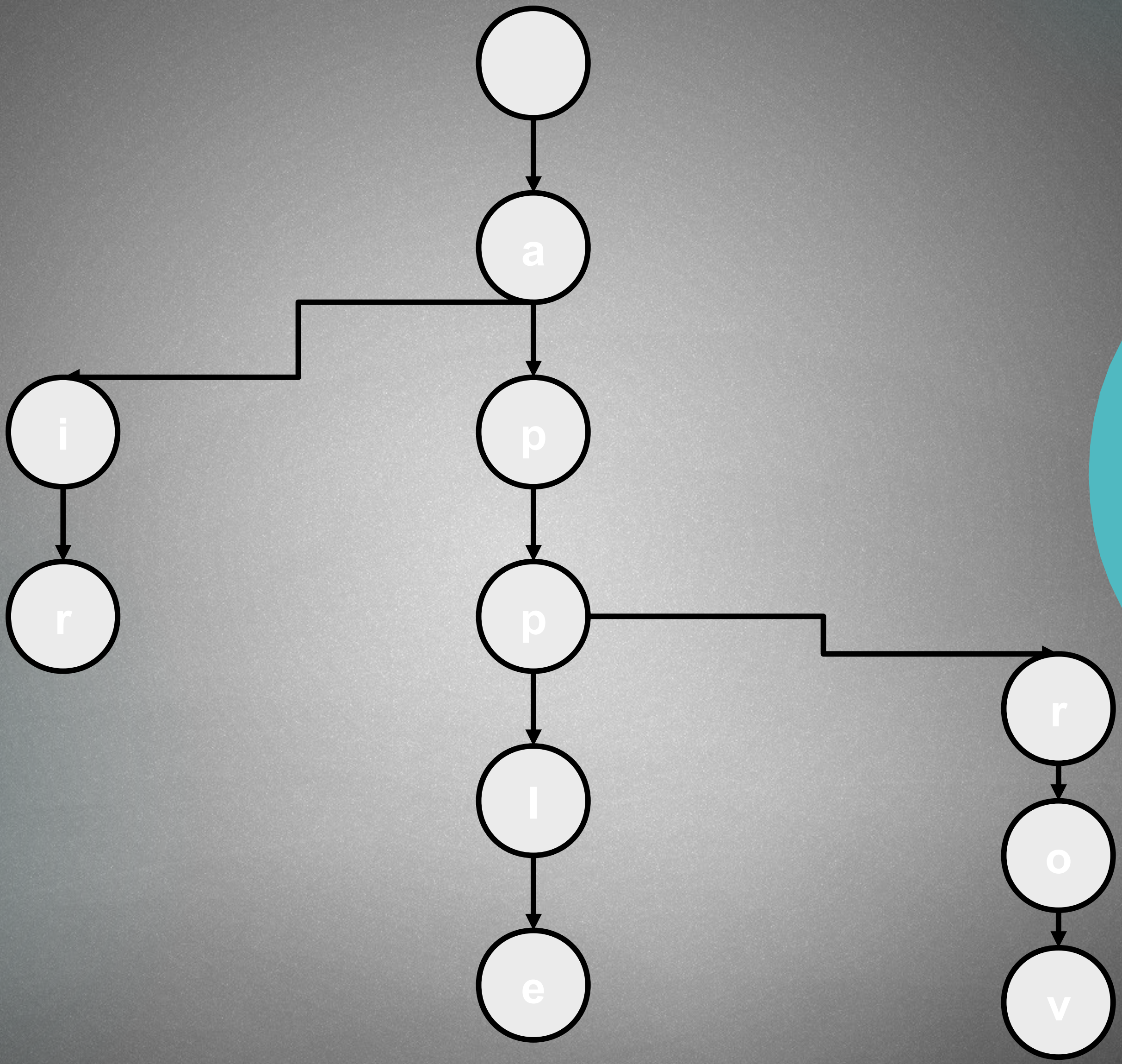
Insertion

insert(„**approve**”)



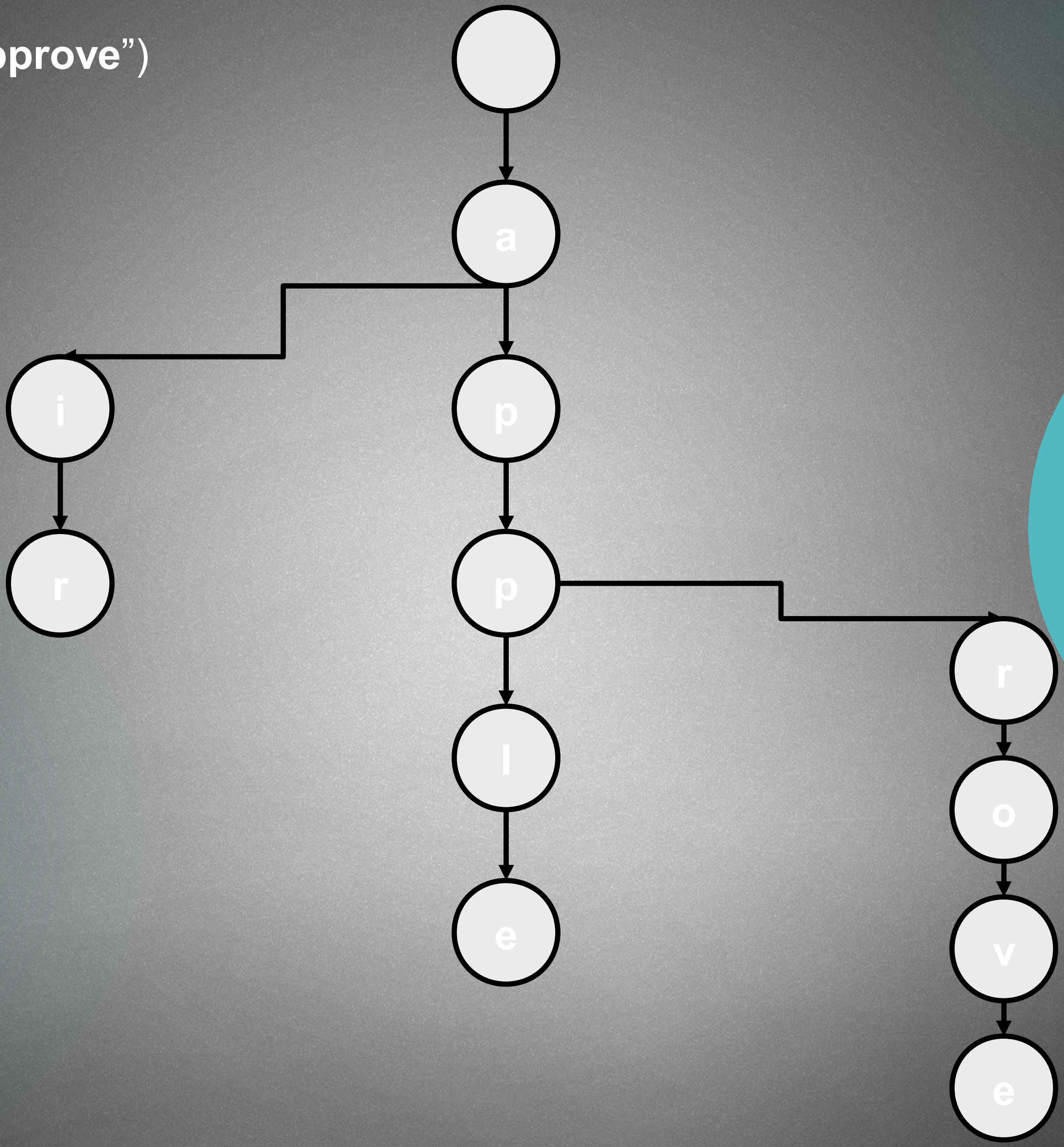
Insertion

insert(„**approve**”)

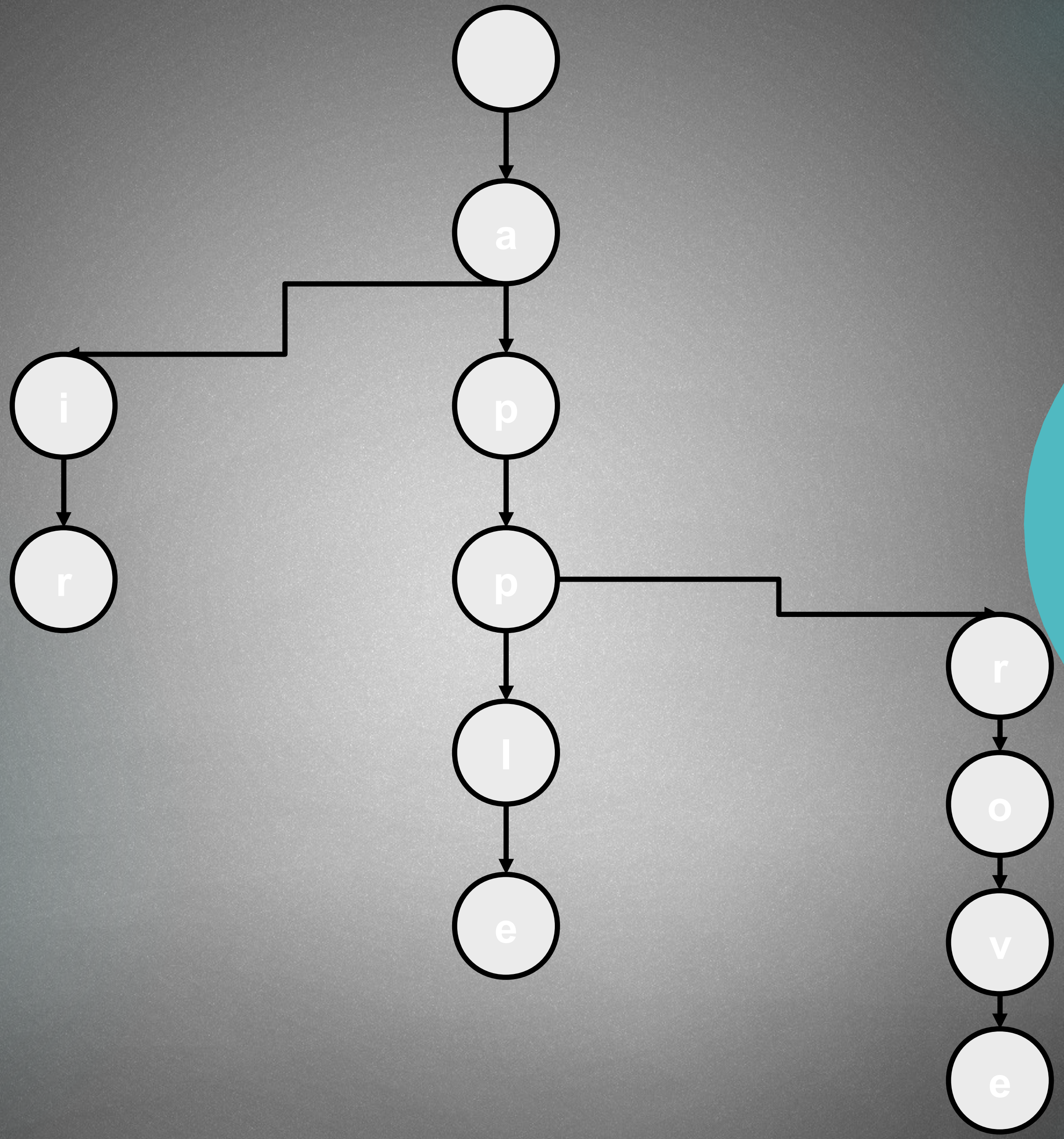


Insertion

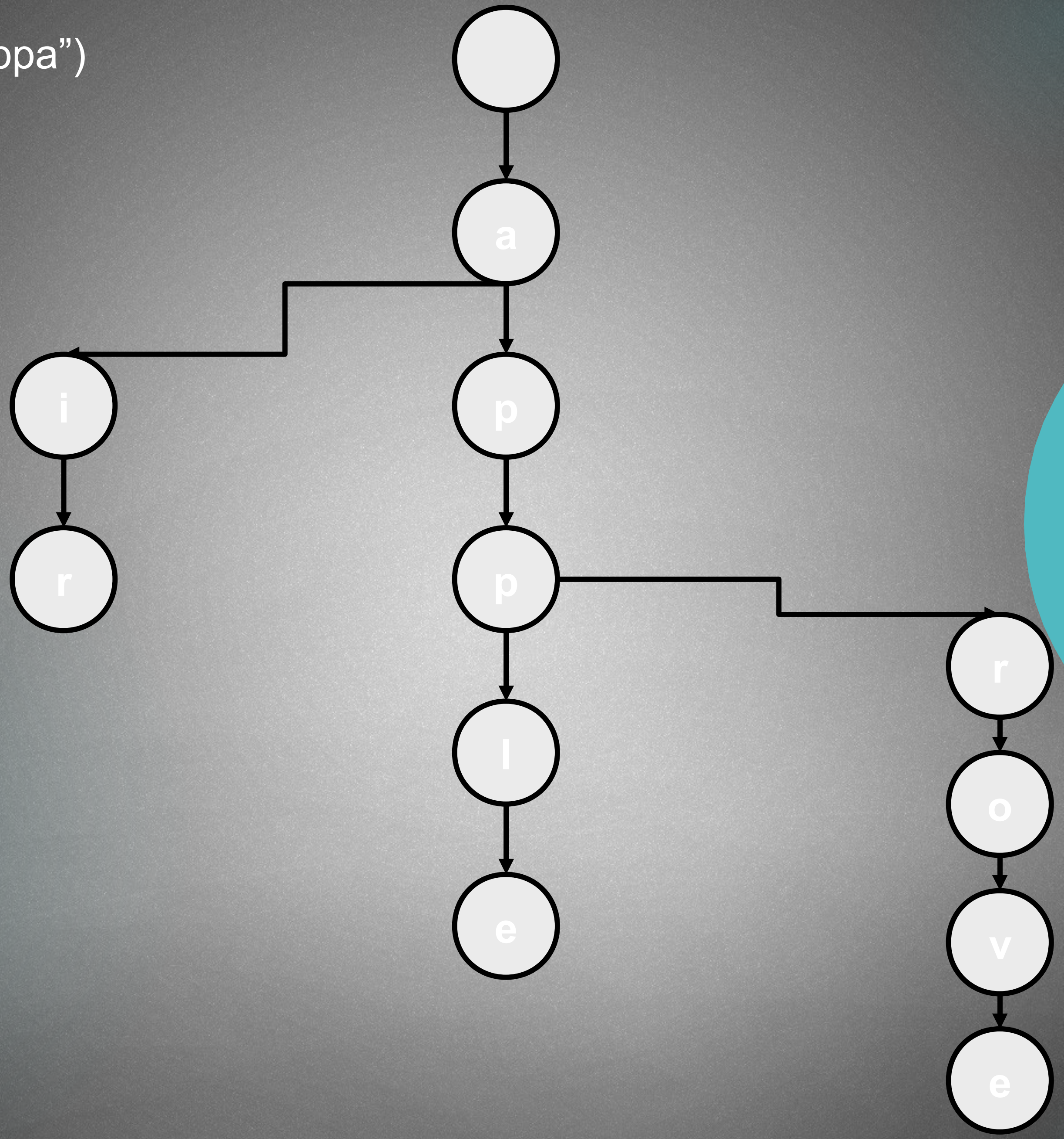
insert(„**approve**”)



Insertion

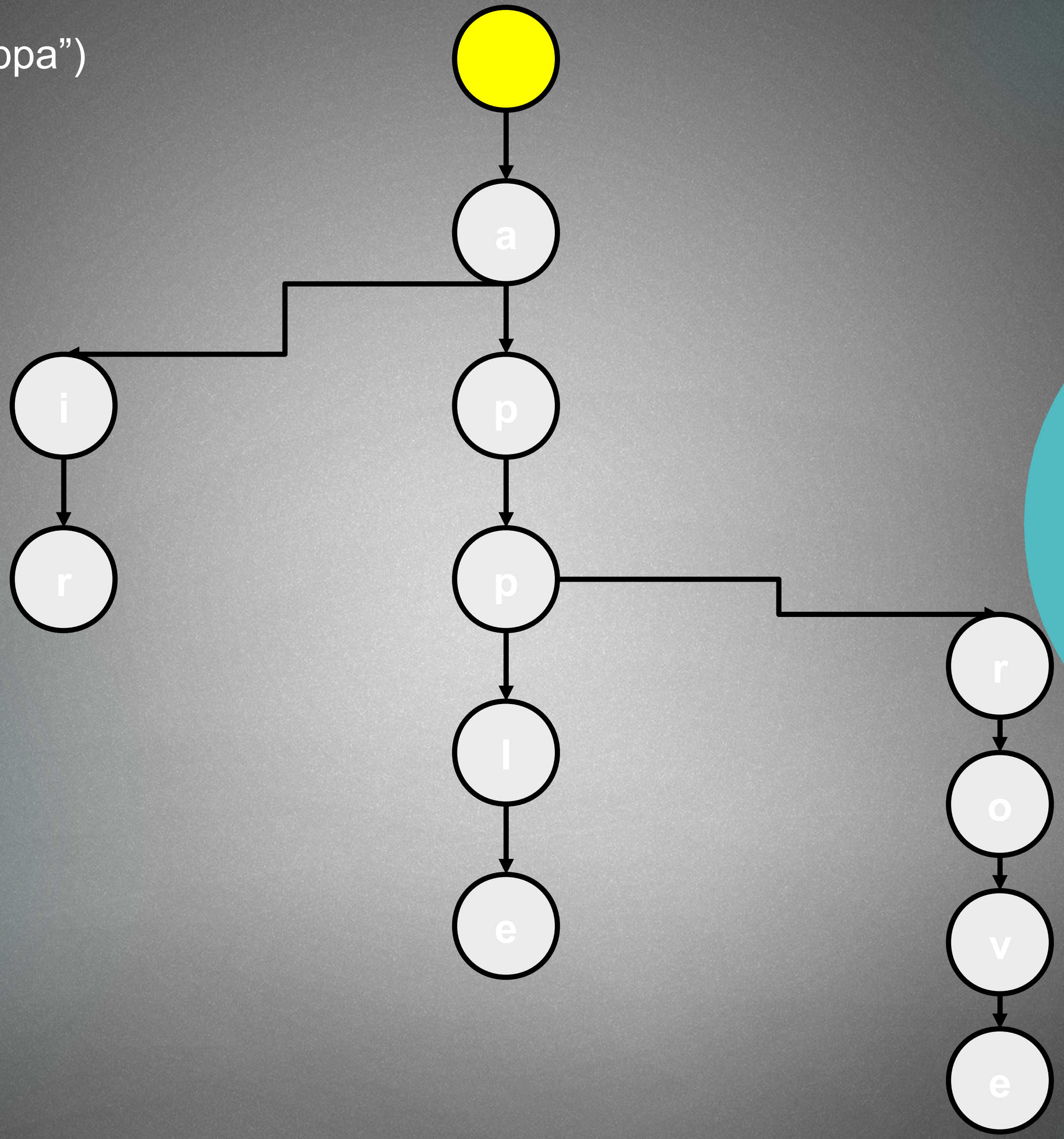


Insertion insert(„appa”)

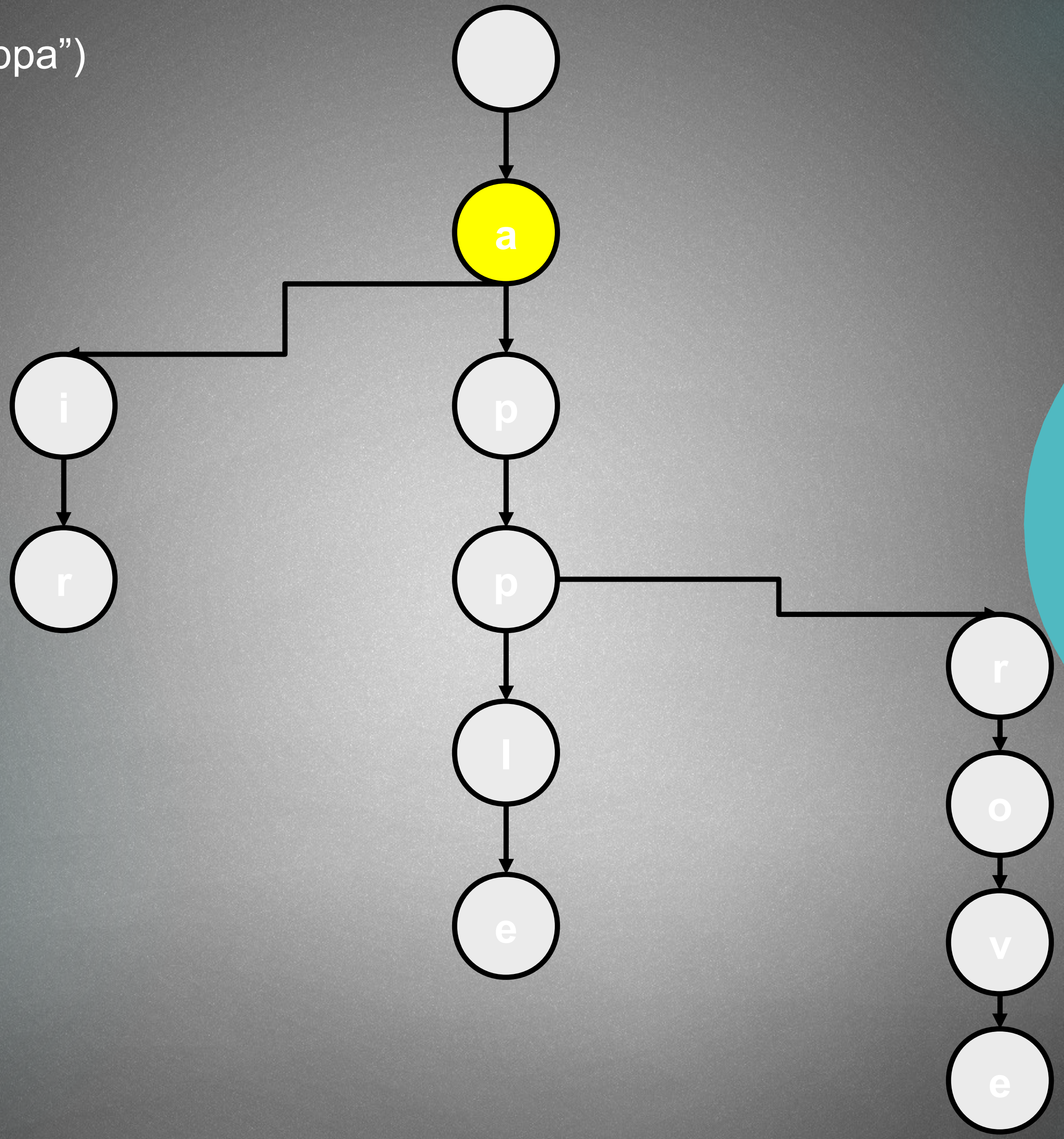


Insertion

insert(„appa”)

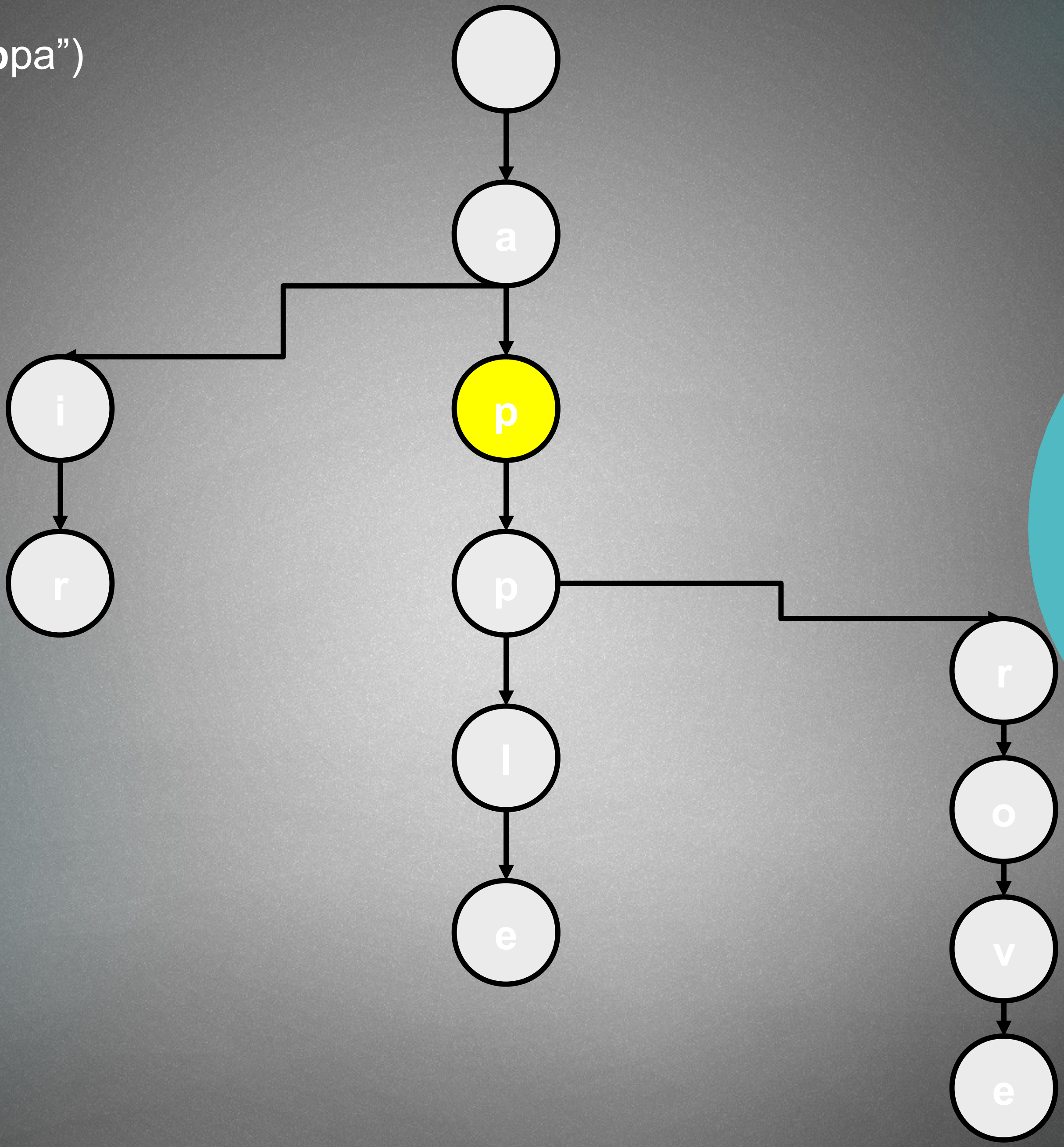


Insertion insert(„appa”)



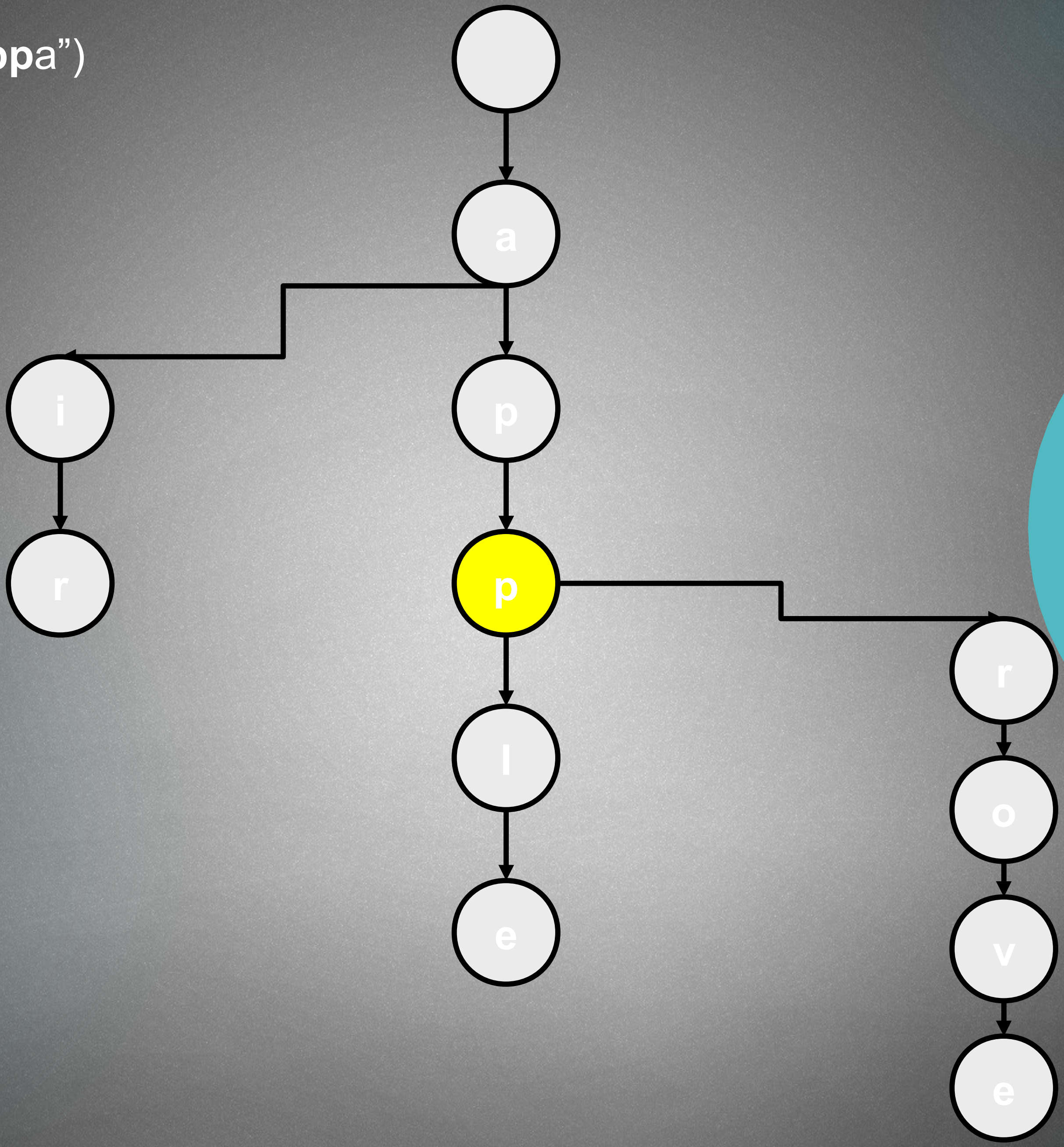
Insertion

insert(„**ap**pa”)



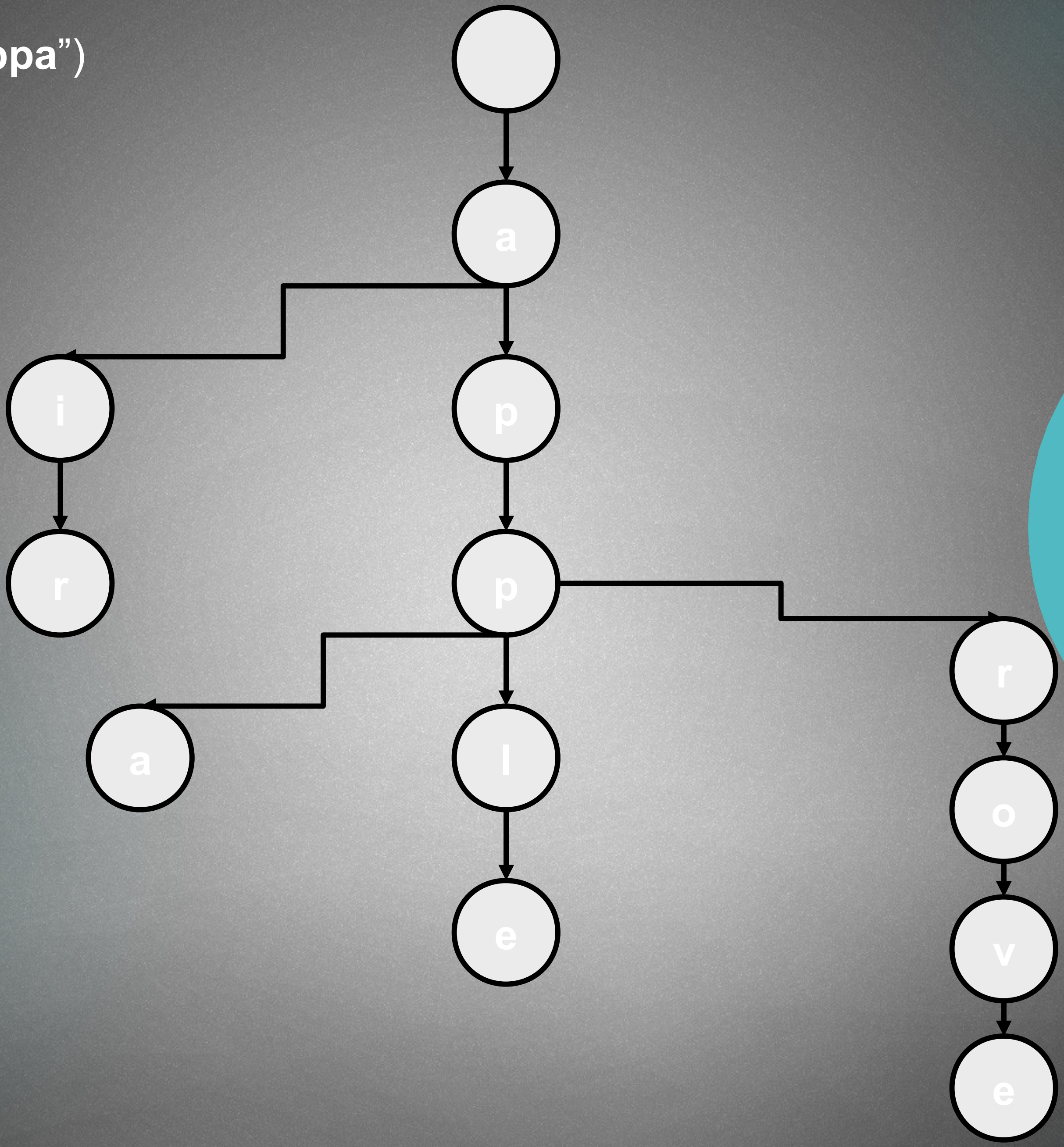
Insertion

insert(„**appa**”)

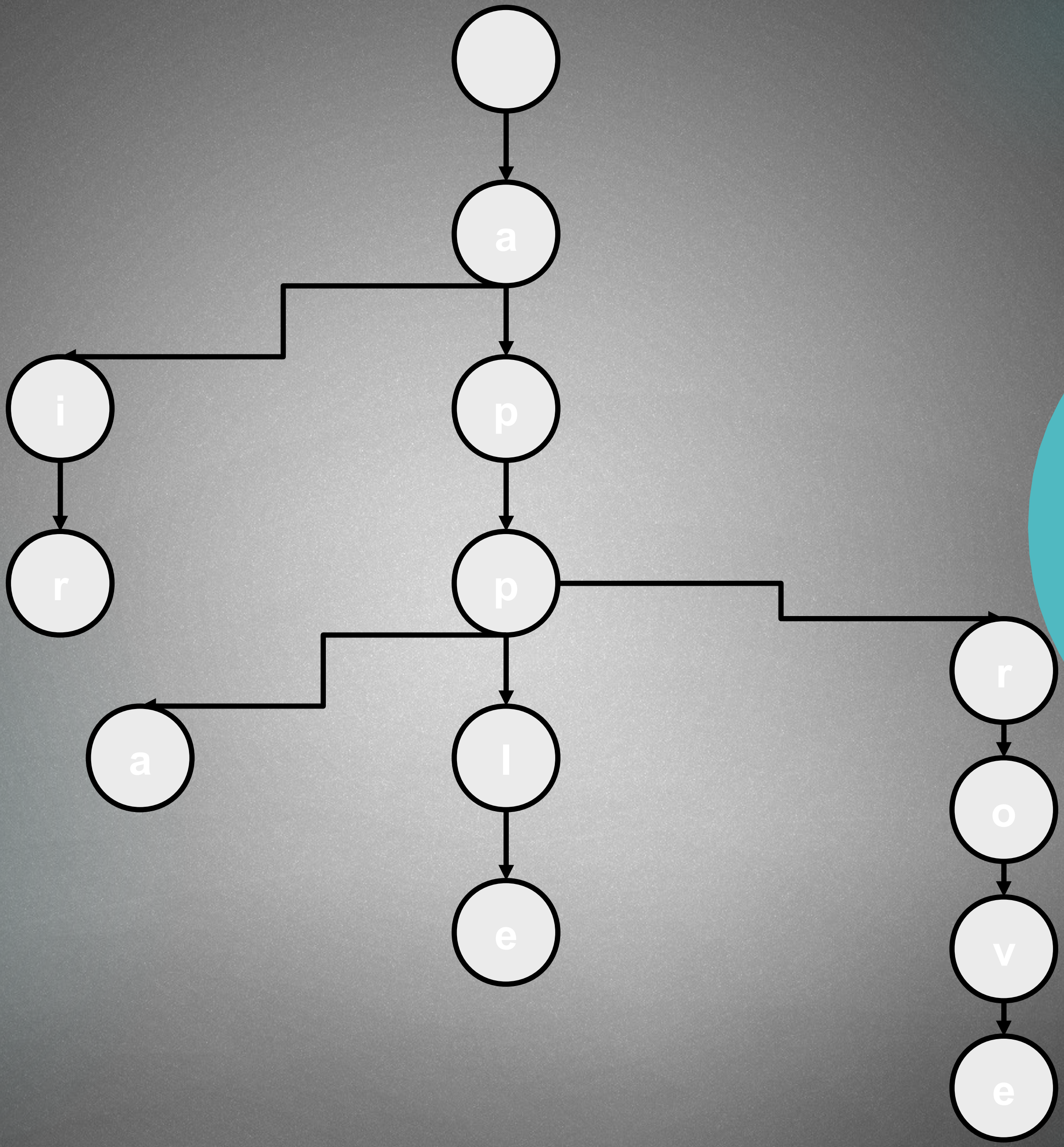


Insertion

insert(„**appa**")

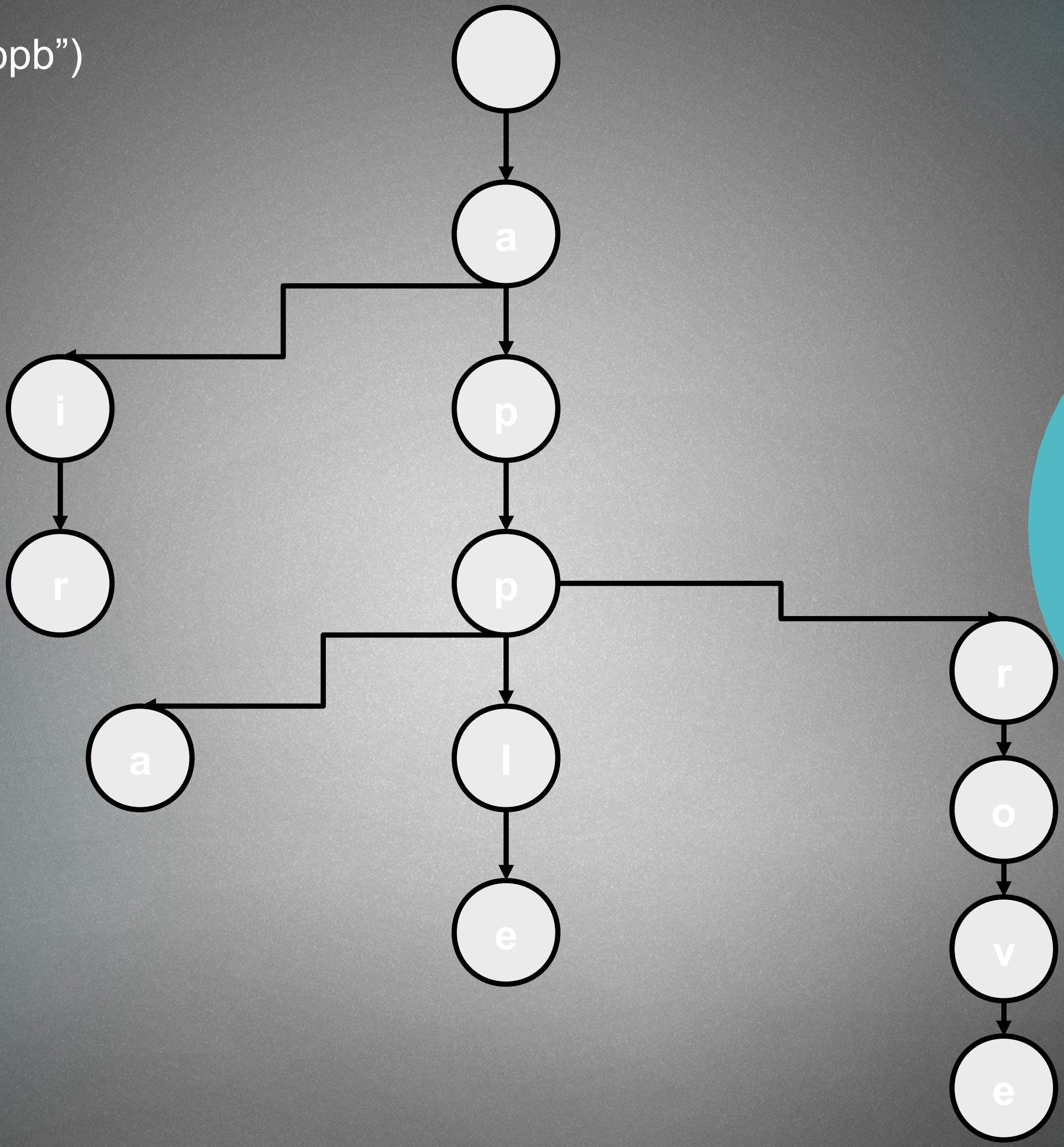


Insertion



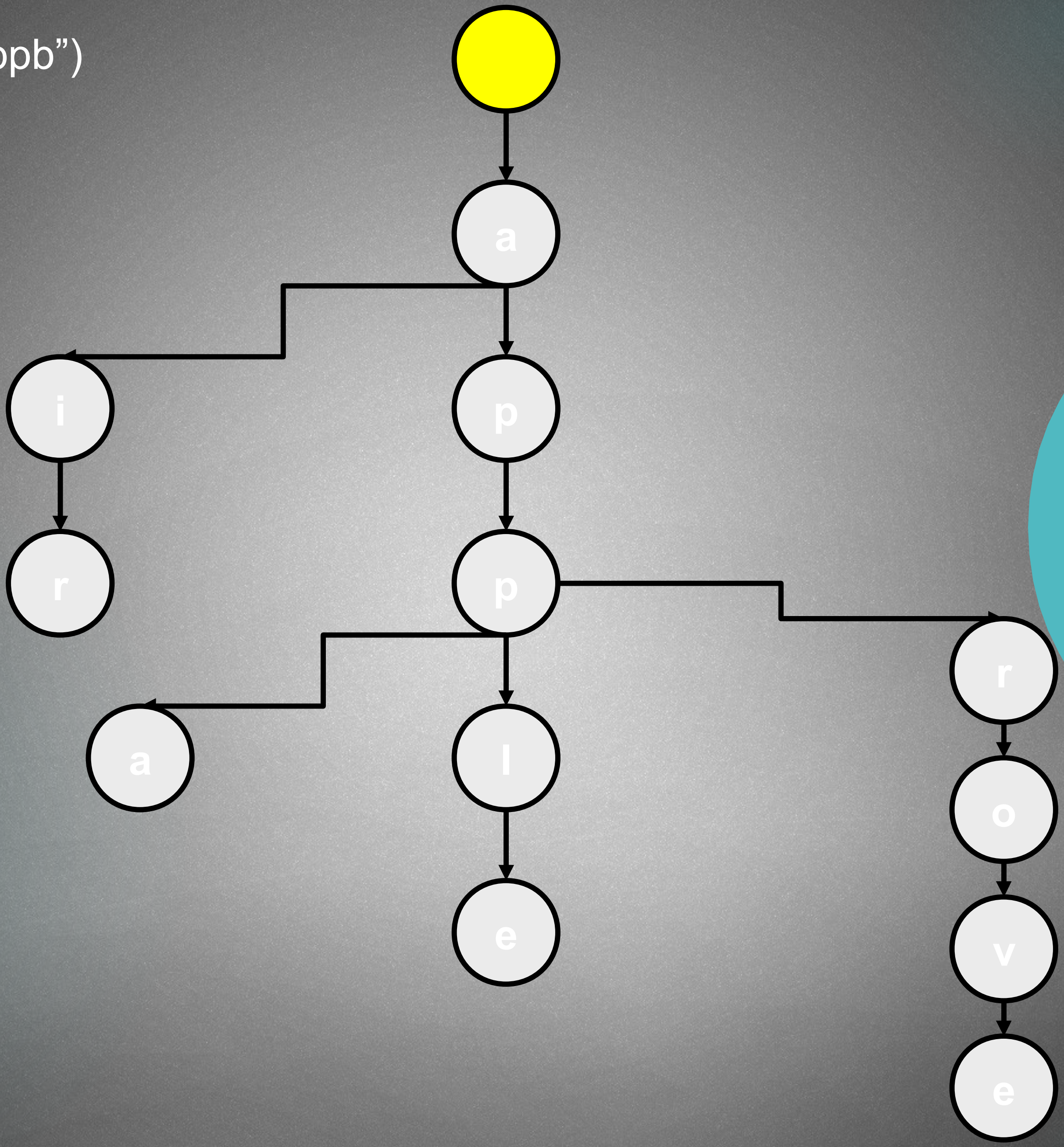
Insertion

insert(„appb”)



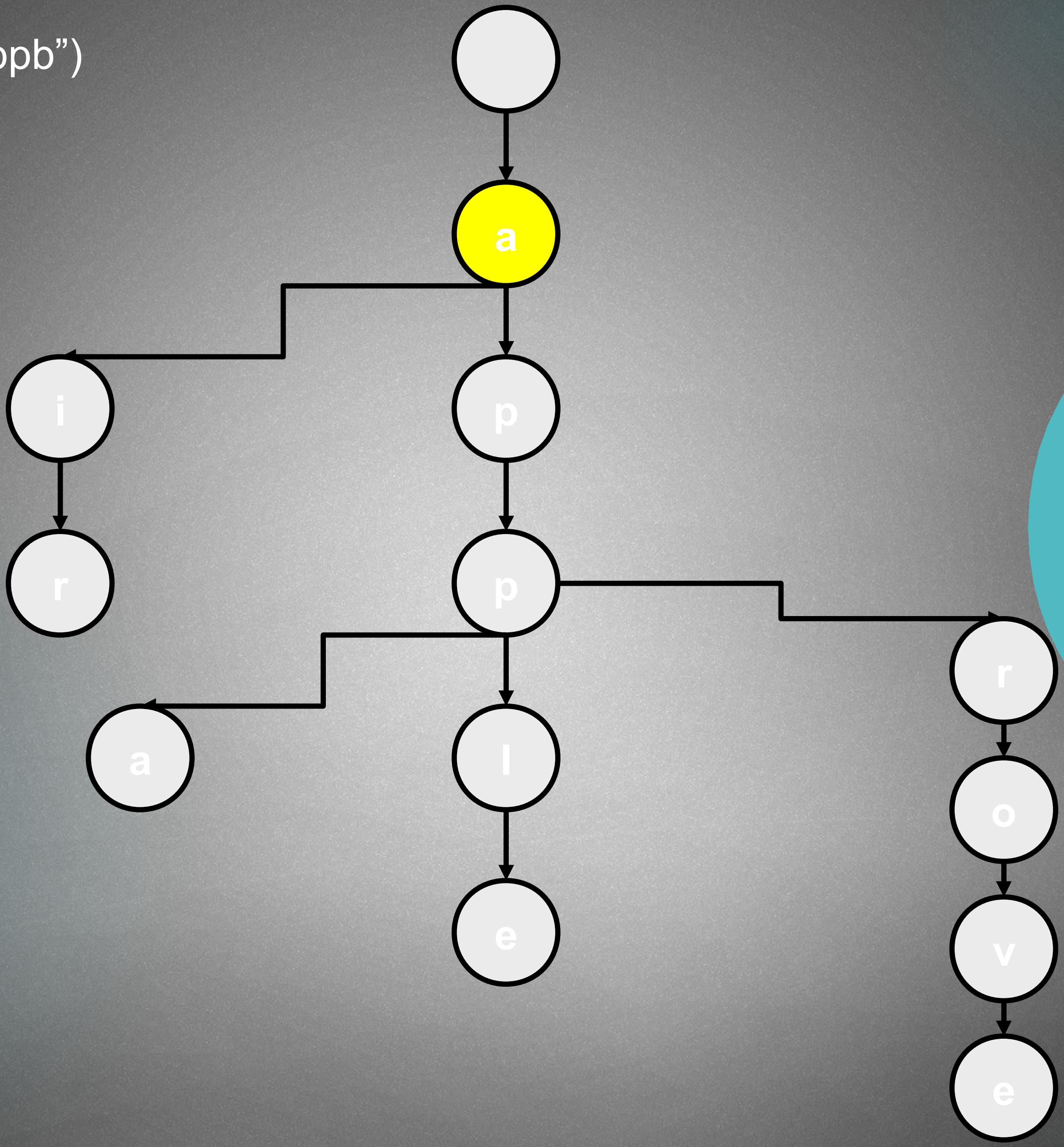
Insertion

insert(„appb”)



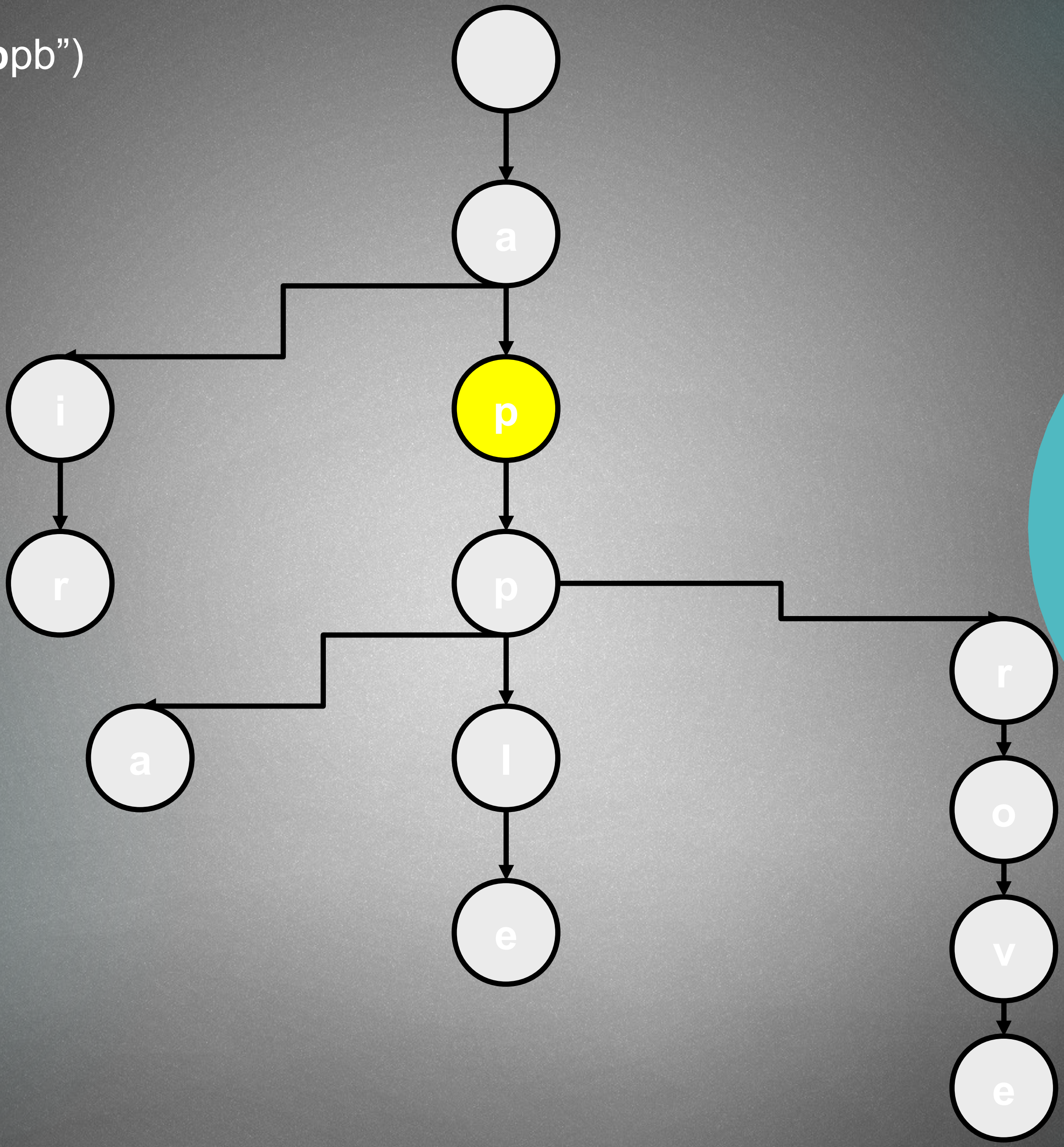
Insertion

insert(„appb”)



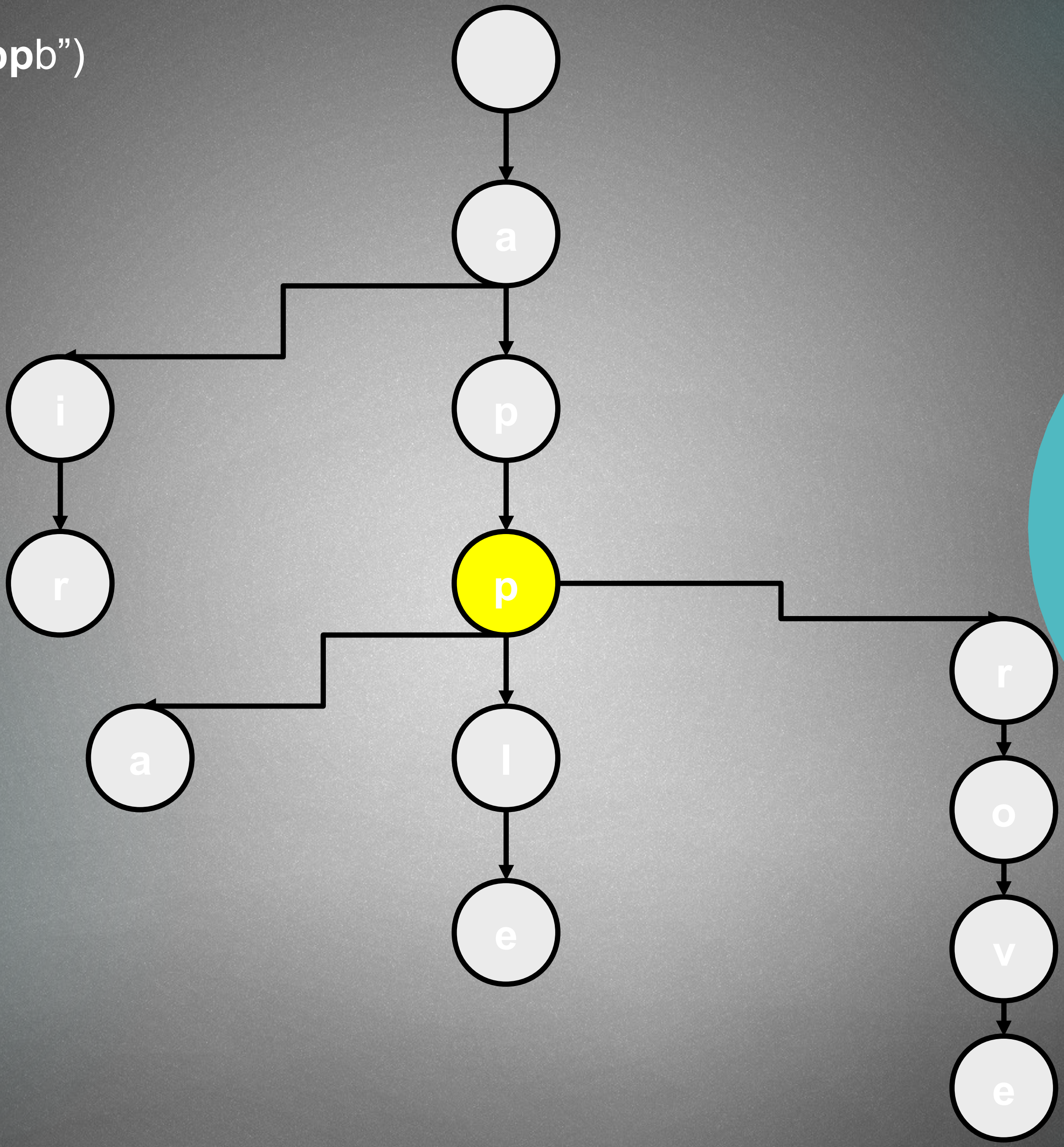
Insertion

insert(„**ap**pb”)

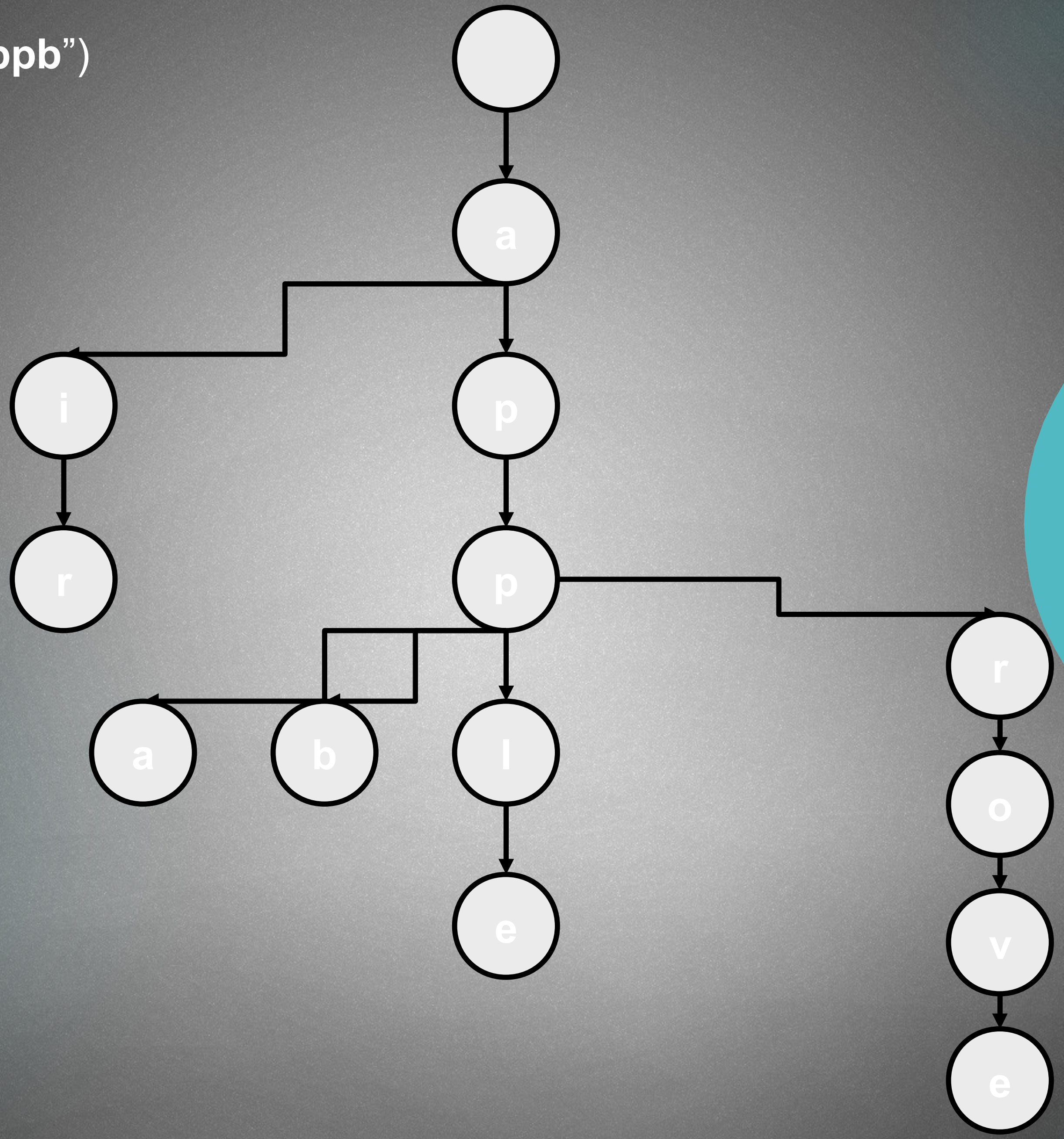


Insertion

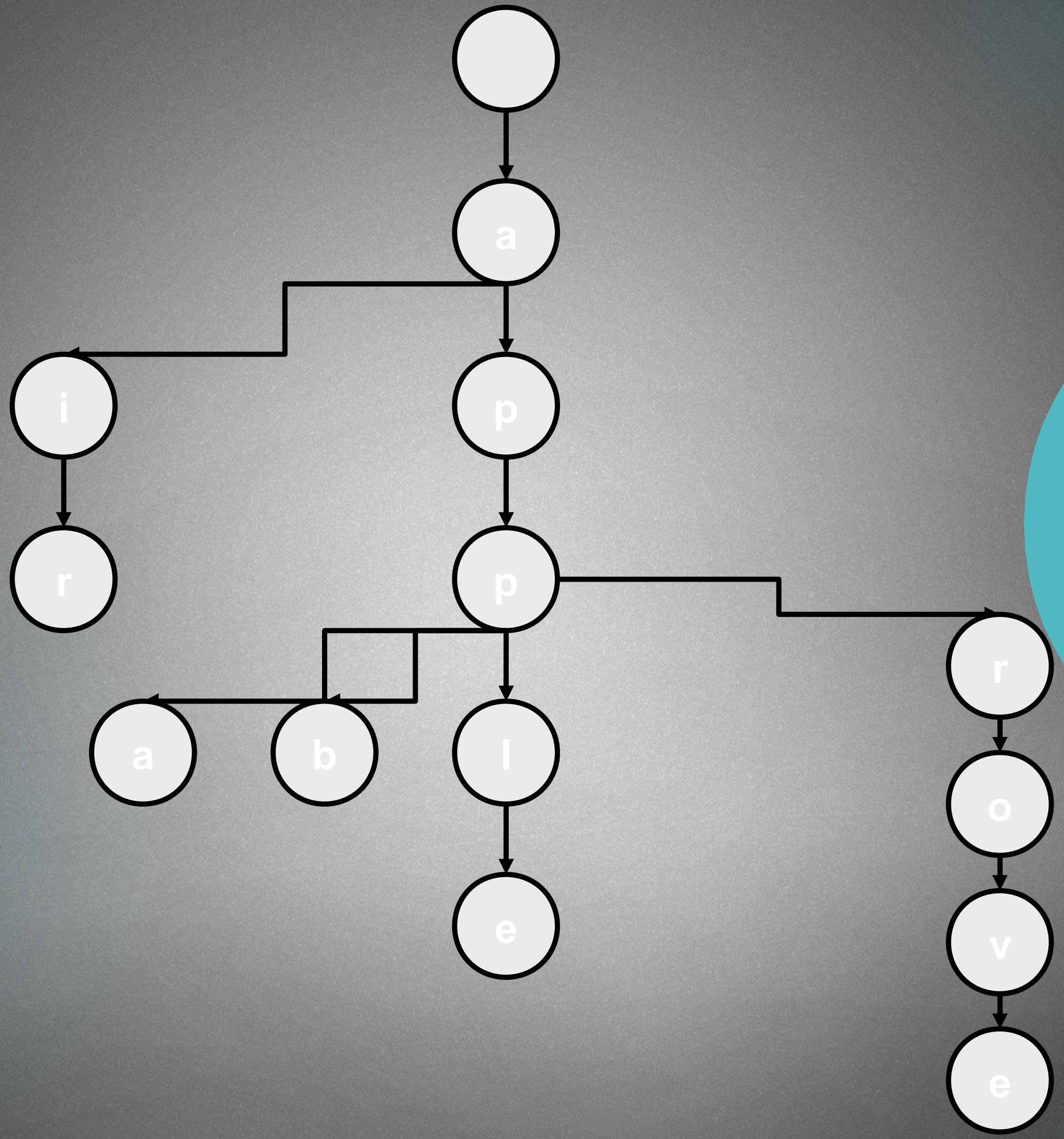
insert(„**app**b”)



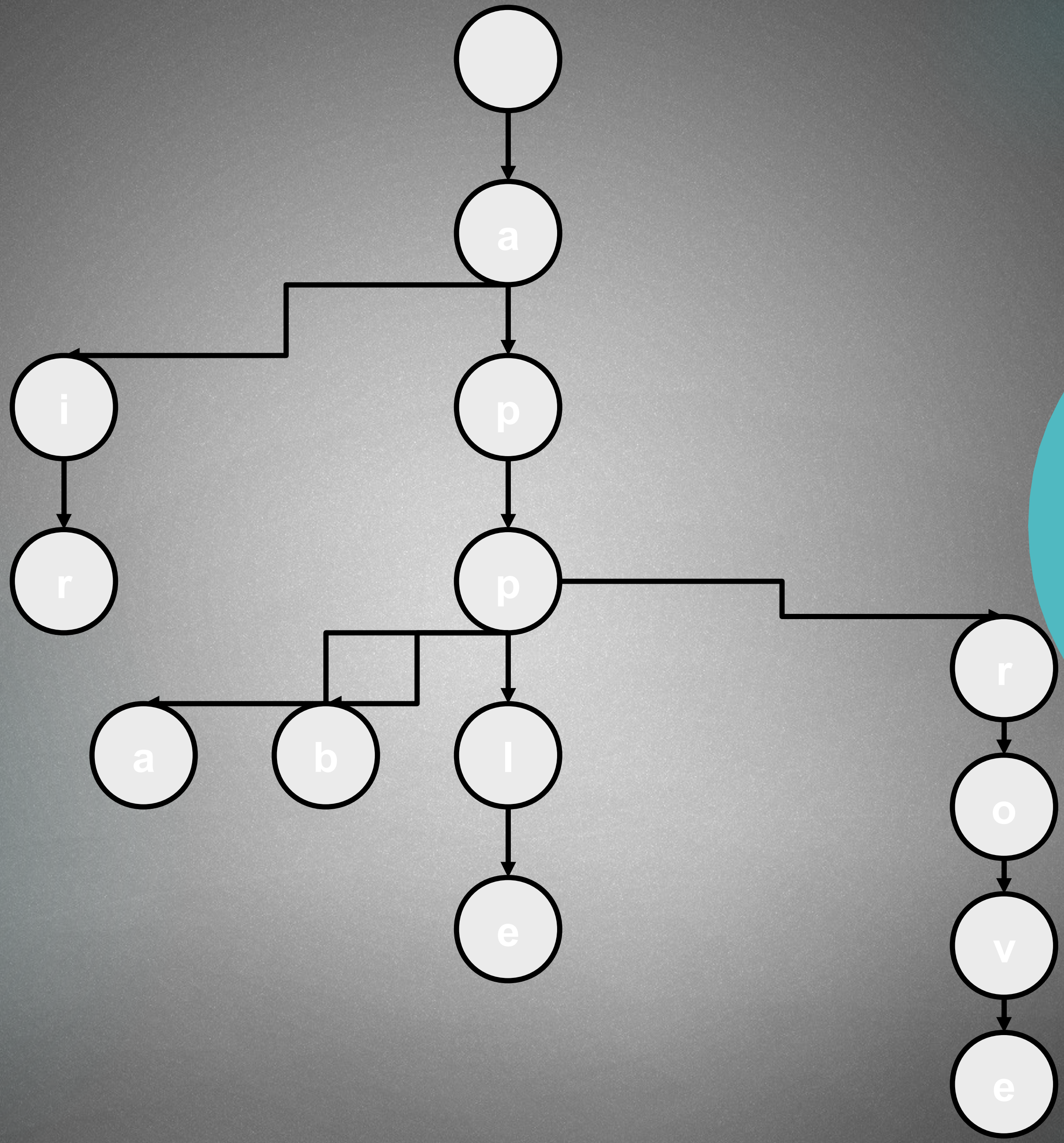
Insertion insert(„**appb**”)

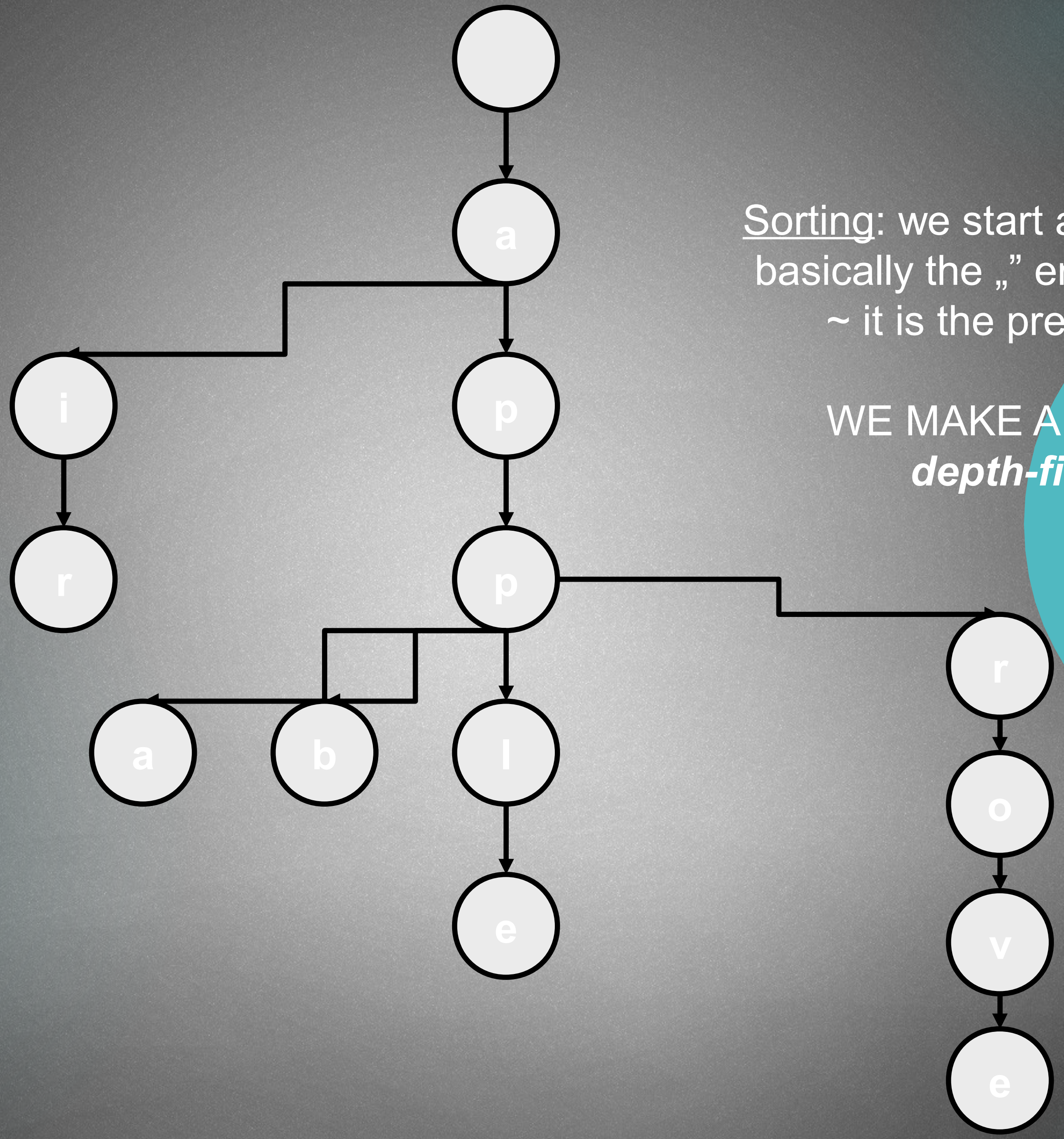


Insertion



Sorting


a
i
p
r
p
a
b
l
r
e
o
v
e



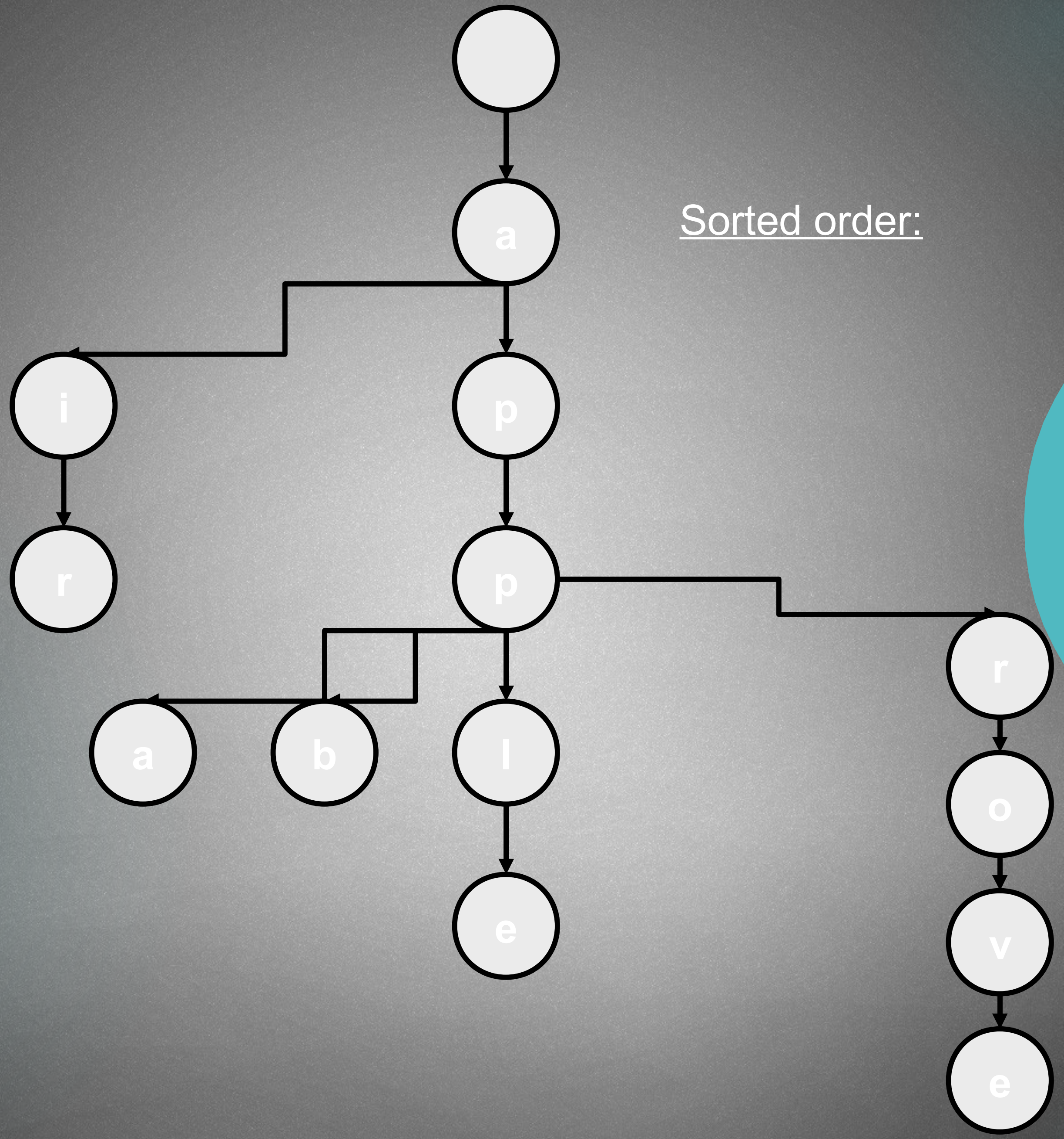
Sorting: we start at the root, which is basically the „” empty string

~ it is the prefix of every item in the trie

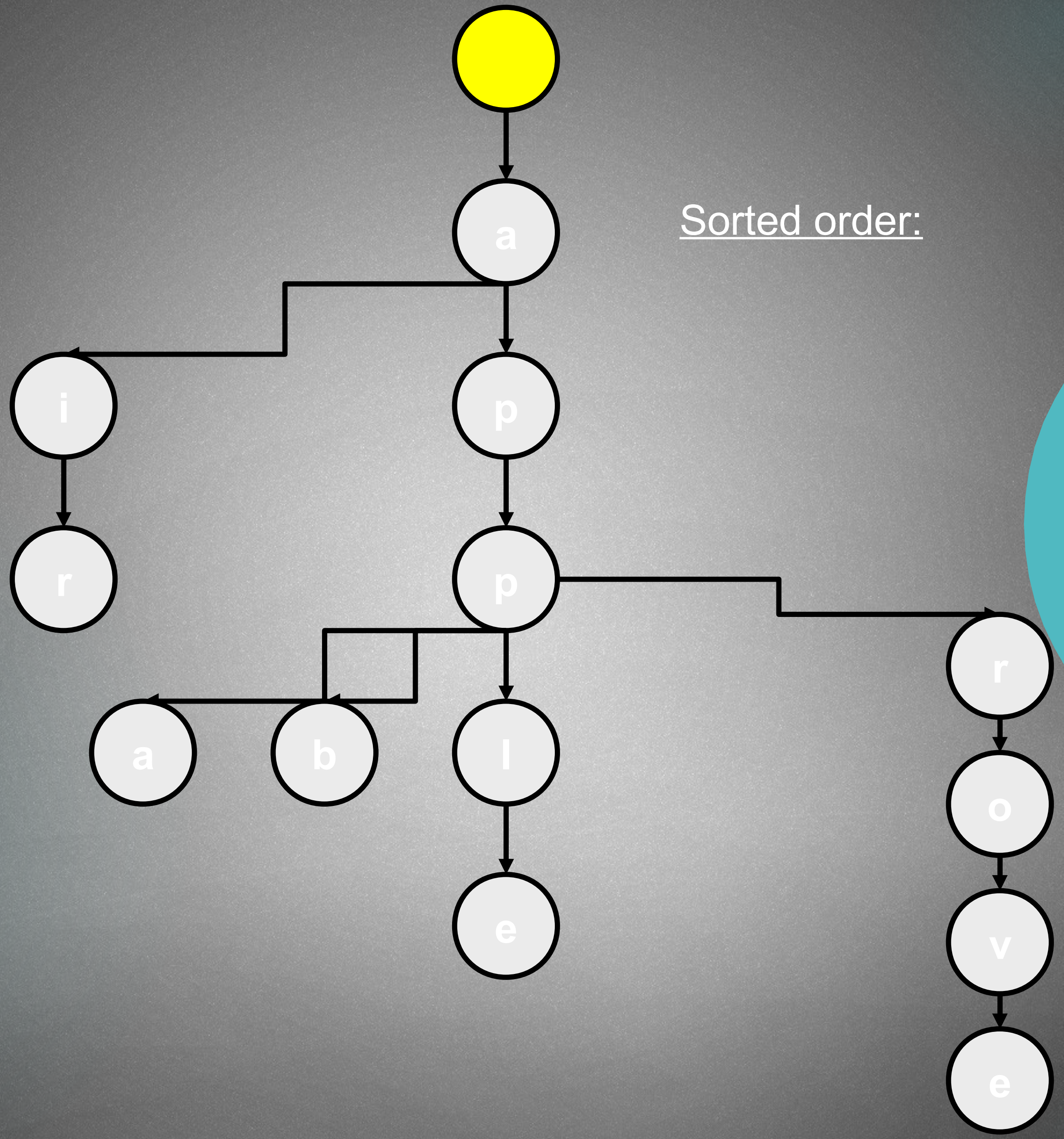
WE MAKE A SIMPLE DFS !!!

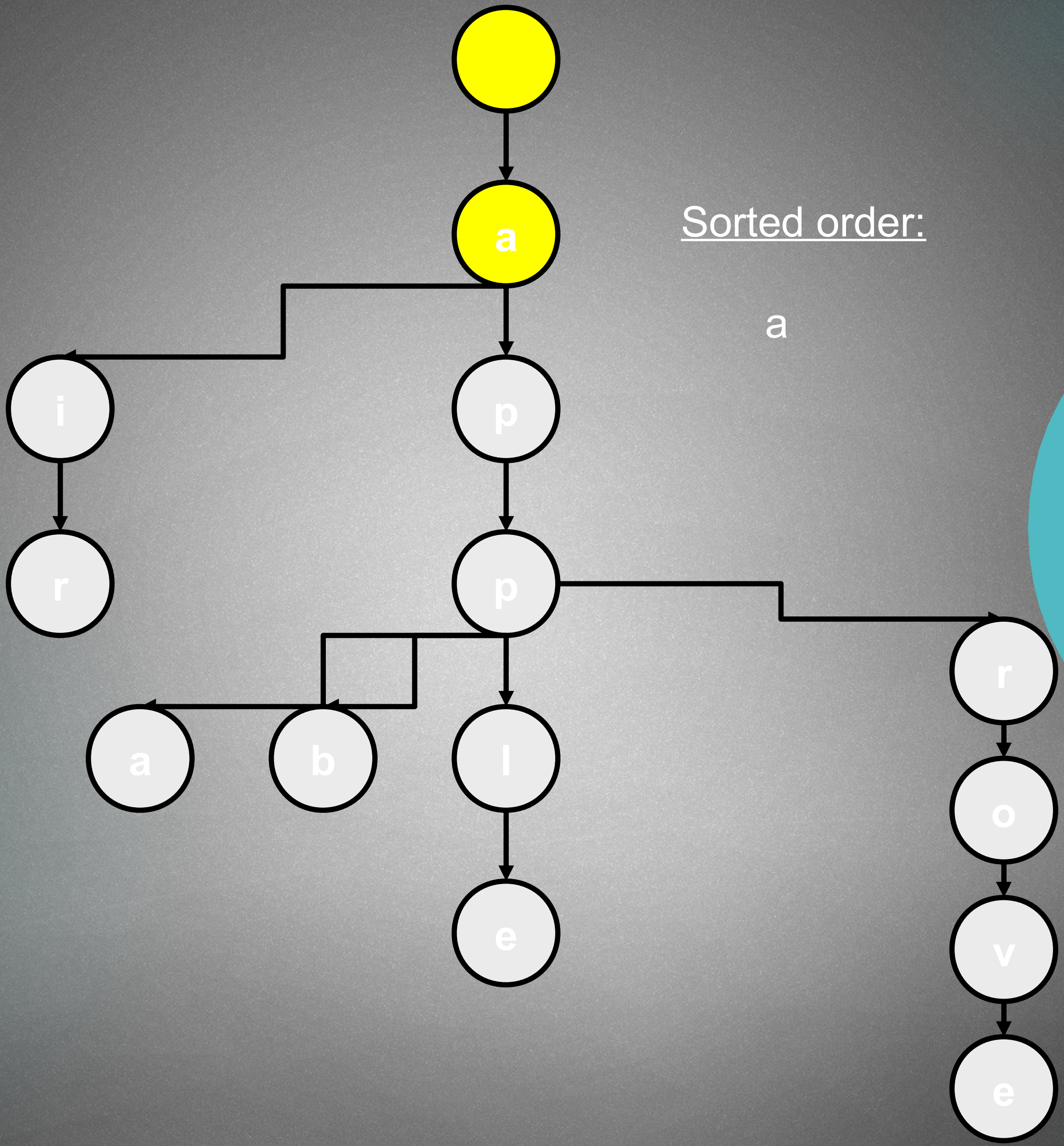
depth-first search

Sorted order:



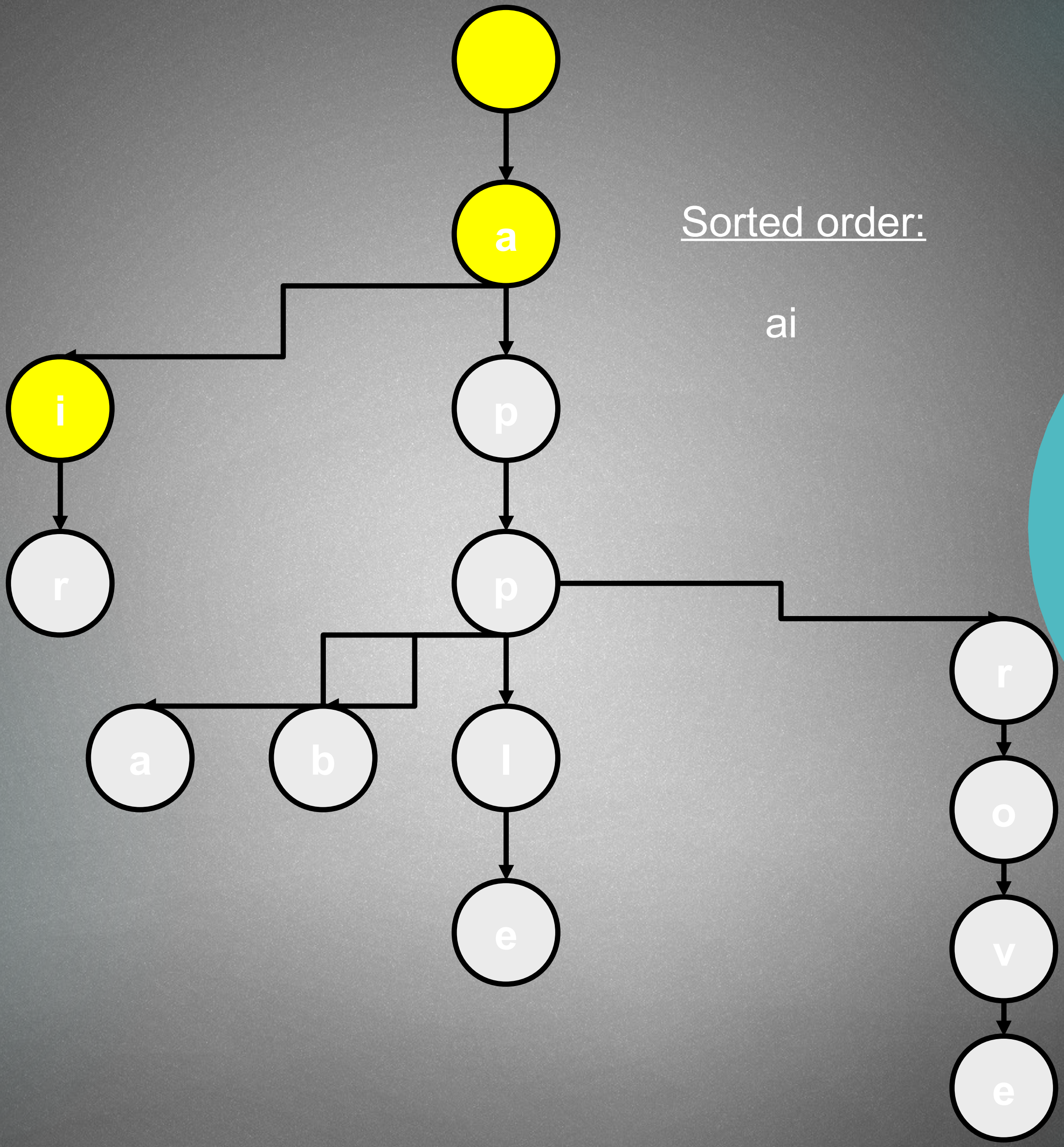
Sorted order:





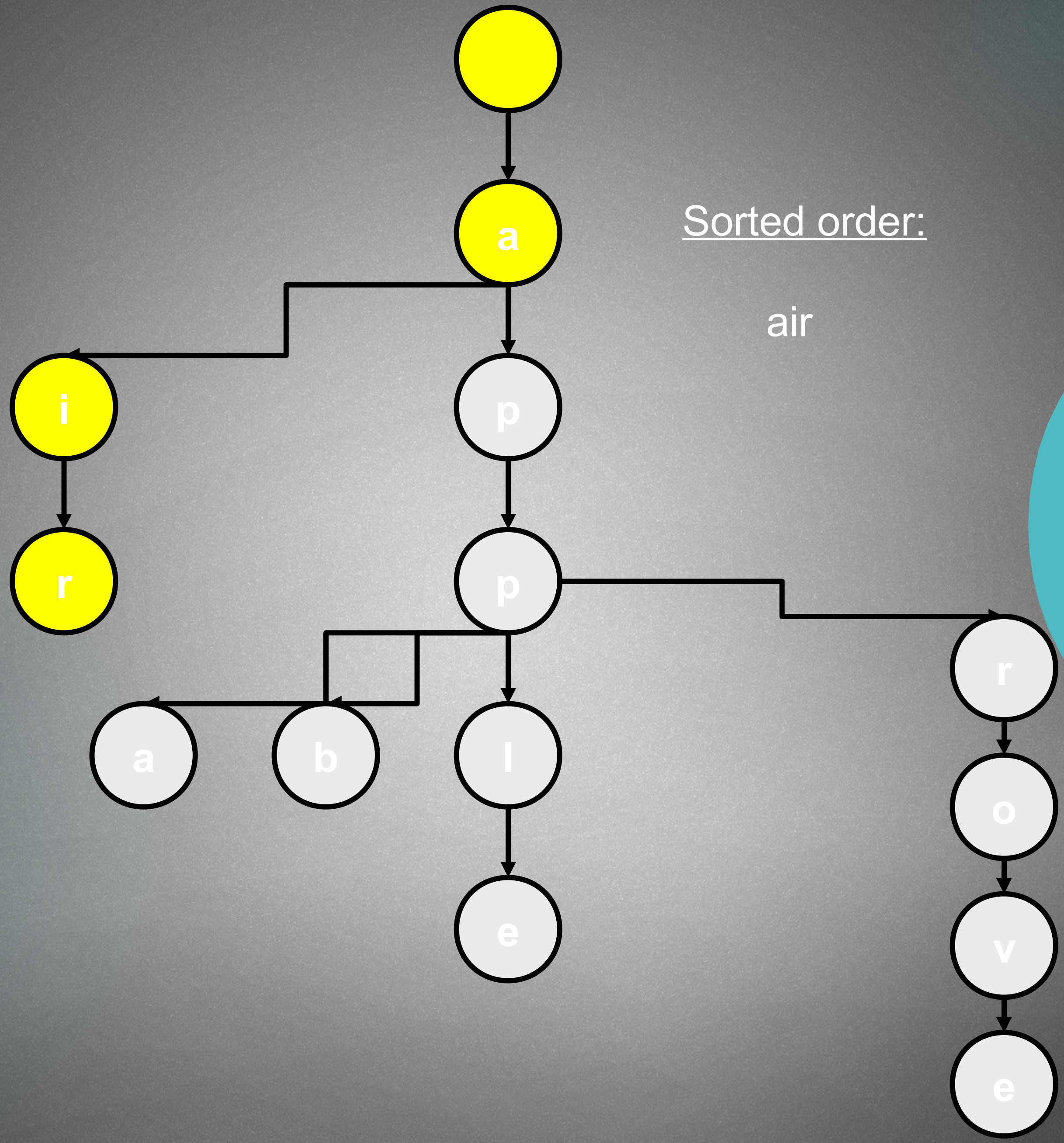
Sorted order:

a



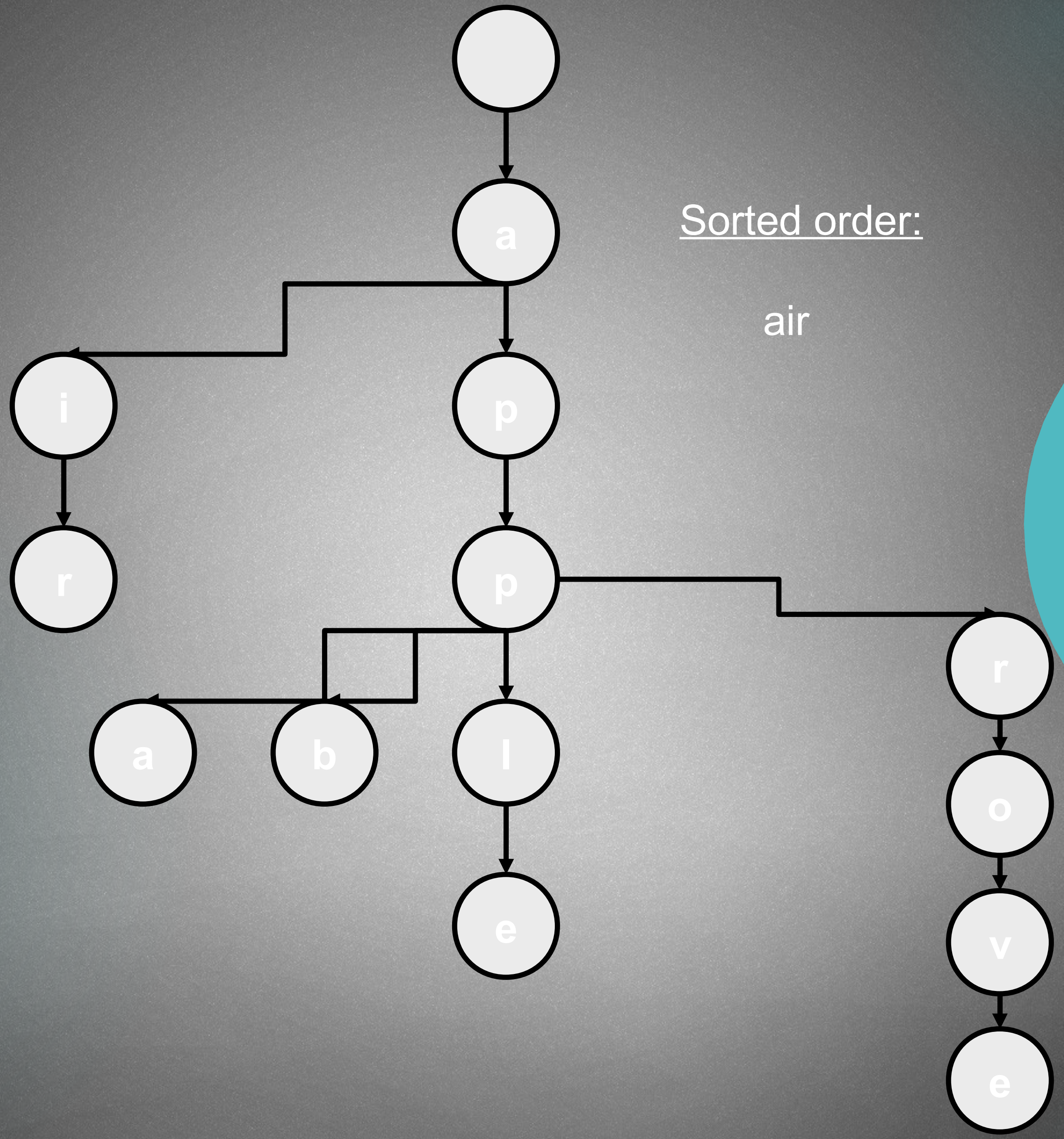
Sorted order:

ai



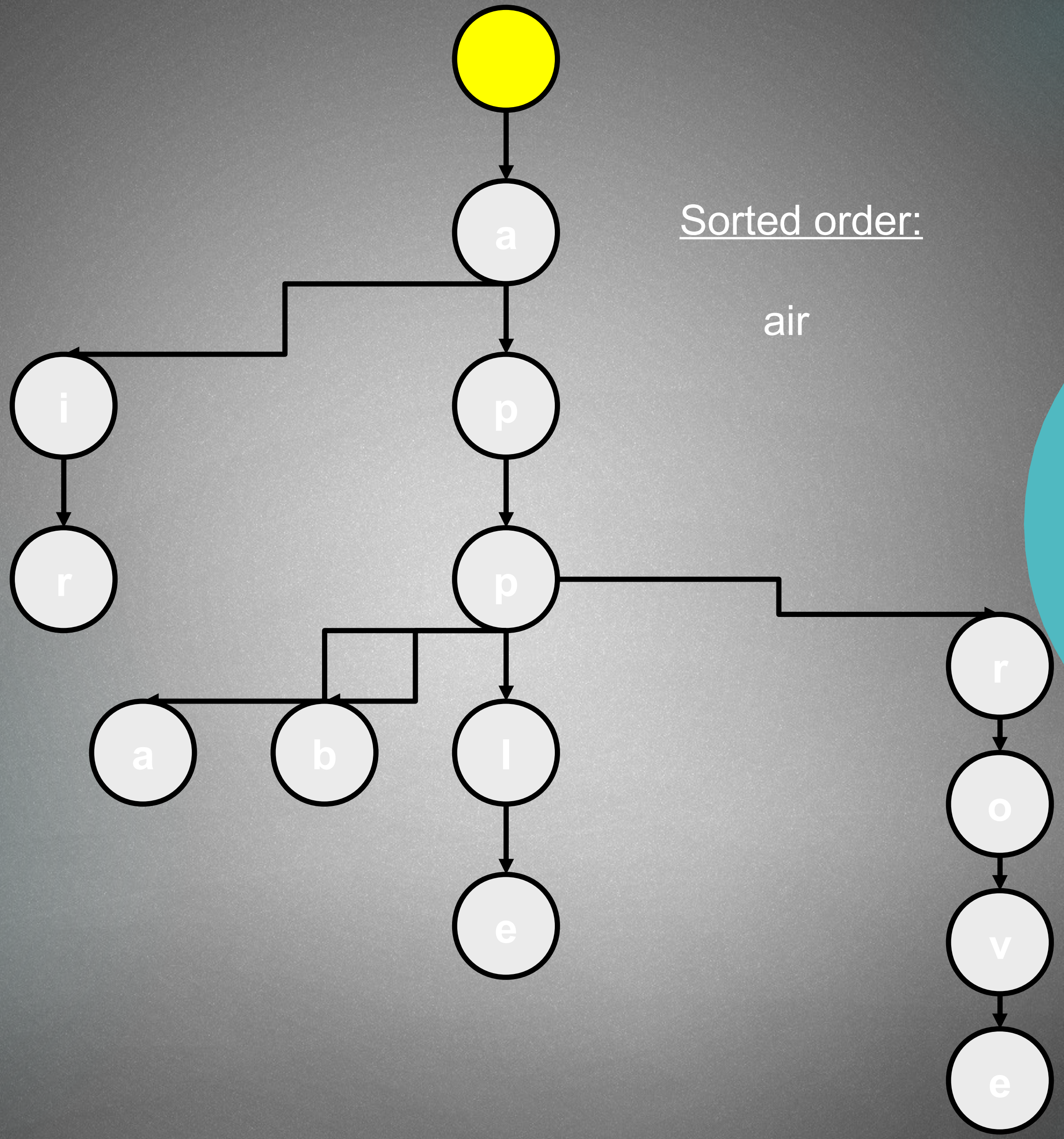
Sorted order:

air



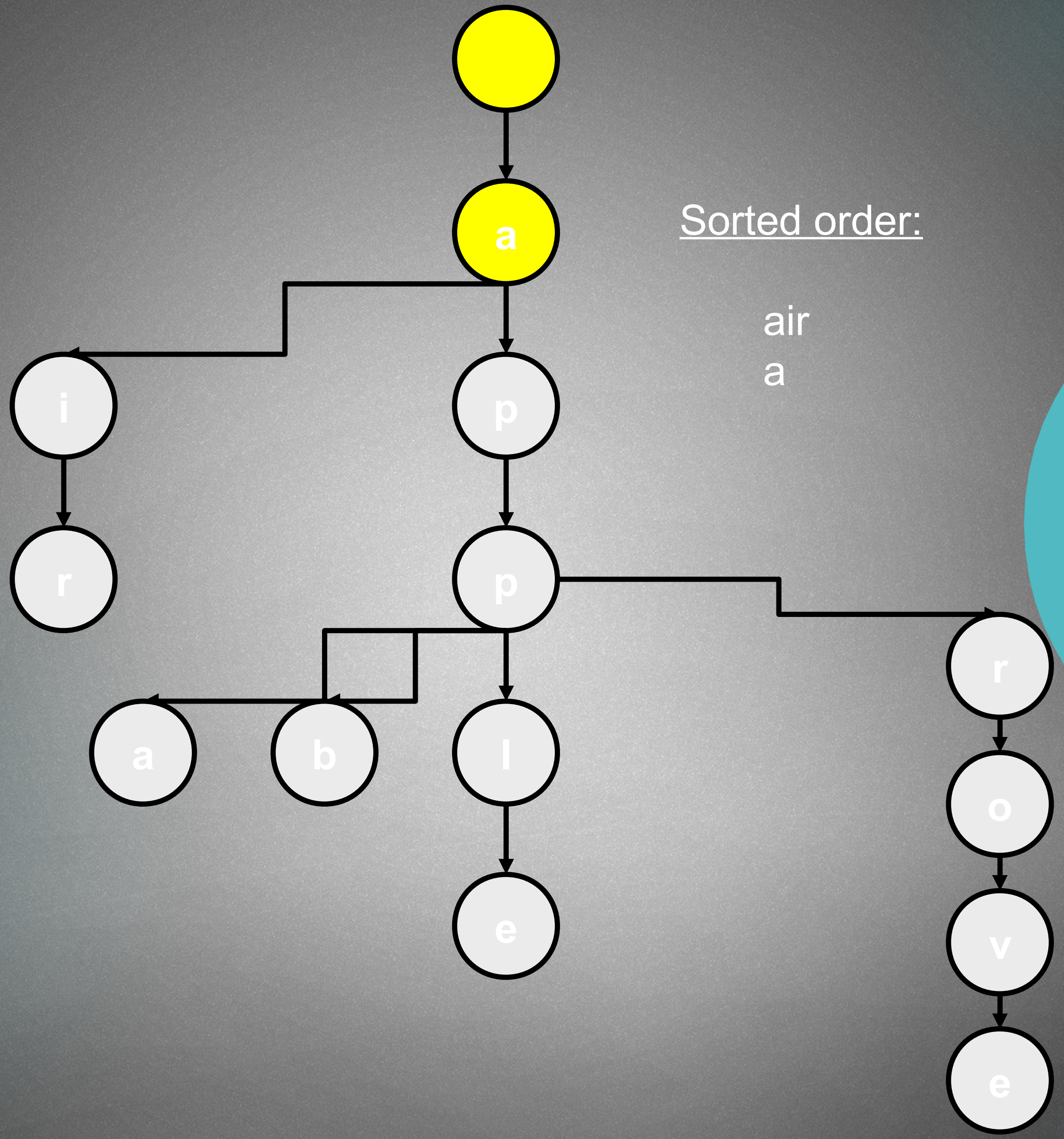
Sorted order:

air



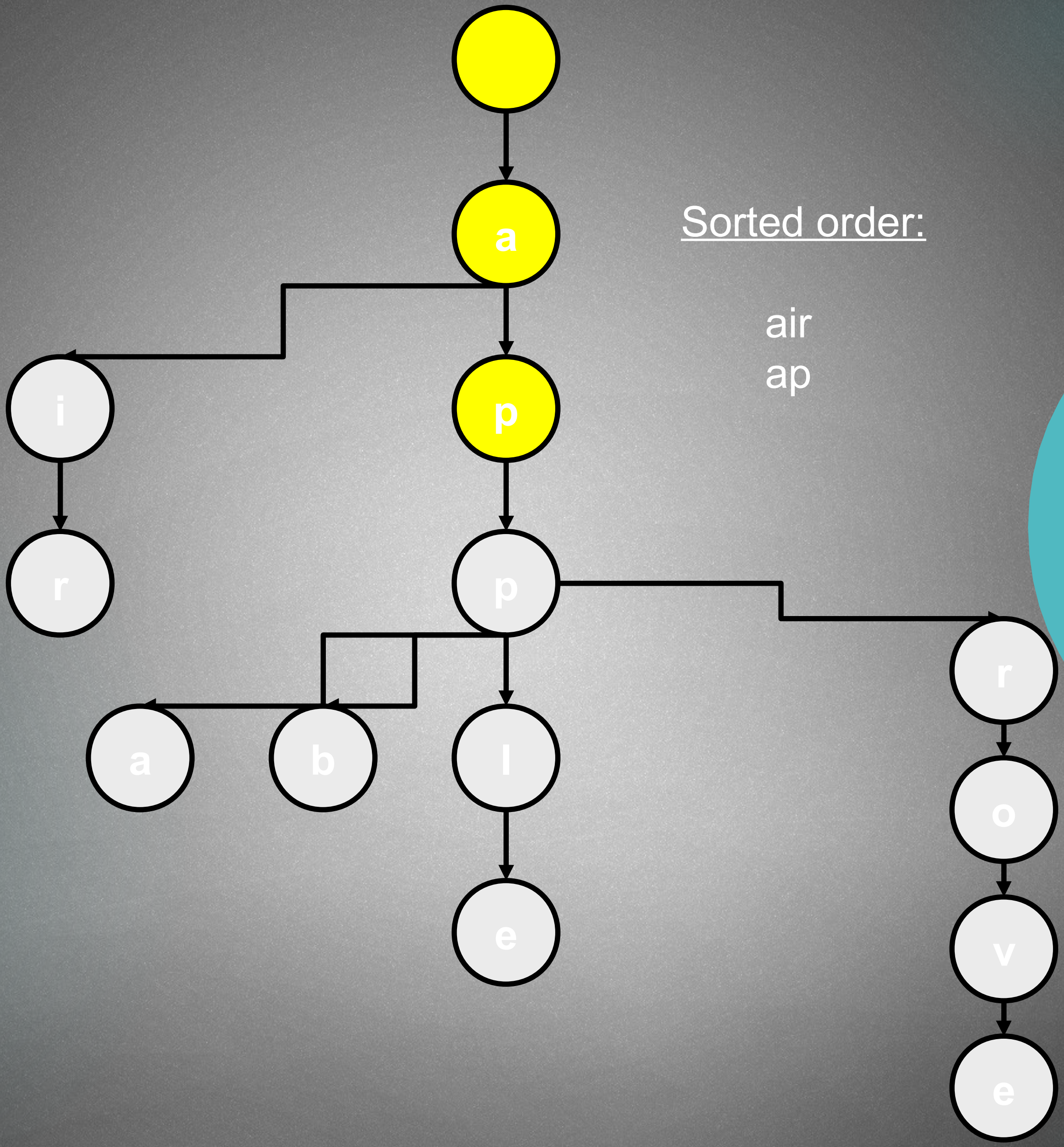
Sorted order:

air



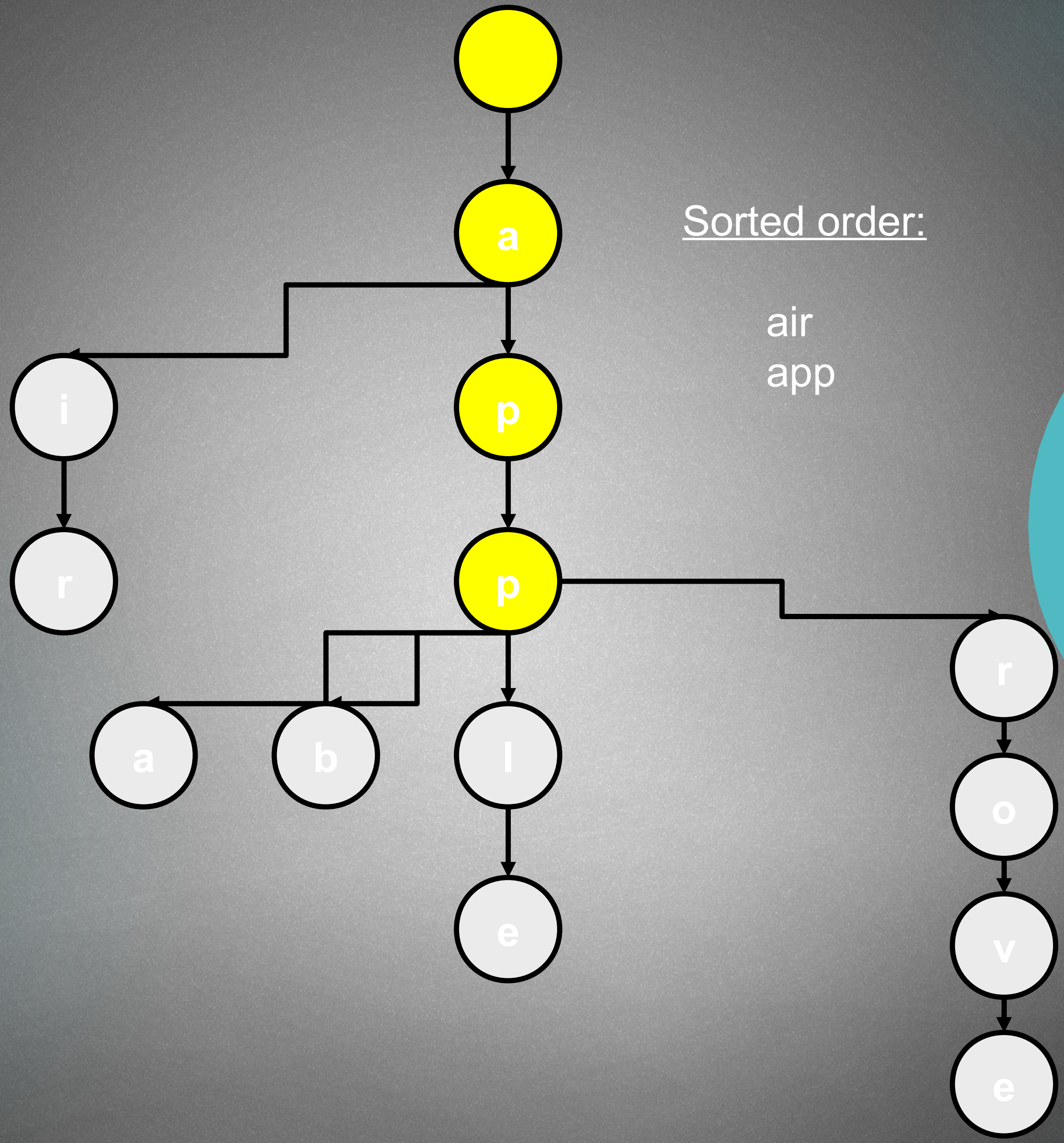
Sorted order:

air
a



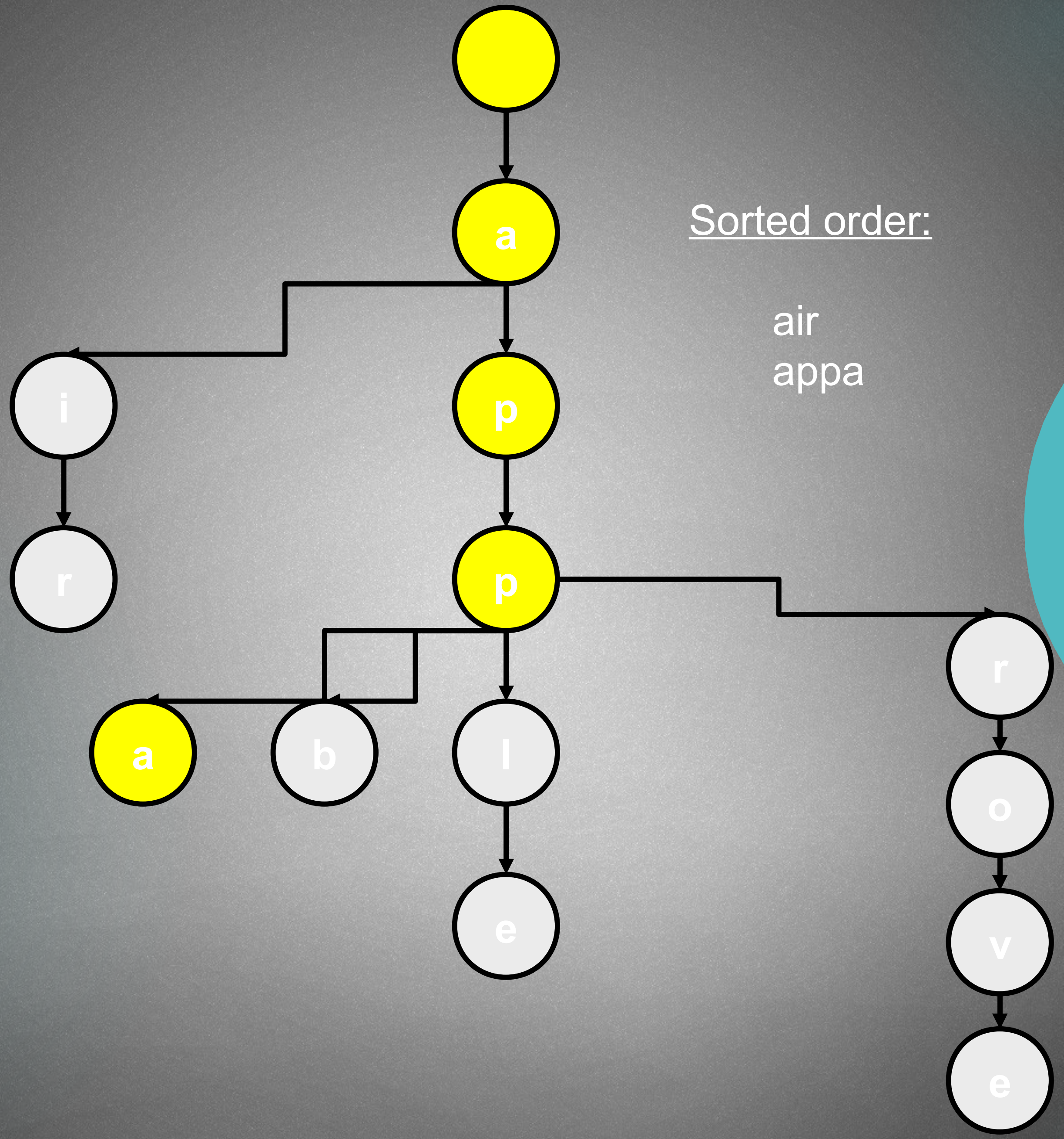
Sorted order:

air
ap



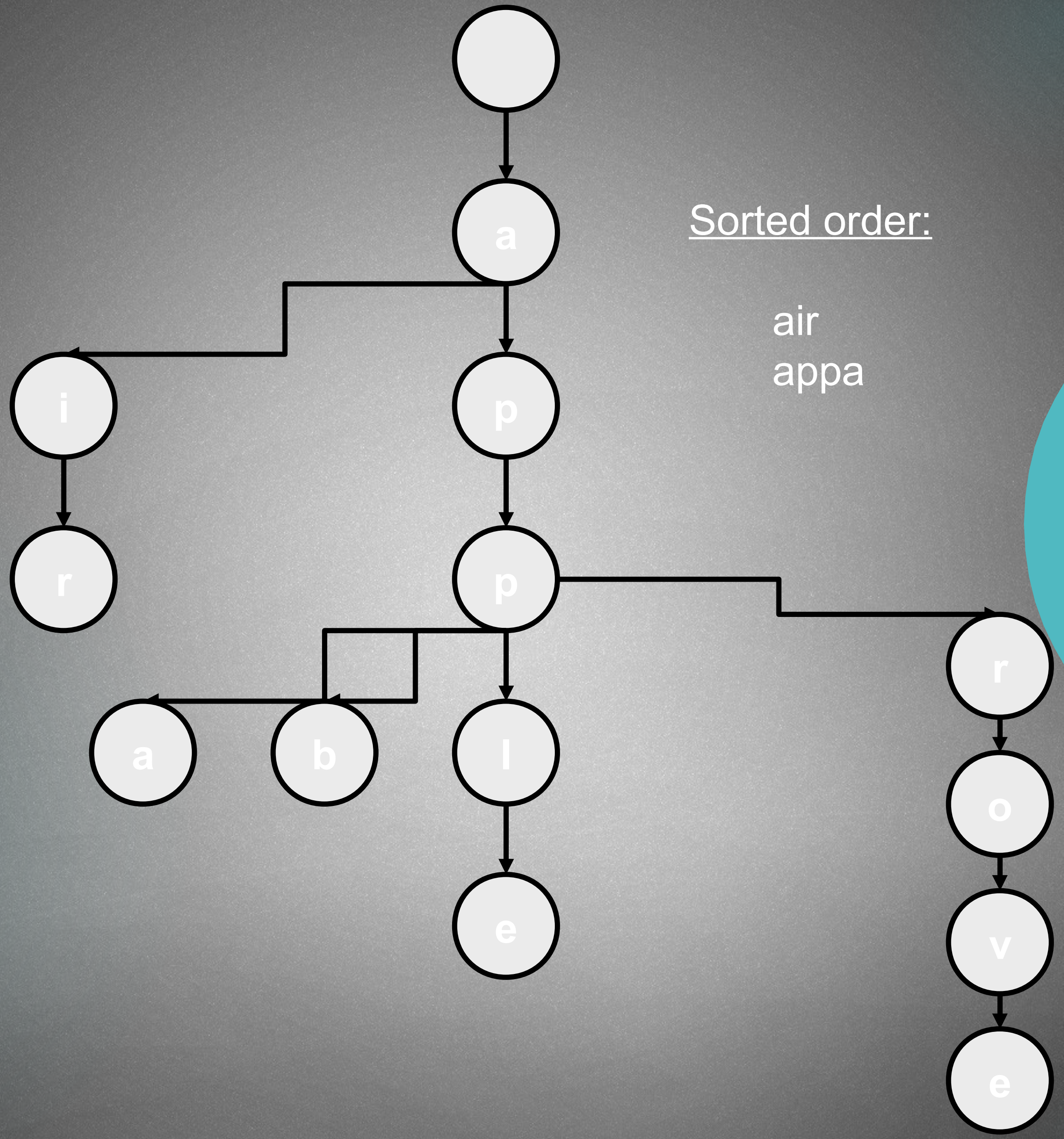
Sorted order:

air
app



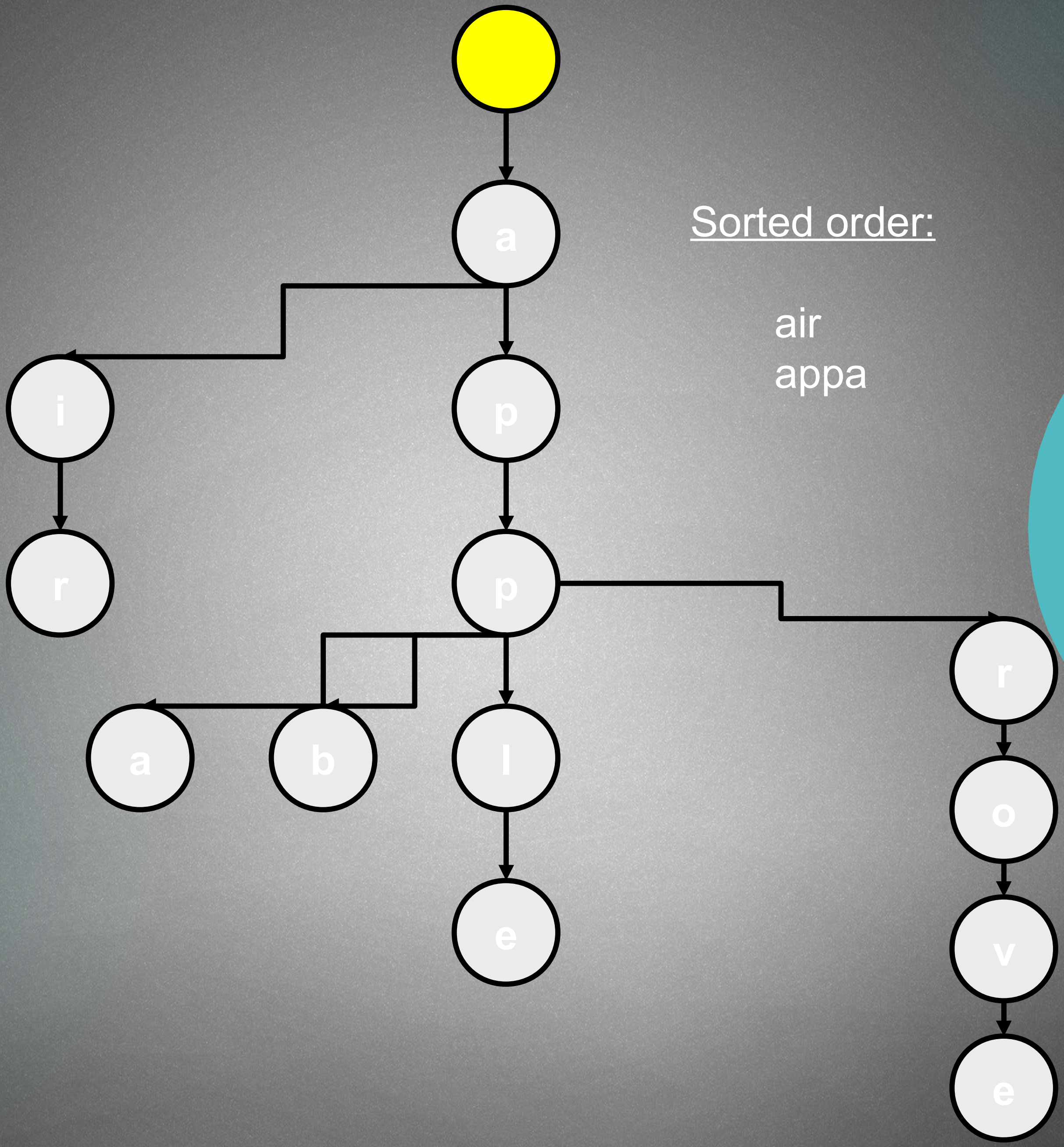
Sorted order:

air
appa



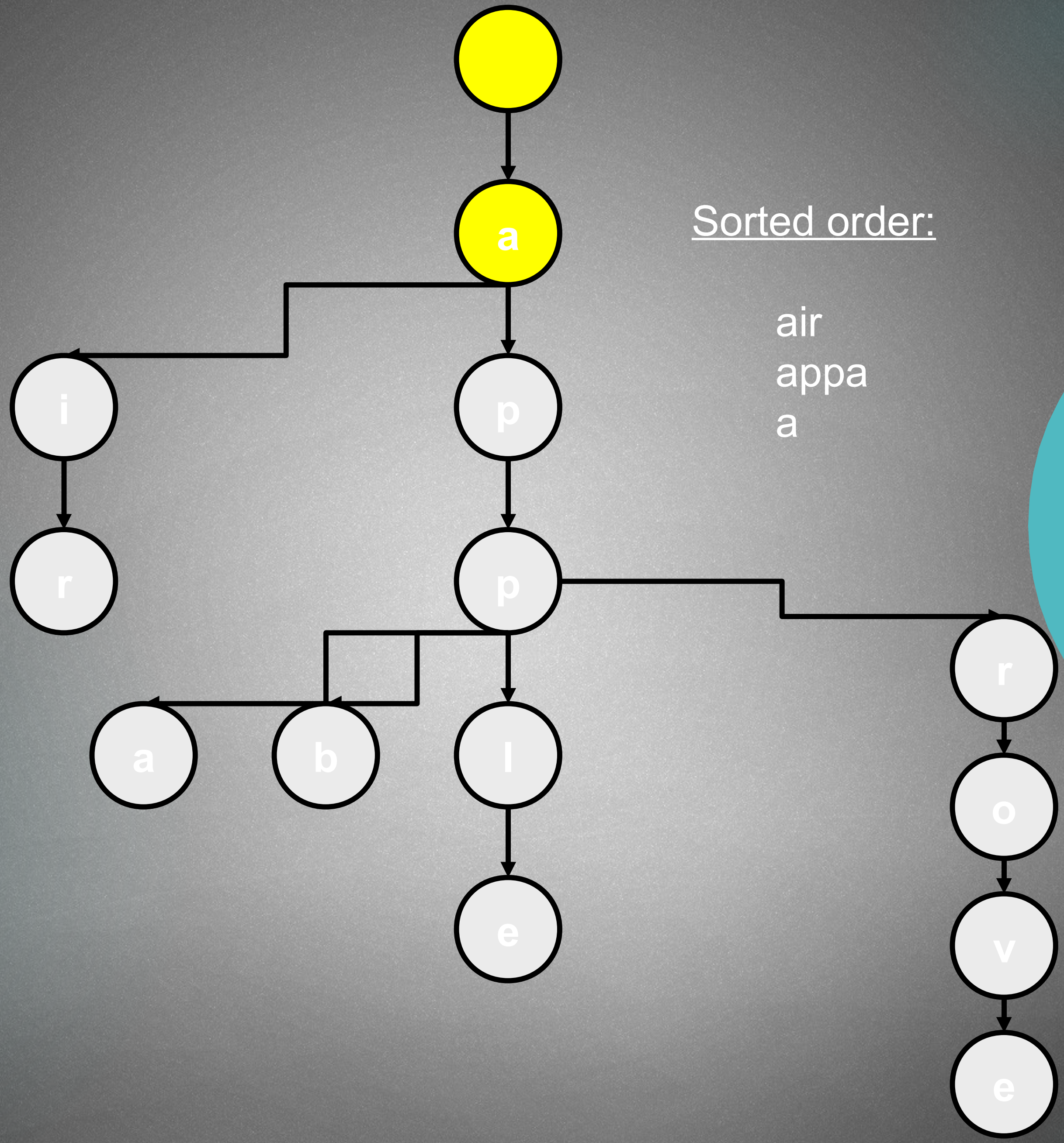
Sorted order:

air
appa



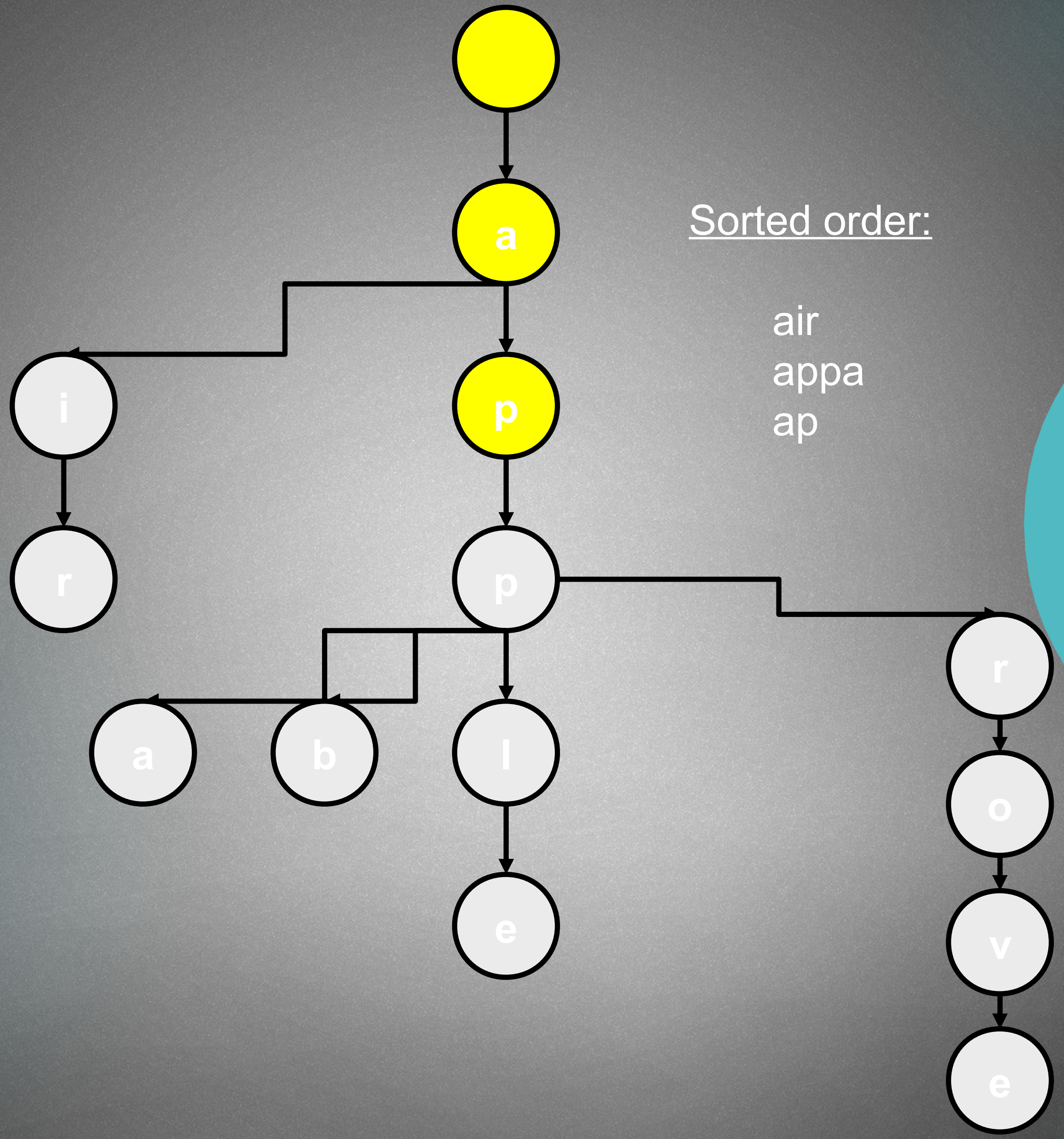
Sorted order:

air
appa



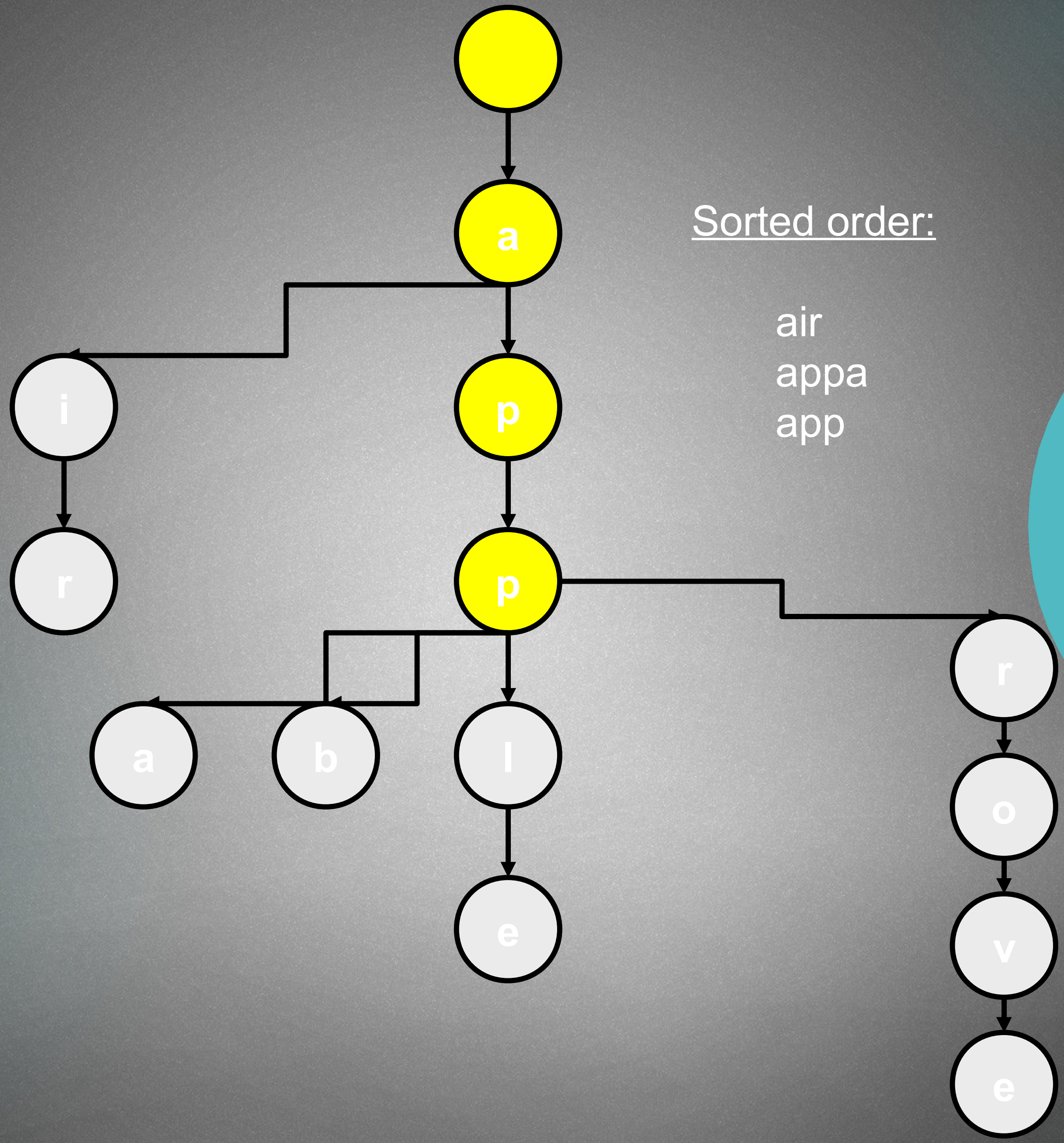
Sorted order:

air
appa
a



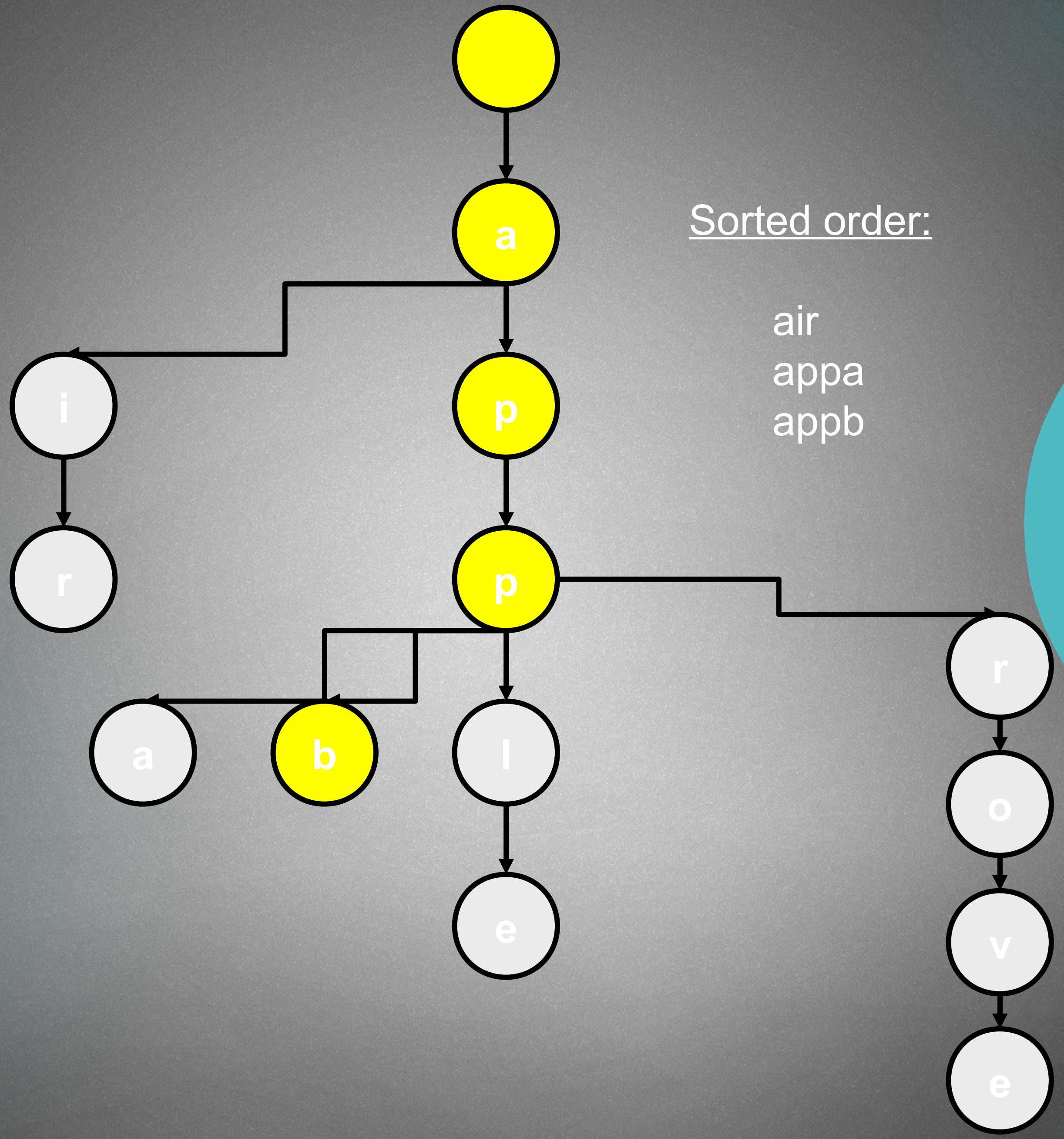
Sorted order:

air
appa
ap



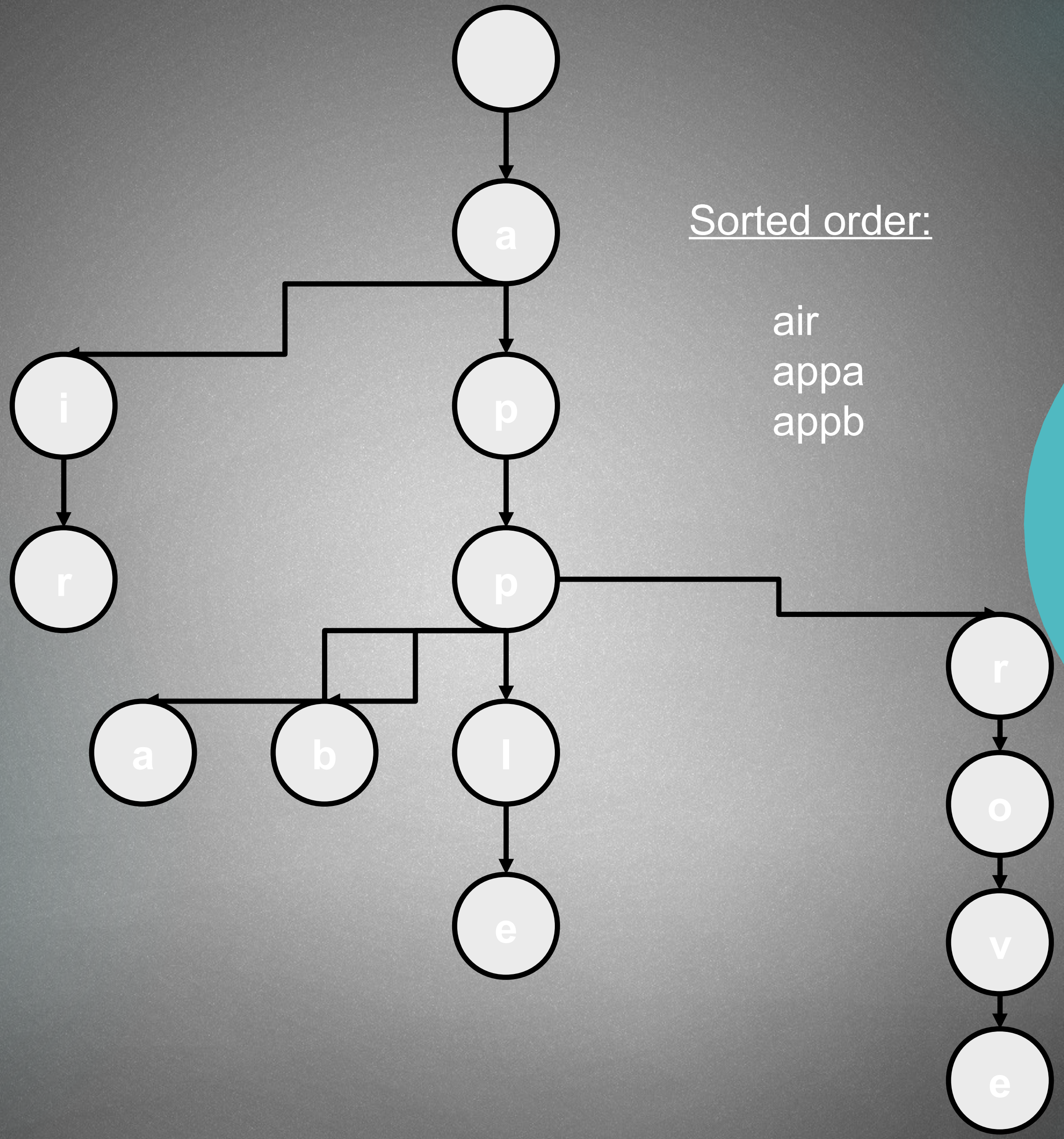
Sorted order:

air
appa
app



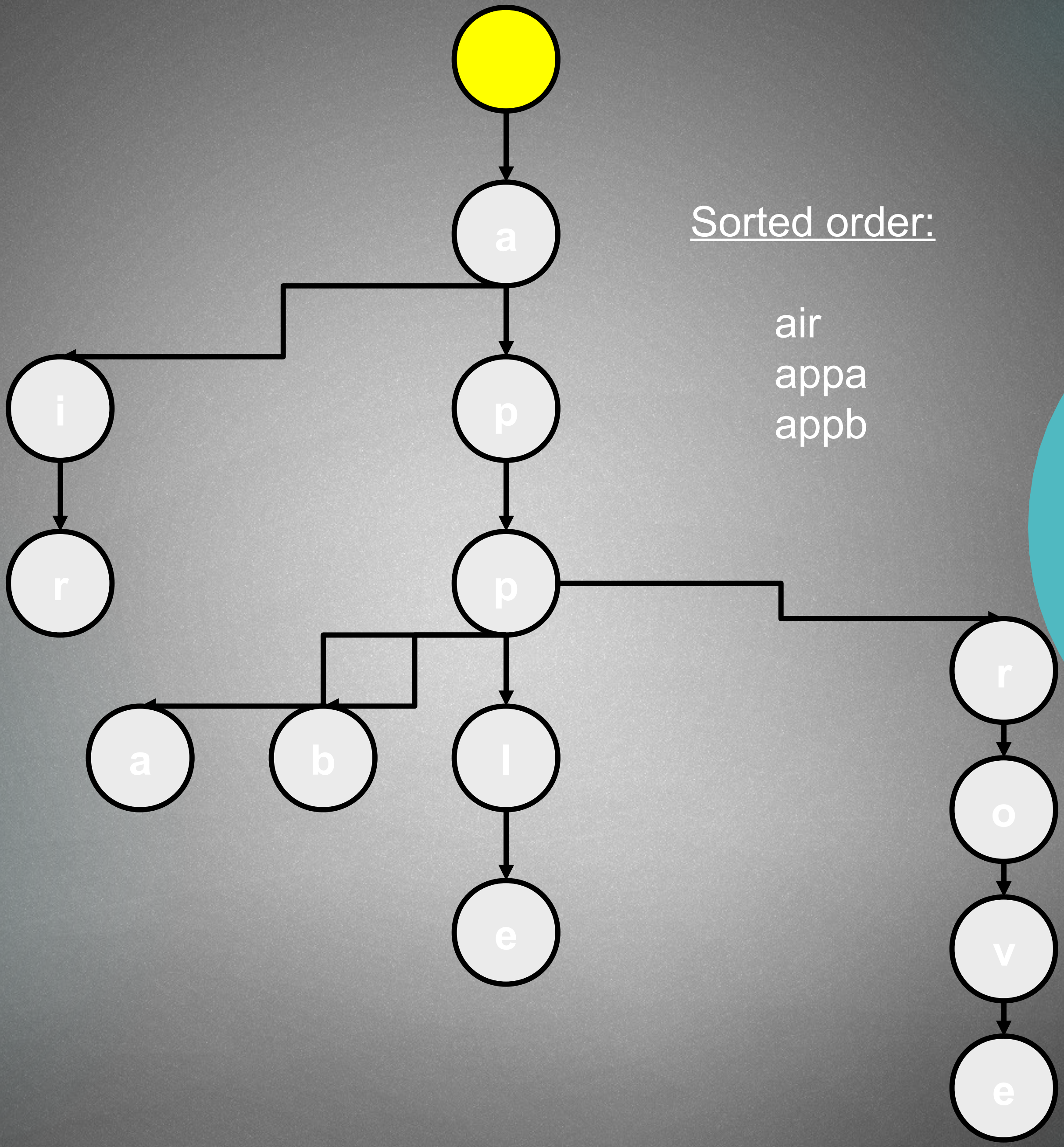
Sorted order:

air
appa
appb



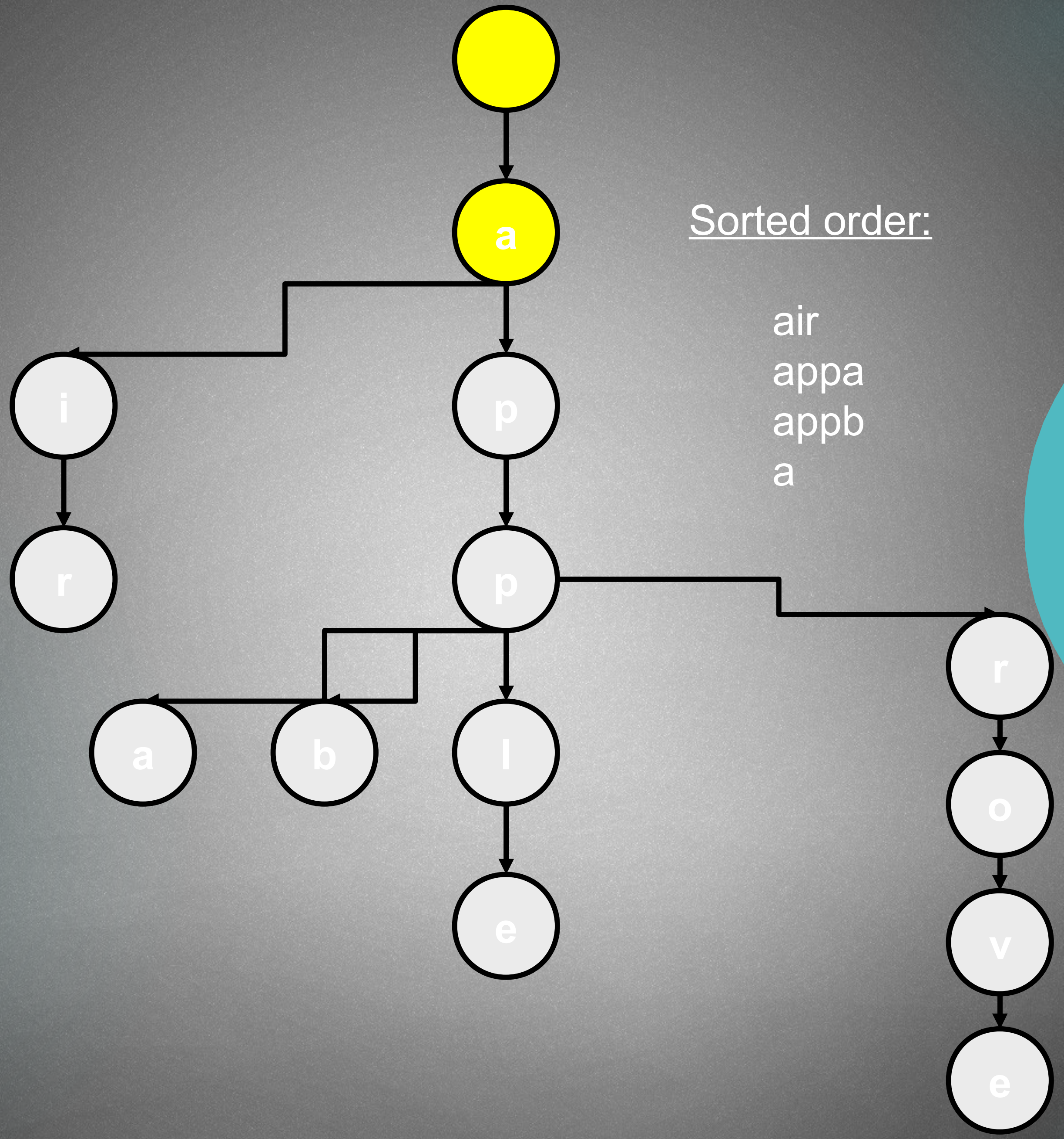
Sorted order:

air
appa
appb



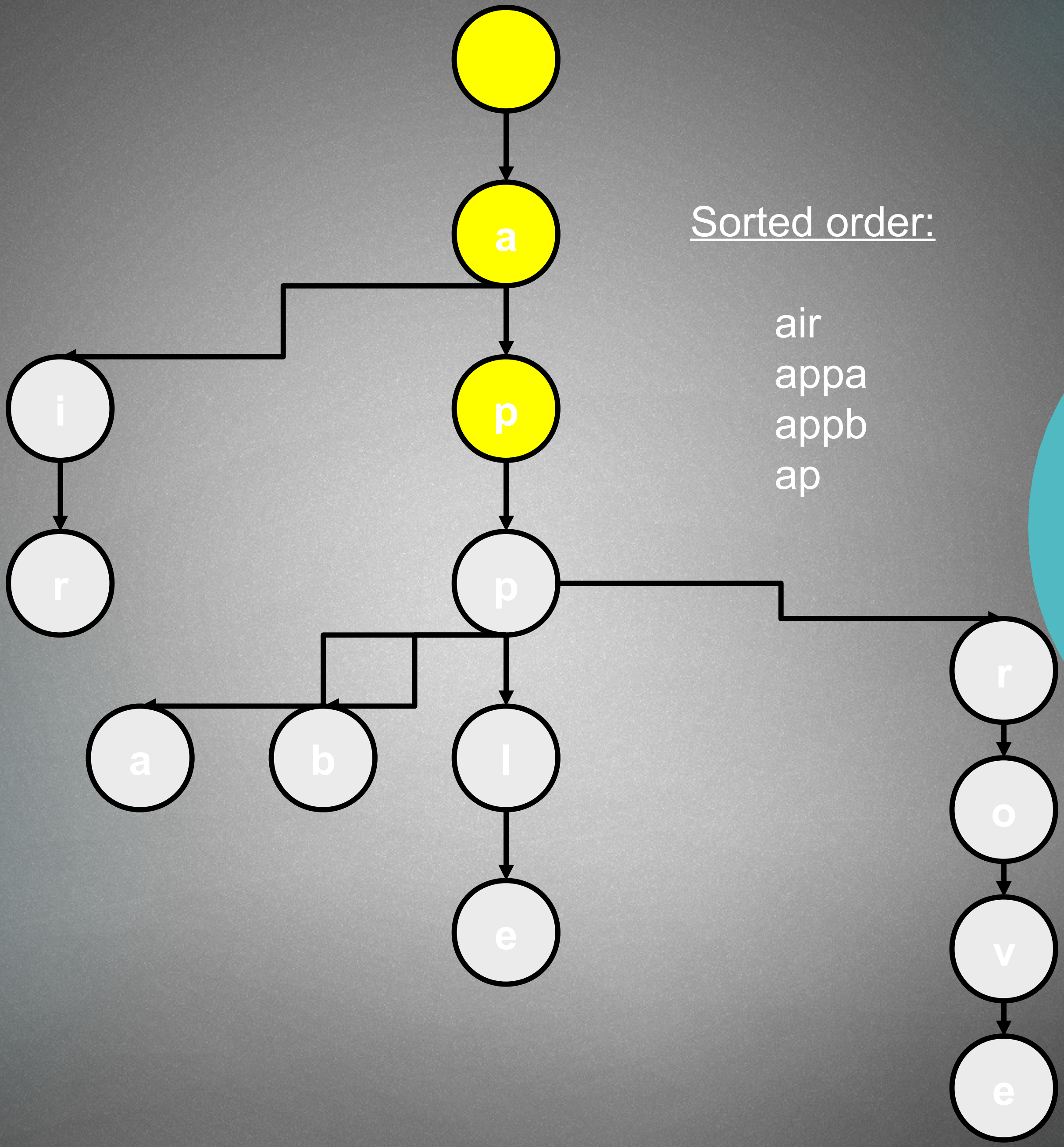
Sorted order:

air
appa
appb



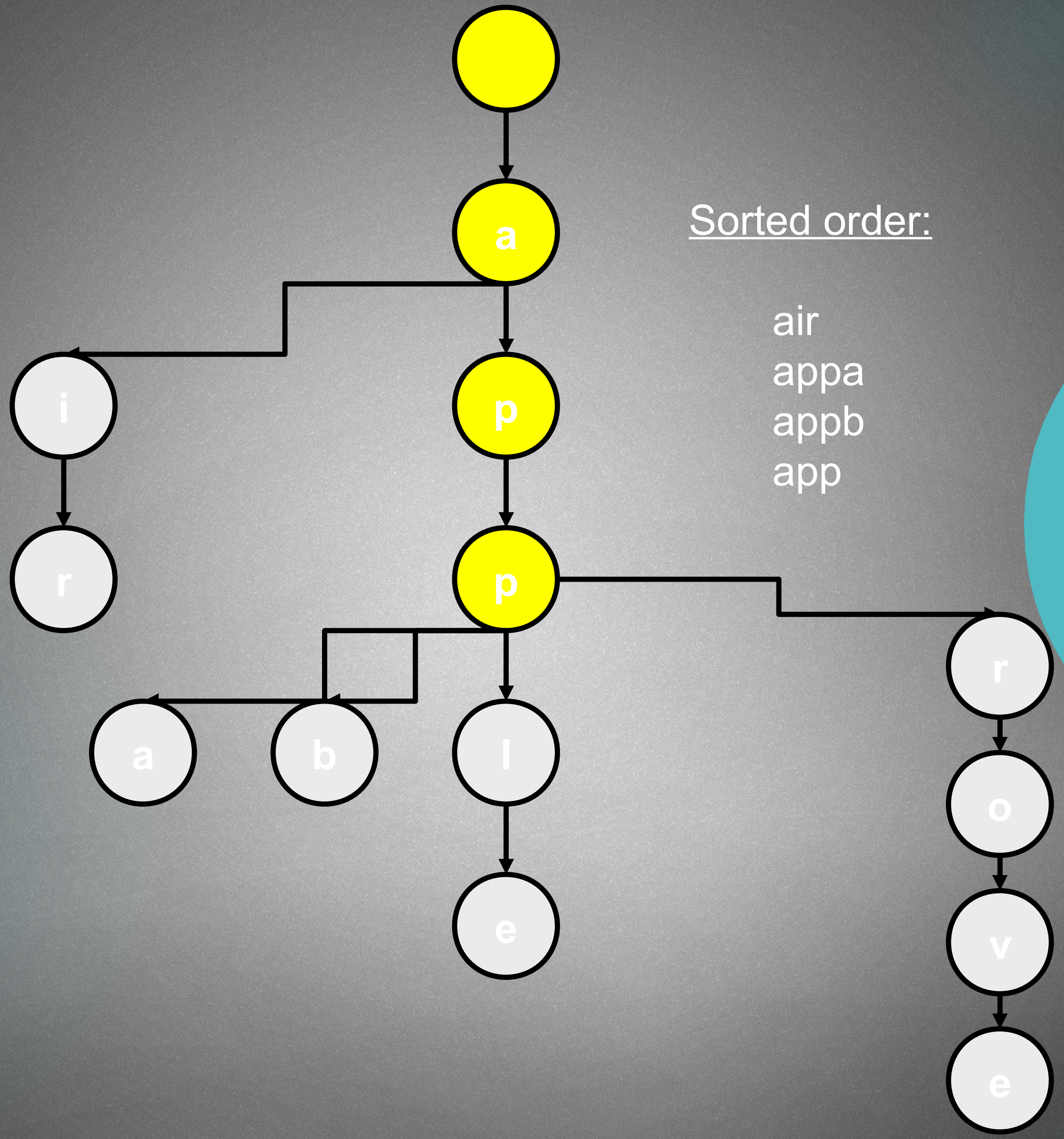
Sorted order:

air
appa
appb
a



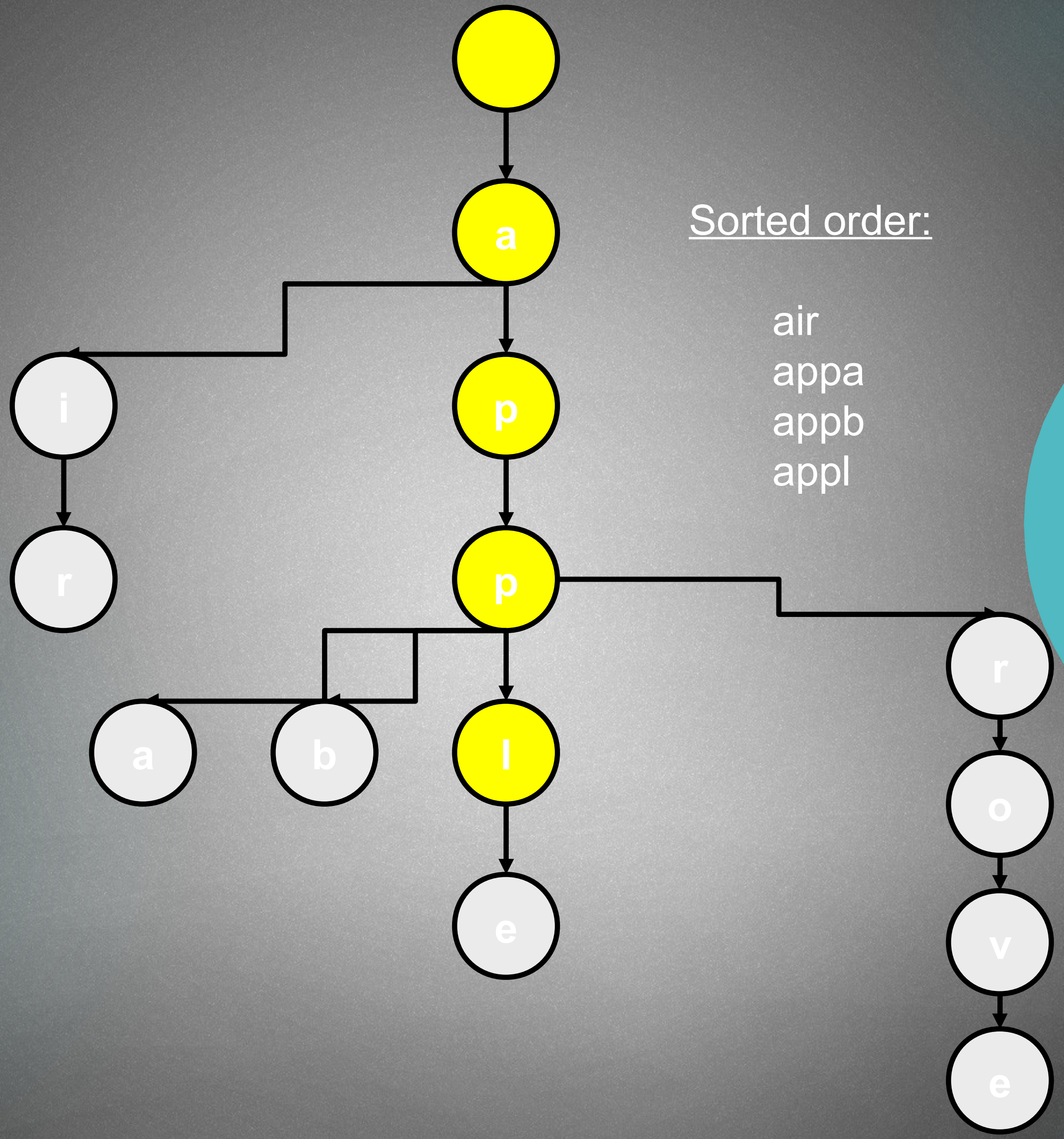
Sorted order:

air
appa
appb
ap



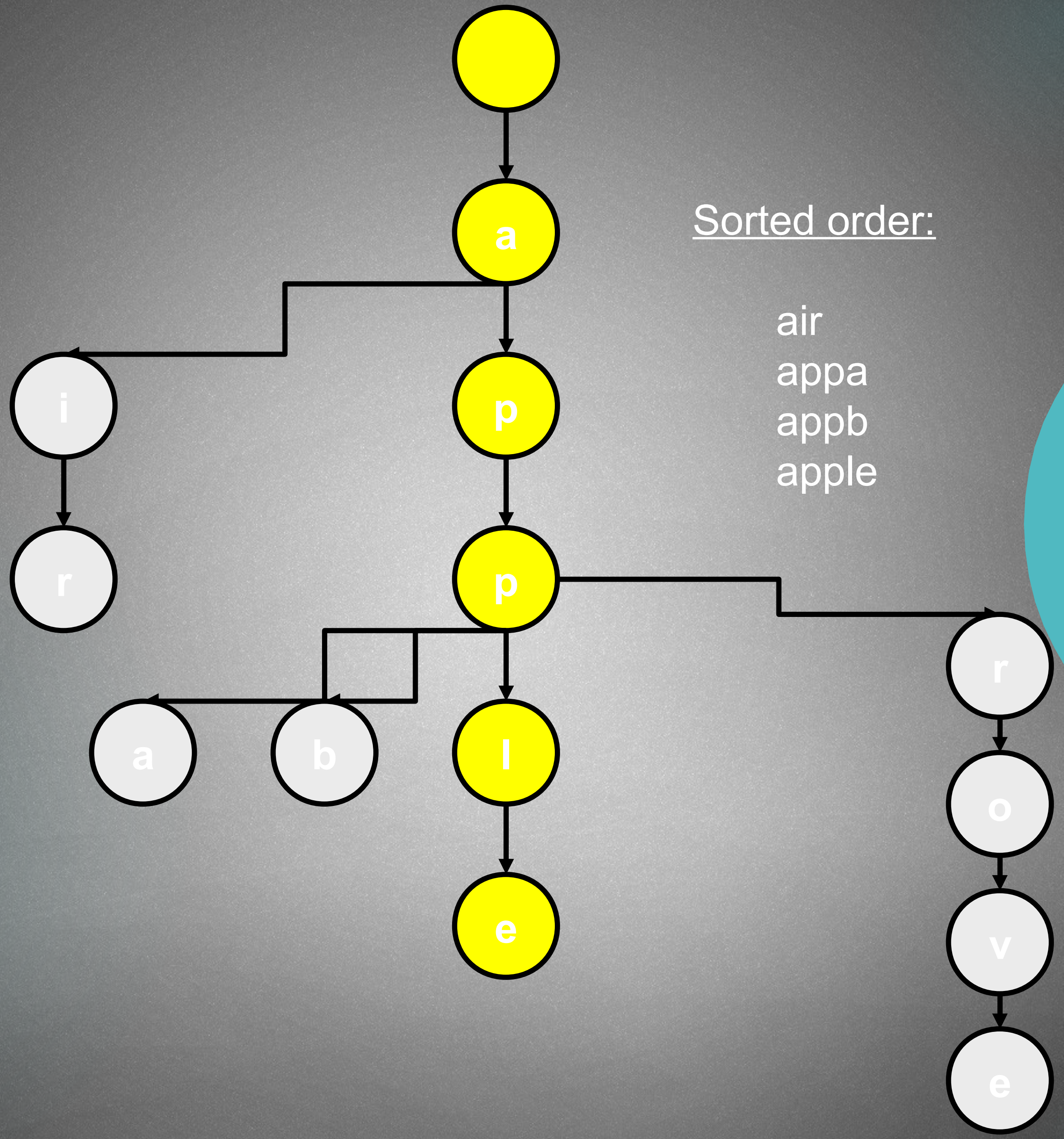
Sorted order:

air
appa
appb
app



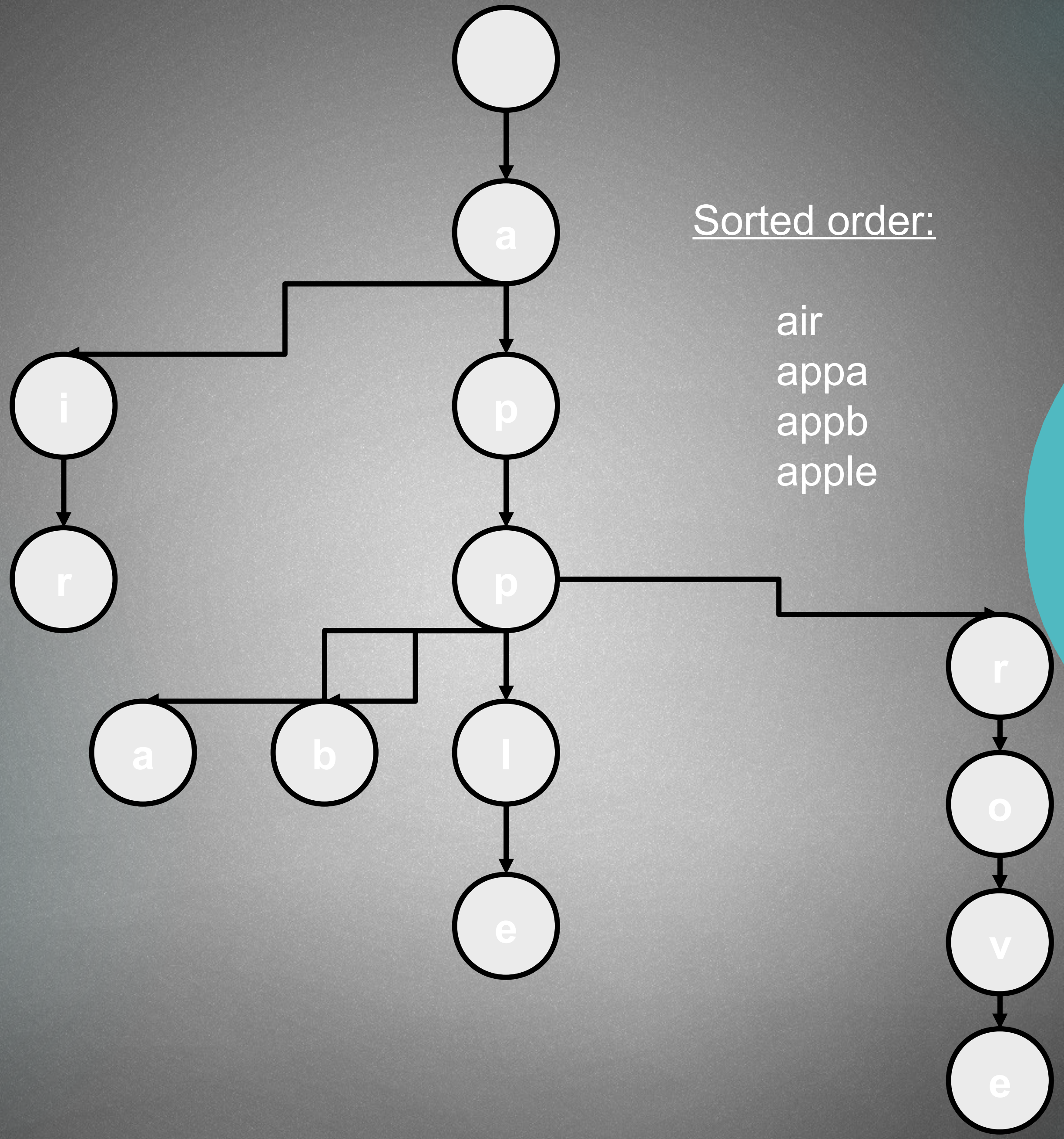
Sorted order:

air
appa
appb
appl



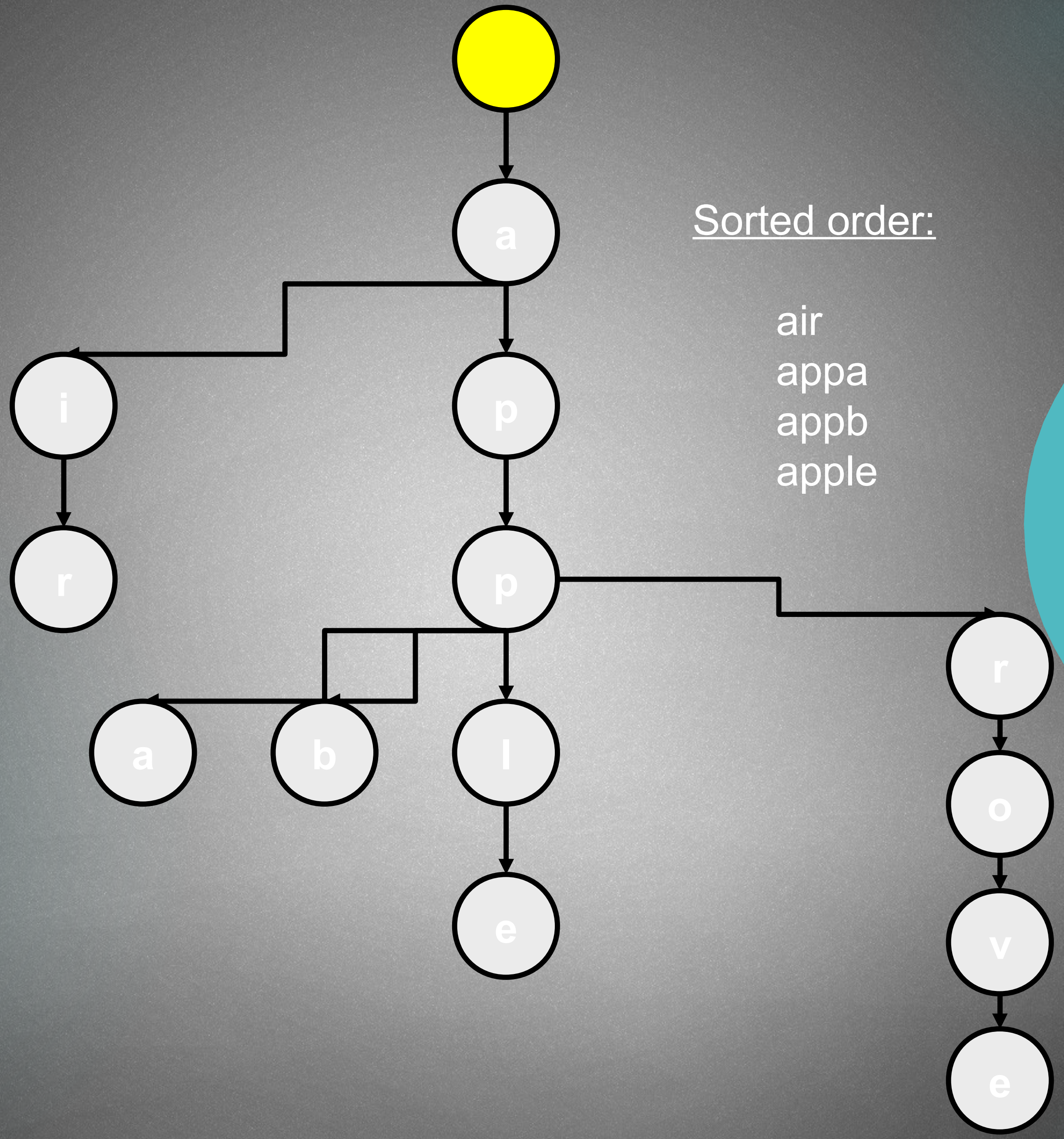
Sorted order:

air
appa
appb
apple



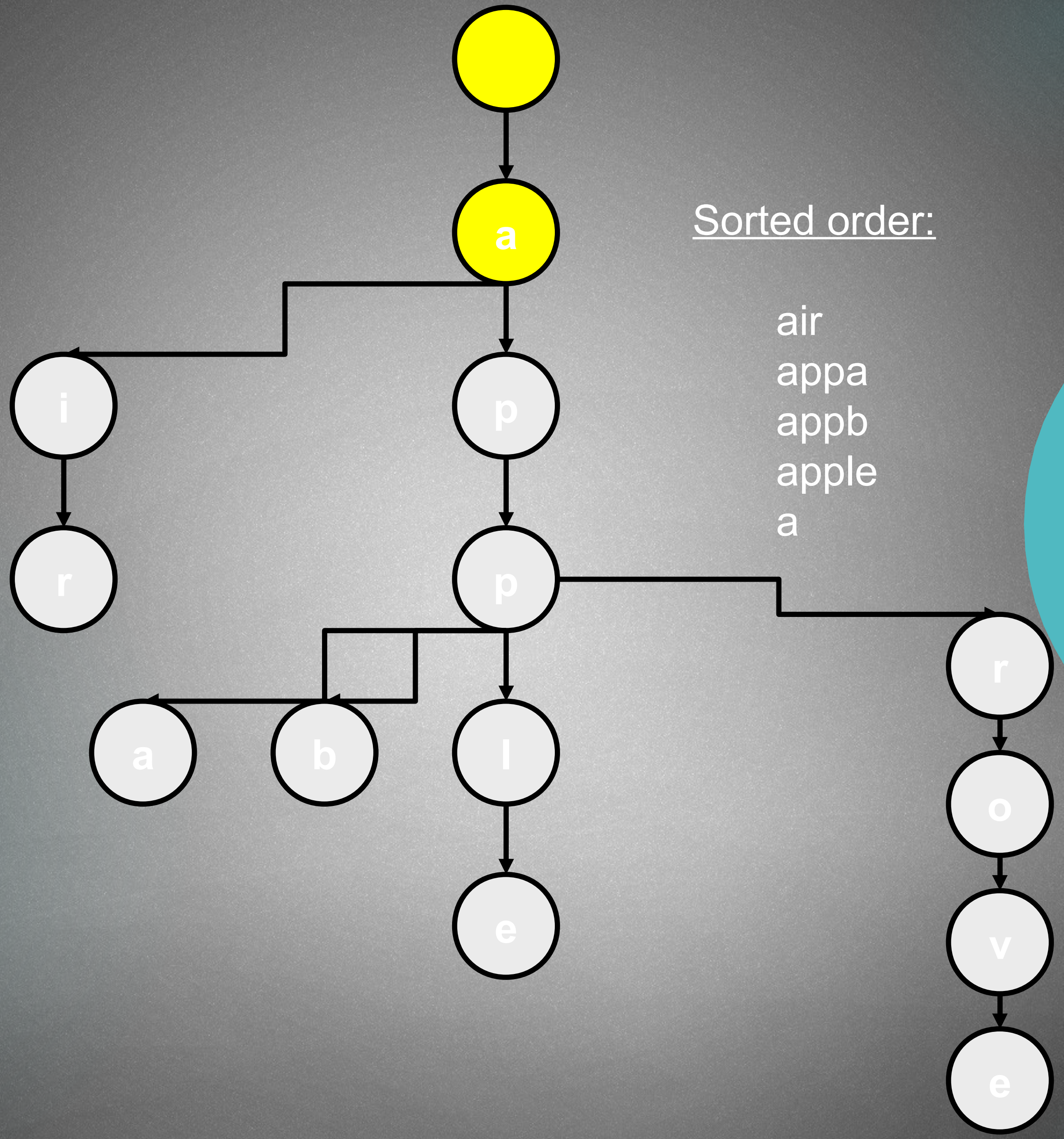
Sorted order:

air
appa
appb
apple



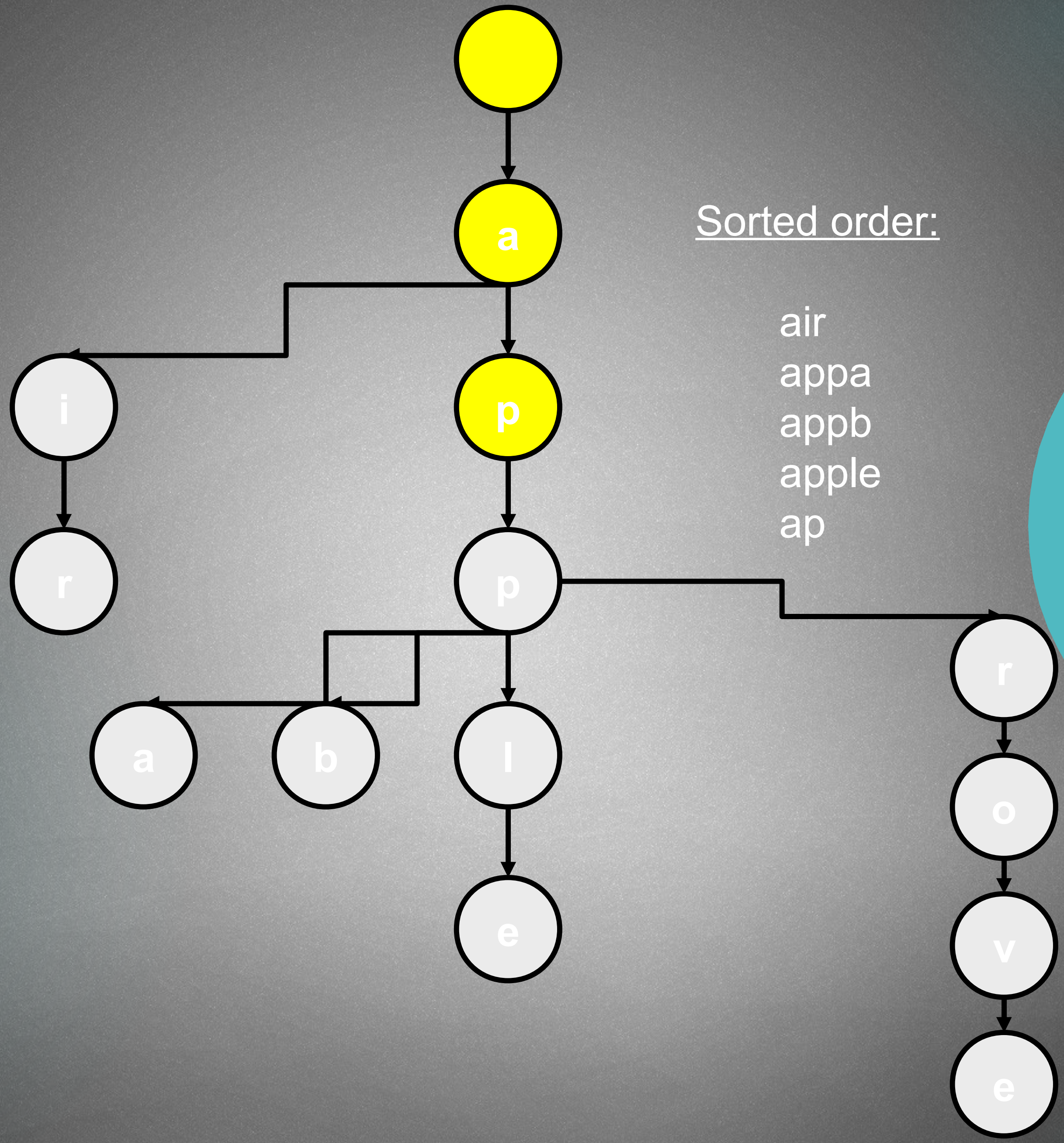
Sorted order:

air
appa
appb
apple



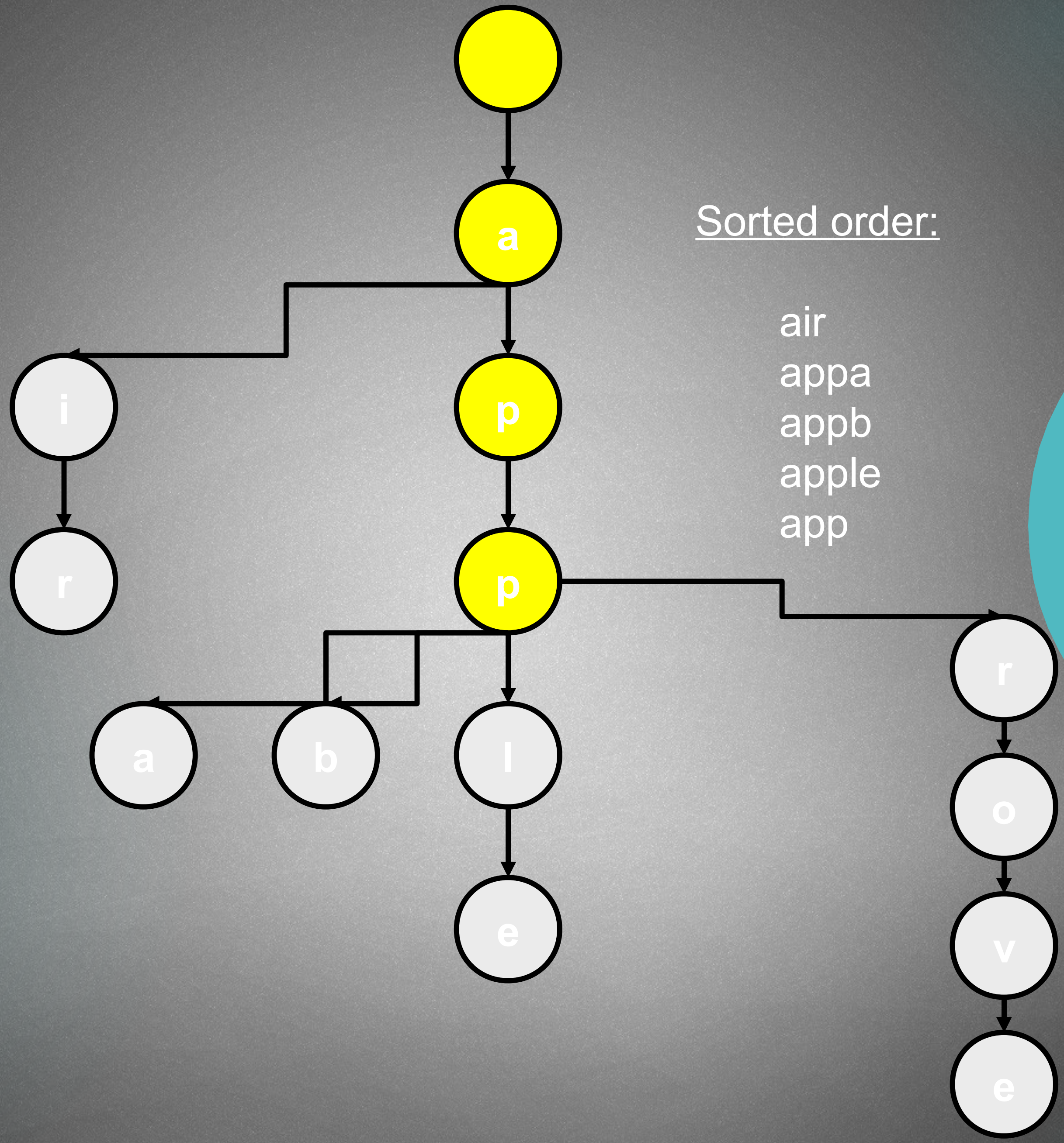
Sorted order:

air
appa
appb
apple
a



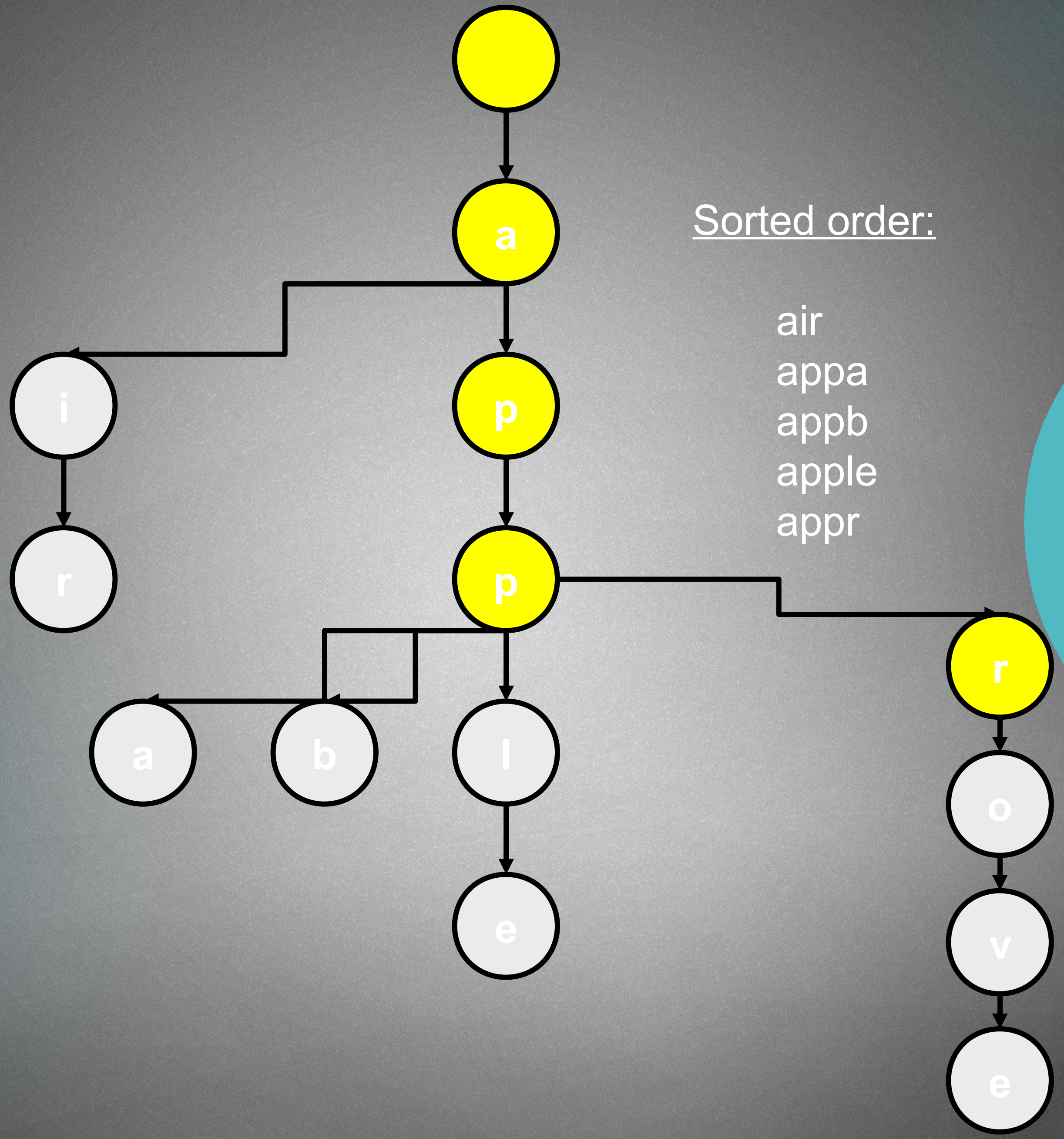
Sorted order:

air
appa
appb
apple
ap



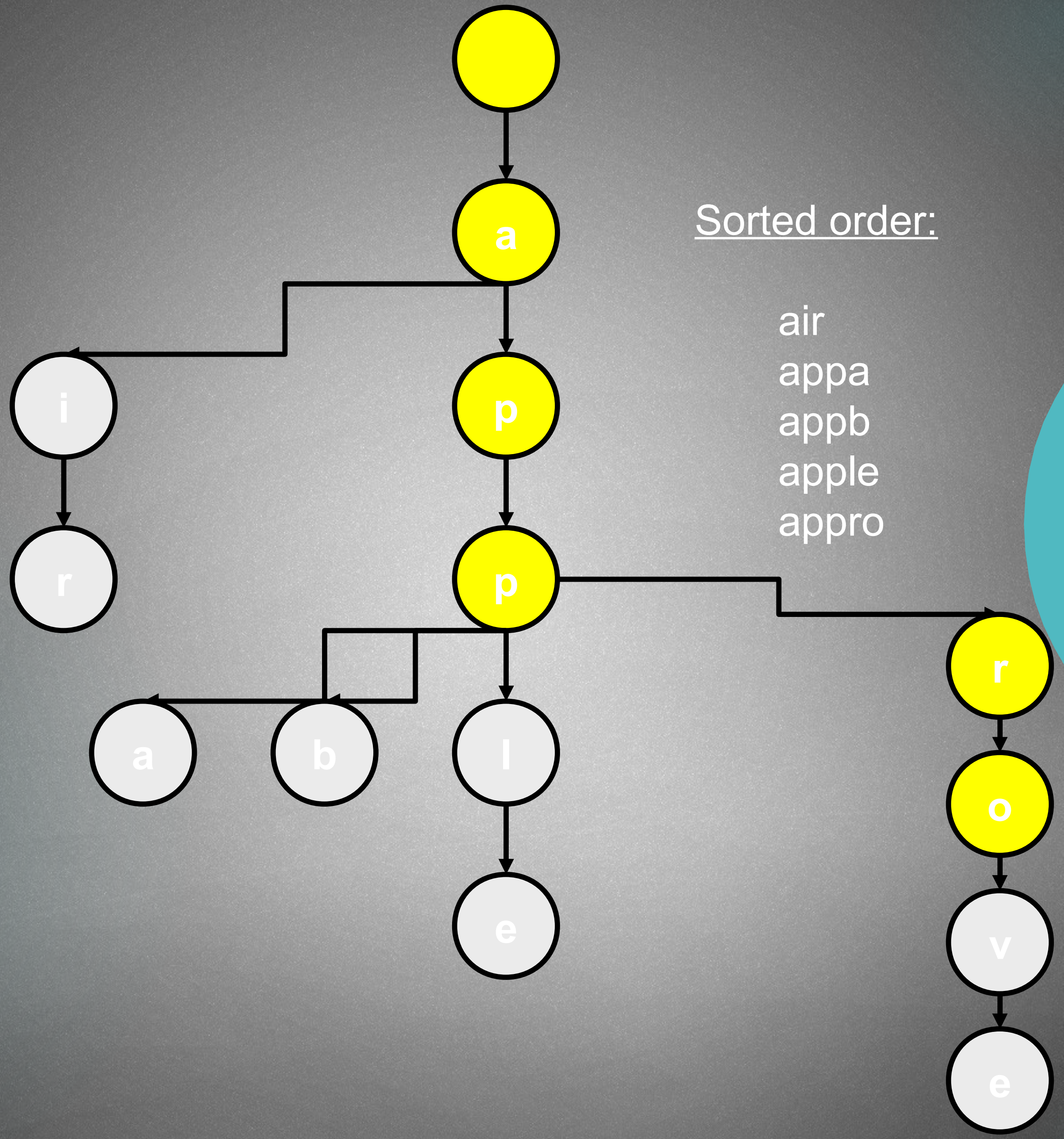
Sorted order:

air
appa
appb
apple
app



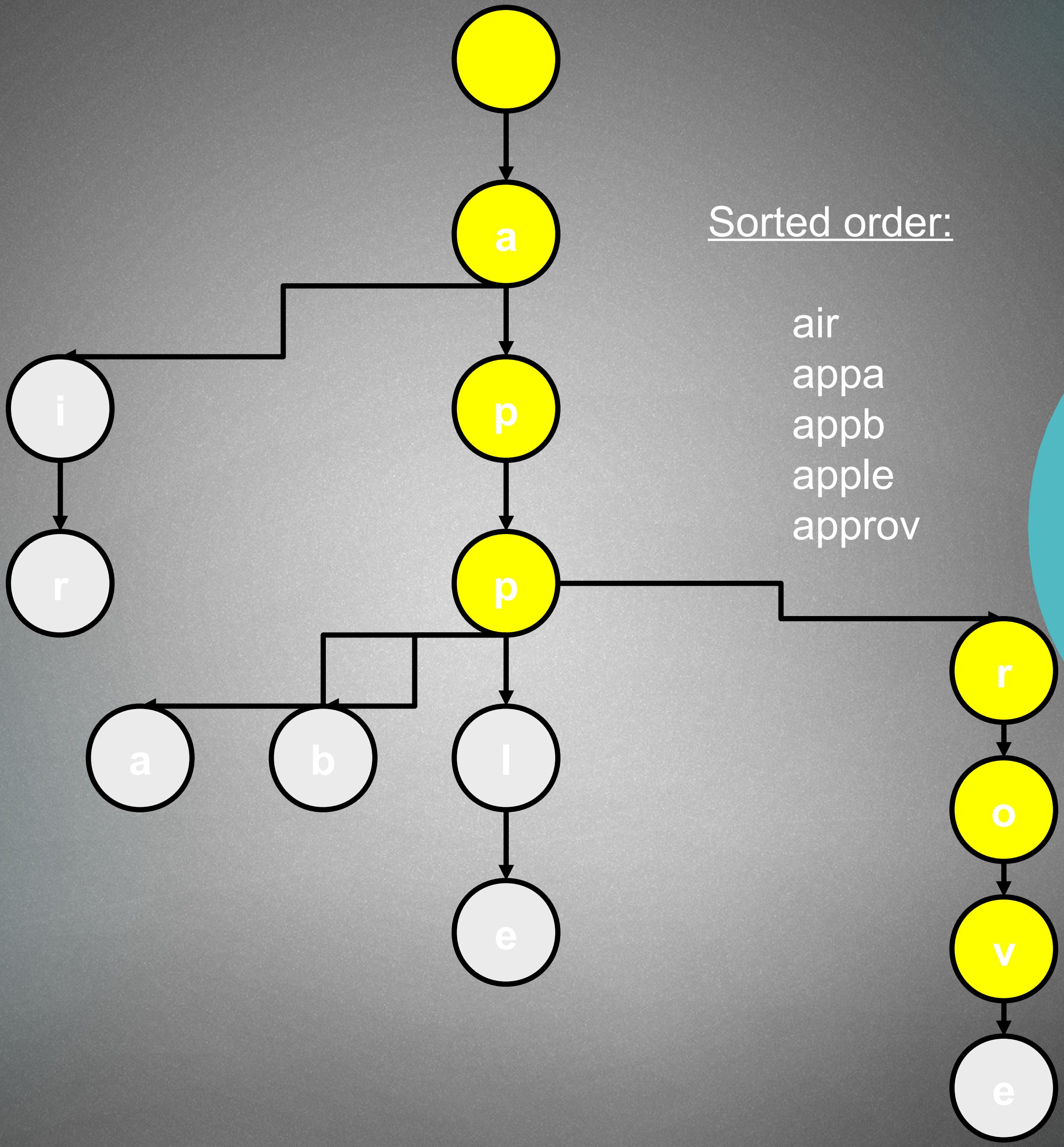
Sorted order:

air
appa
appb
apple
appr



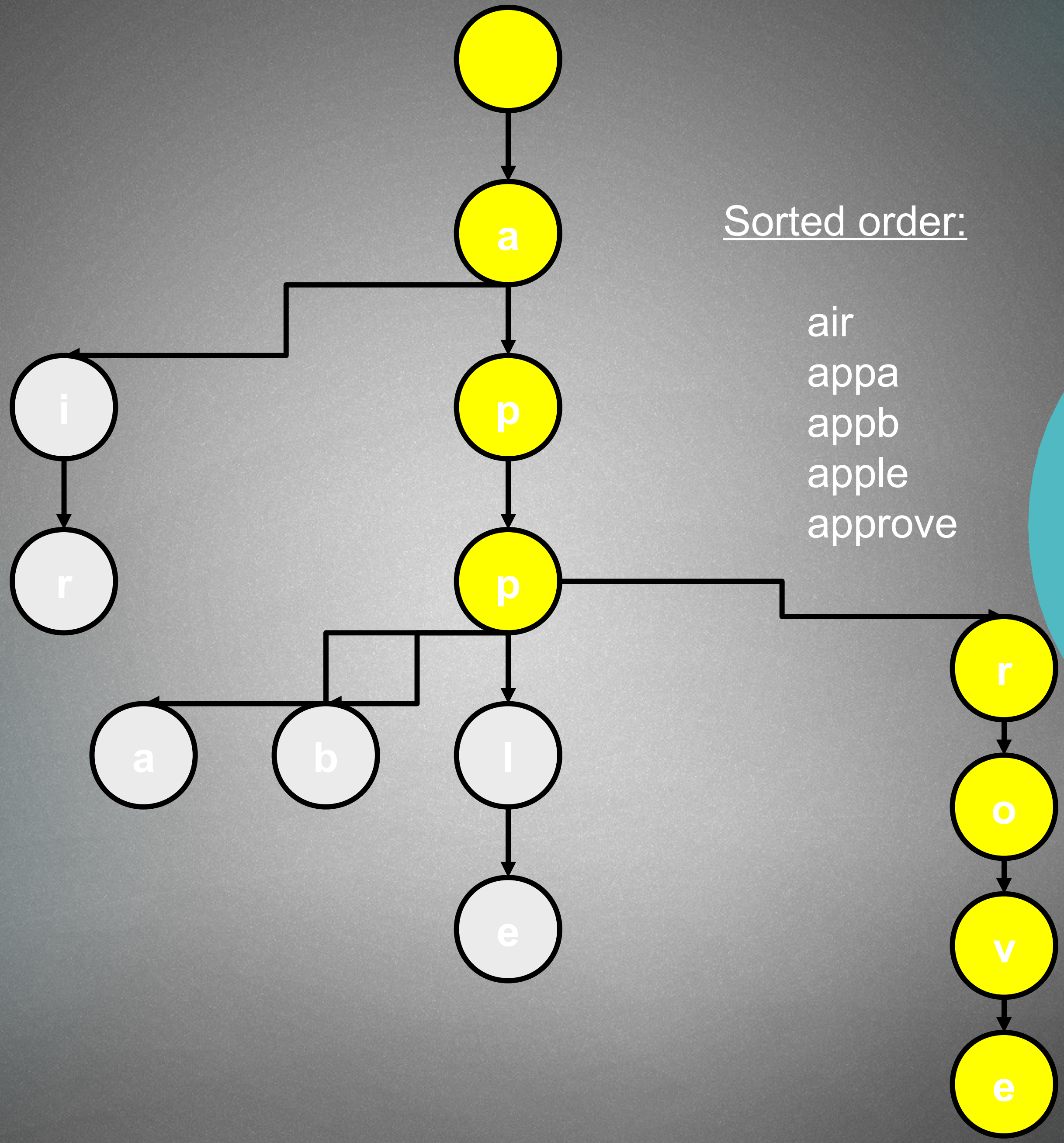
Sorted order:

air
appa
appb
apple
appro



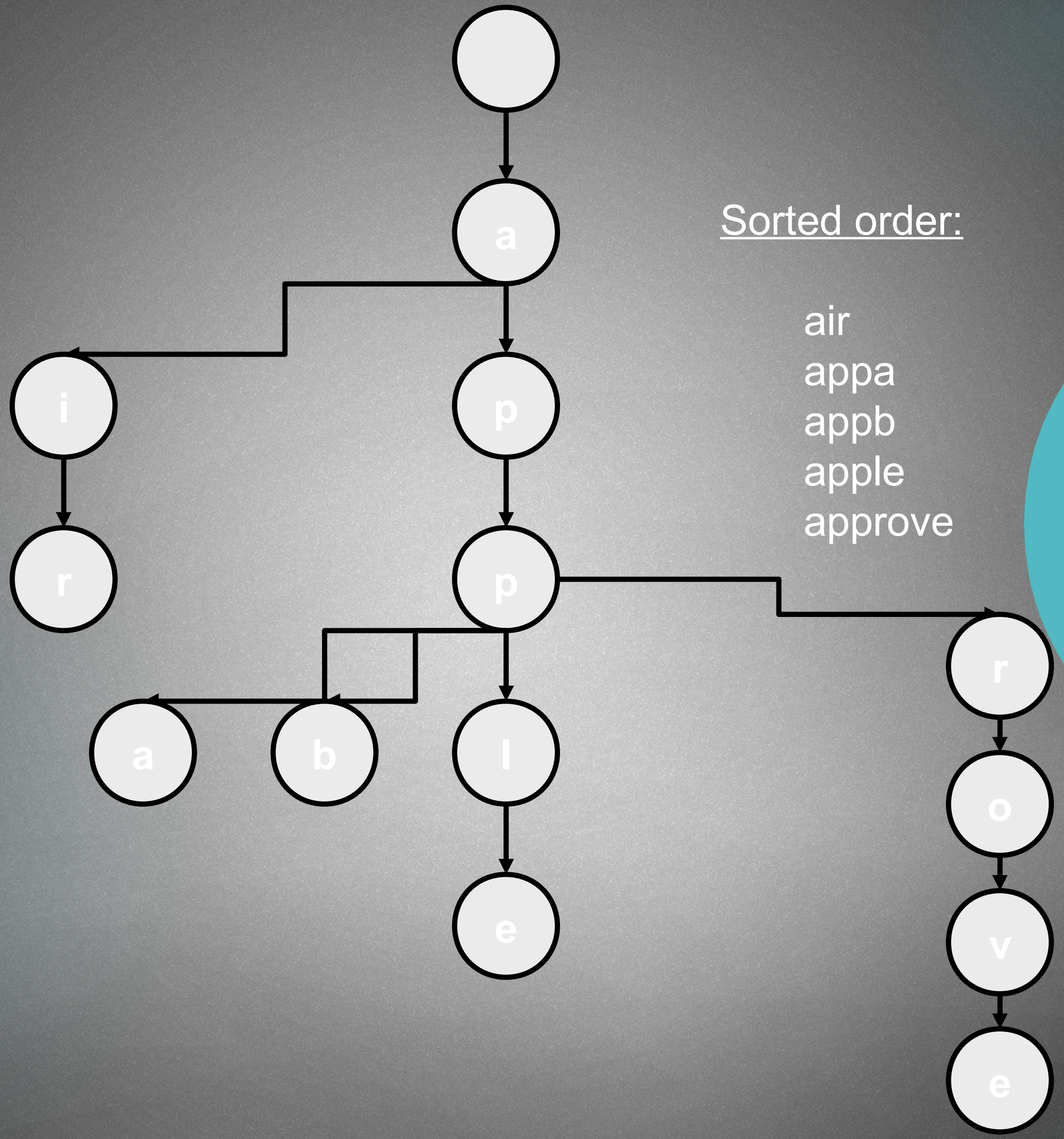
Sorted order:

air
appa
appb
apple
approv



Sorted order:

air
appa
appb
apple
approve



Sorted order:

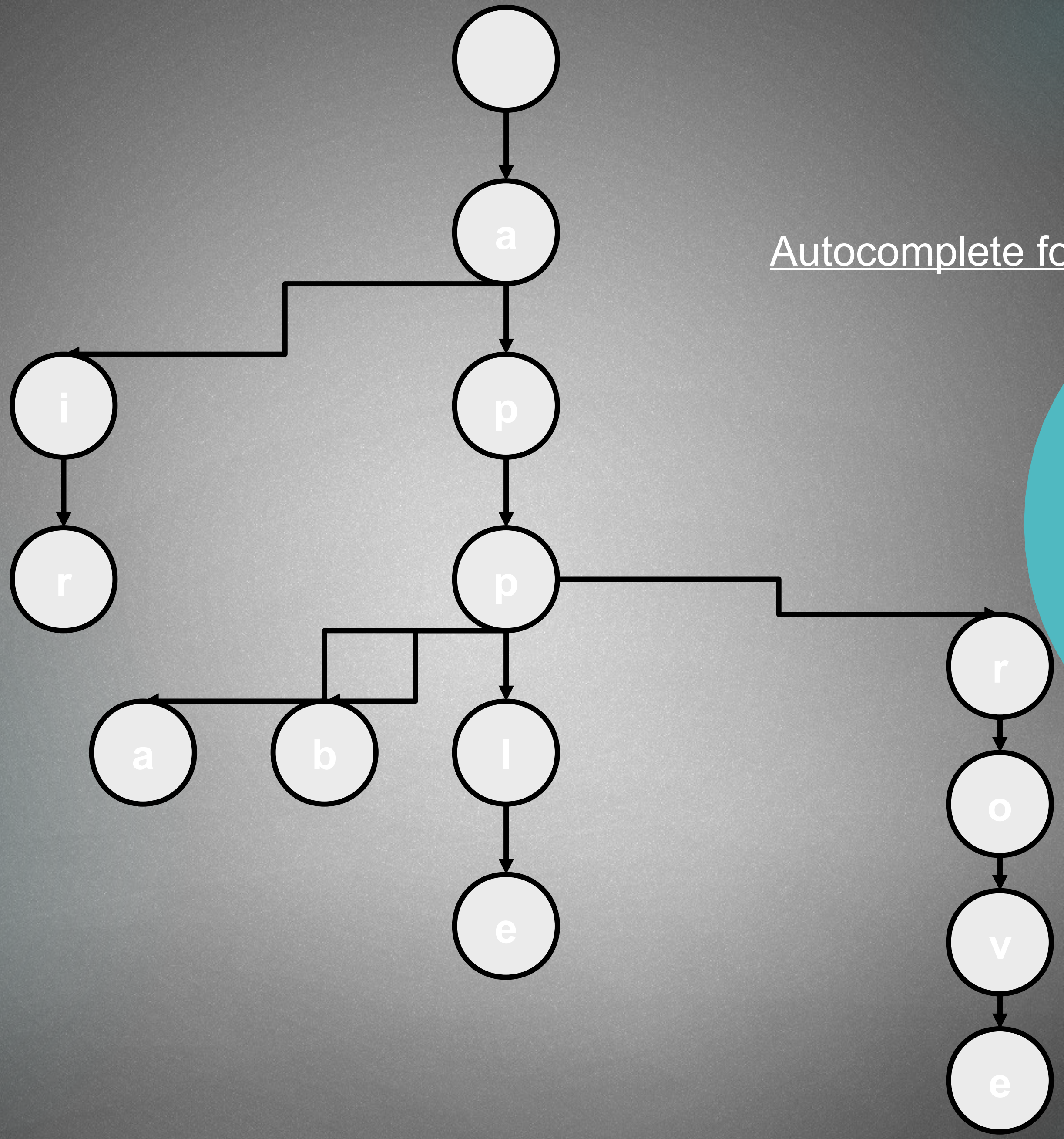
air
appa
appb
apple
approve

Autocomplete

It is the same as we have seen for the sorting

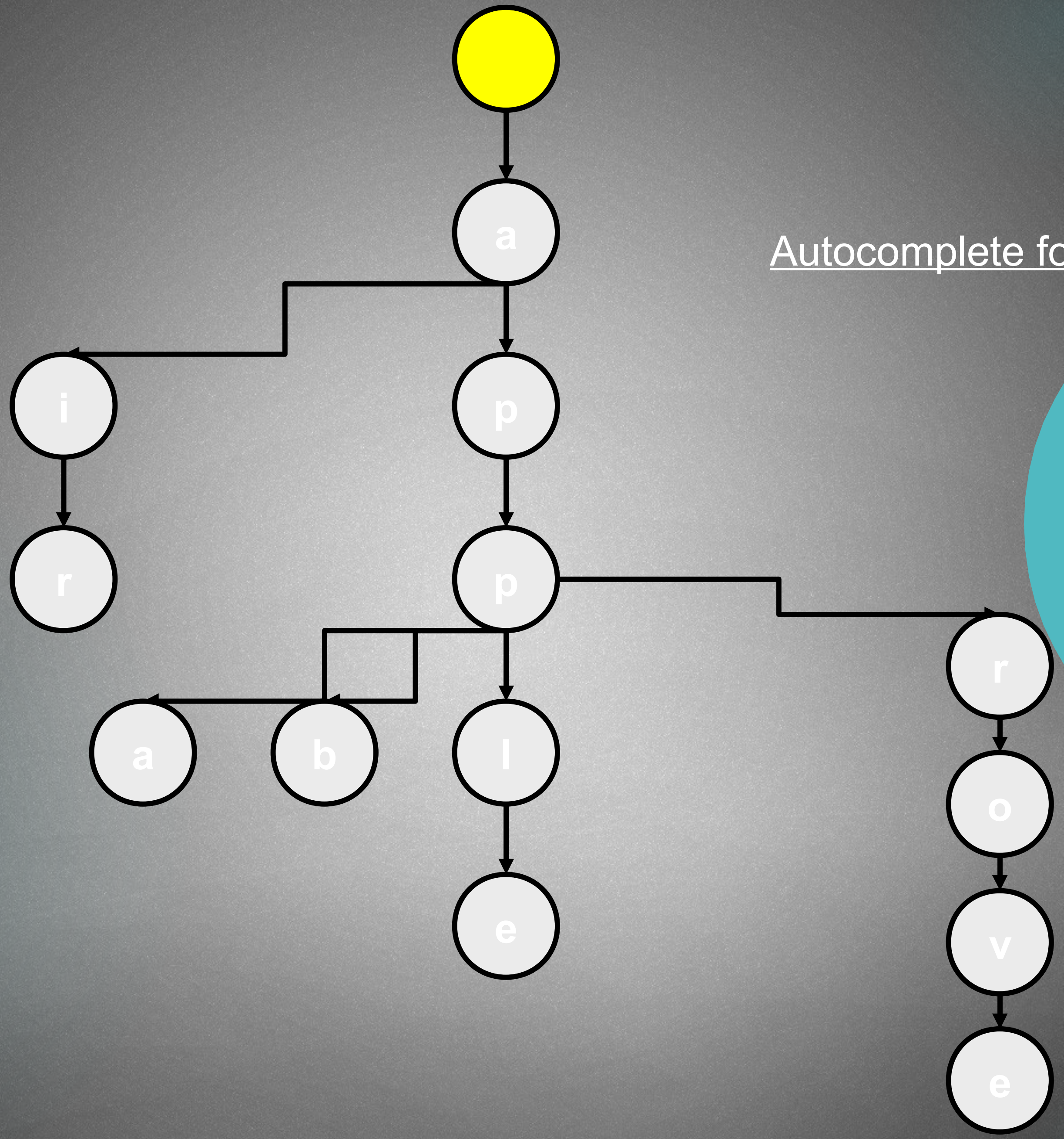
BUT here we have to look for the prefix first

+ make a depth-first traversal starting with that node

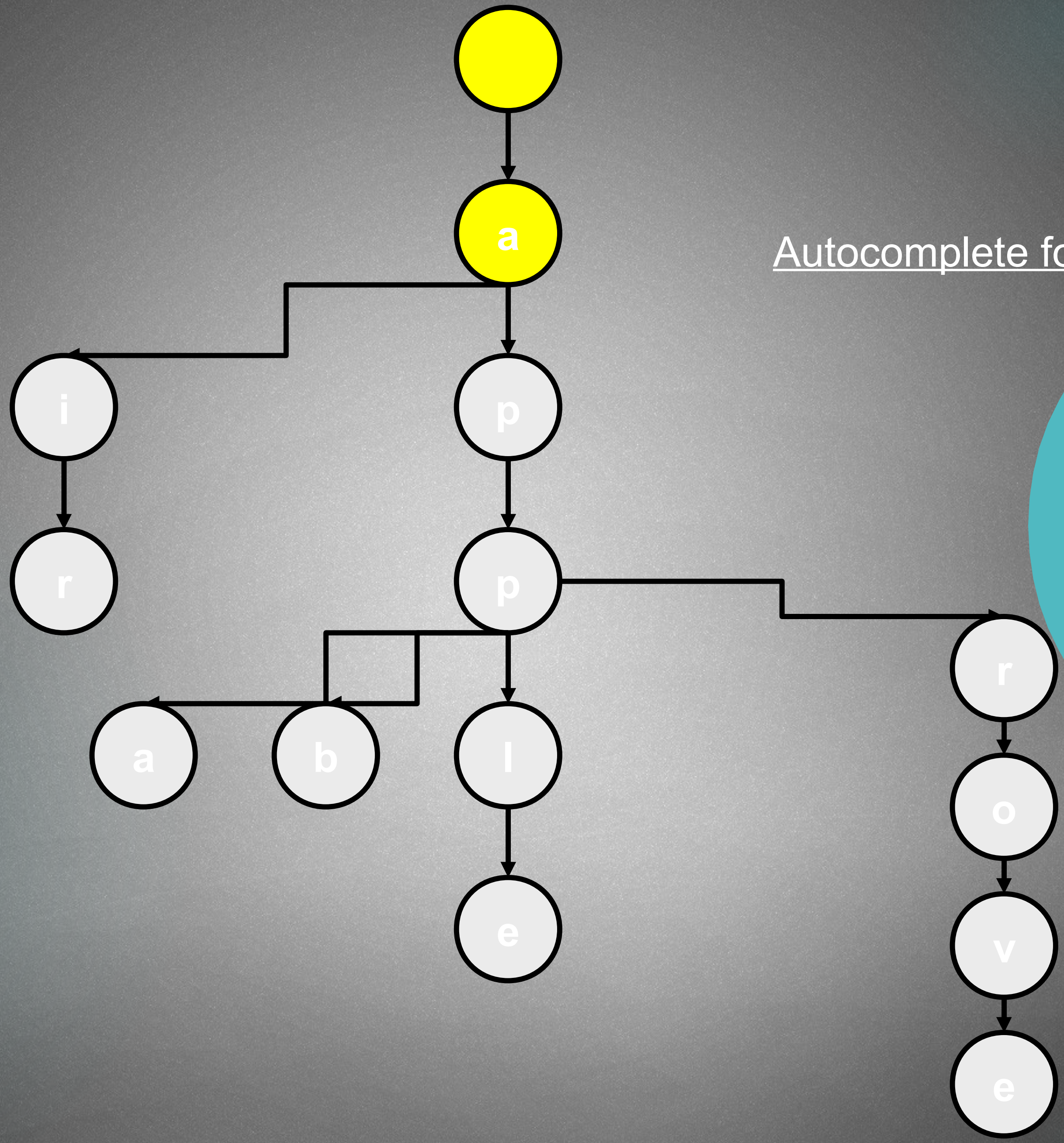


Autocomplete for: app

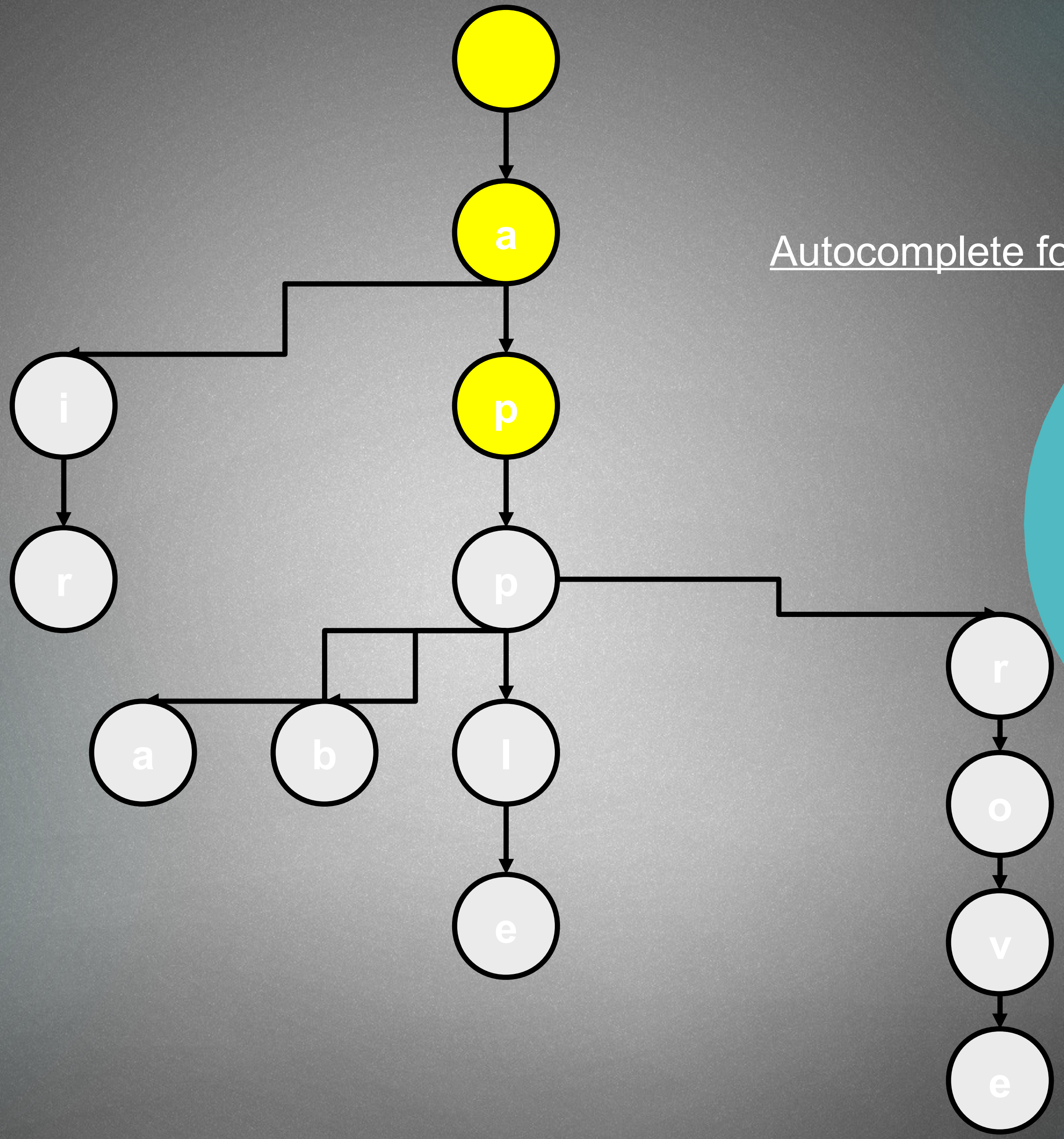
Autocomplete for: app



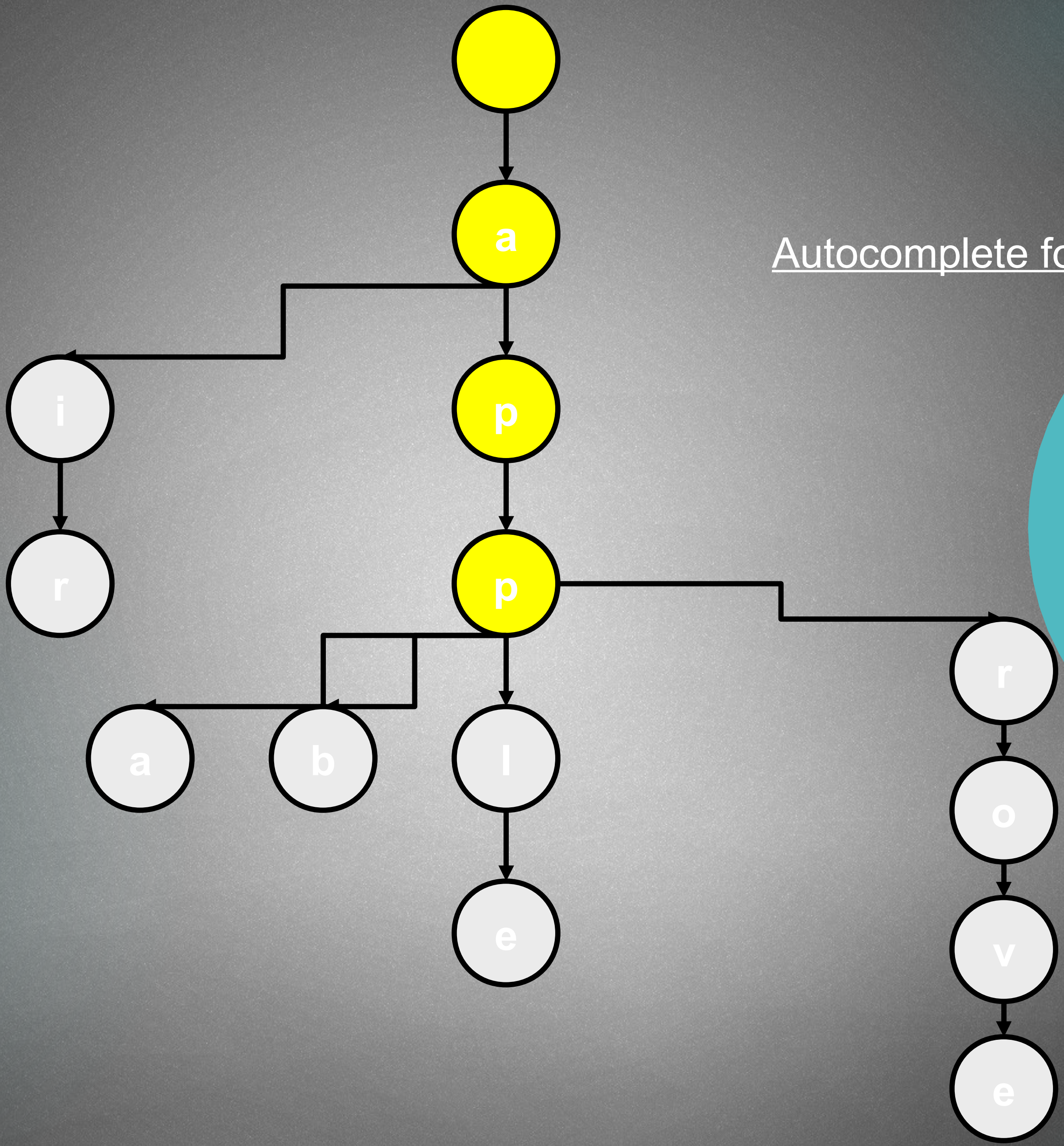
Autocomplete for: app



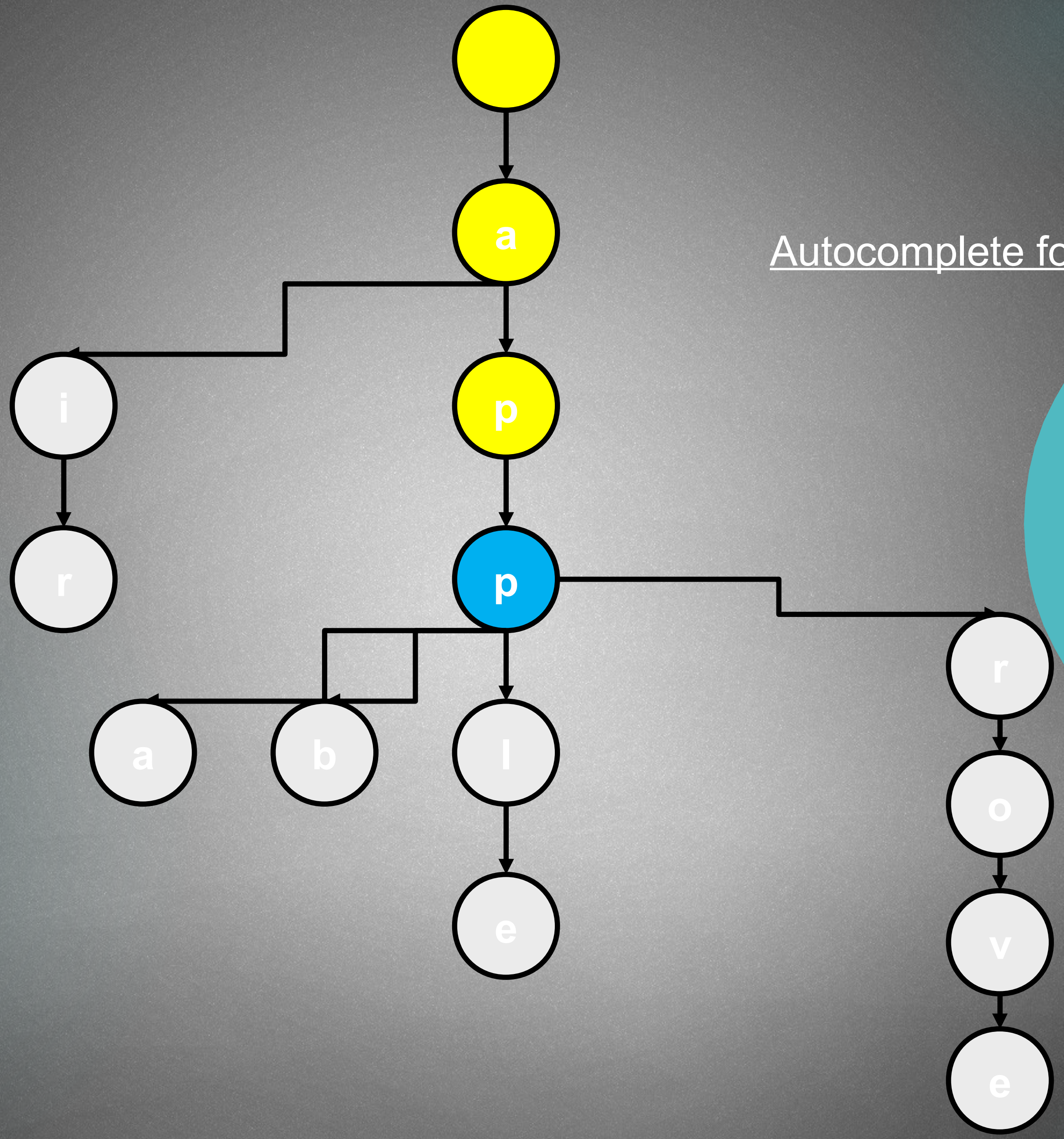
Autocomplete for: app

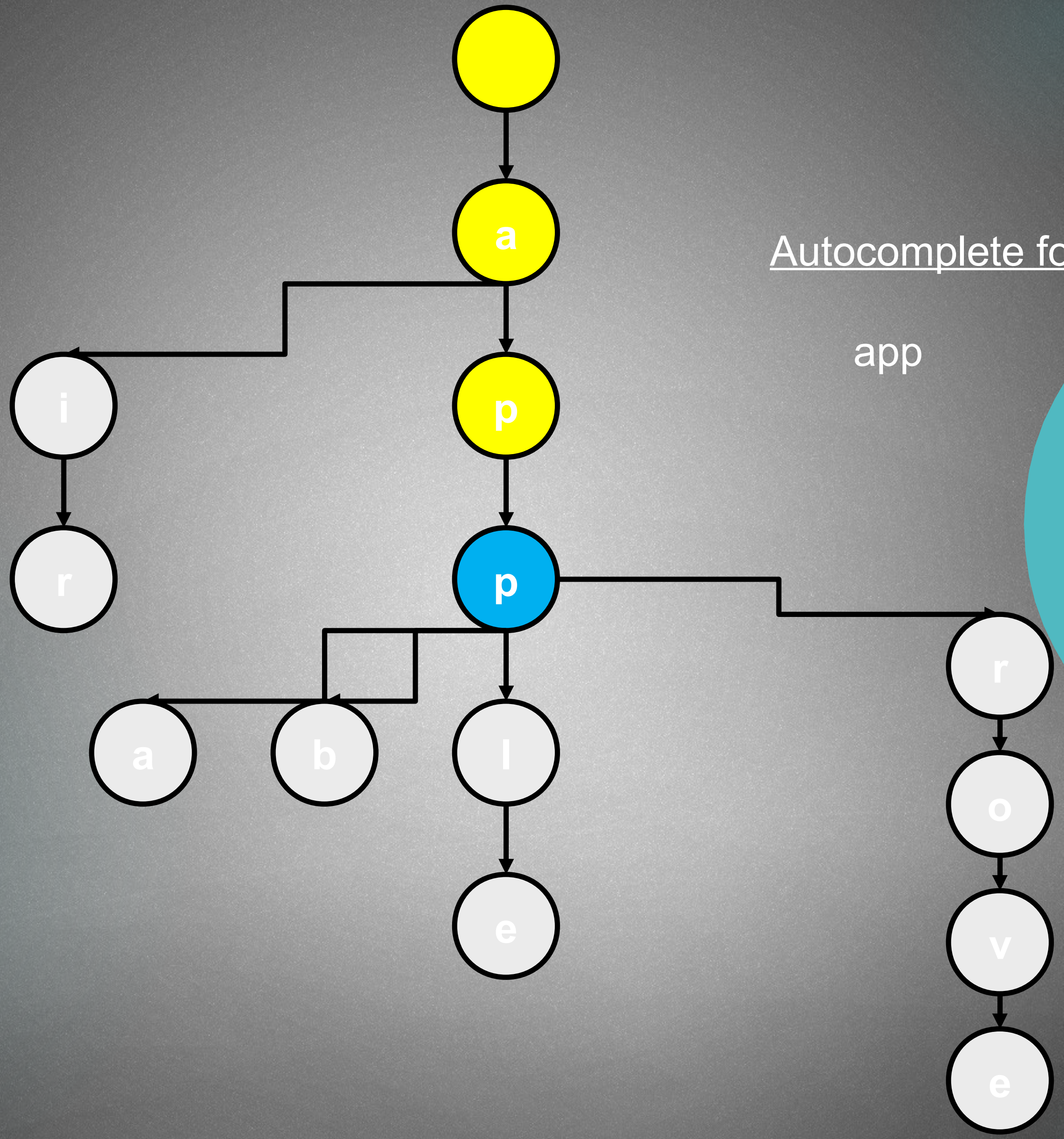


Autocomplete for: app



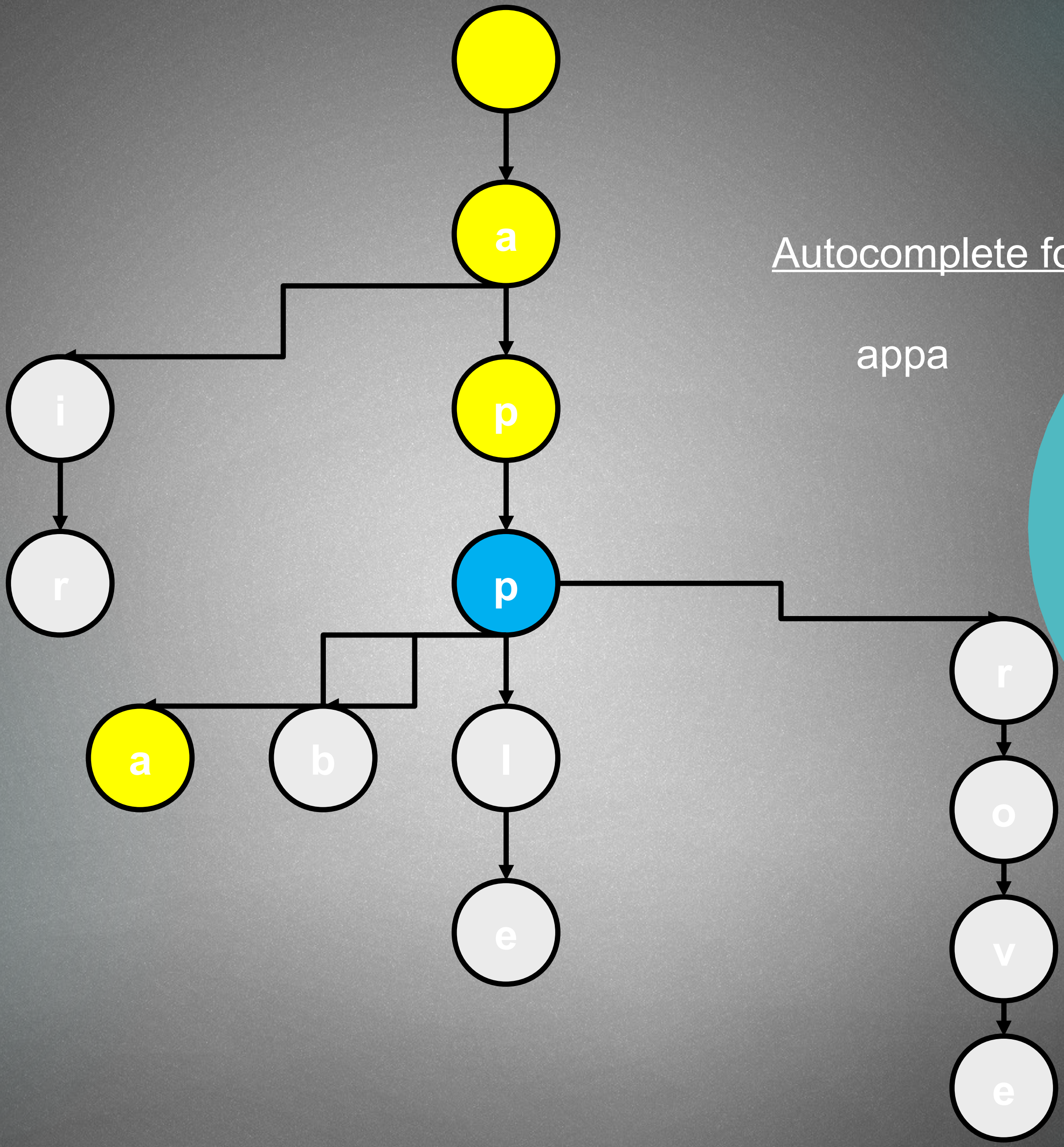
Autocomplete for: app





Autocomplete for: app

app

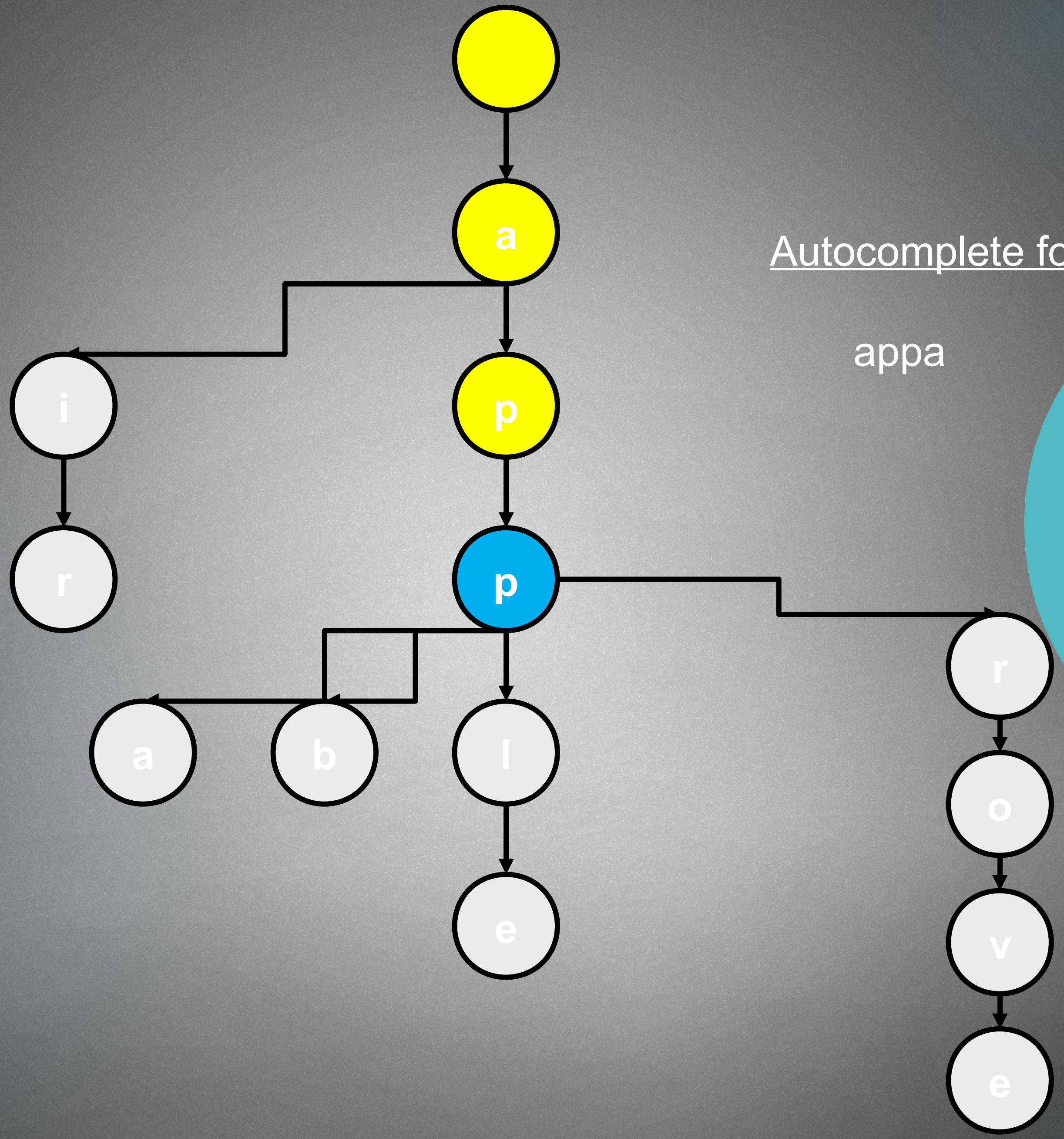


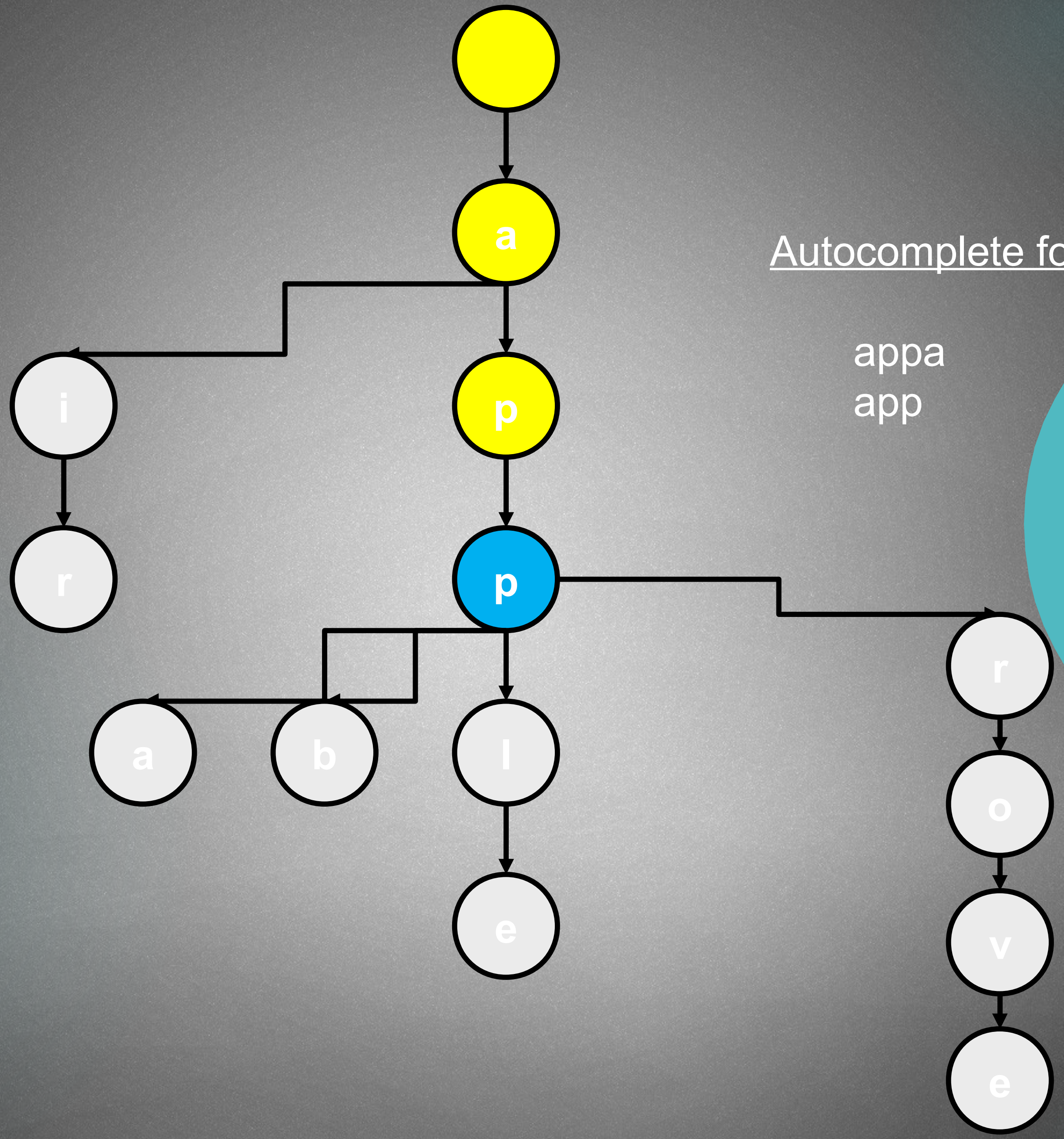
Autocomplete for: app

appa

Autocomplete for: app

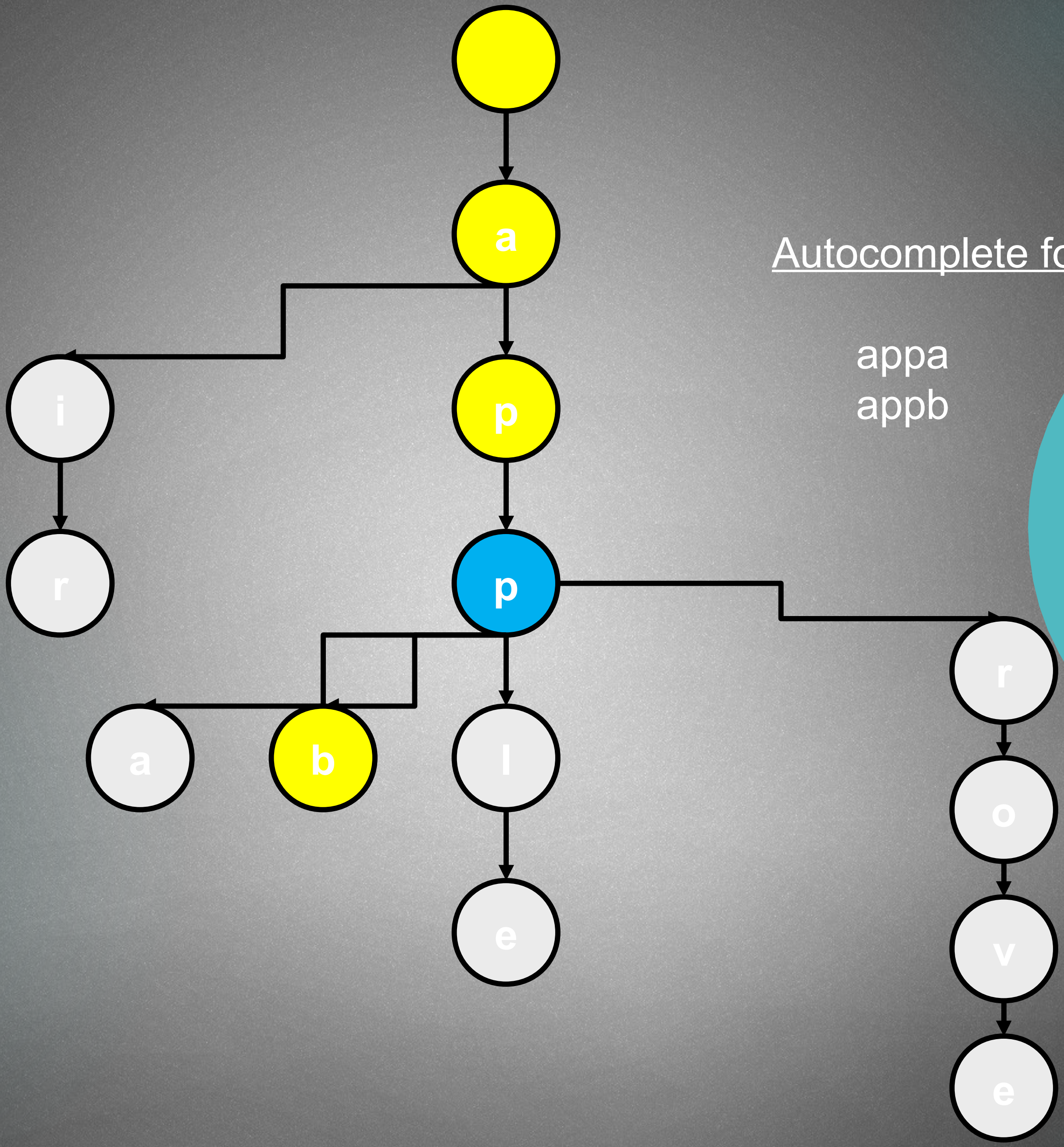
appa





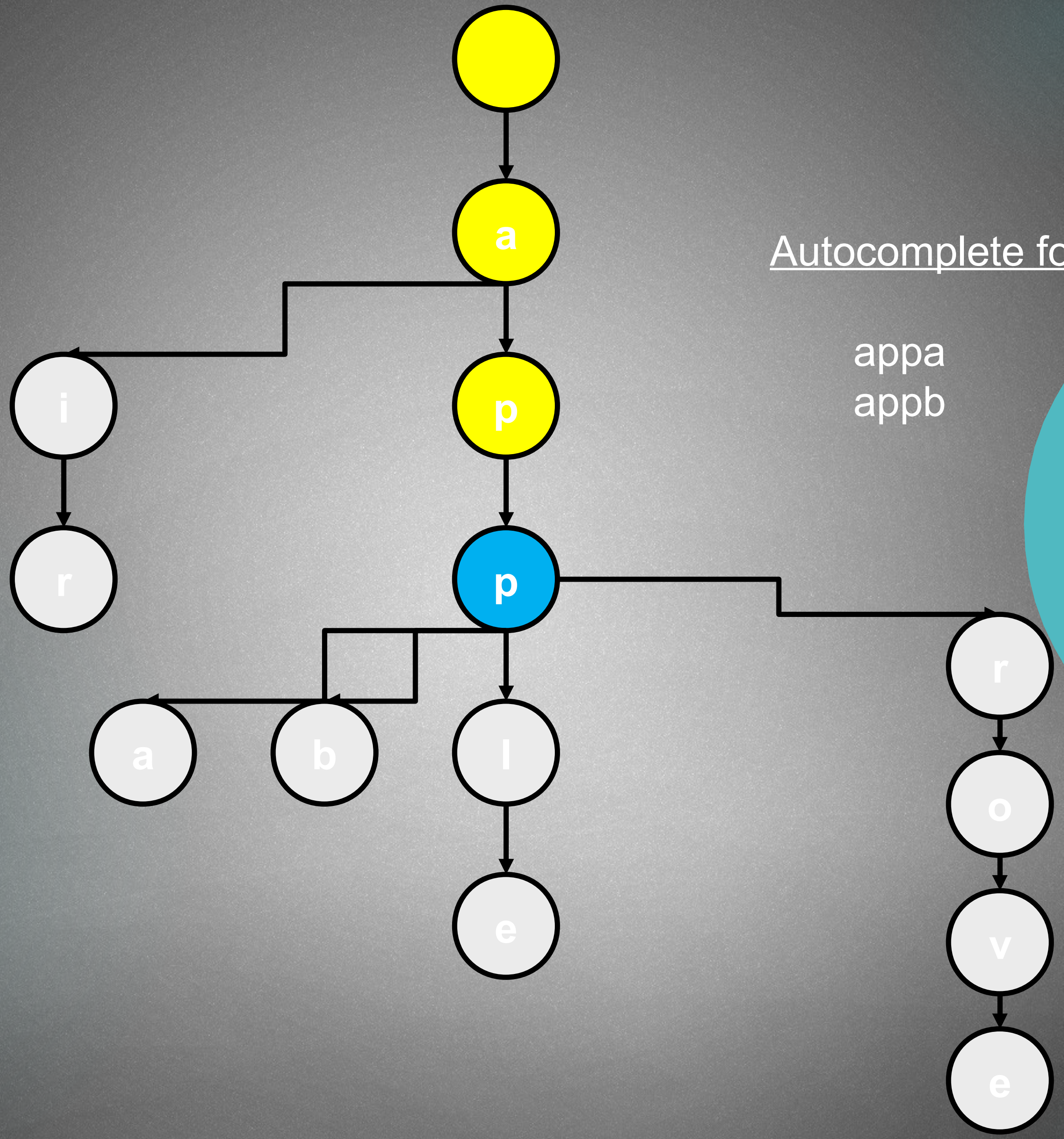
Autocomplete for: app

appa
app



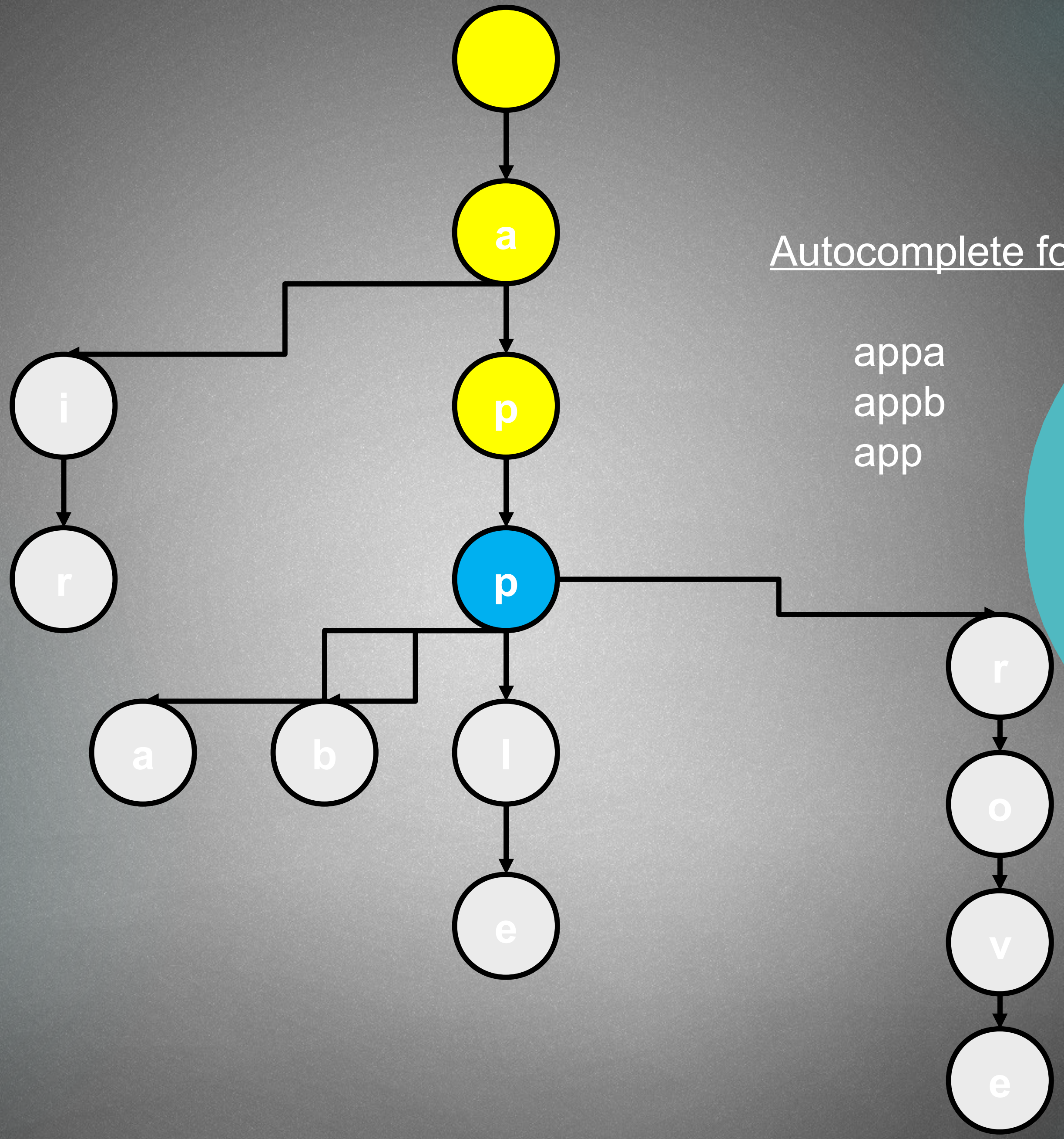
Autocomplete for: app

appa
appb



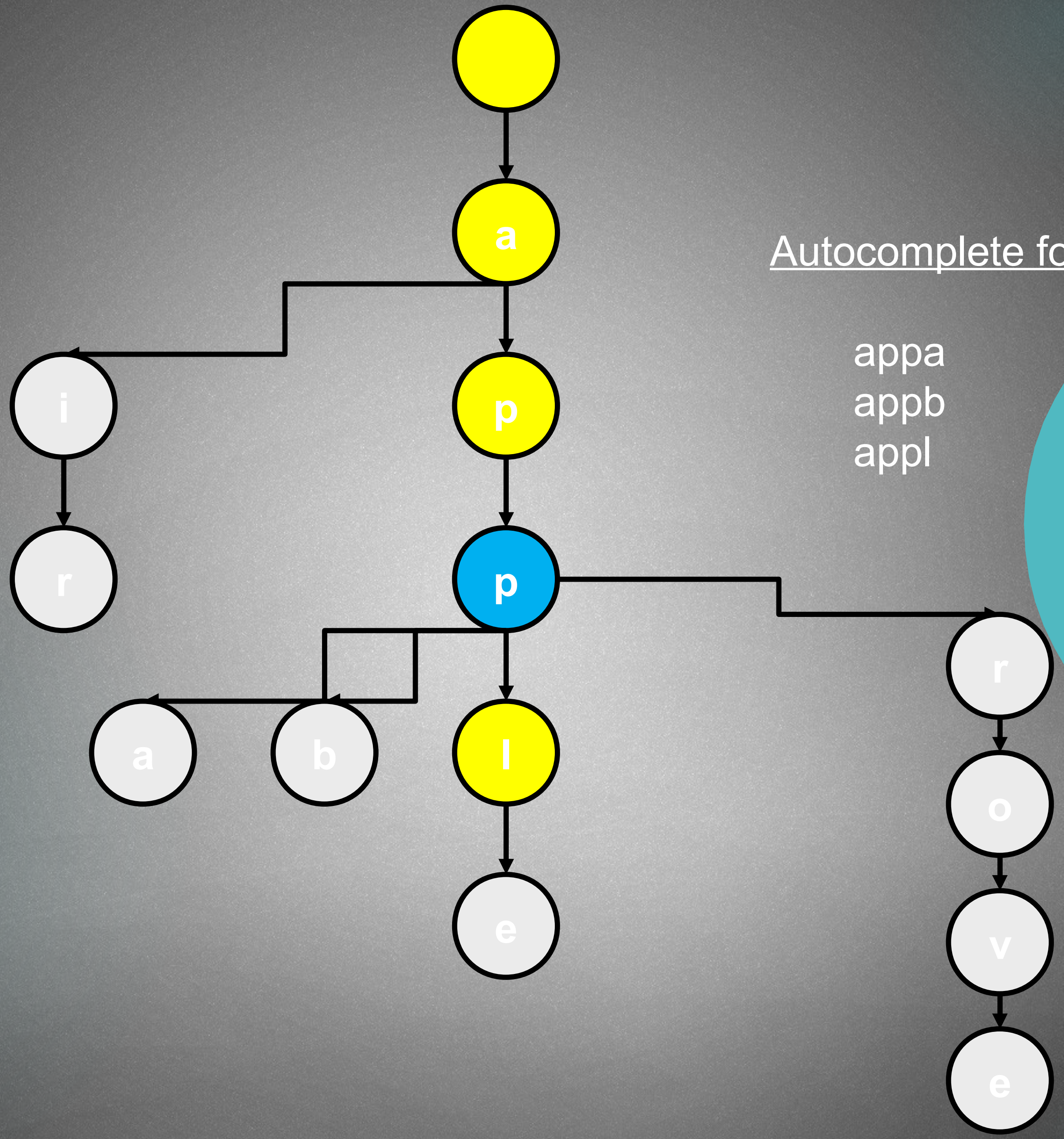
Autocomplete for: app

appa
appb



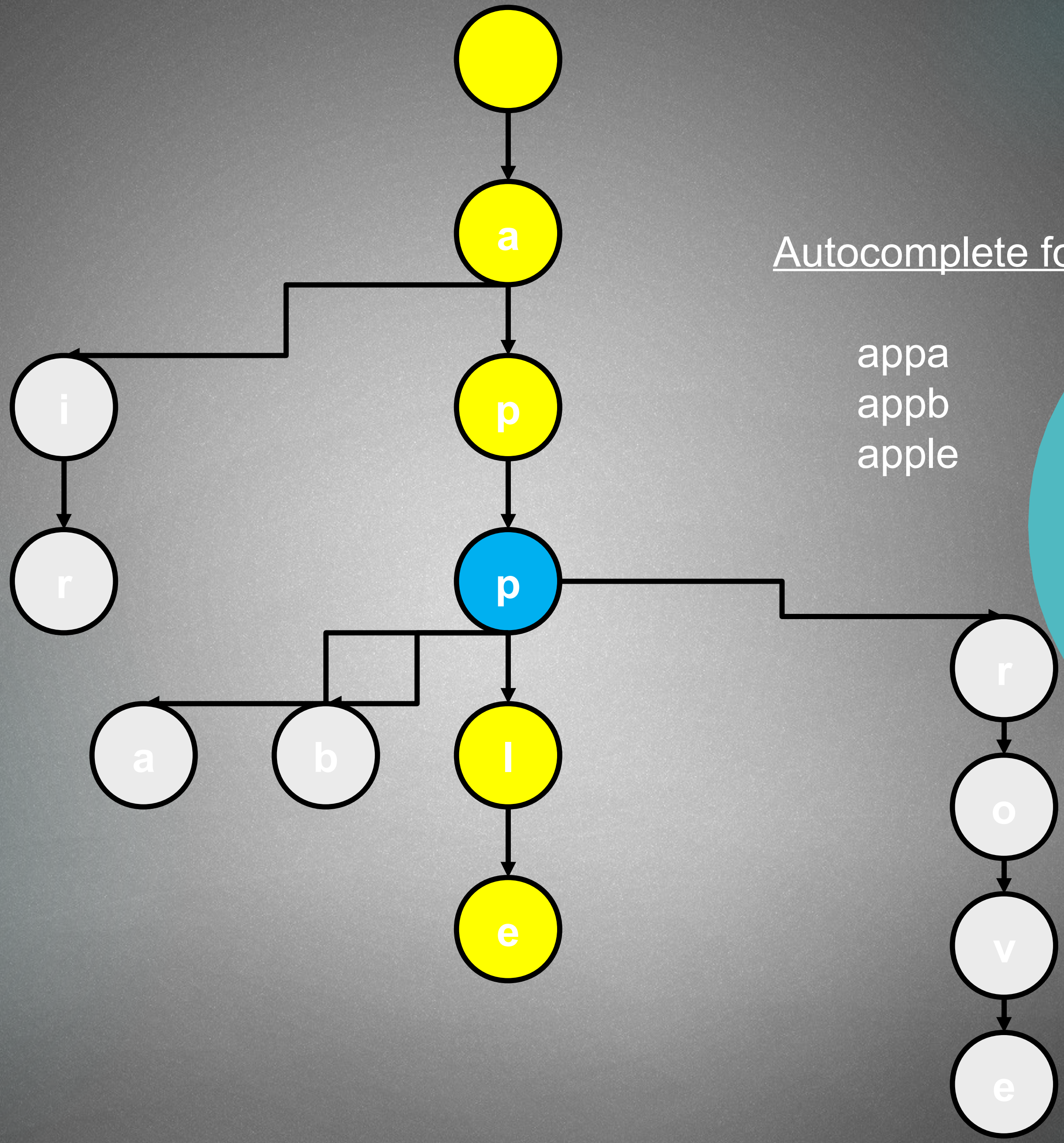
Autocomplete for: app

appa
appb
app



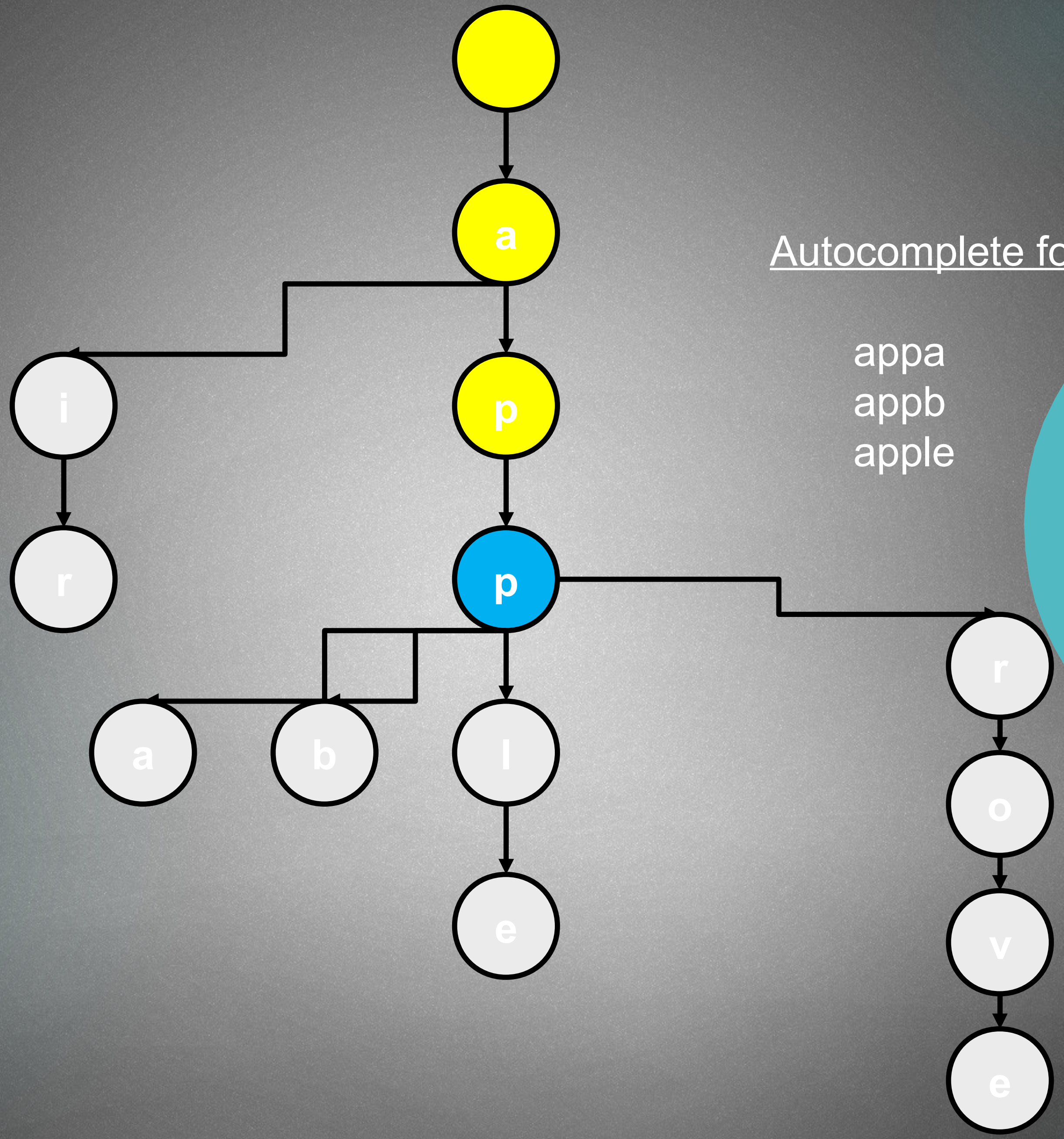
Autocomplete for: app

appa
appb
appl



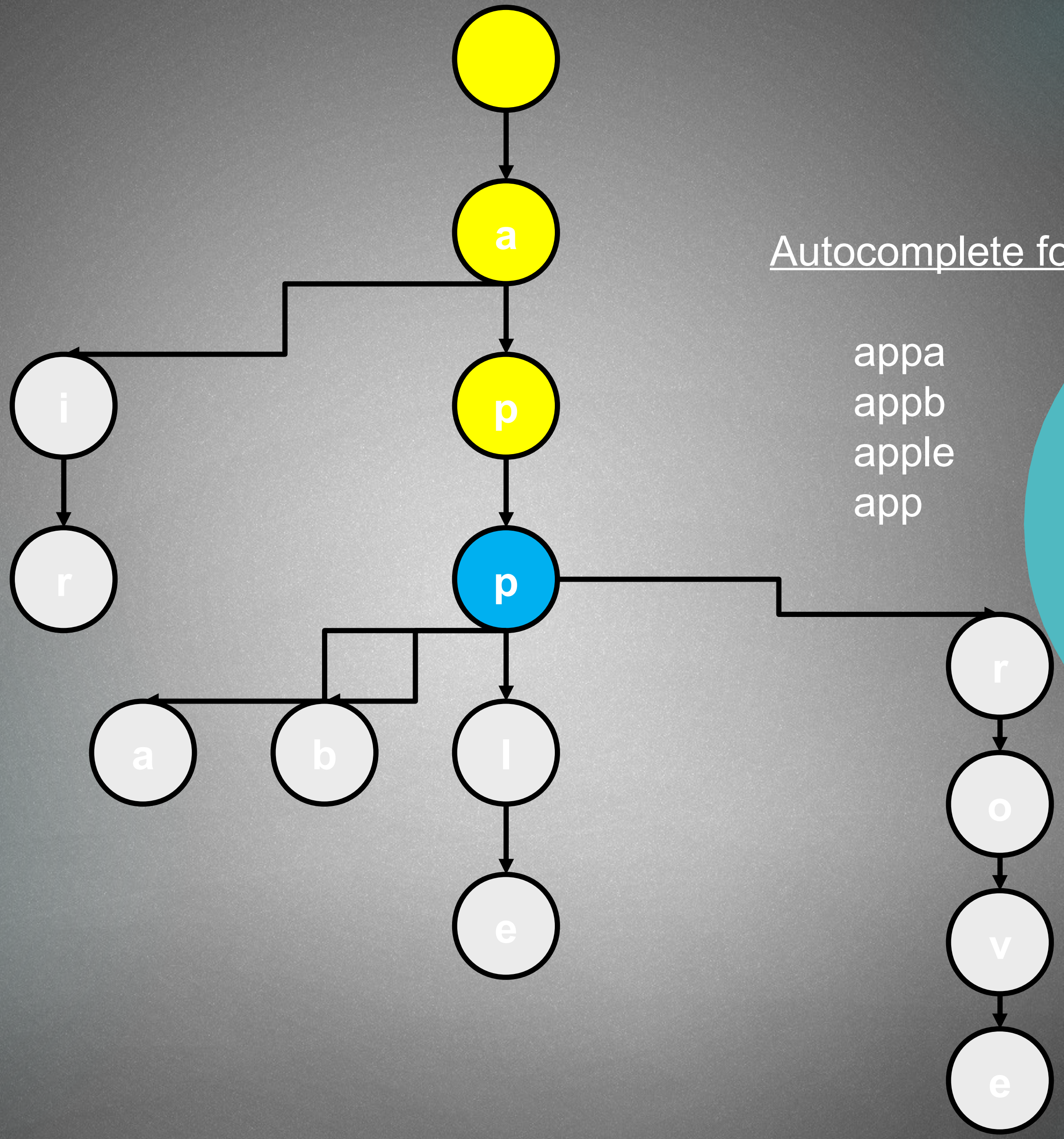
Autocomplete for: app

appa
appb
apple



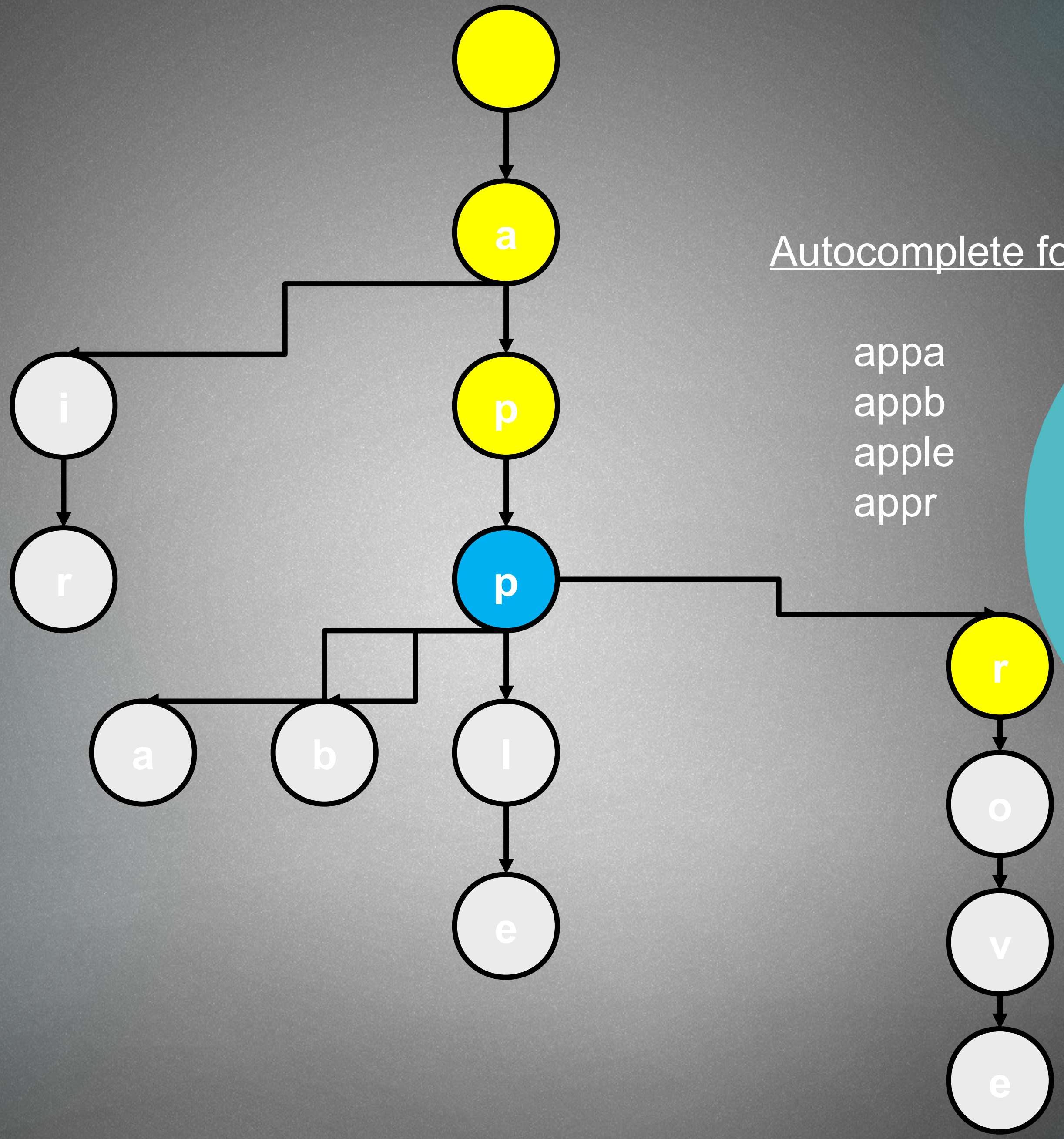
Autocomplete for: app

appa
appb
apple



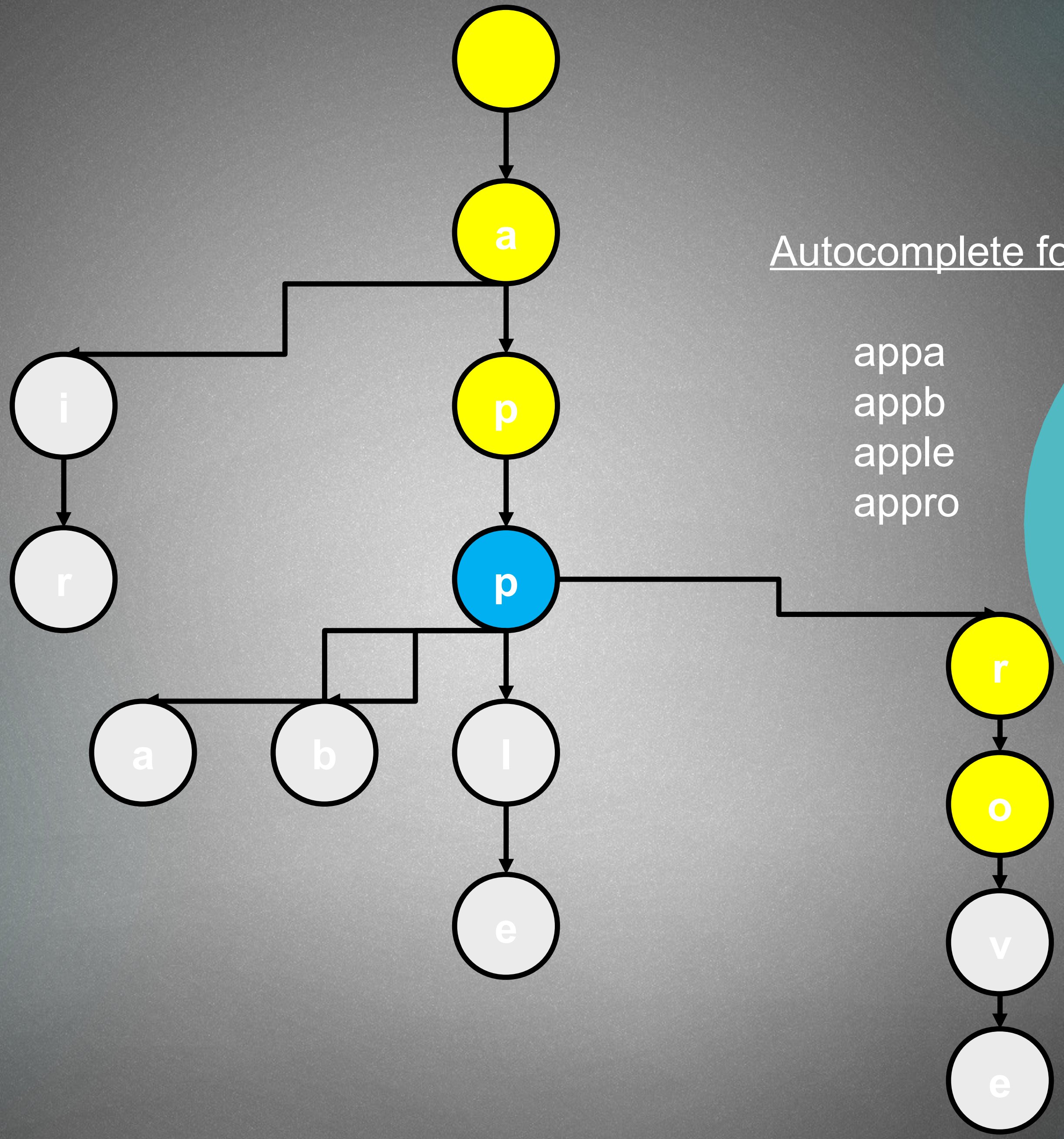
Autocomplete for: app

appa
appb
apple
app



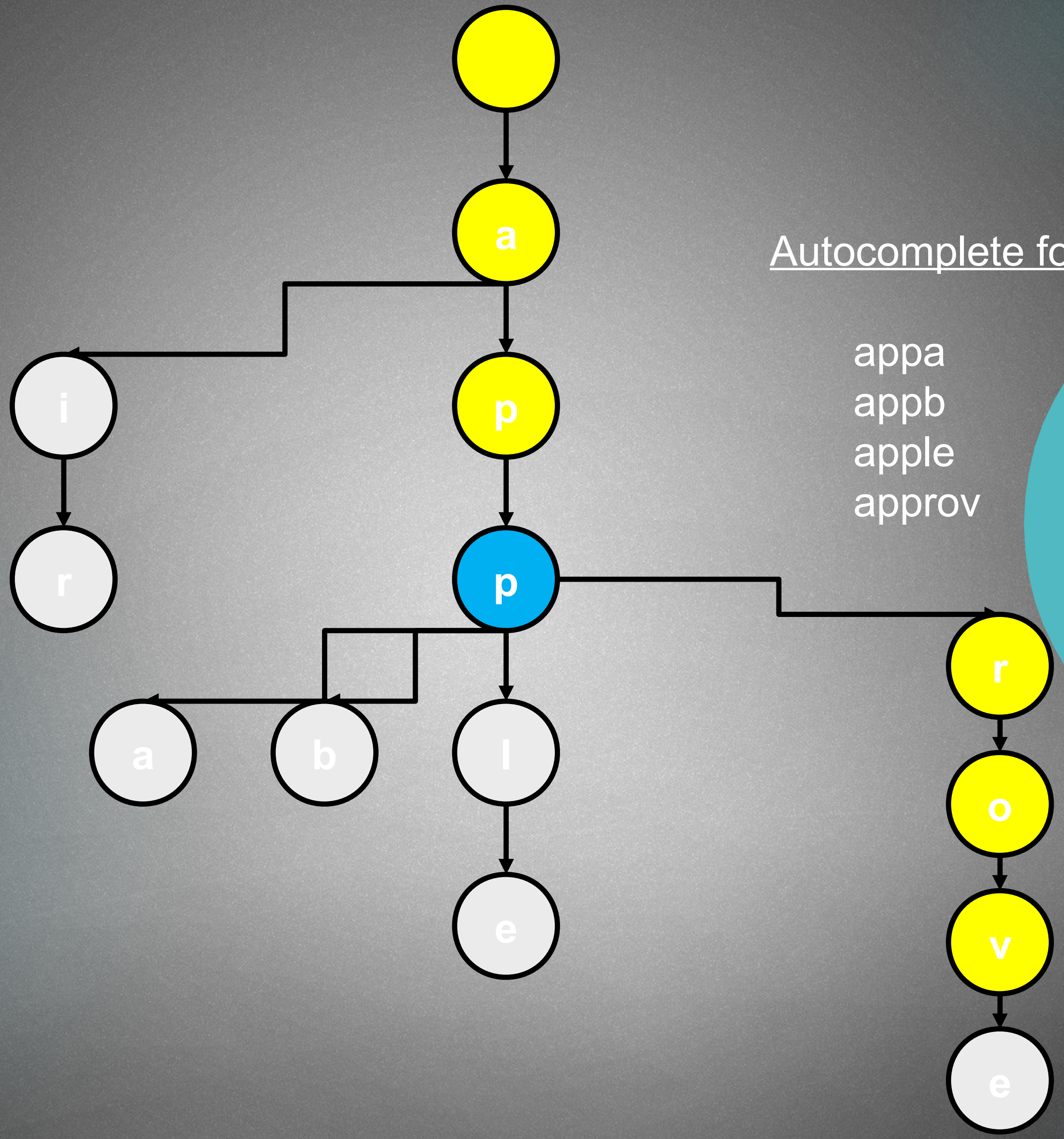
Autocomplete for: app

appa
appb
apple
appr



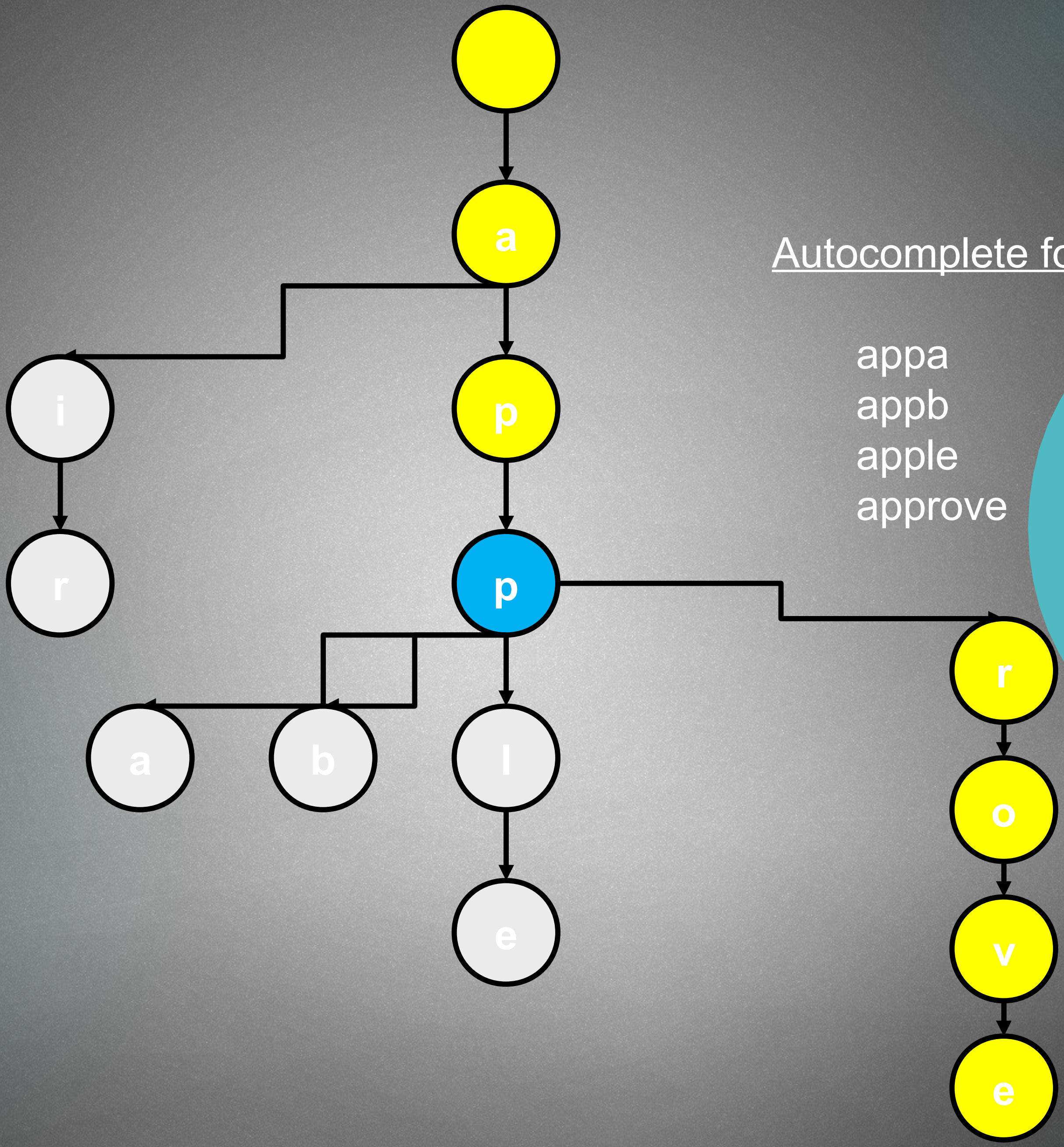
<u>Autocomplete for</u>: app

appa
appb
apple
appro



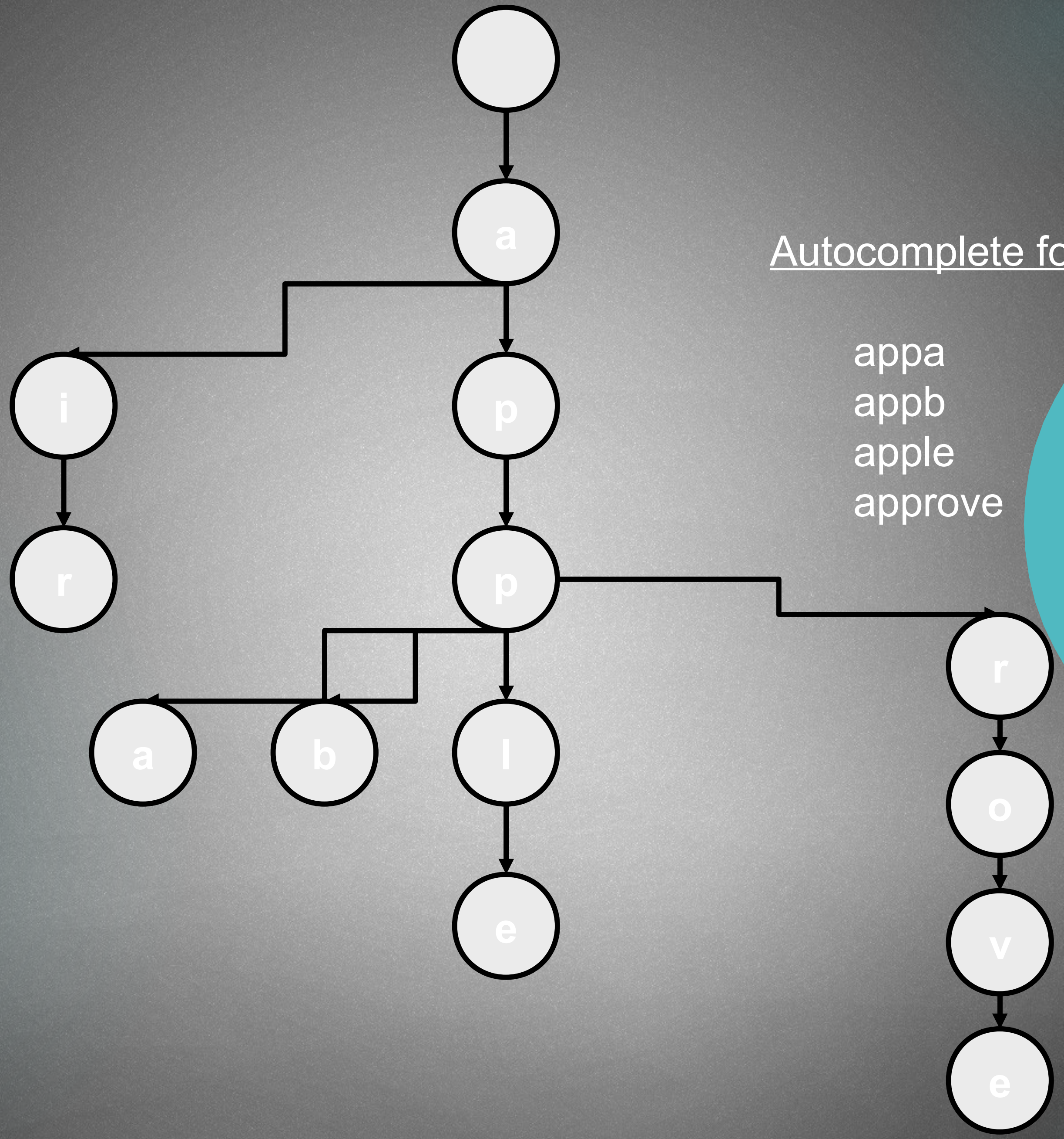
Autocomplete for: app

appa
appb
apple
approv



Autocomplete for: app

appa
appb
apple
approve

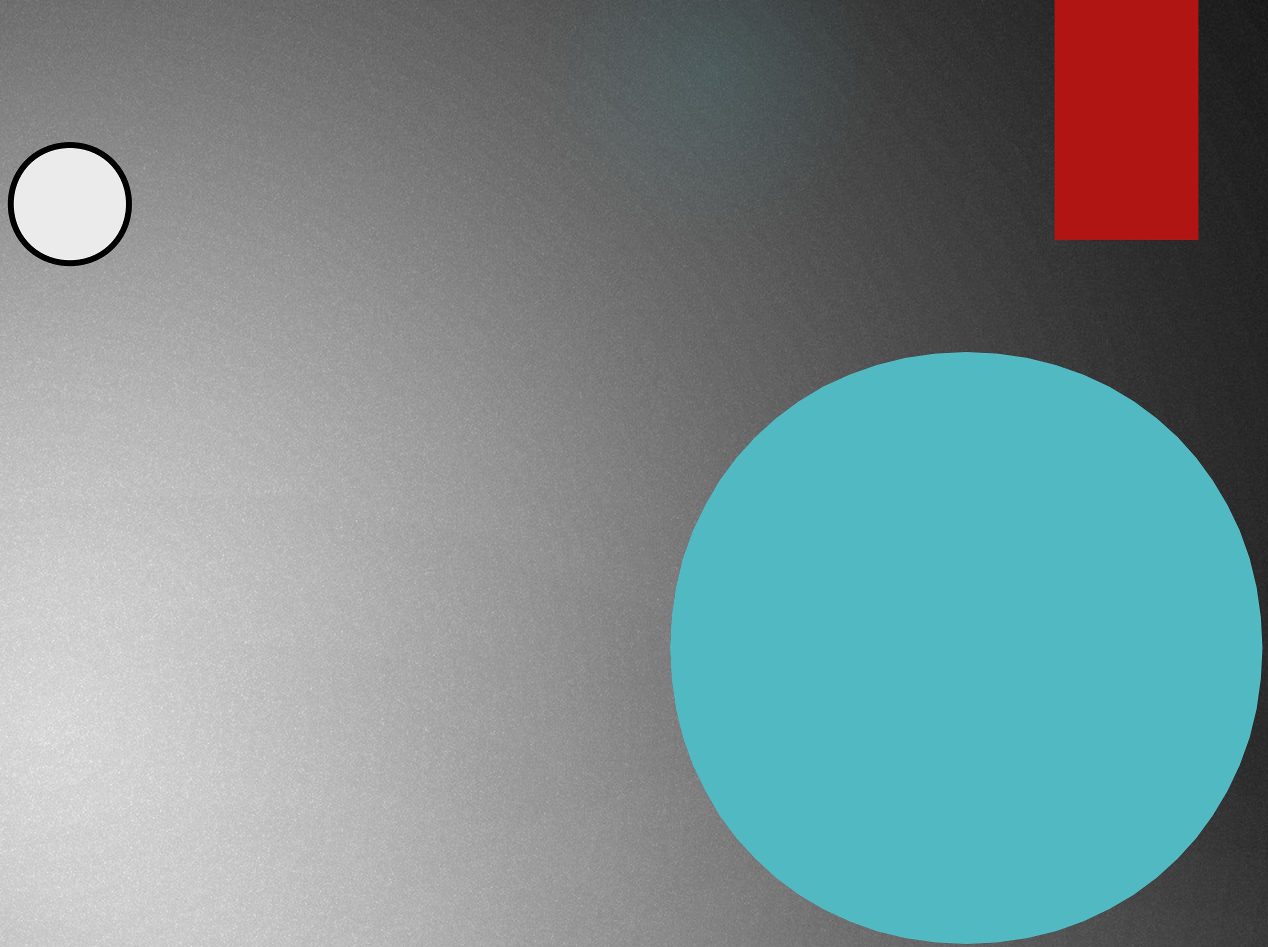


<u>Autocomplete for</u>: app

appa
appb
apple
approve

Trie as a map

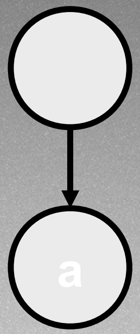
Insertion put(„apple”, 1)



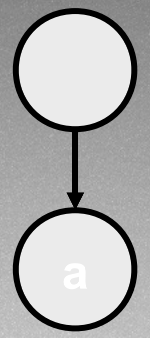
Insertion

put(„**a**pple", 1)

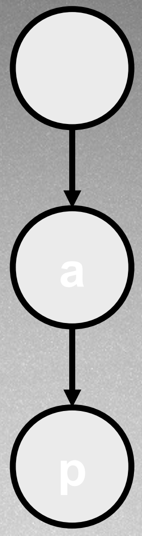
Insertion put(„**a**pple", 1)



Insertion put(„**ap**ple”, 1)

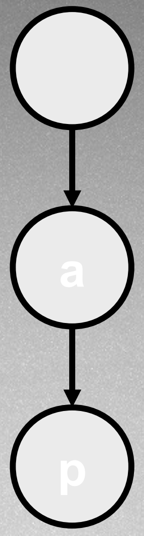


Insertion put(„**ap**ple", 1)



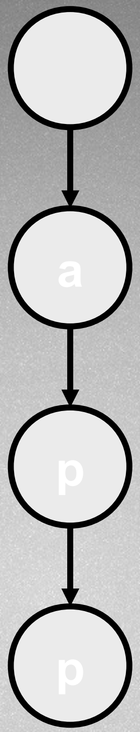
Insertion

put(„**app**le”, 1)

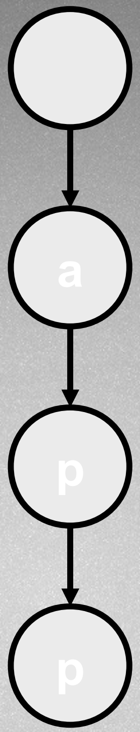


Insertion

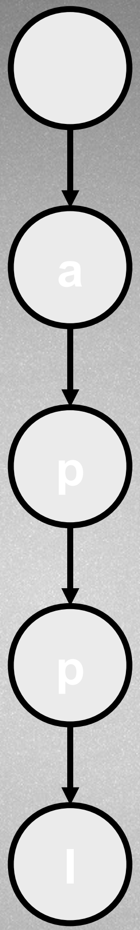
put(„**app**le”, 1)



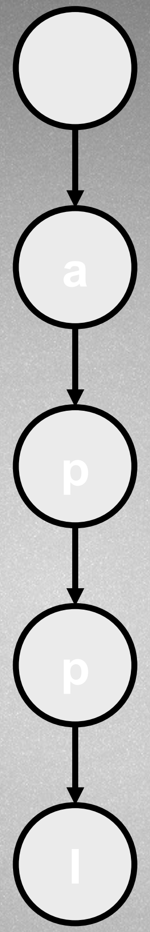
Insertion put(„**appl**e”, 1)



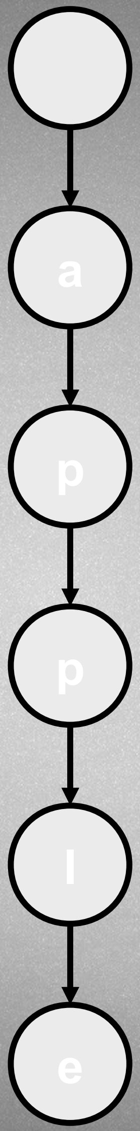
Insertion put(„**apple**", 1)



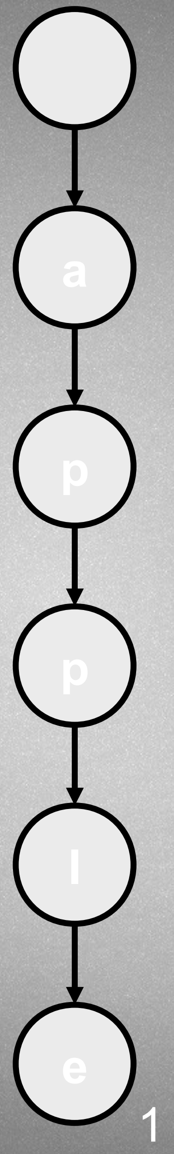
Insertion put(„**apple**", 1)



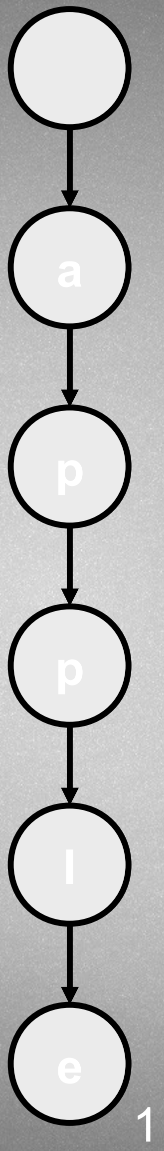
Insertion put(„**apple**", 1)



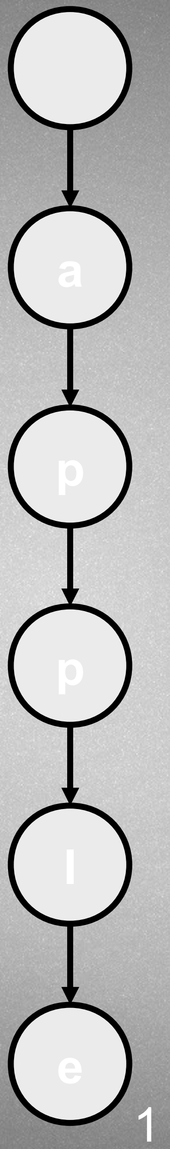
Insertion put(„**apple**", 1)



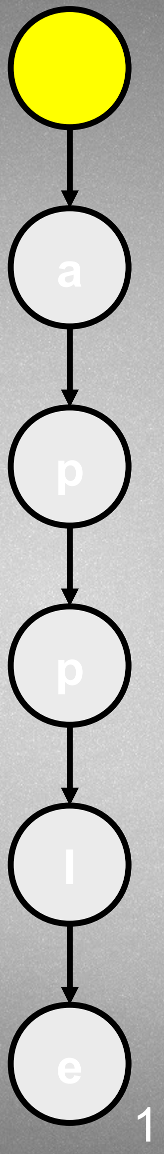
Insertion



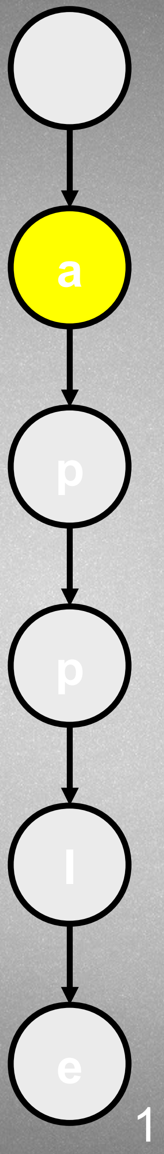
Insertion put(„air”, 2)



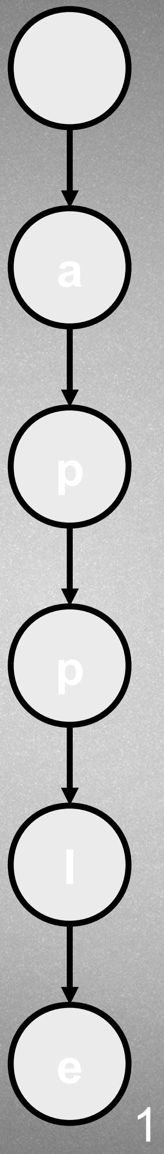
Insertion put(„air”, 2)



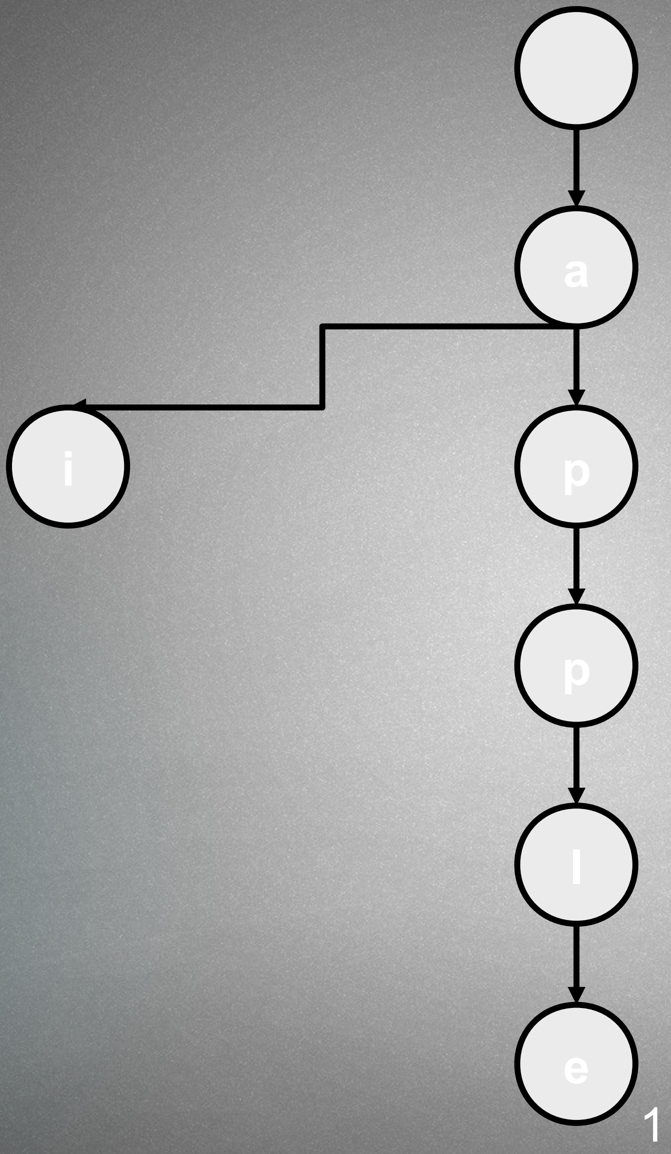
Insertion put(„air”, 2)



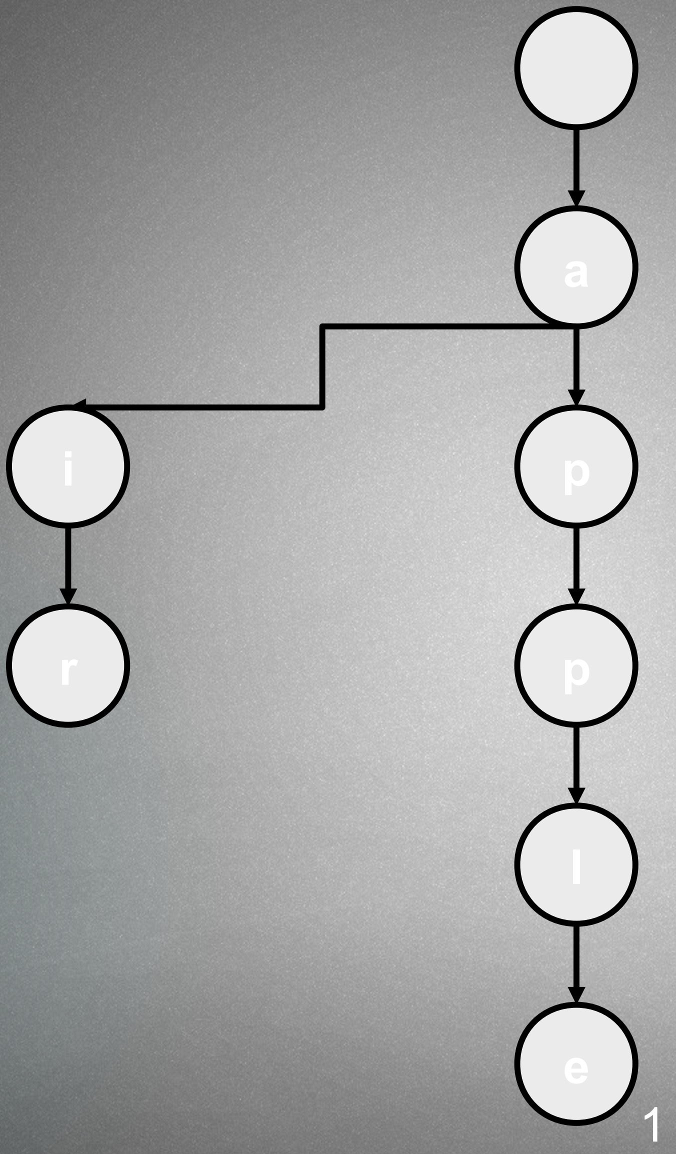
Insertion put(„air”, 2)



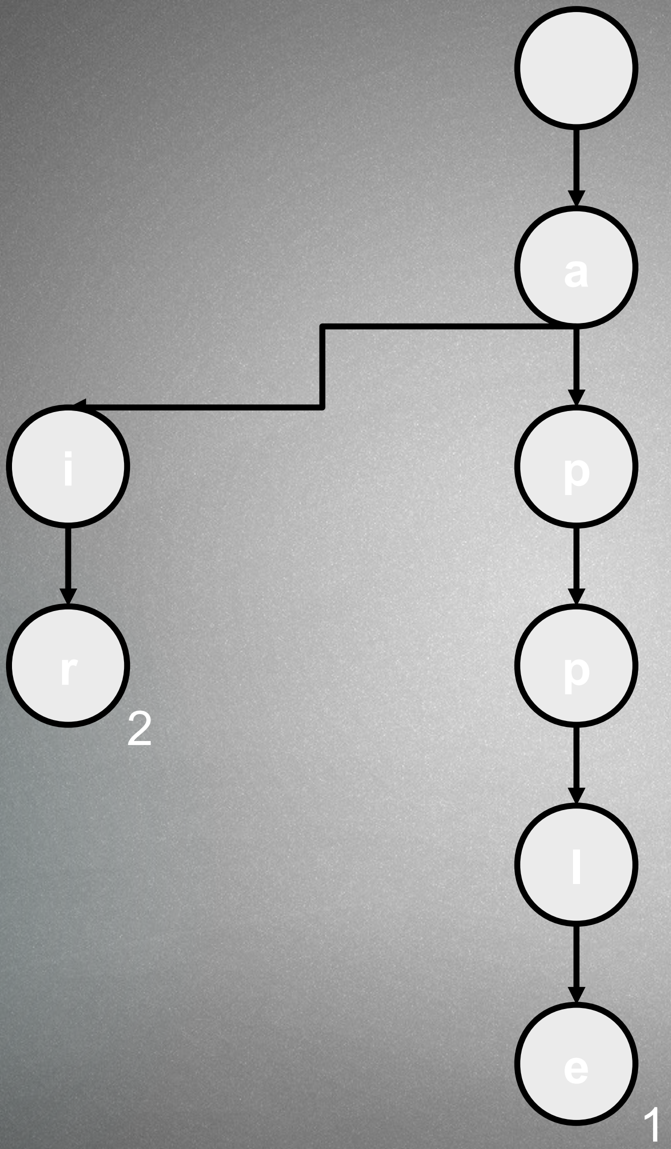
Insertion put(„**air**", 2)



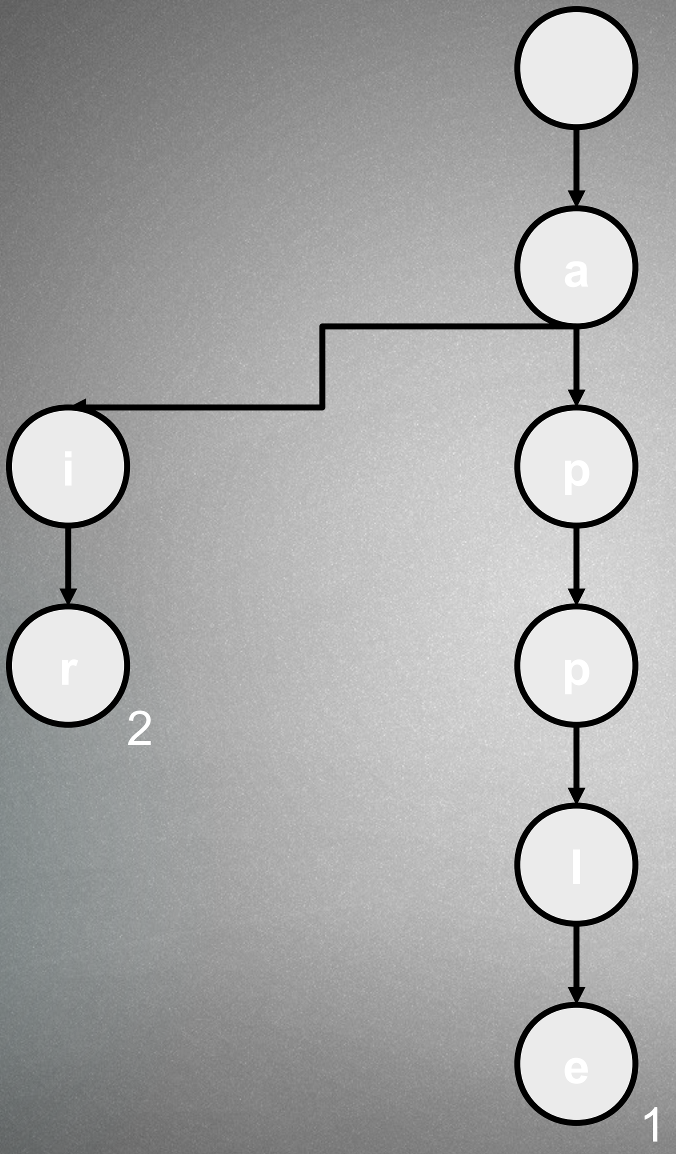
Insertion put(„**air**”, 2)



Insertion

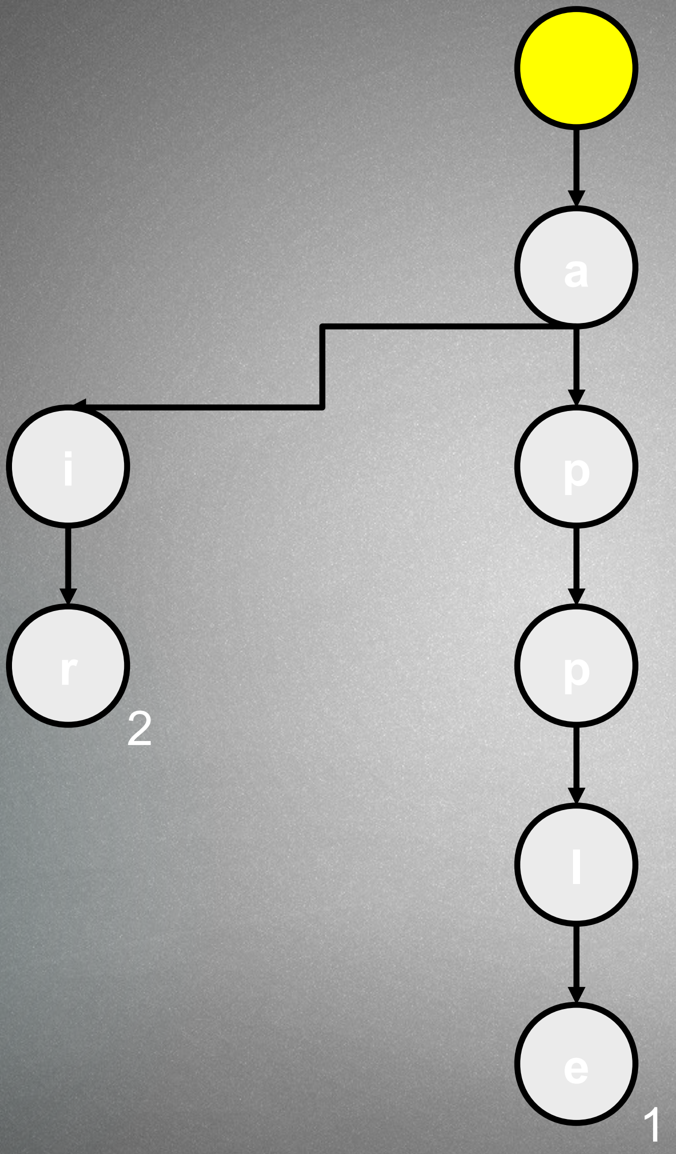


Insertion put(„approve", 3)

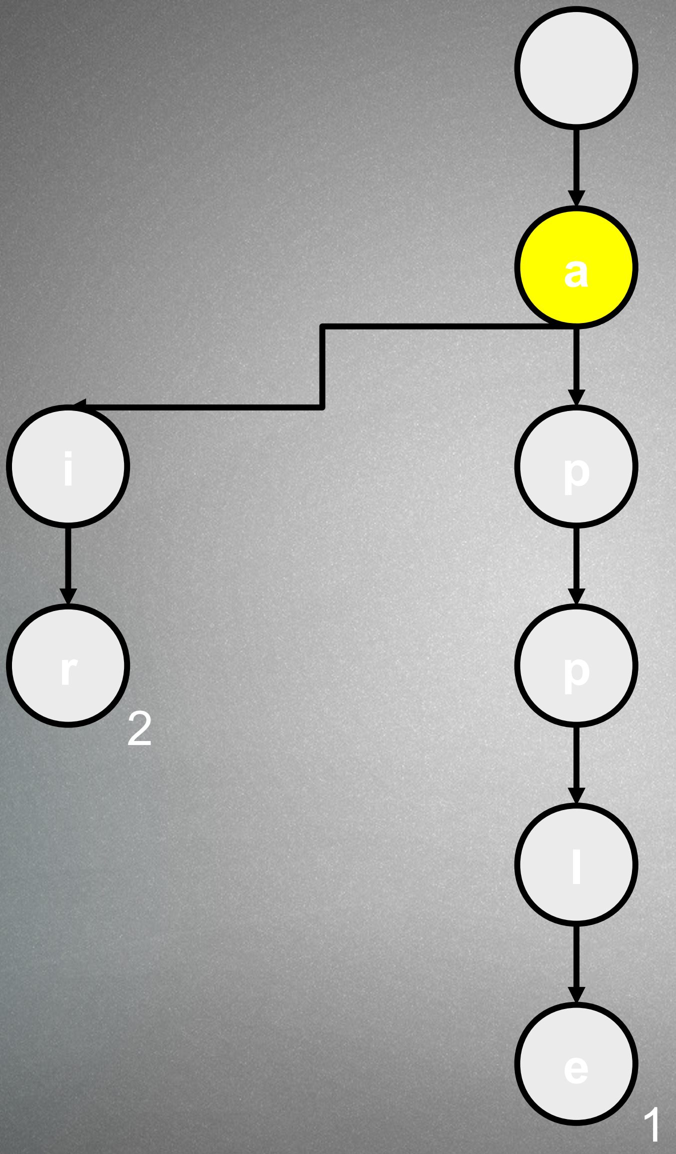


Insertion

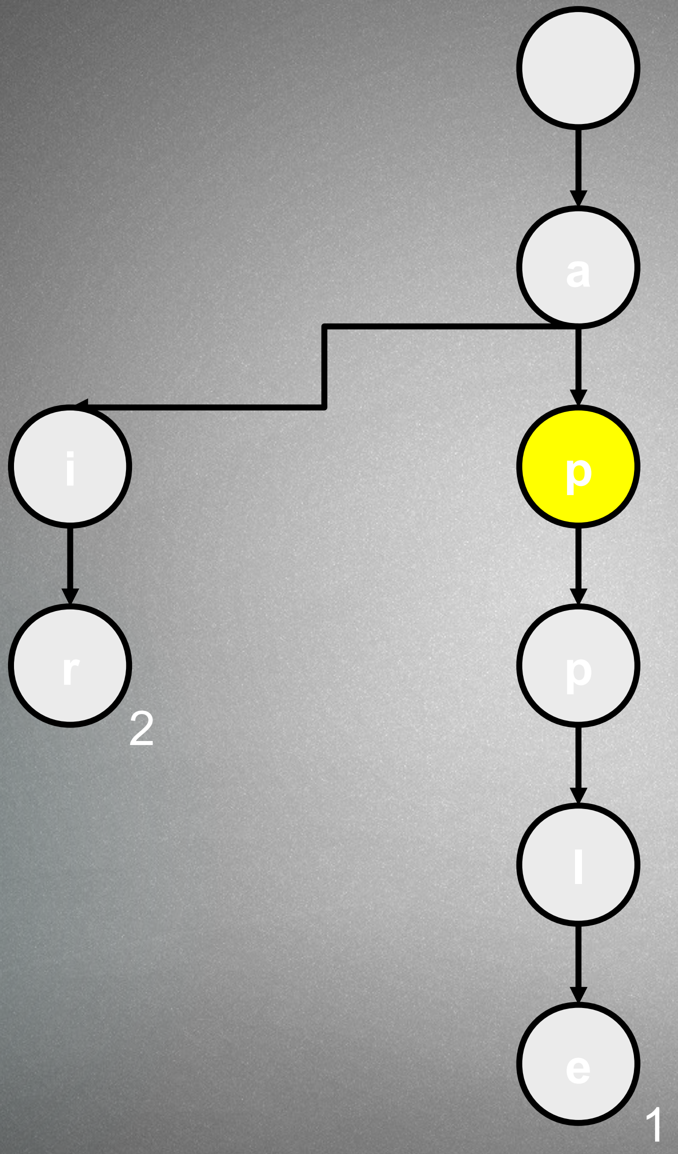
put(„approve”, 3)



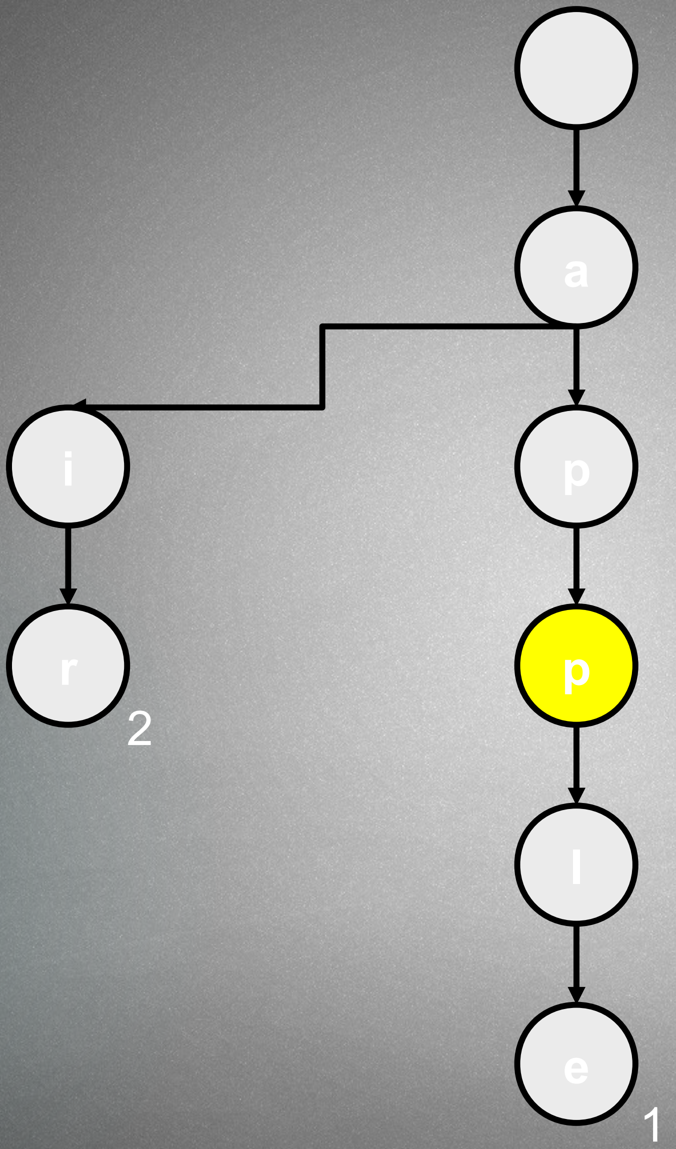
Insertion put(„approve”, 3)



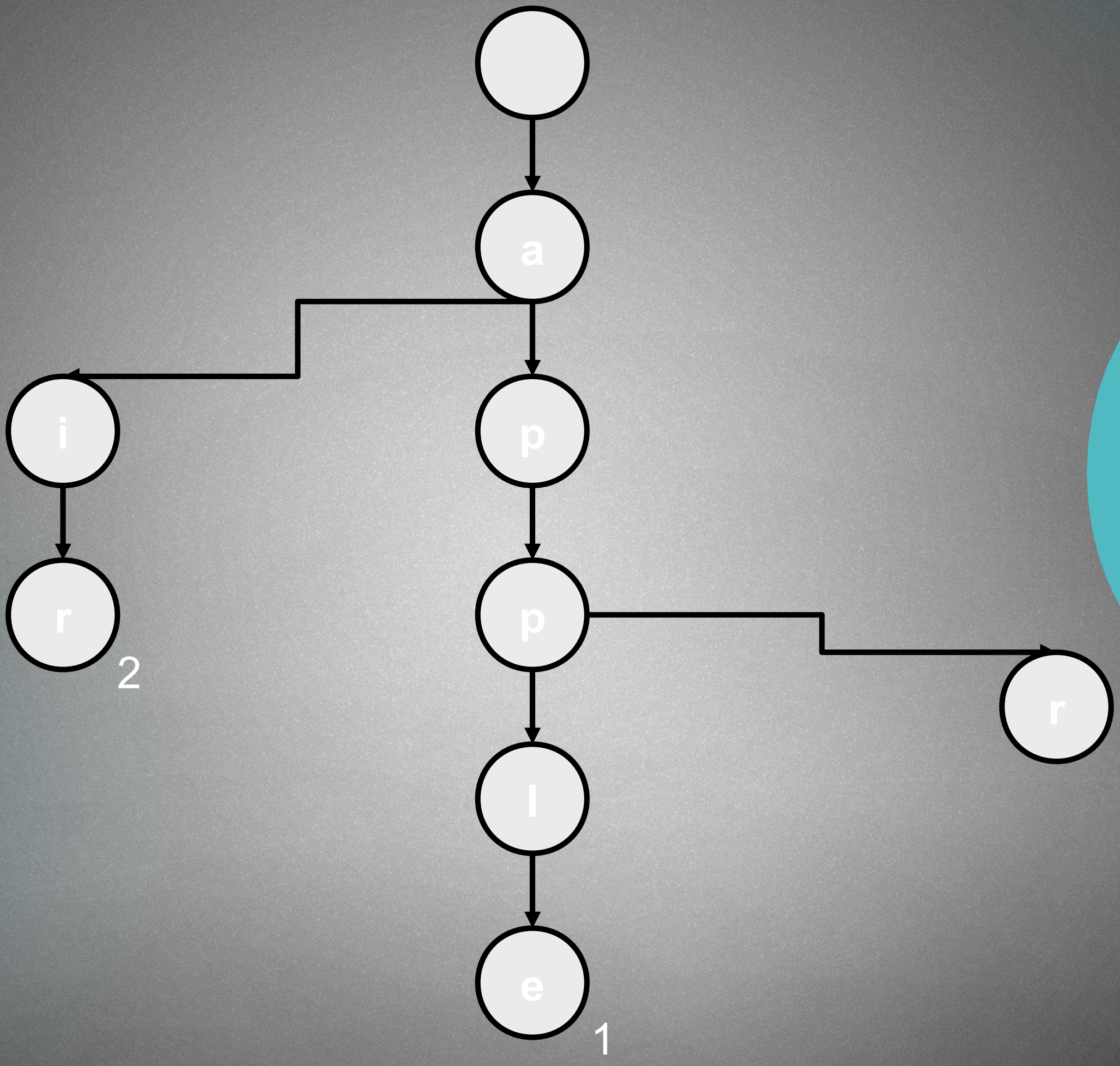
Insertion put(„**ap**prove", 3)



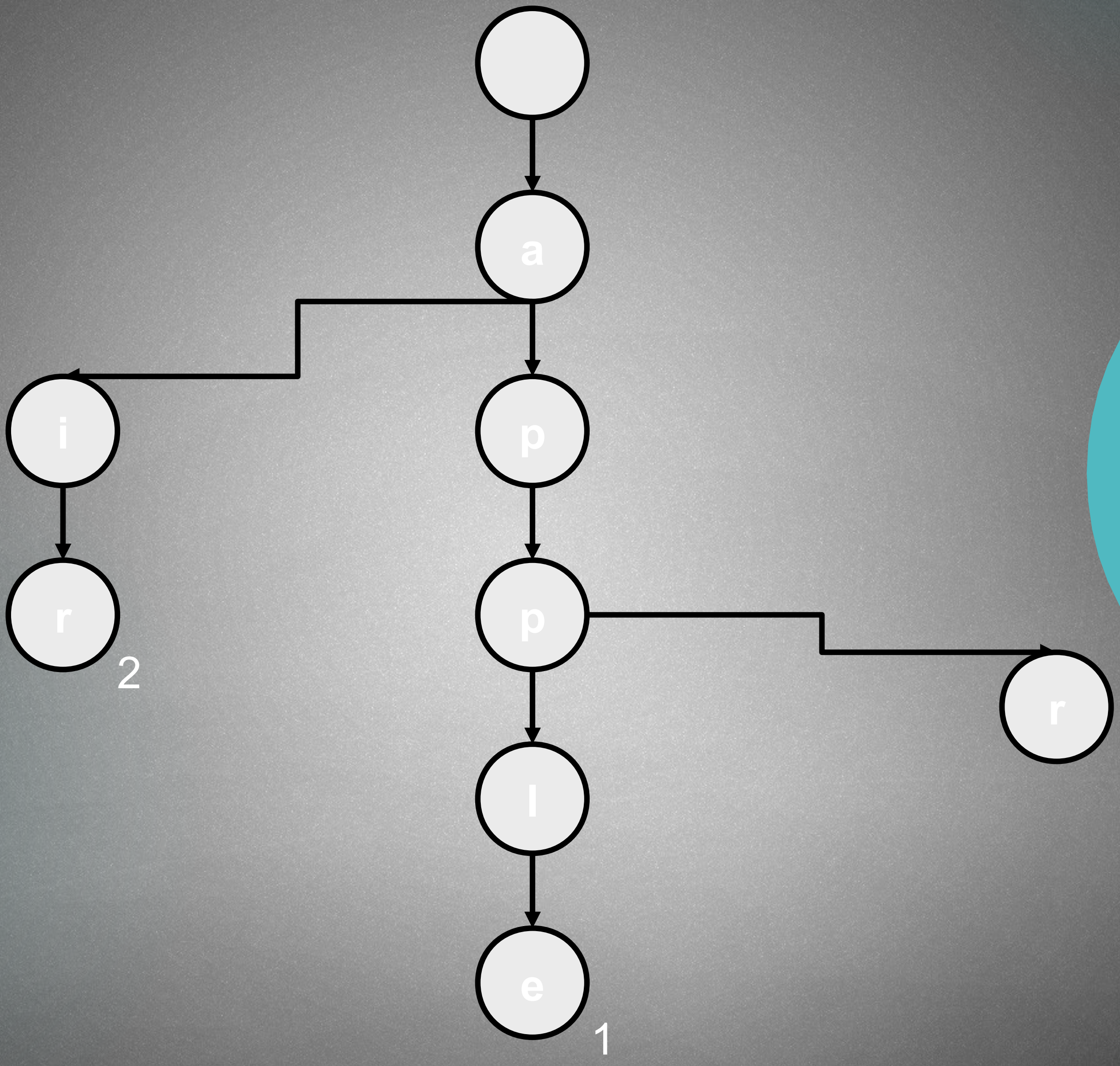
Insertion put(„**app**rove", 3)



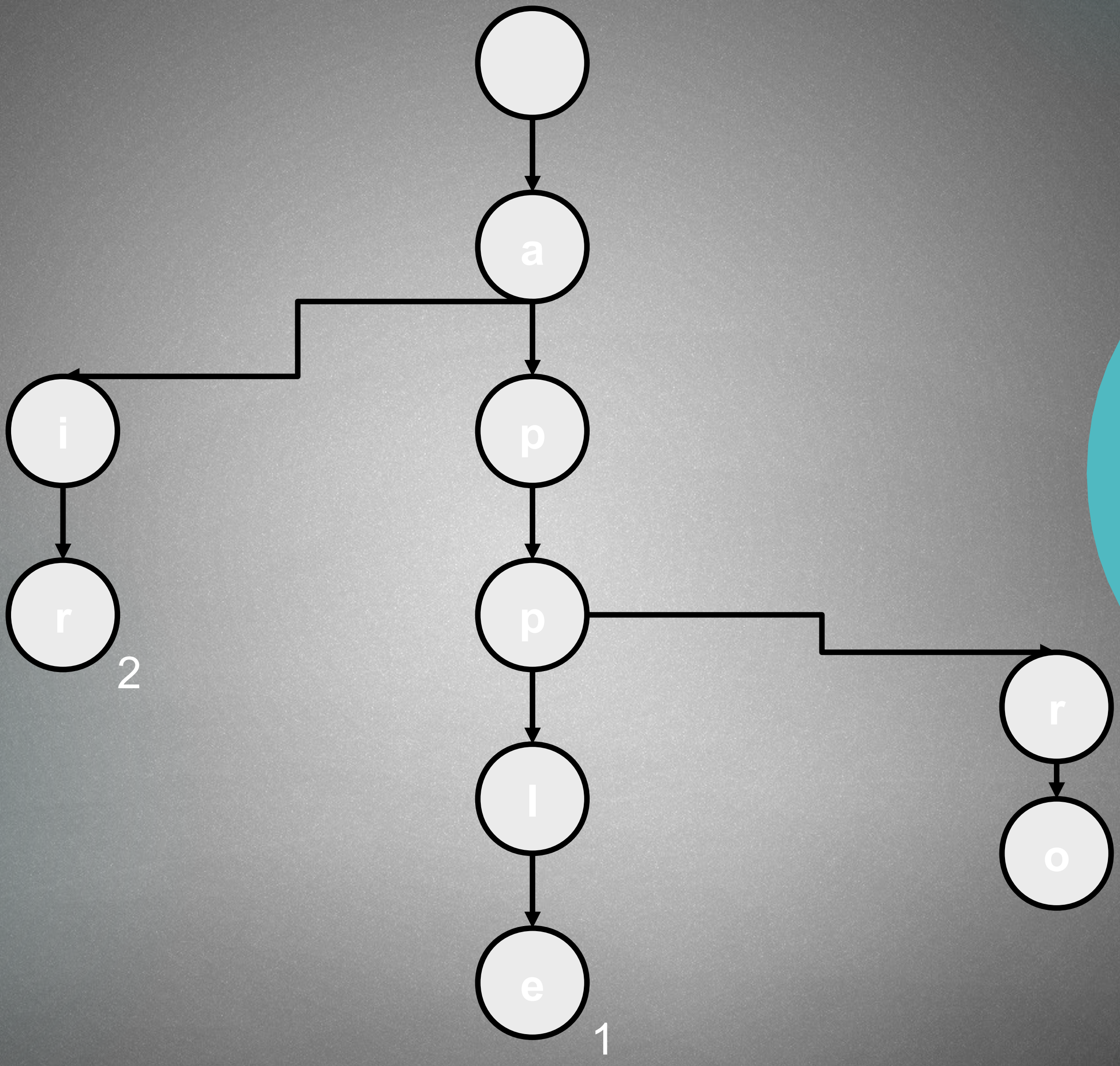
Insertion put(„**app**r**o**ve", 3)



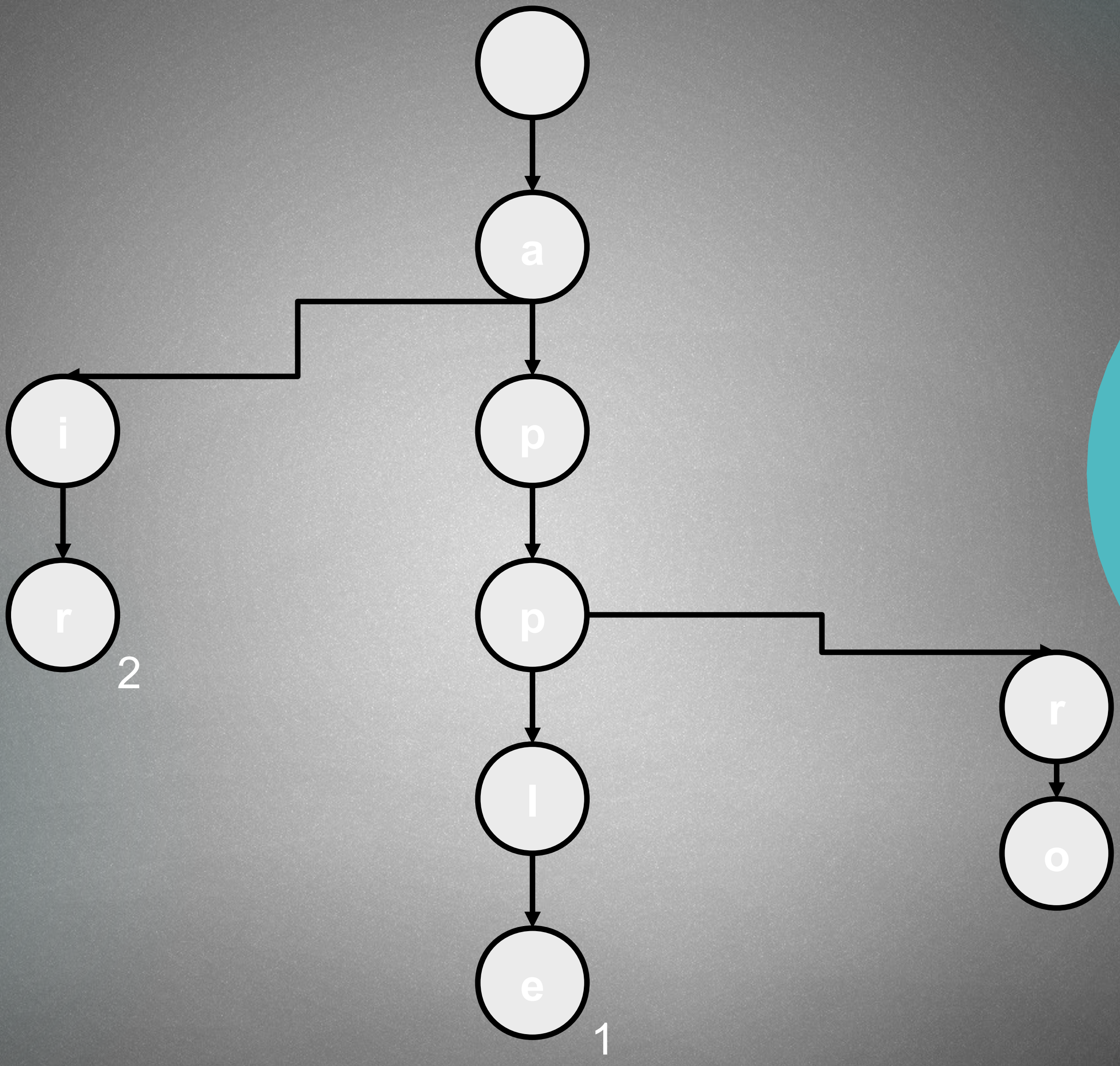
Insertion put(„**appro**ve”, 3)



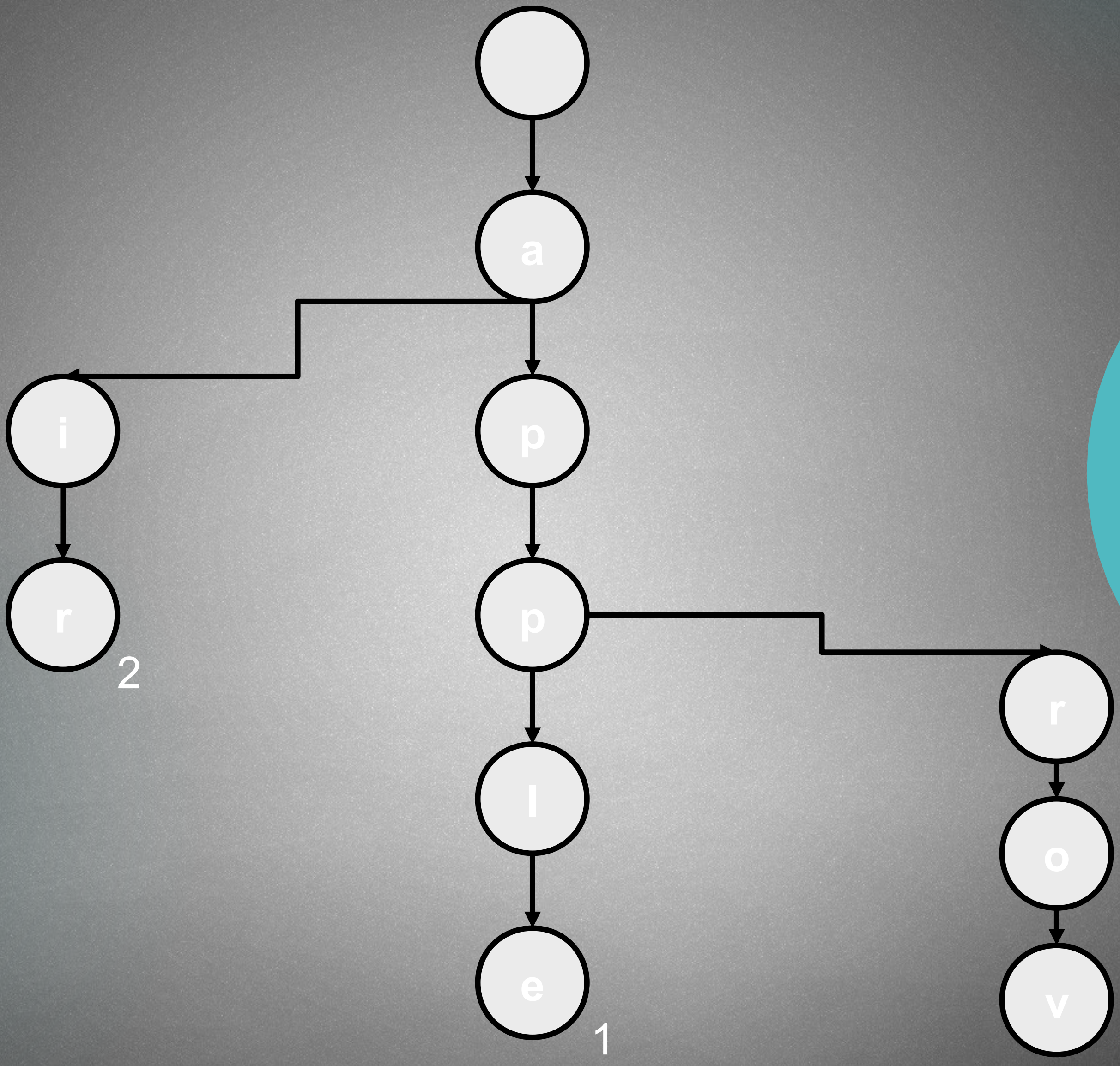
Insertion put(„**approve**", 3)



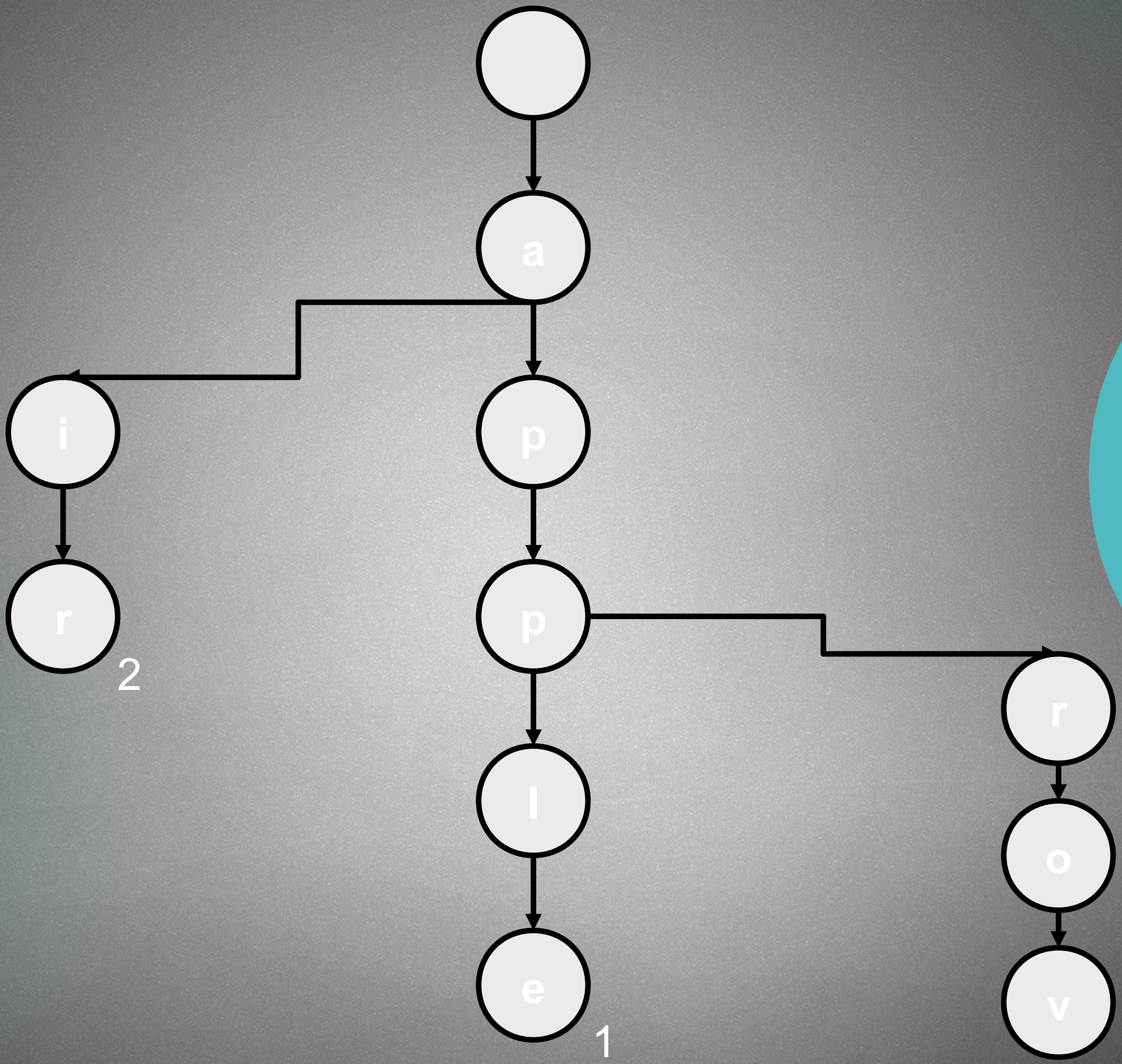
Insertion put(„**approve**”, 3)



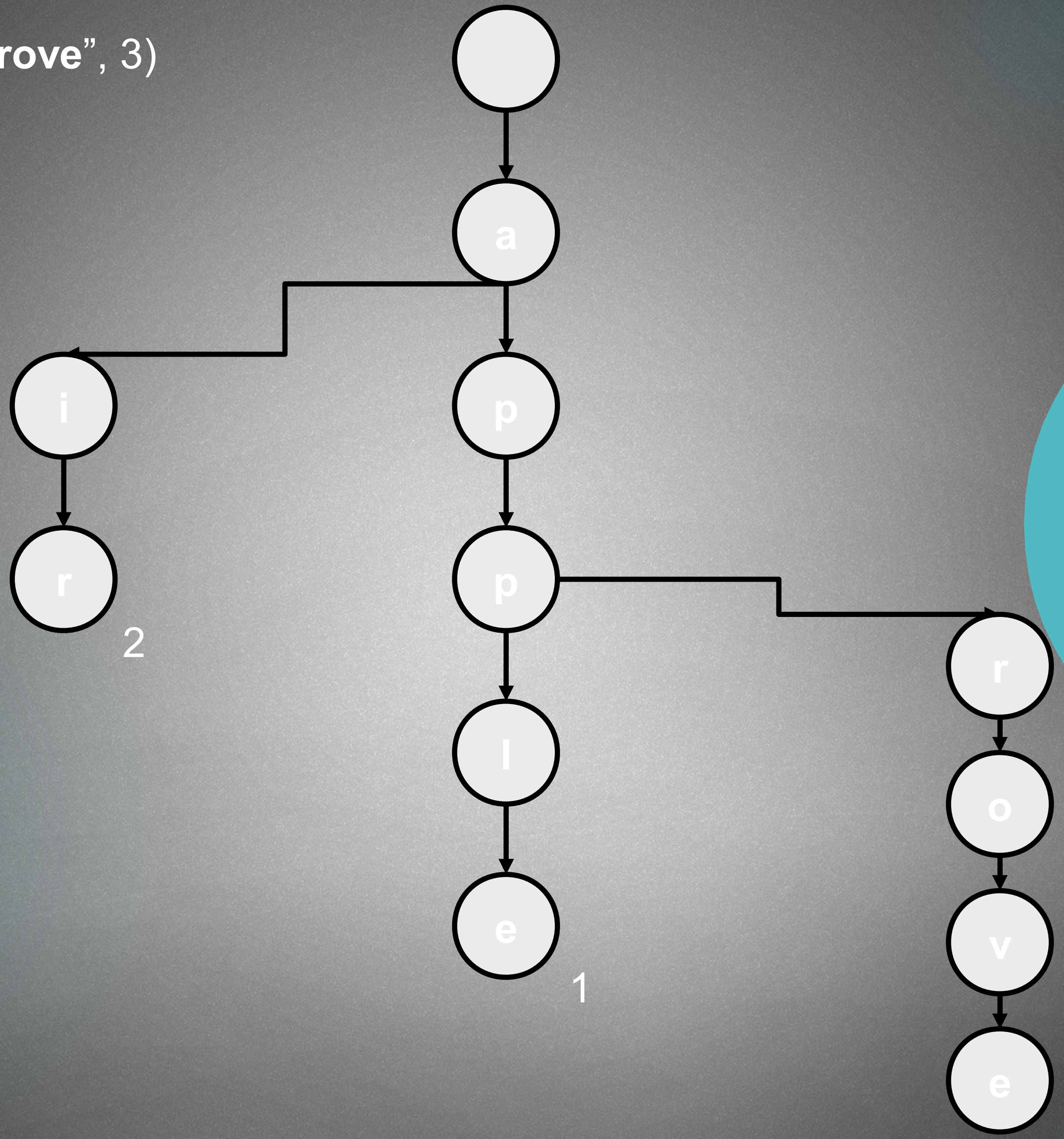
Insertion put(„**approve**", 3)



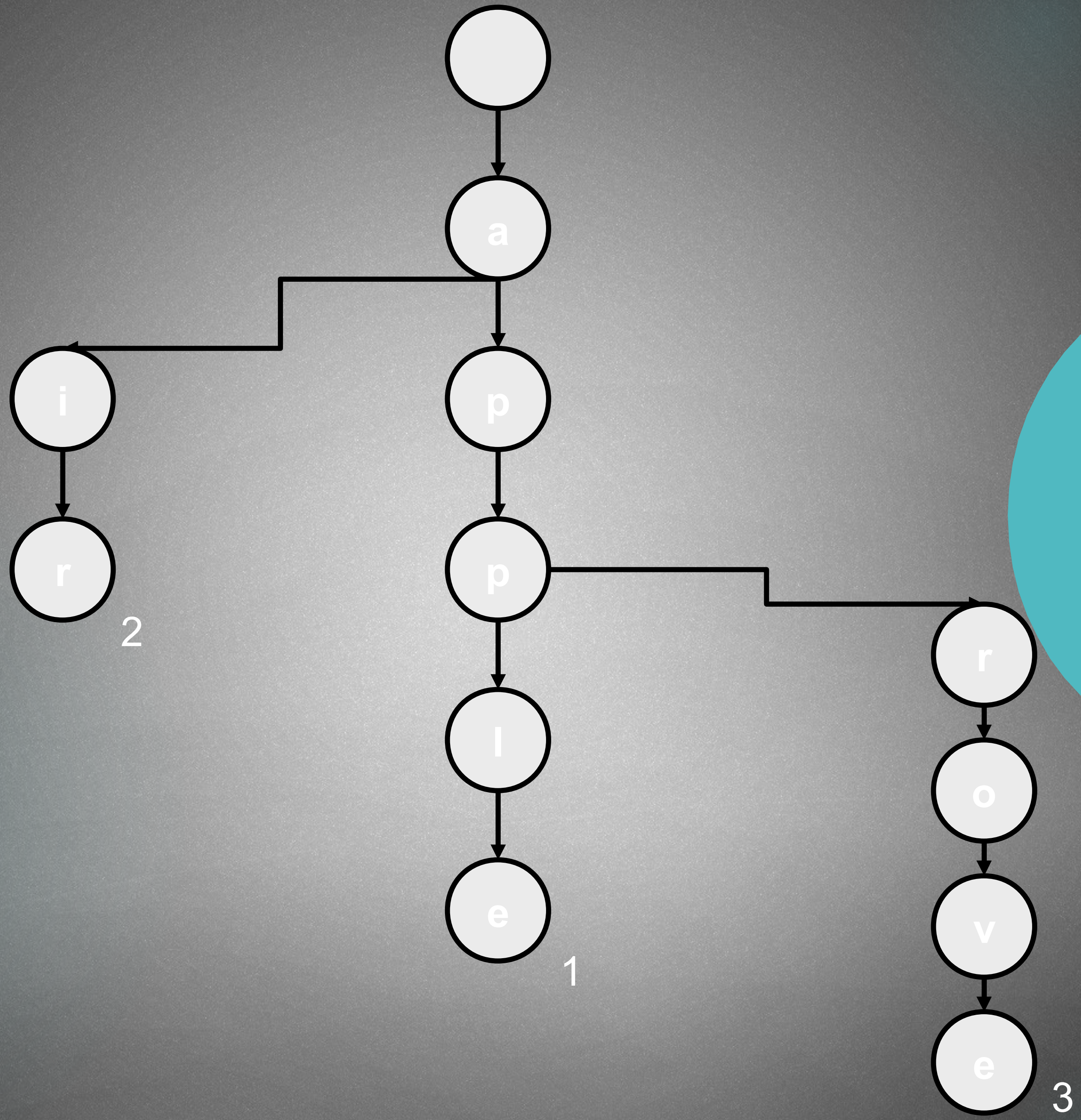
Insertion put(„**approve**”, 3)



Insertion put(„**approve**”, 3)

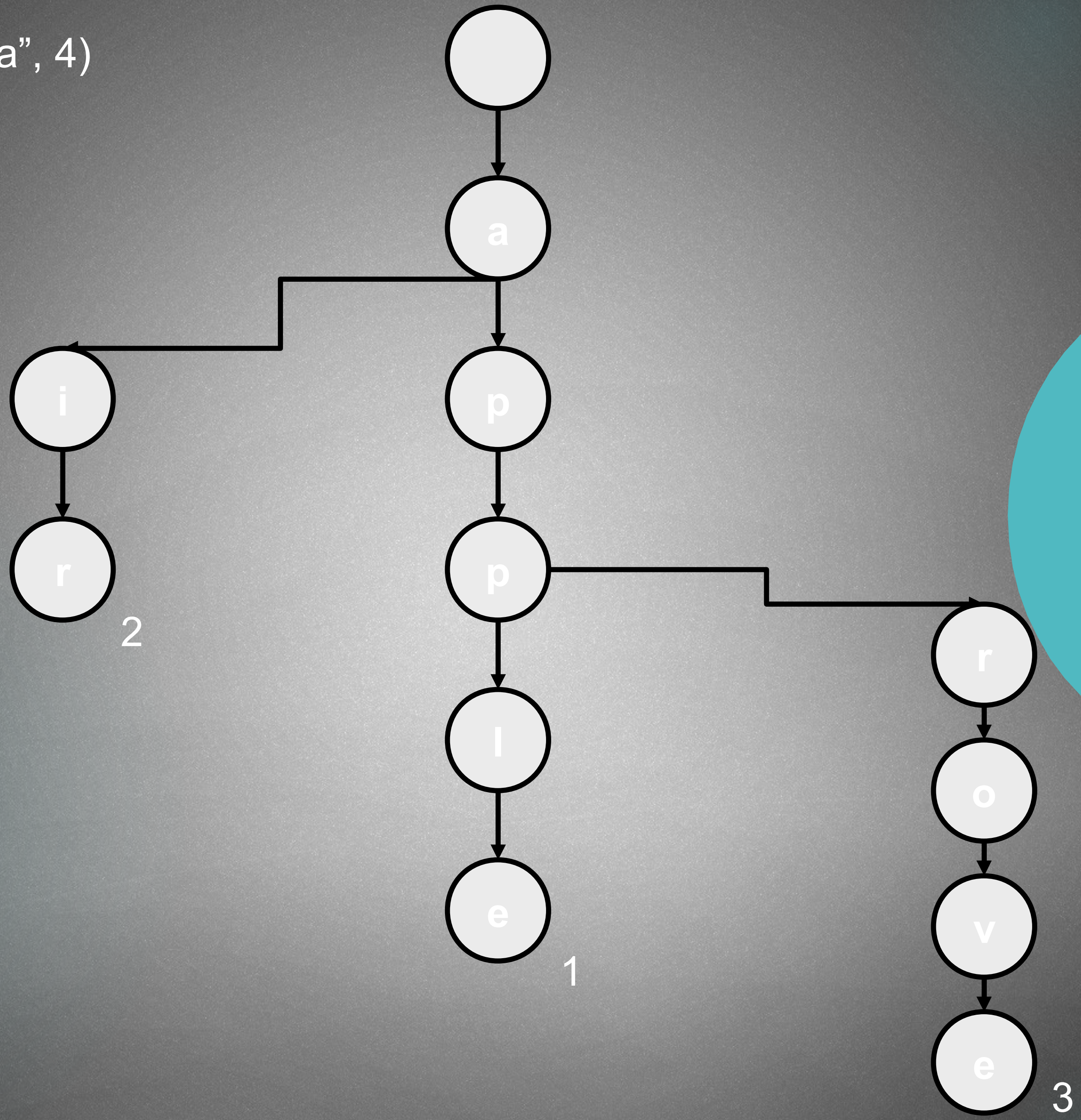


Insertion

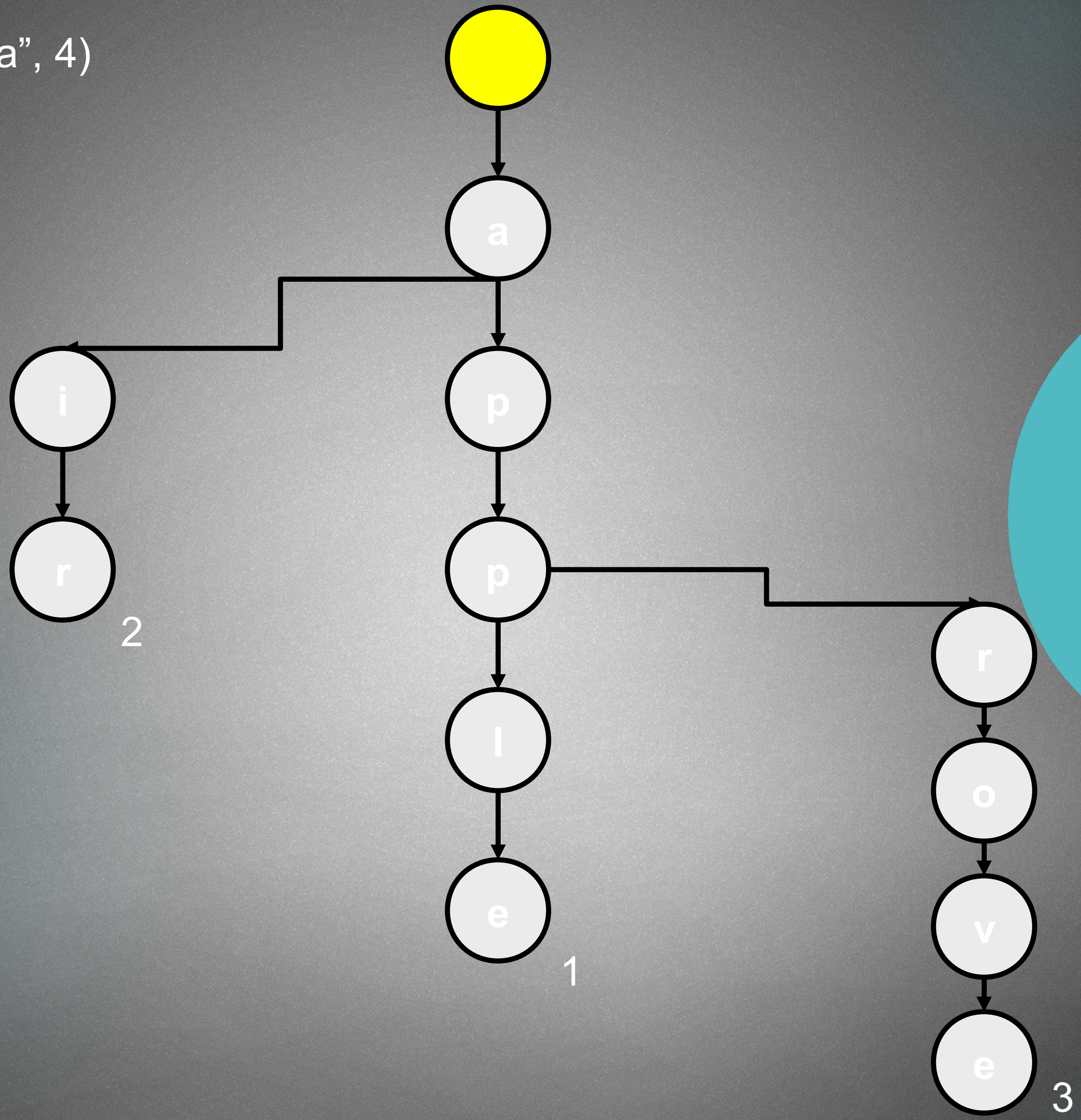


Insertion

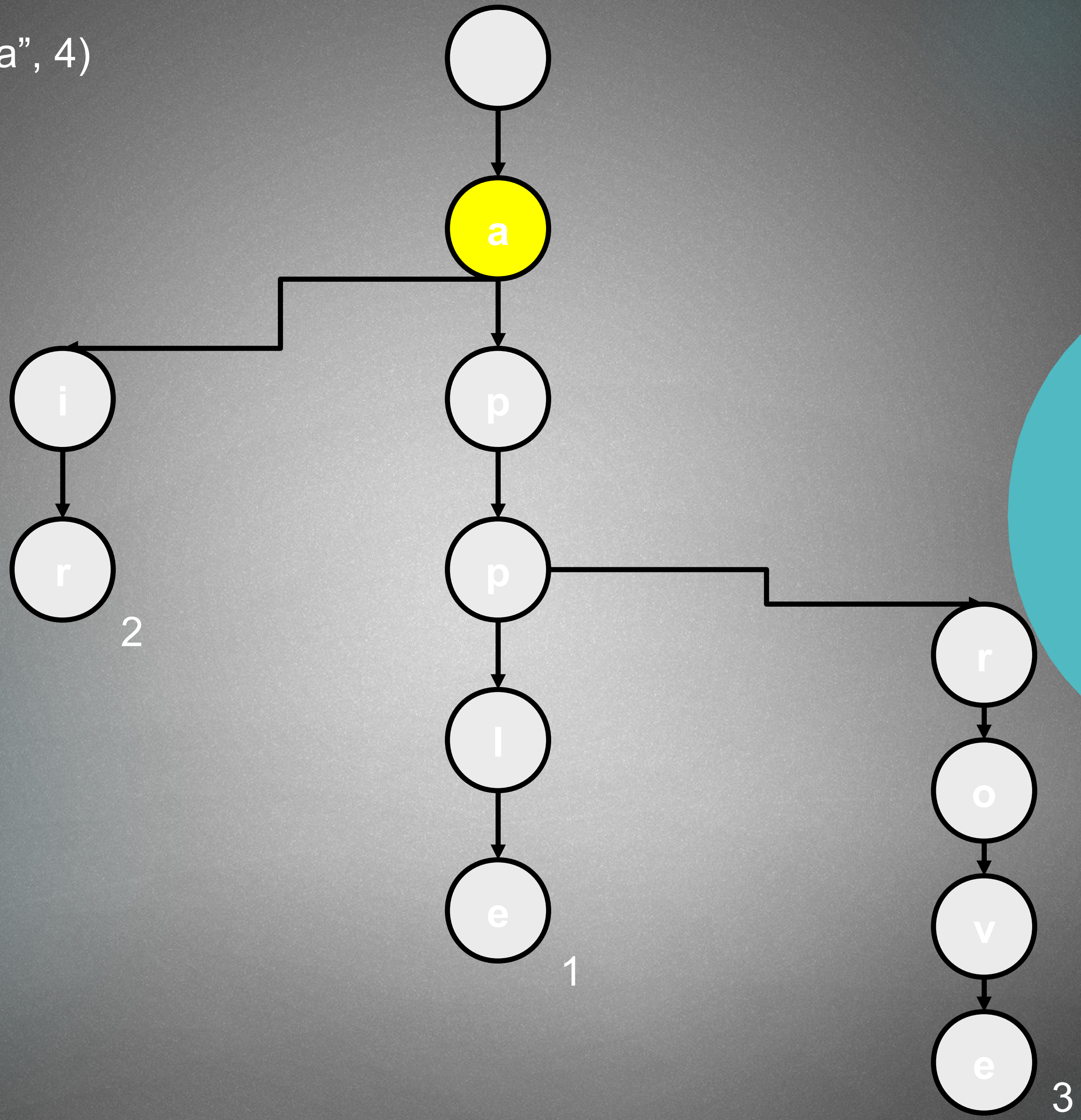
put(„appa", 4)



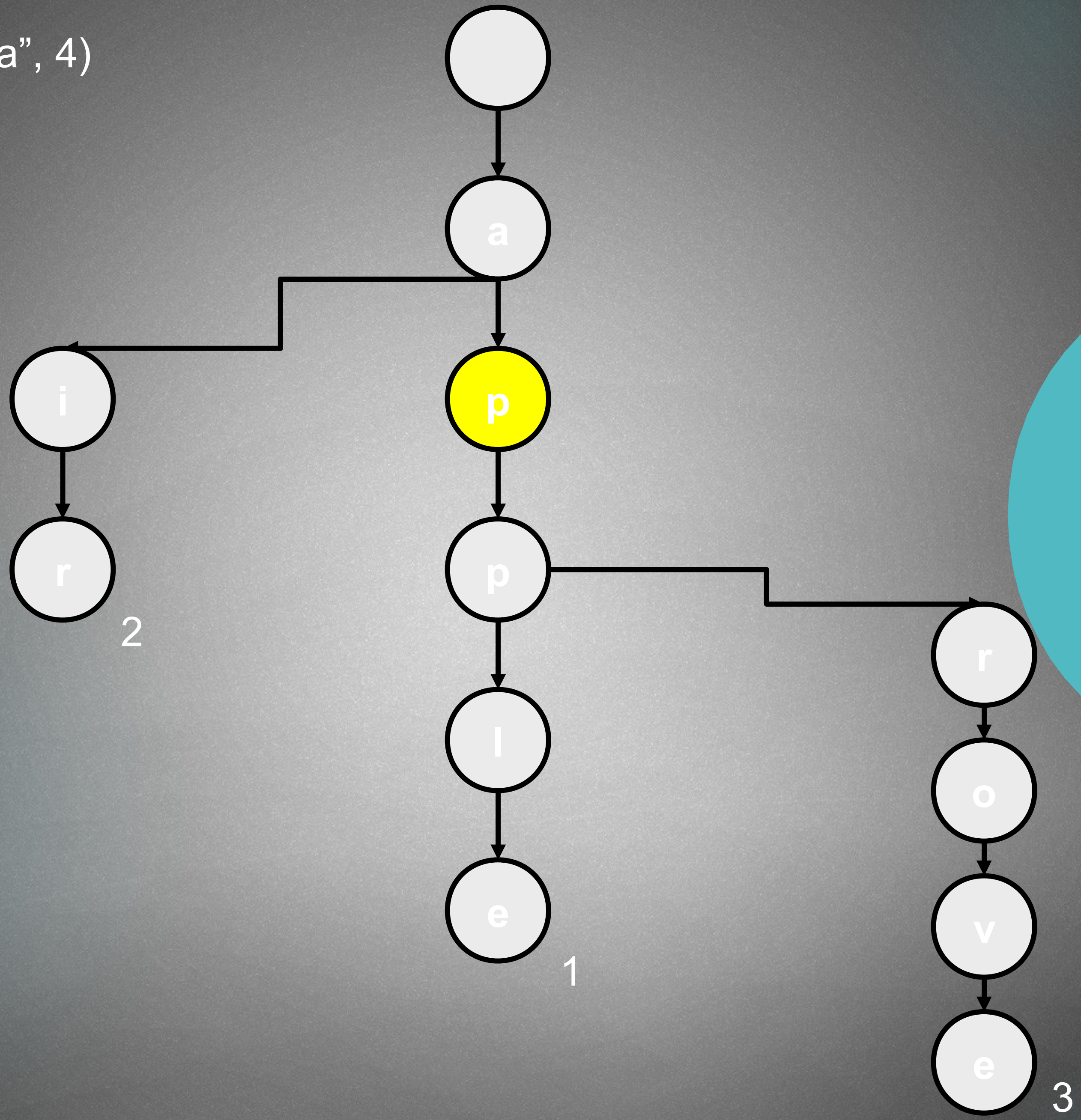
Insertion put(„appa”, 4)



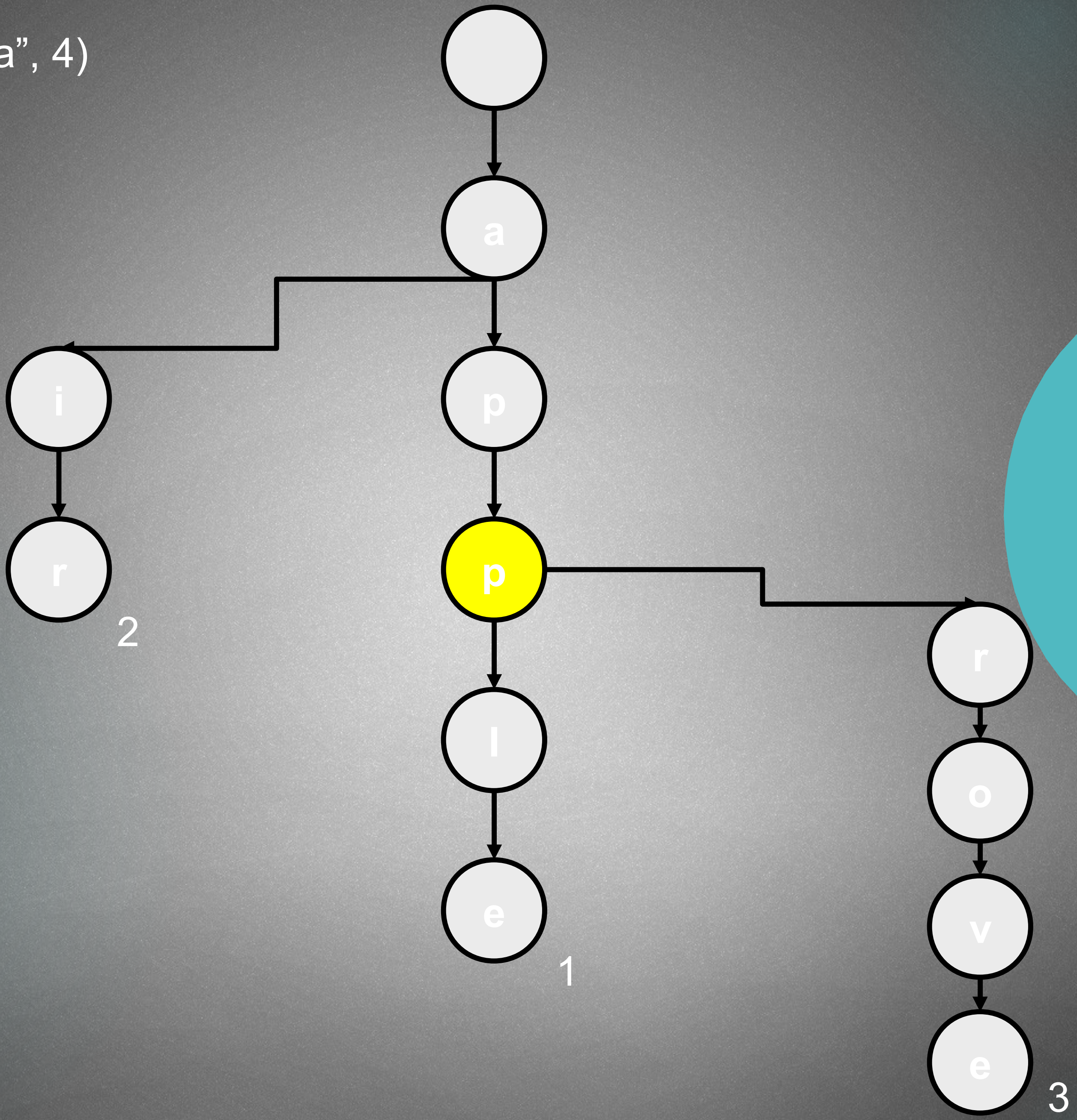
Insertion put(„appa”, 4)



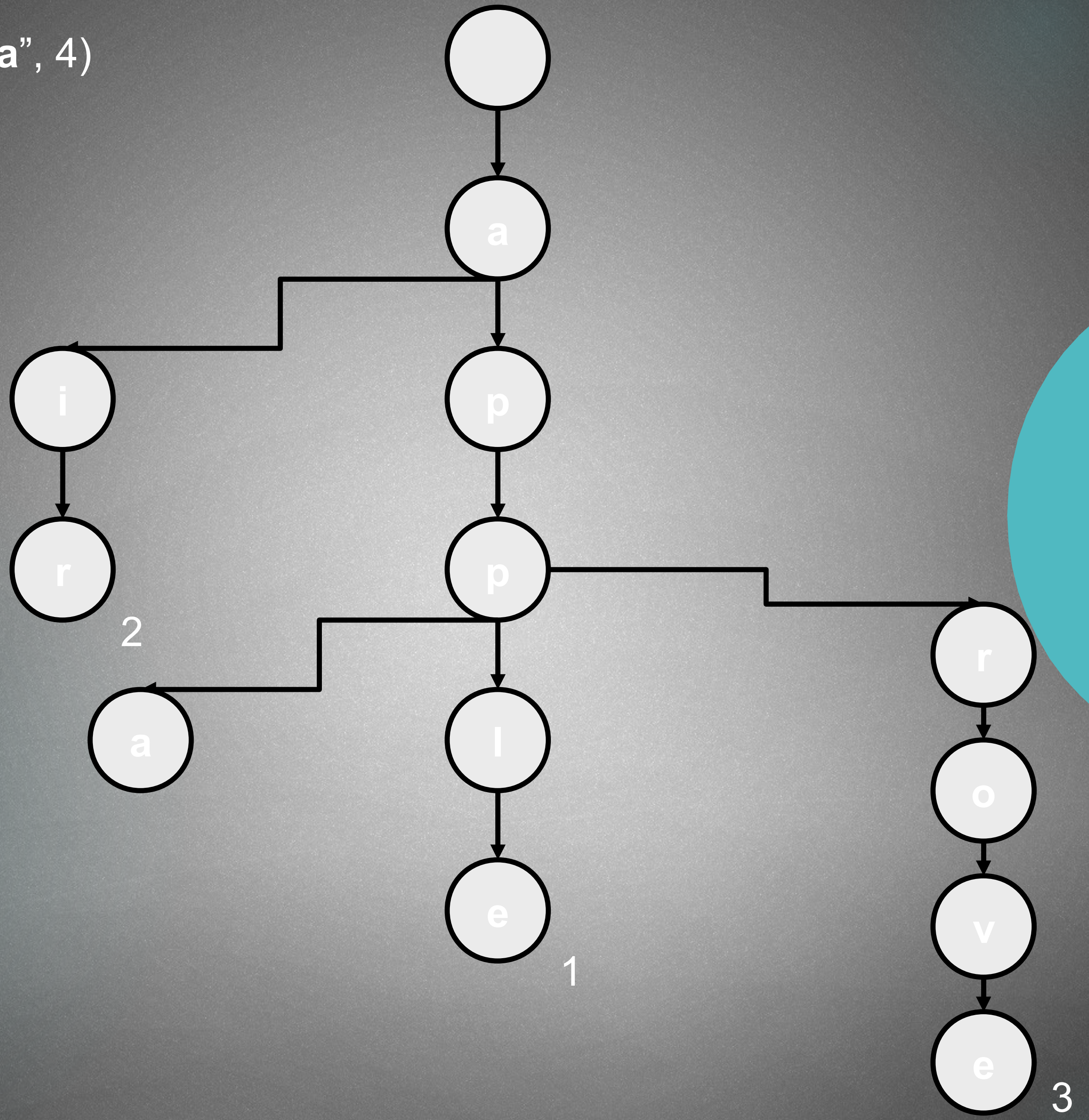
Insertion put(„**ap**pa", 4)



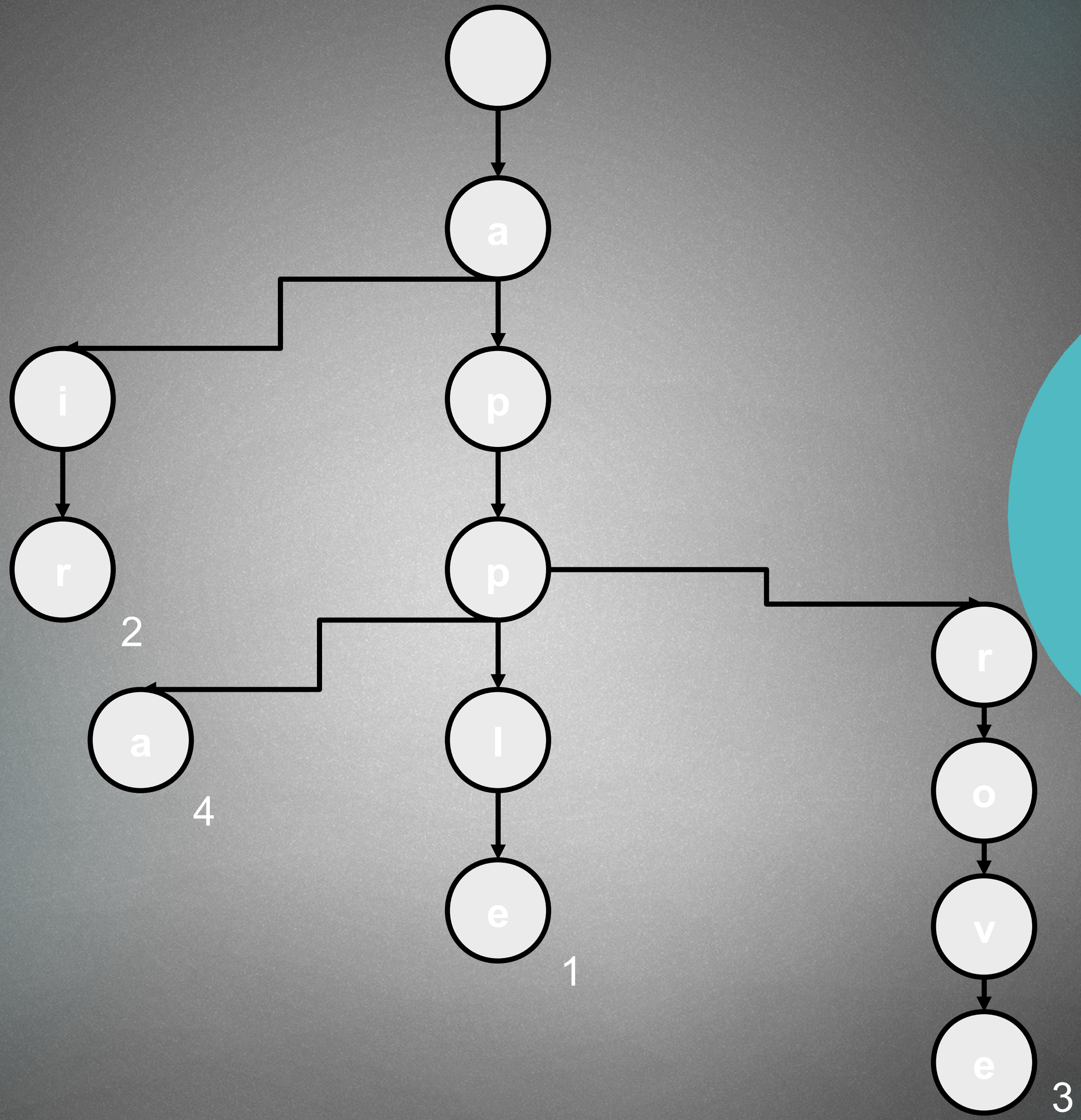
Insertion put(„**appa**", 4)



Insertion put(„**appa**”, 4)

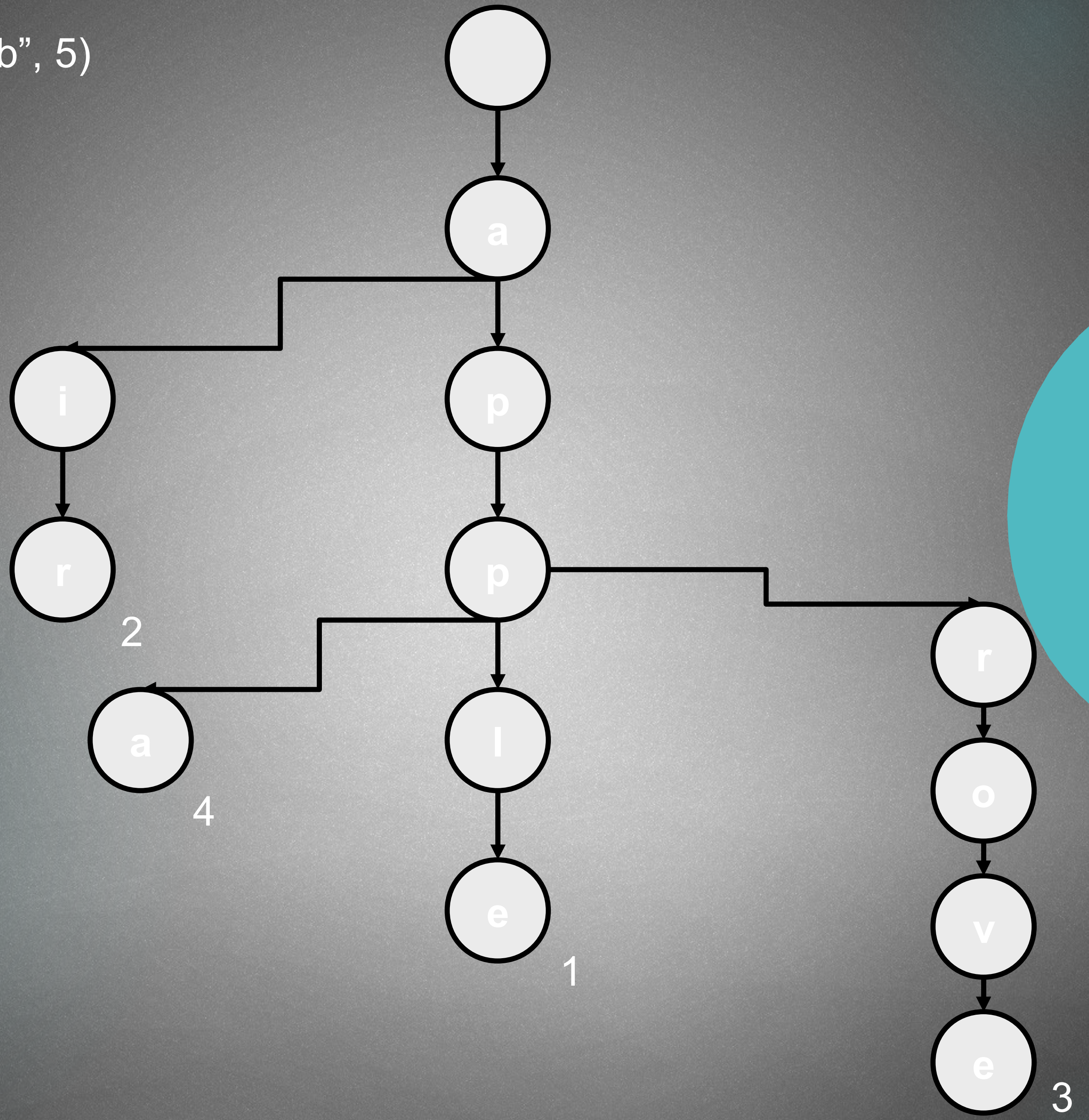


Insertion

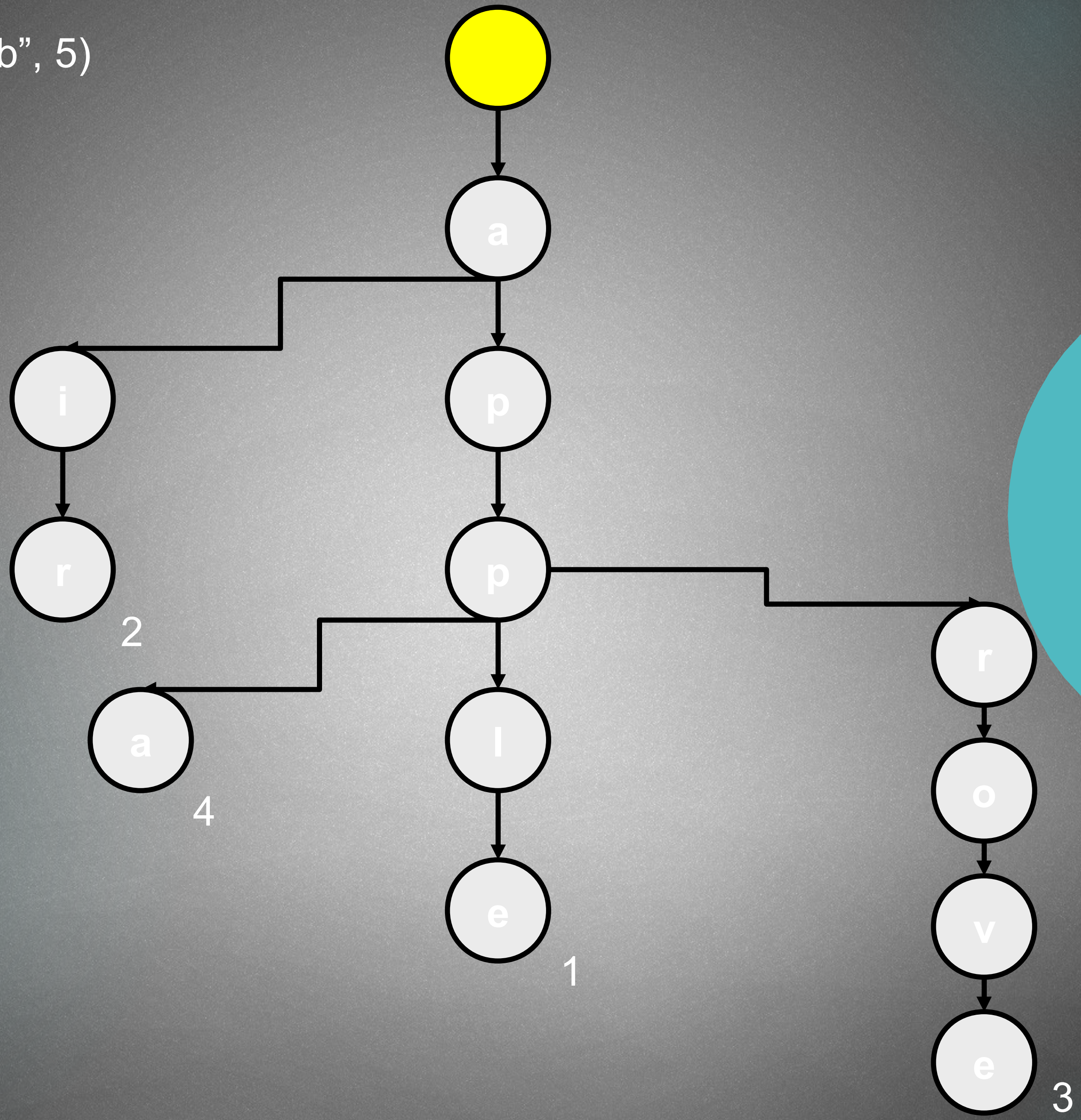


Insertion

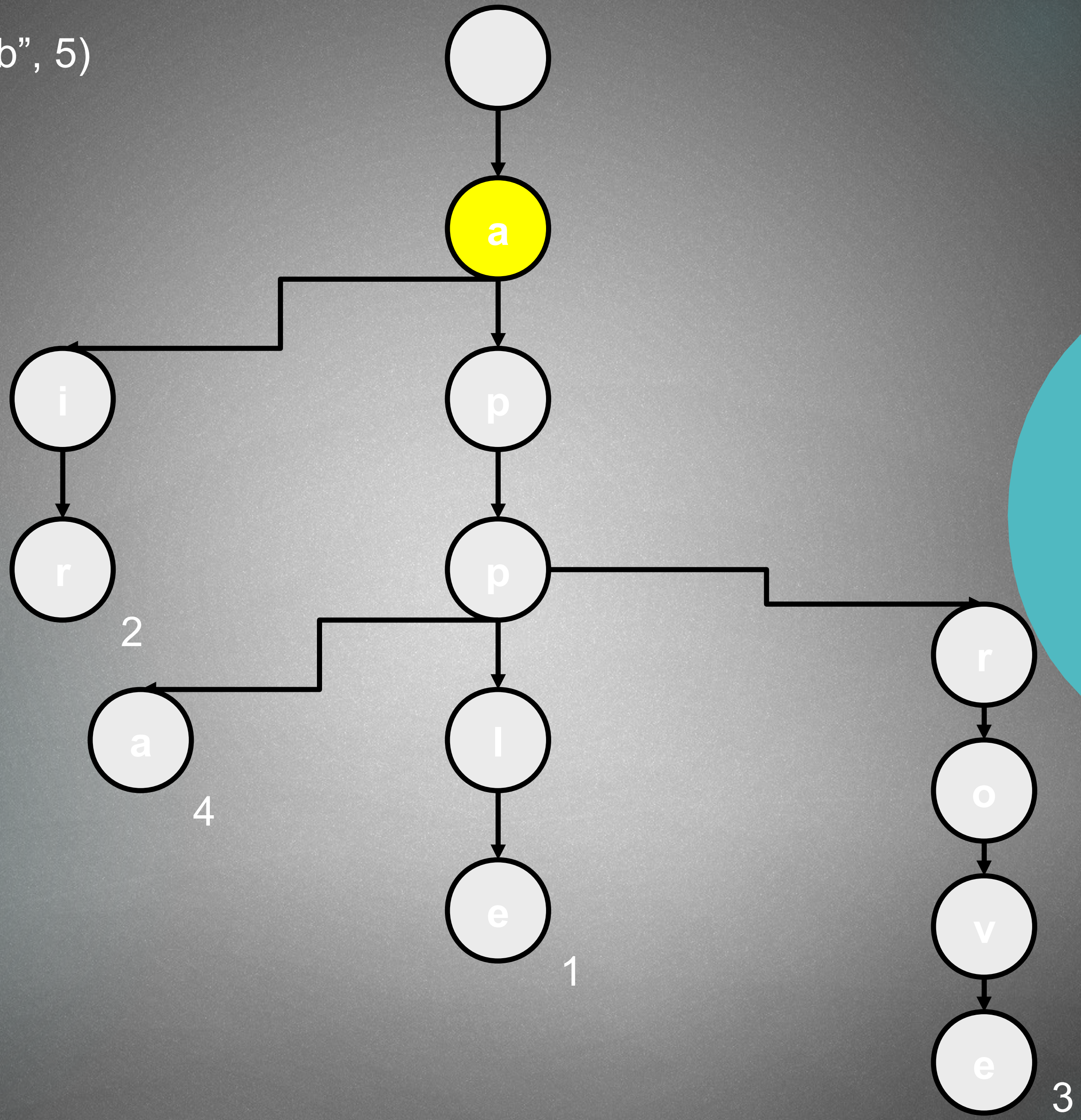
put(„appb”, 5)



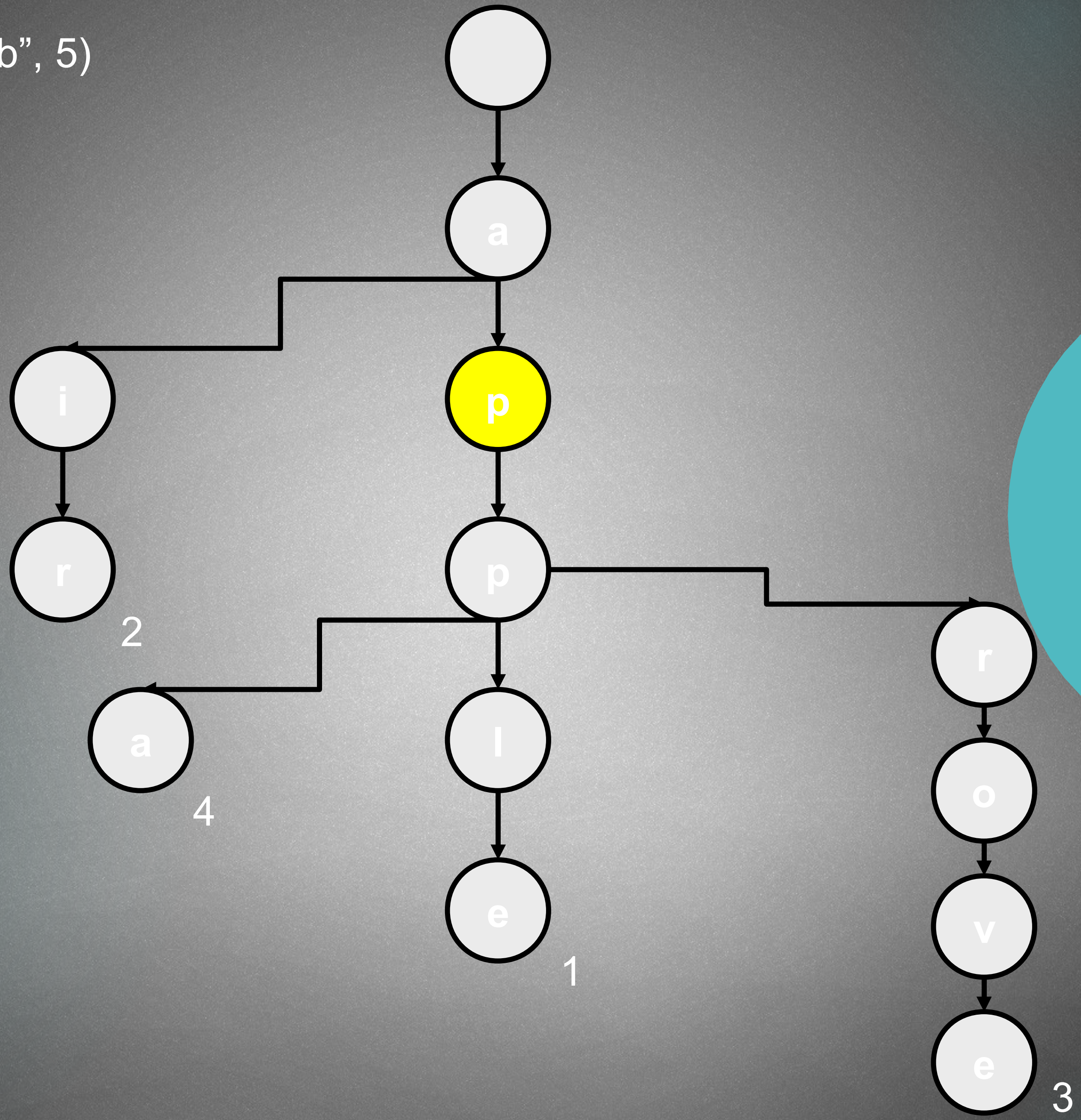
Insertion put(„appb", 5)



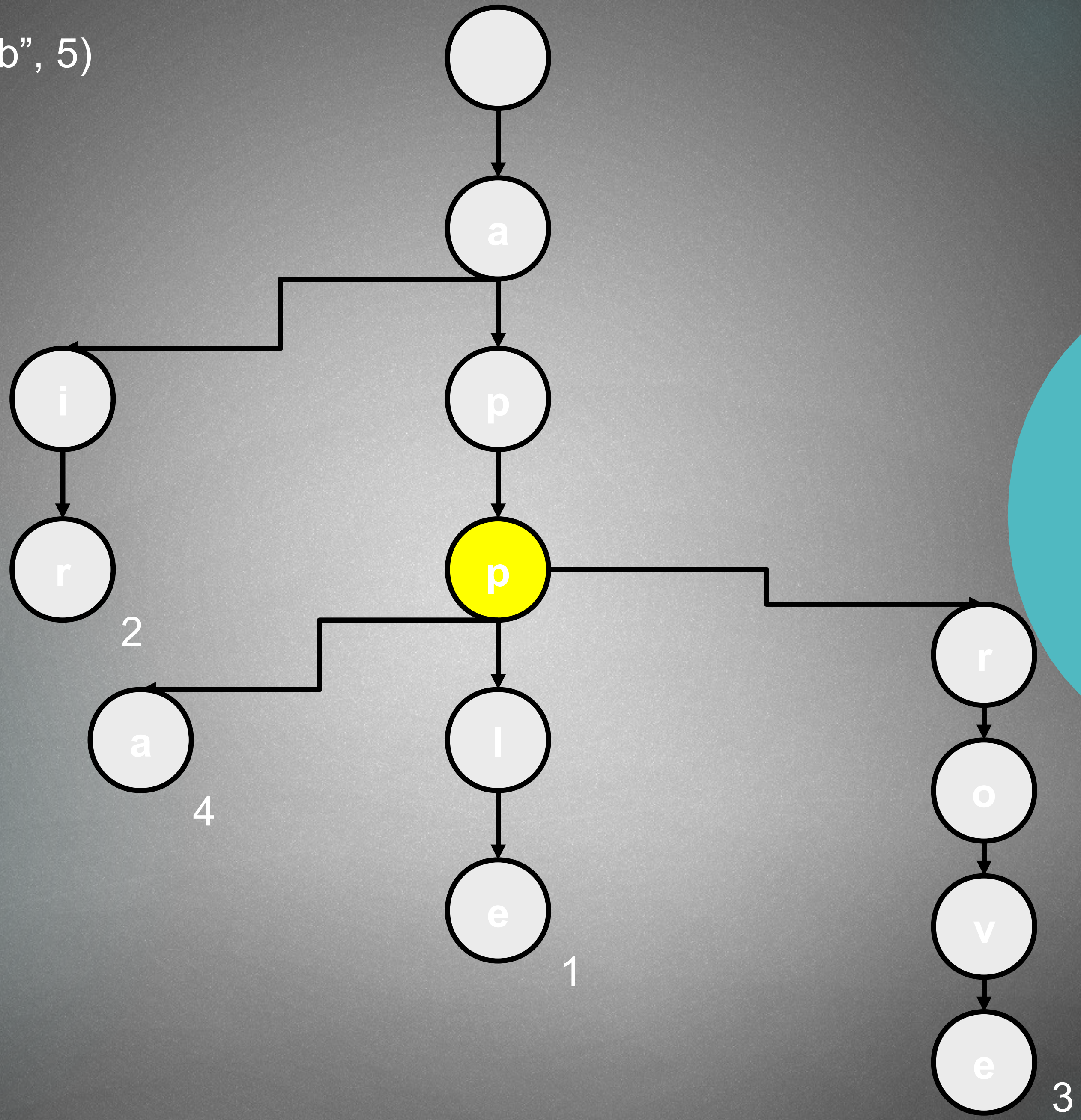
Insertion put(„appb”, 5)



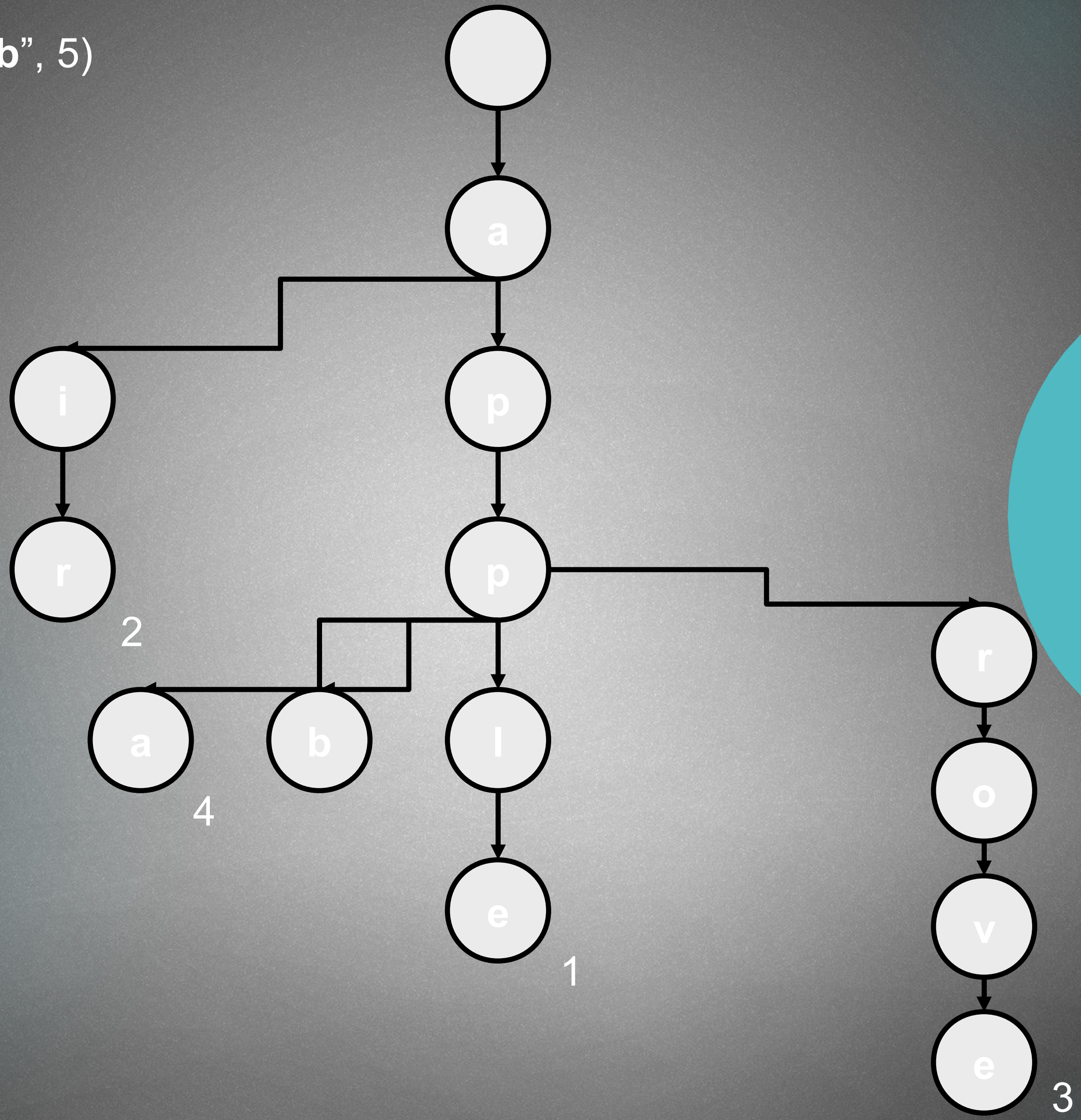
Insertion put(„**ap**pb", 5)

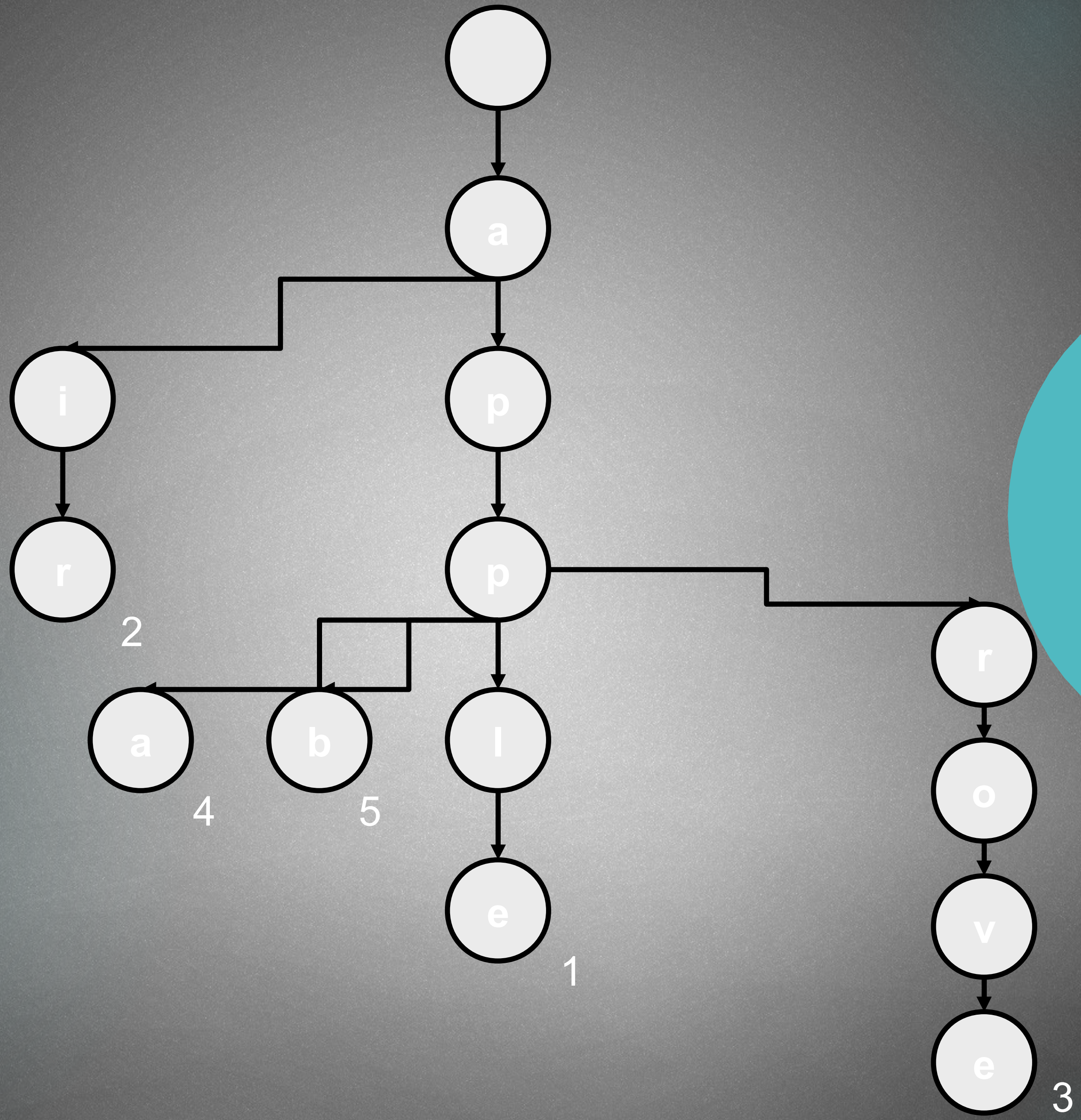


Insertion put(„**app**b", 5)

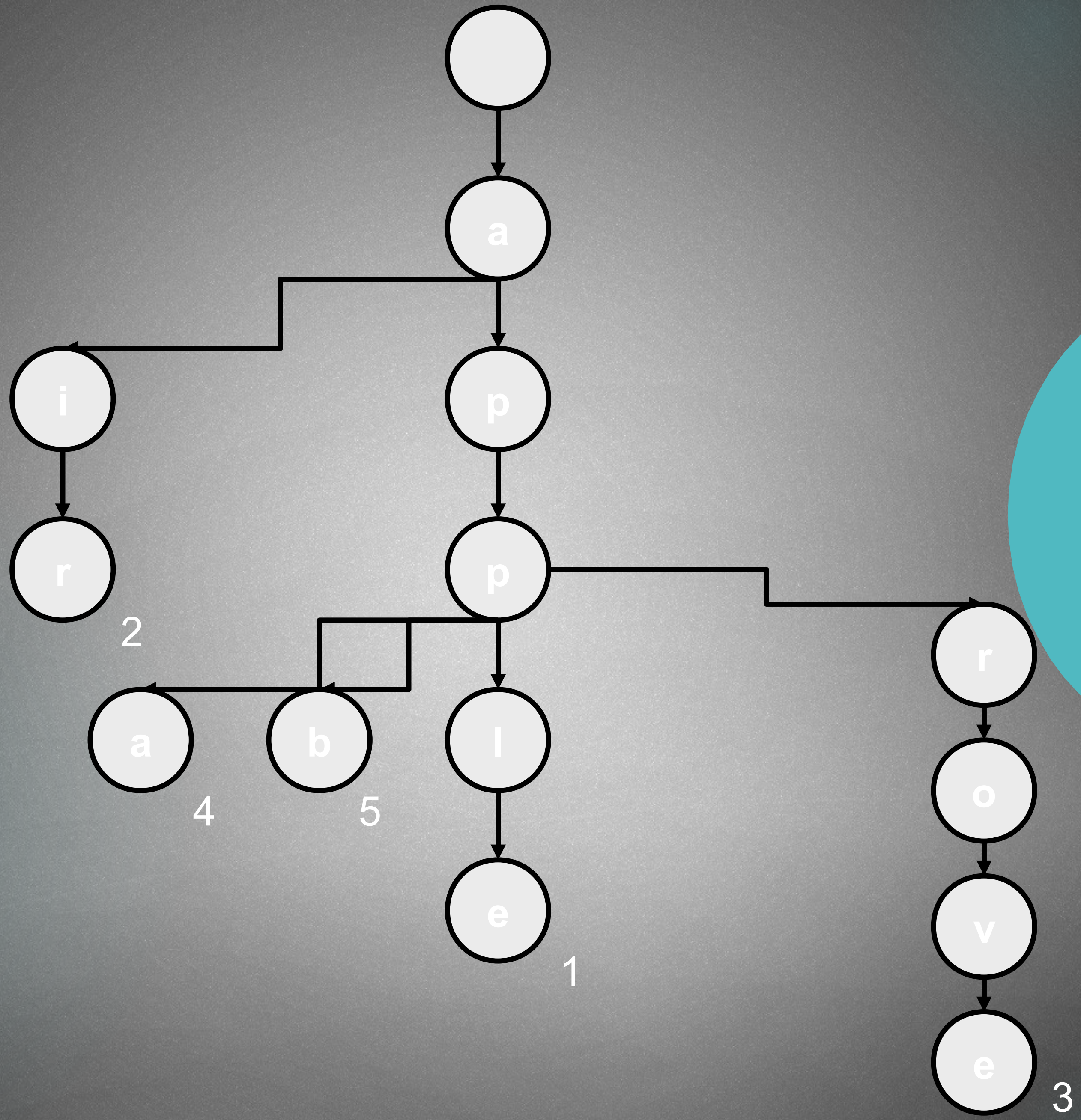


Insertion put(„**appb**”, 5)

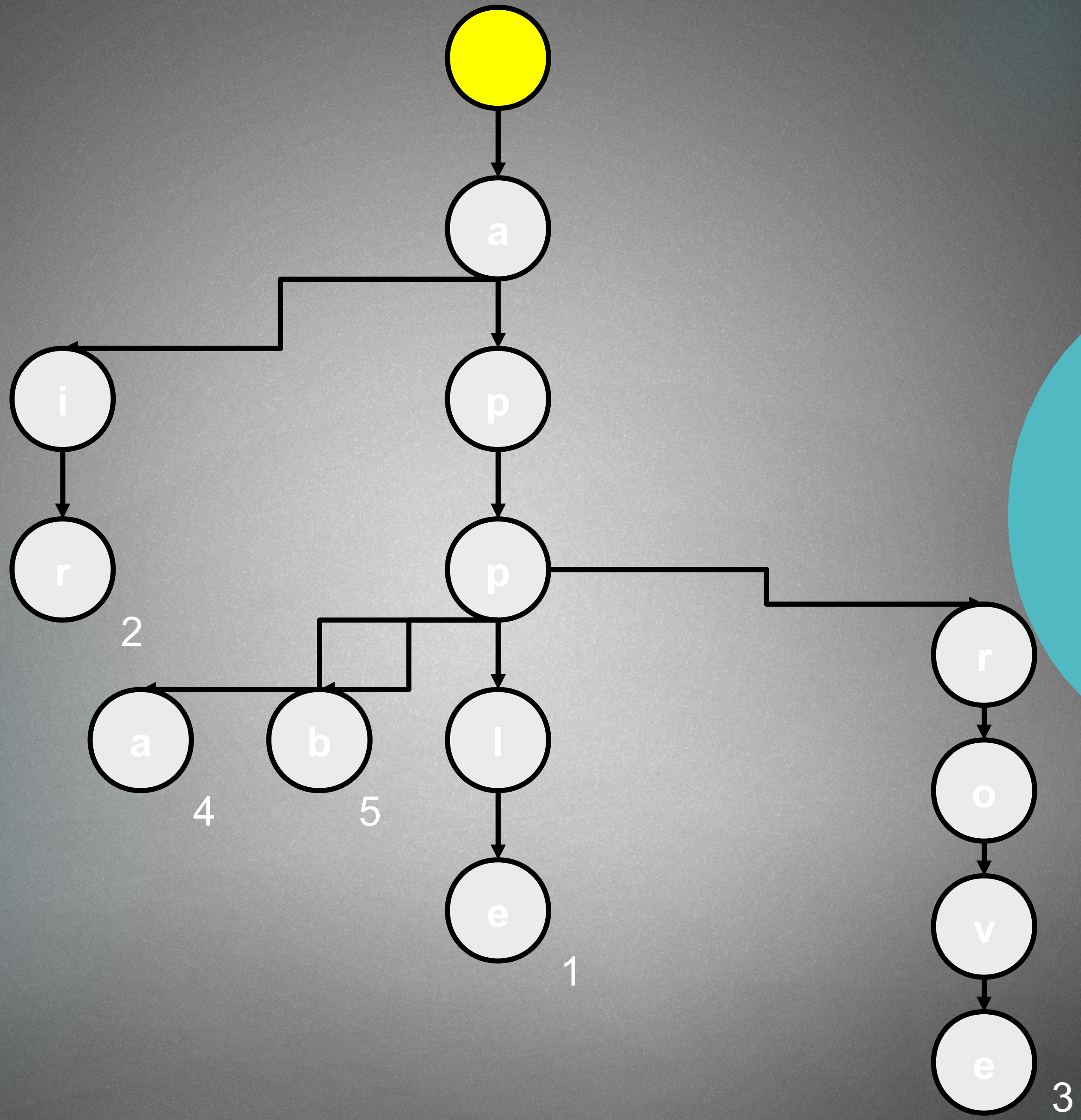



a
i
p
r
p
2
a
b
l
r
4
5
o
e
v
1
e
3

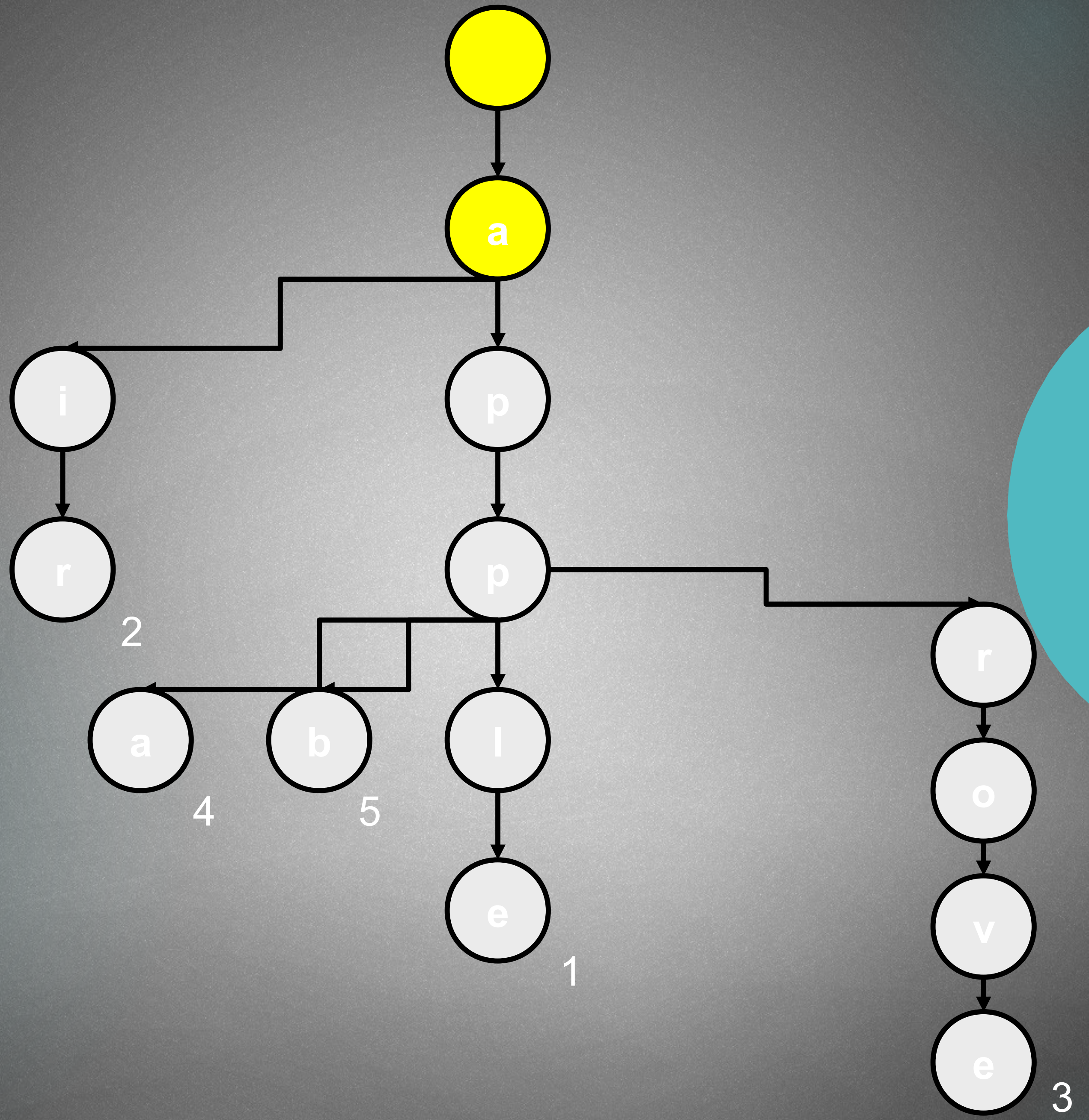
get(„apple")



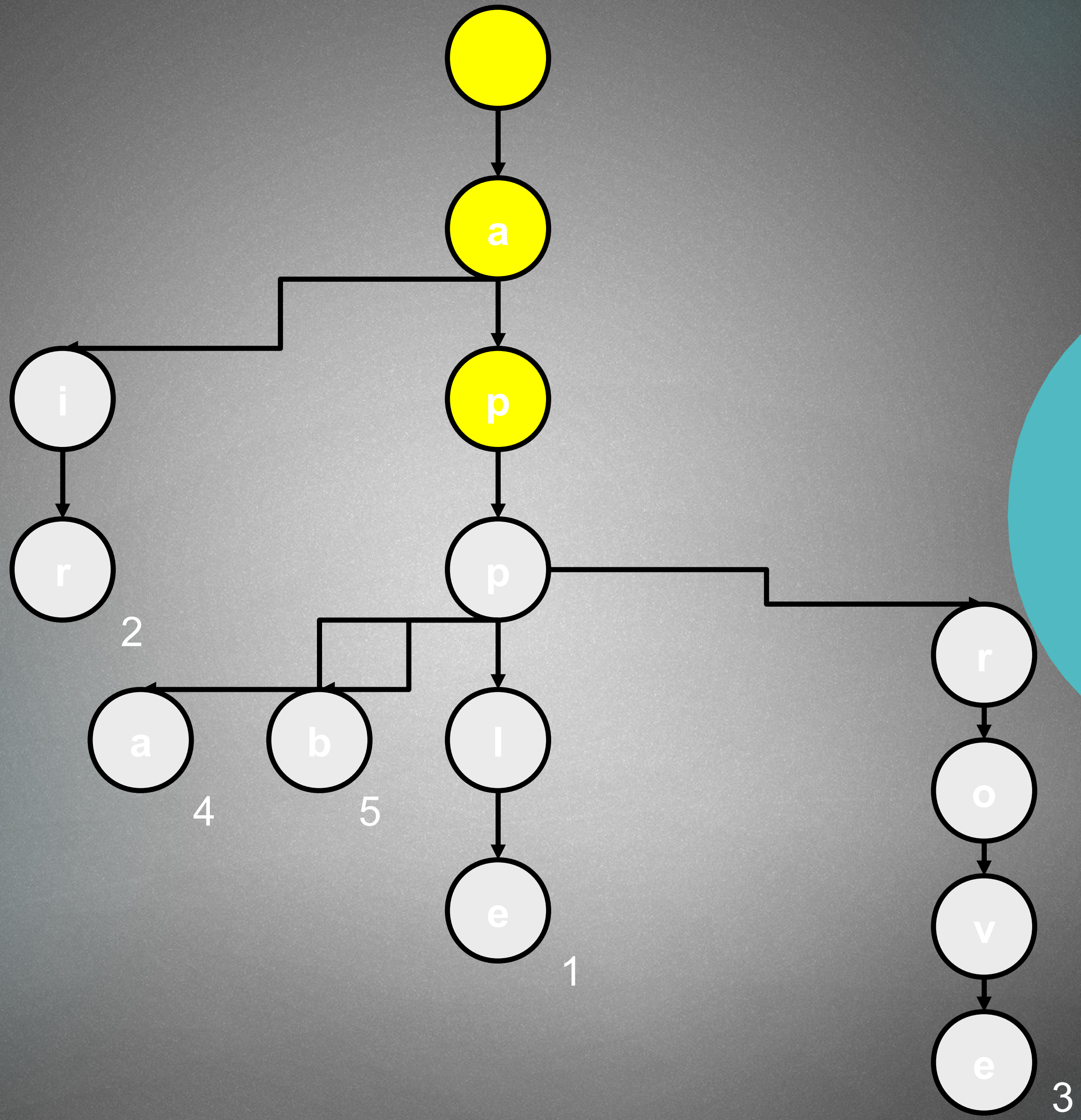
get(„apple”)



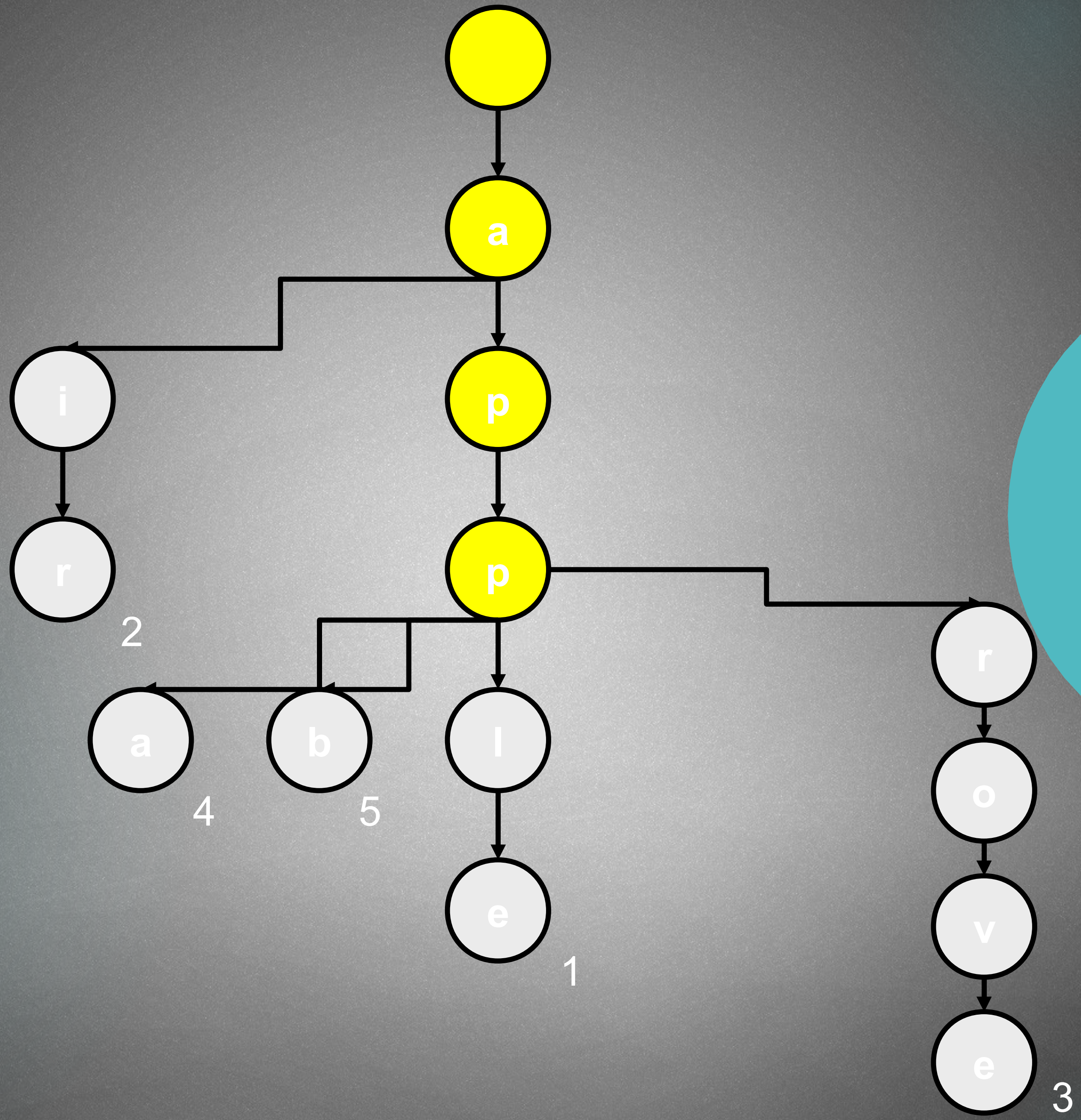
get(„apple”)



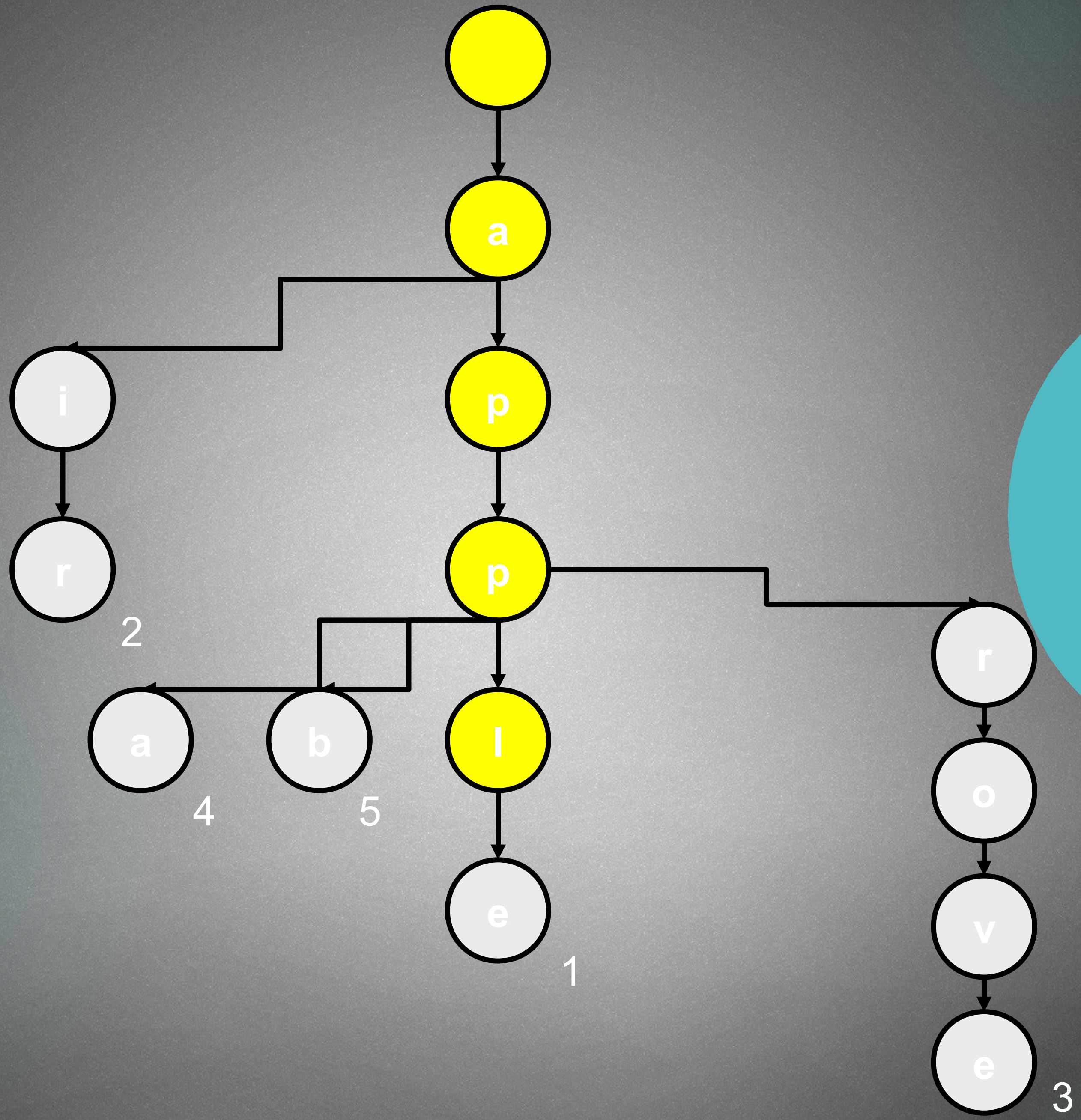
get(„**ap**ple”)



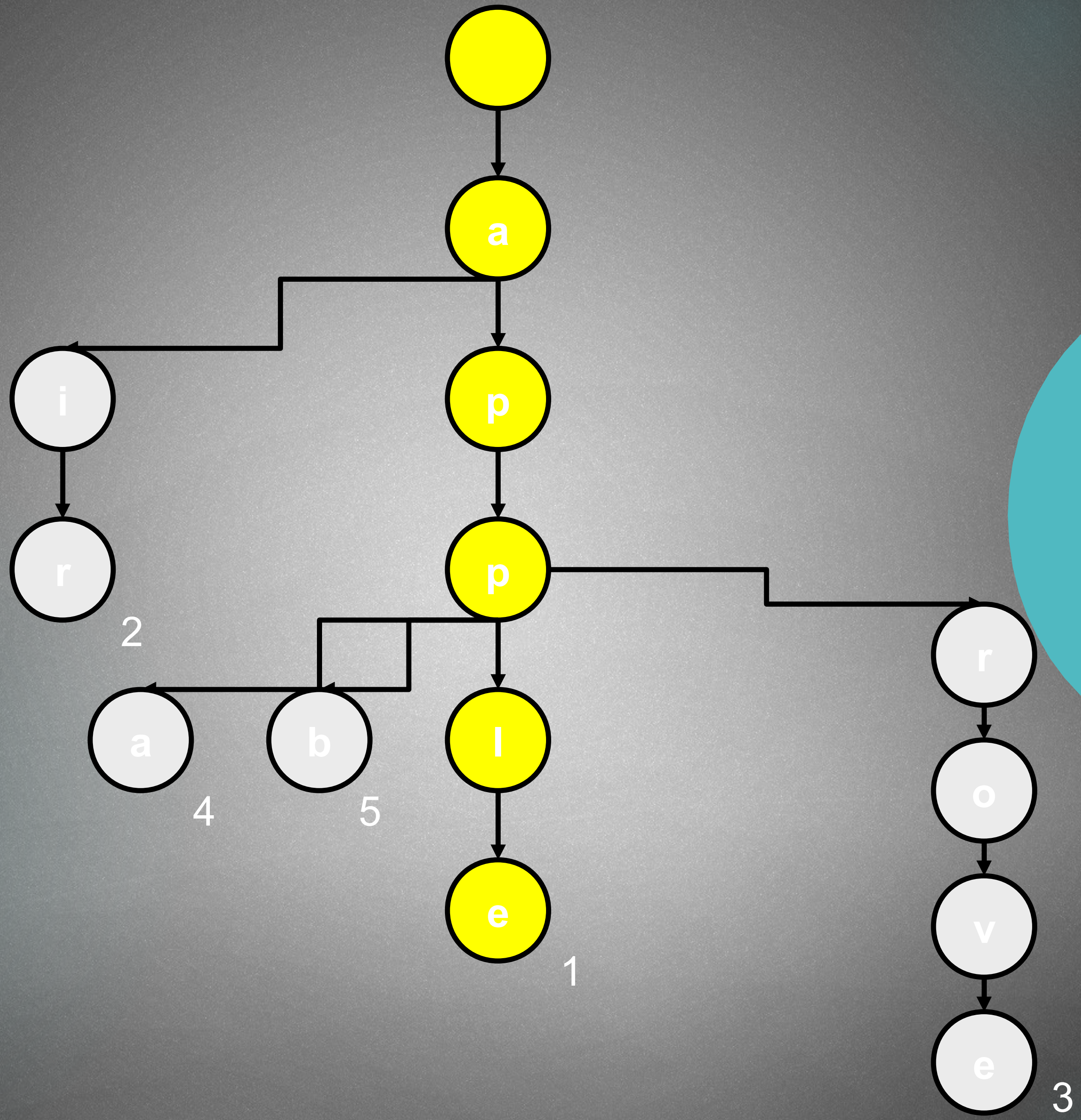
get(„apple”)



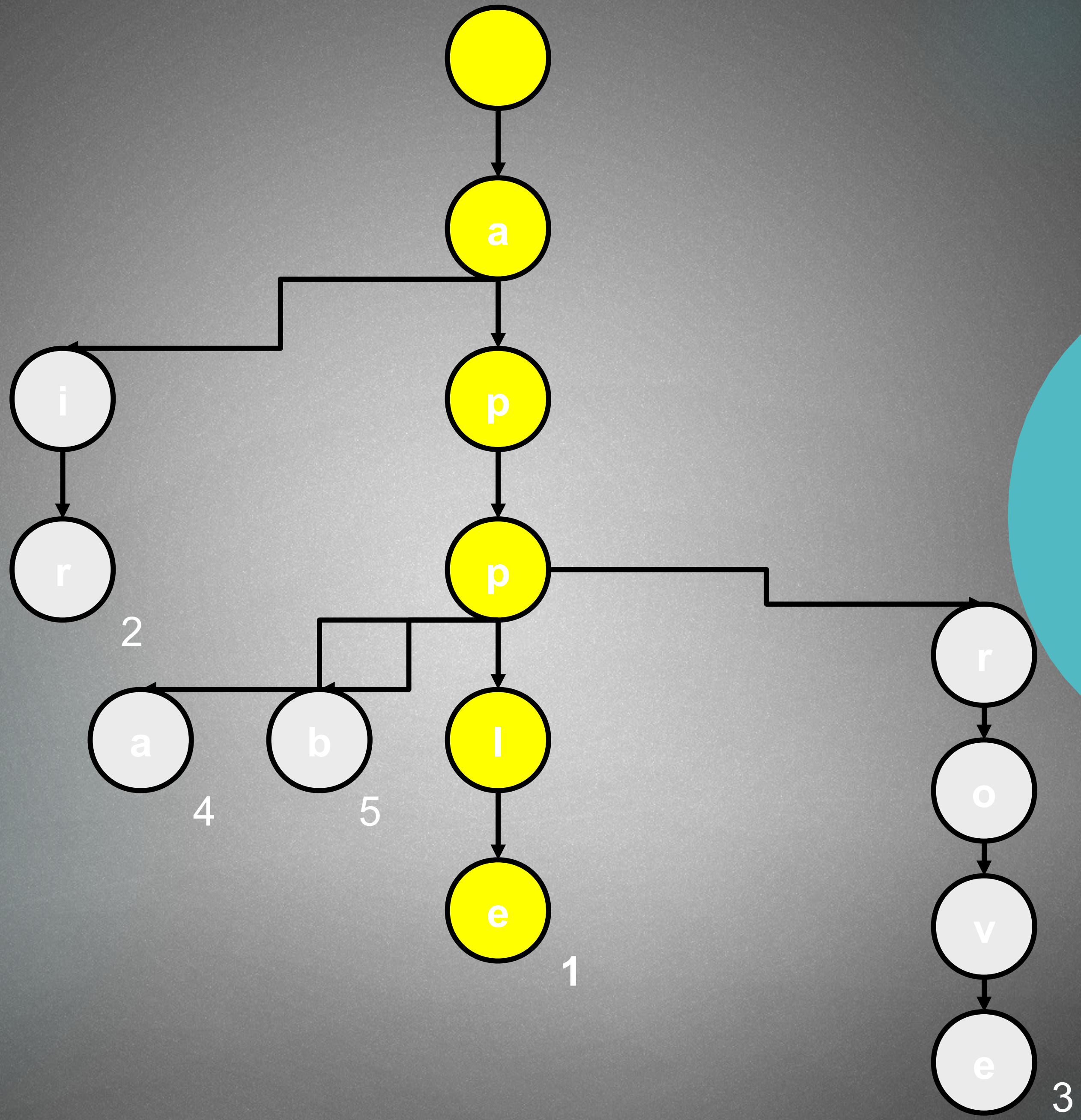
get(„**apple**”)

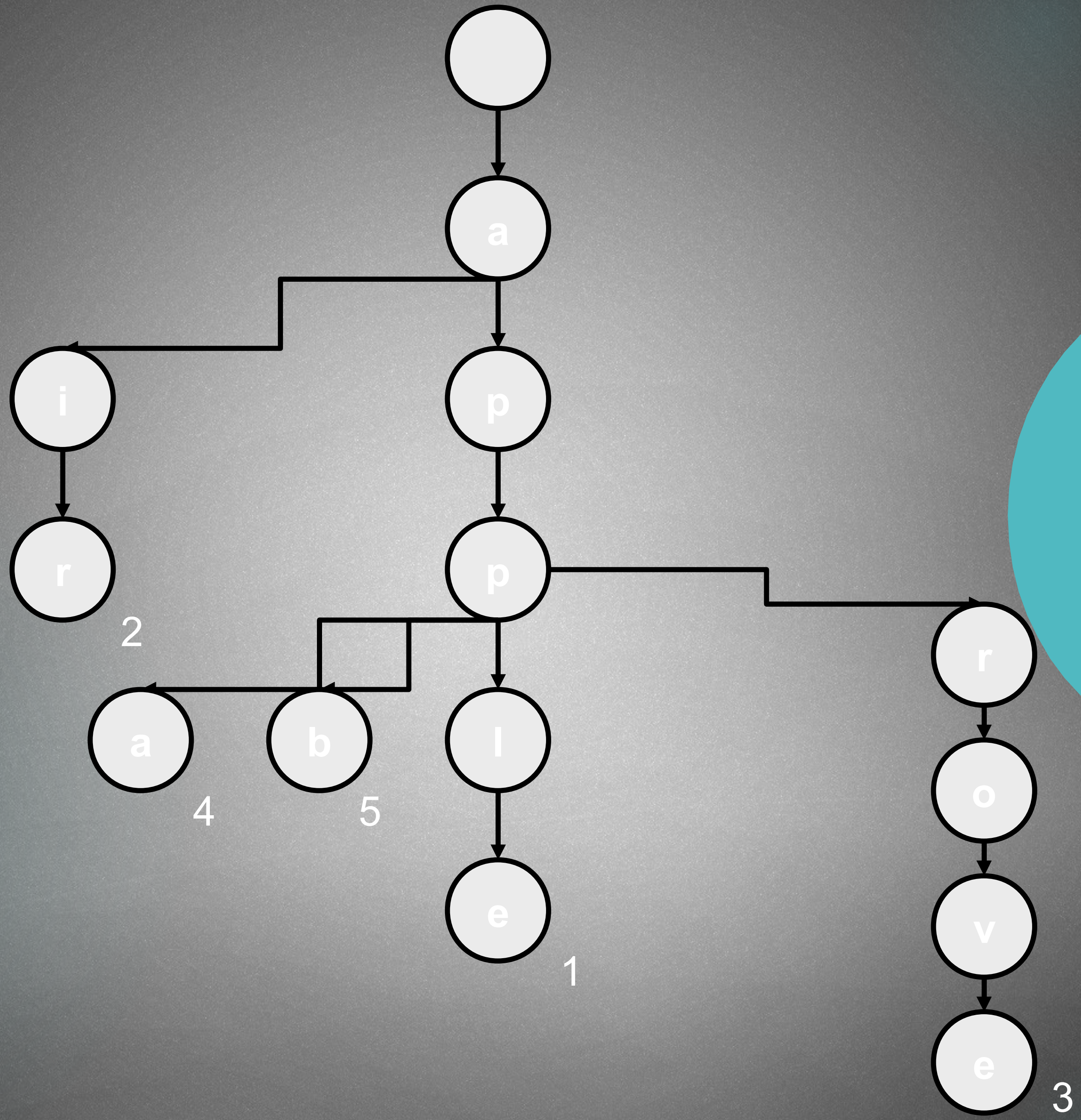


get(„**apple**”)



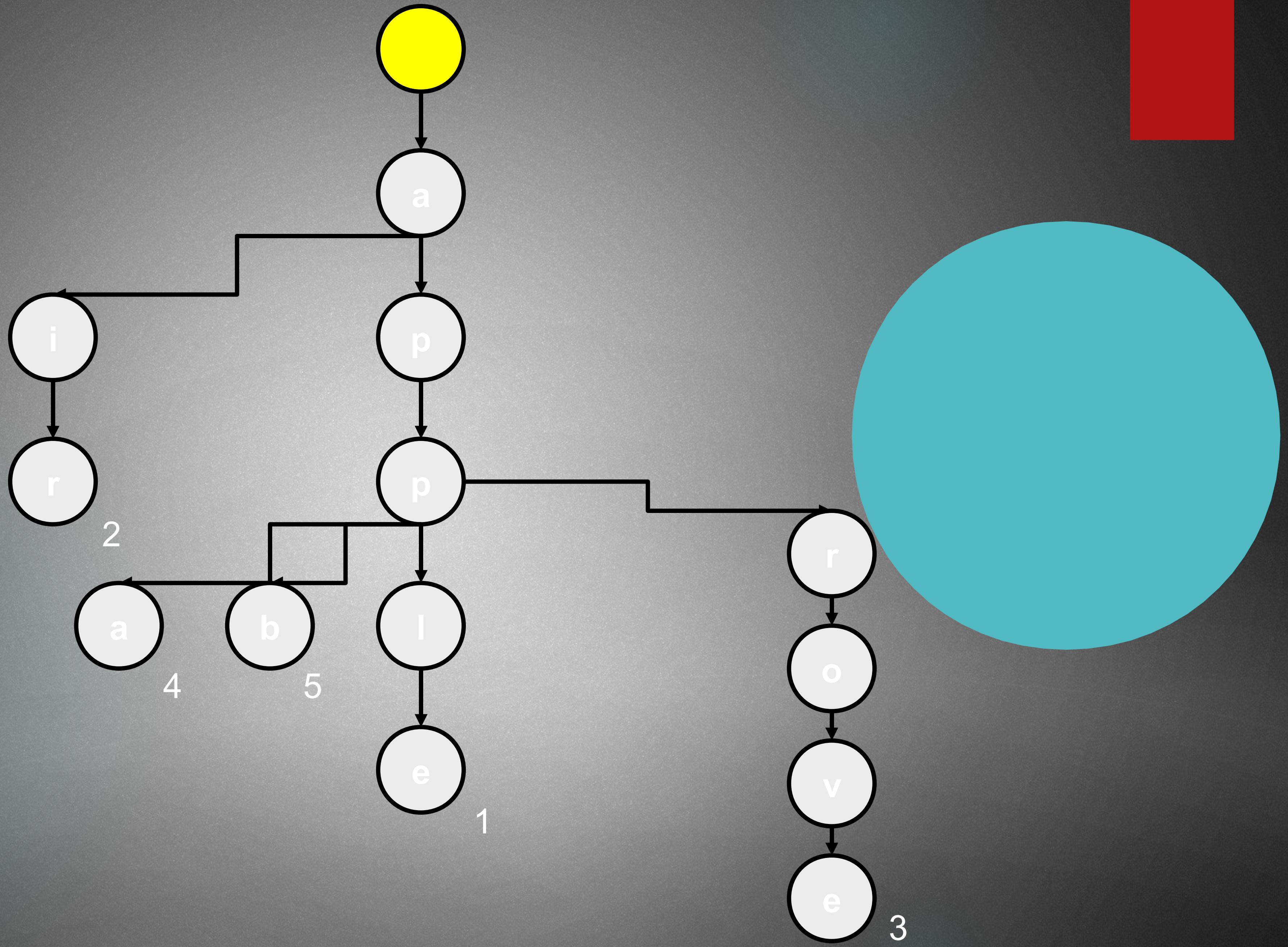
get(„**apple**")



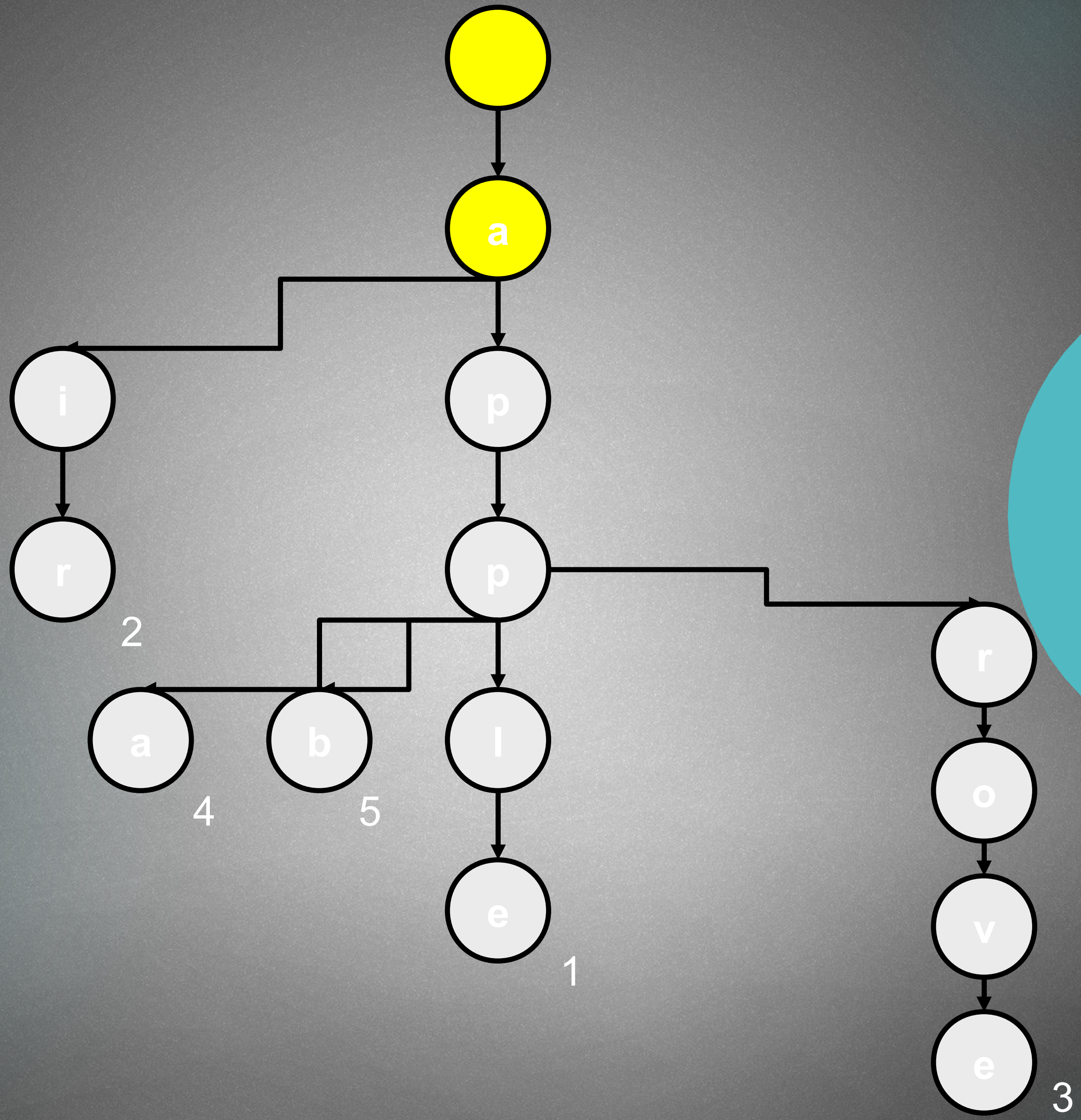


a
i
p
r
2
p
a
b
l
r
4
5
e
o
1
v
e
3

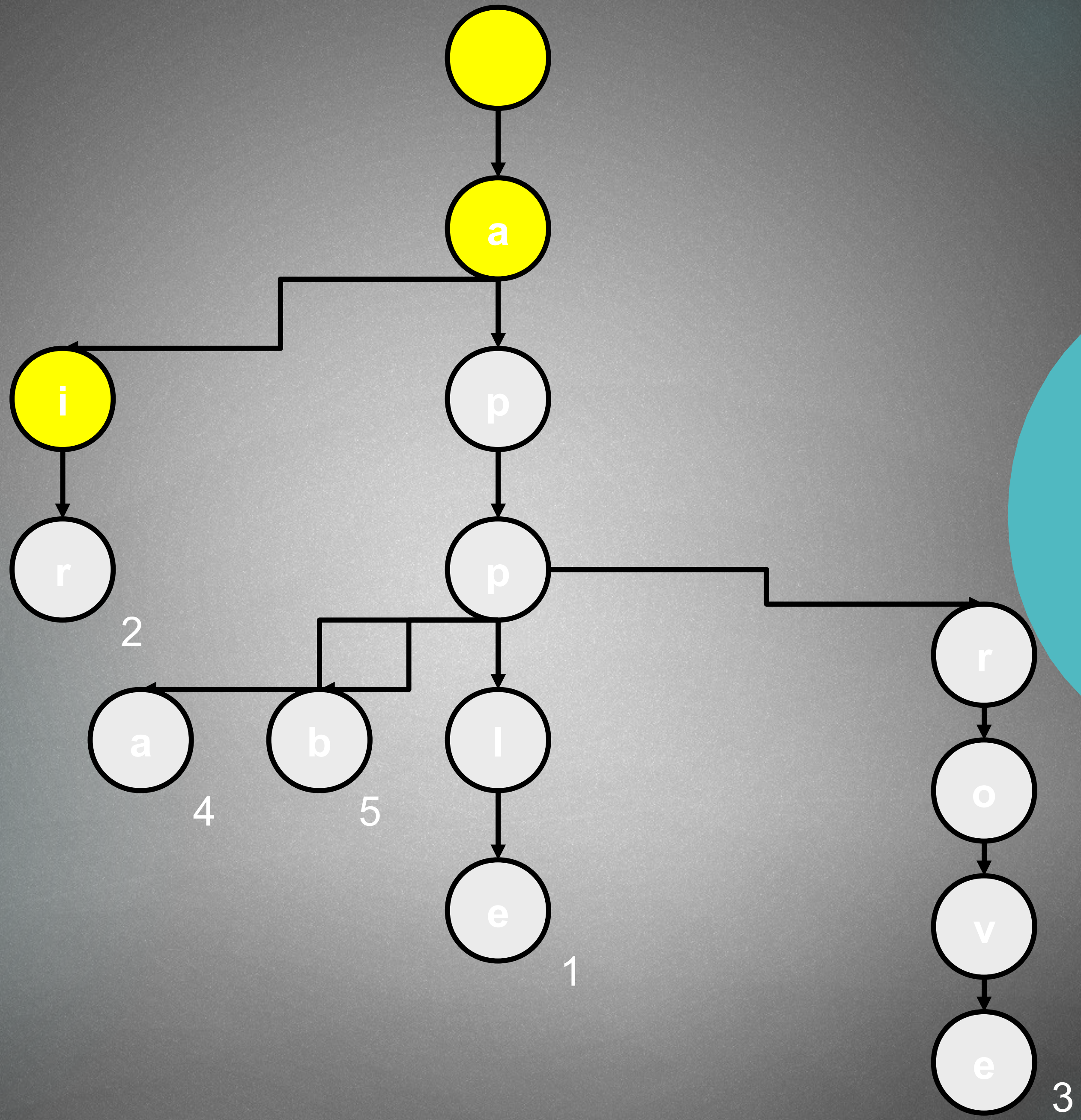
get(„air”)



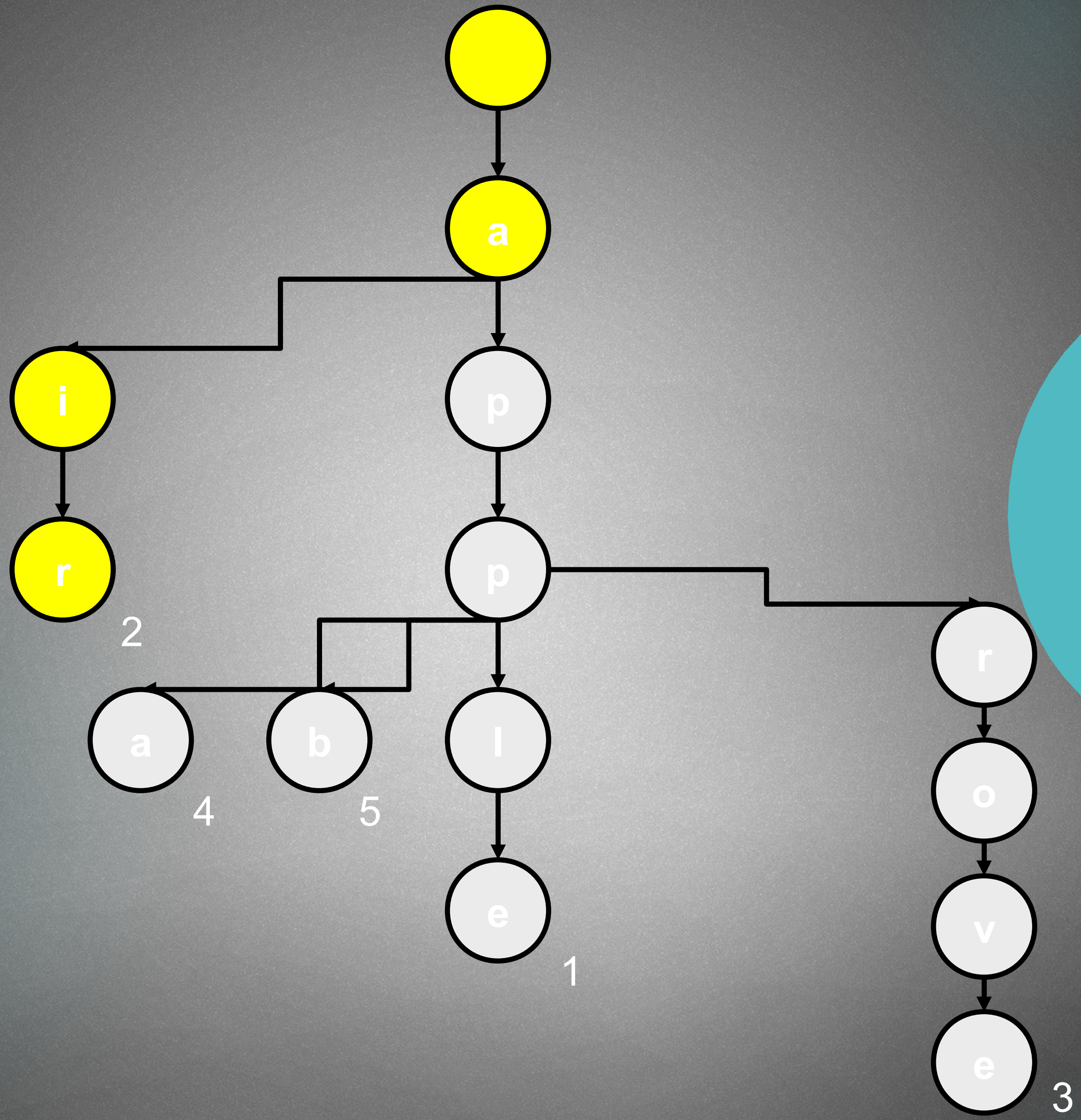
get(„air”)



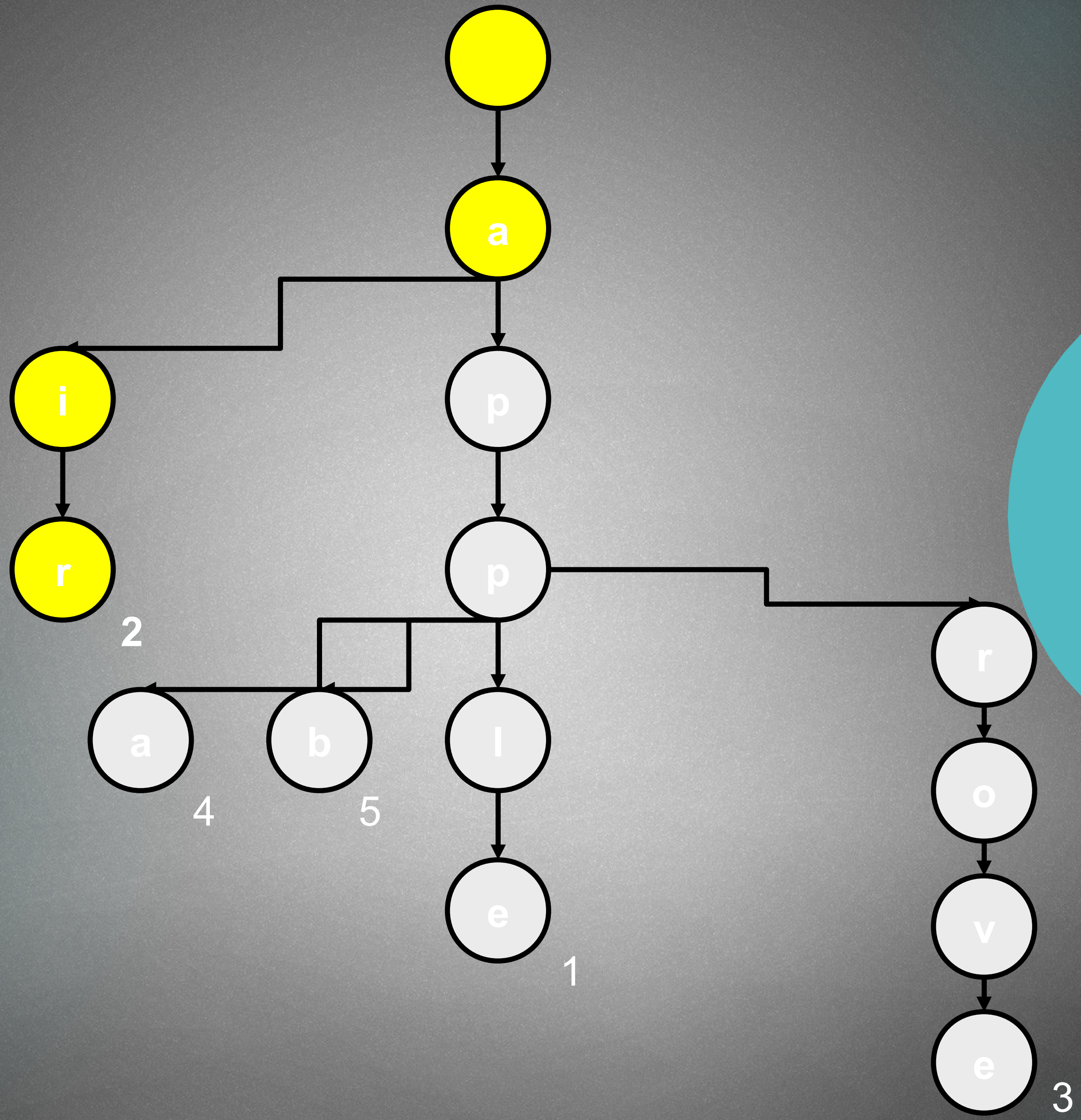
get(„**air**”)



get(„**air**”)



get(„**air**”)



Hashing VS tries

- We can replace hash tables with tries → tries are more effective for search misses

Hashing VS tries

- We can replace hash tables with tries → tries are more effective for search misses

Hash tables: the key is going to be converted into an array index
with the help of the hash function

For example: key is „apple” → the hash function considers
every single character in the key !!!

Tries: we consider every single character of the key „apple”
BUT we return right when there is a mismatch
// several times we just consider the few first
characters of the key

Hashing VS tries

- We can replace hash tables with tries → tries are more effective for search misses
- So it is faster to use tries in the worst case

Hashing VS tries

- We can replace hash tables with tries → tries are more effective for search misses
- So it is faster to use tries in the worst case

Hash tables: we end up with a linked list, so searching is with **O(N)** running time in this case
N: number of items in the hash table

Tries: worst case is that we have to consider every character of the key → it is **O(m)** complexity
m: length of the key

USUALLY N >> m !!!

Hashing VS tries

- We can replace hash tables with tries → tries are more effective for search misses
- So it is faster to use tries in the worst case
- For tries there are no collisions !!!

Hashing VS tries

- We can replace hash tables with tries → tries are more effective for search misses
- So it is faster to use tries in the worst case
- For tries there are no collisions !!!
- Tries can provide sorting → so alphabetical ordering of the entries by keys !! ~ hash tables does not
- No hash function needed for tries and designing a perfect hash function is a very complex task

Hashing VS tries

- Tries may be slower than hash tables
- Searching on secondary storage (for example HDD hard drive disk)
- Random-access time is high compared to main memory
- Hash tables → we just search by the indexes once when the index is generated by the hash function
- Tries → there is a random-access every time we consider the next character in the key
- Sometimes tries need more memory: a memory chunk is allocated for every single character with tries BUT for hash tables there is just a single chunk of memory !!!

Applications

Predictive text it is sort of an input technology

On smart phones it is quite popular

~ each key press results in a prediction: the predictions come from tries

Autocomplete we start typing – for example - in the browser

and the suggestions are going to be appeared

~ a trie may be the underlying data structure, and the suggestions are the entries with same prefixes !!!

Spell checking a trie data structure can be used as a spell checker

First we have to construct the trie with the words

~ after the construction → we are able to check whether that given string is present in the trie or not

- so we can check whether the given word is spelled correctly or not
- we can even suggest that what may be the correctly spelled word

TERNARY SEARCH TREES

tst

- With the help of tries we can search and sort strings very very efficiently
- The problem is that tries consume a lot of memory, so we should use ternary search trees instead which stores less references and null objects
- TST stores characters or strings in nodes
- Each node has 3 children: less (lower child), equal (middle child) or greater (higher child)
- Can we balance TST-s with rotations? Yes, but it does not worth the trouble
- It can be used instead of hashmap: it is as efficient as hashing
- Hashing need to examine the entire string key ... TST does not

- TST support sorting operation !!!
- So: TST is better than hashing → especially for search misses + flexible than BST (usually there is no perfect hash function)
- Conclusion: TST is faster than hashmap and more flexible than binary search trees

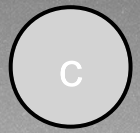
put: with this operation we would like to insert a new element into the ternary search tree with a given key

- if the character is smaller alphabetically: we go to the left
- if the character is equal: we go to the middle
- if the character is greater alphabetically: go right

put: with this operation we would like to insert a new element into the ternary search tree with a given key

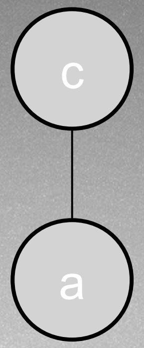
put(„cat”,23)

put: with this operation we would like to insert a new element into the ternary search tree with a given key



put(„cat”,23)

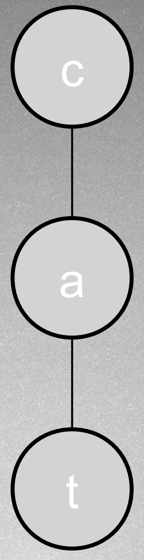
put: with this operation we would like to insert a new element into the ternary search tree with a given key



put(„cat”,23)

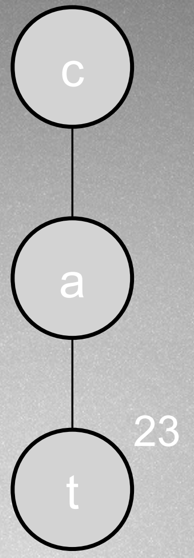
put: with this operation we would like to insert a new element into the ternary search tree with a given key

put(„cat”,23)



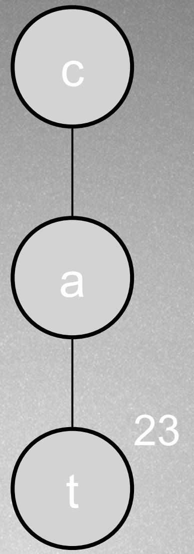
put: with this operation we would like to insert a new element into the ternary search tree with a given key

put(„cat”,23)



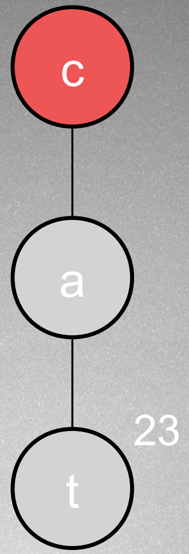
put: with this operation we would like to insert a new element into the ternary search tree with a given key

put(„apple”,46)



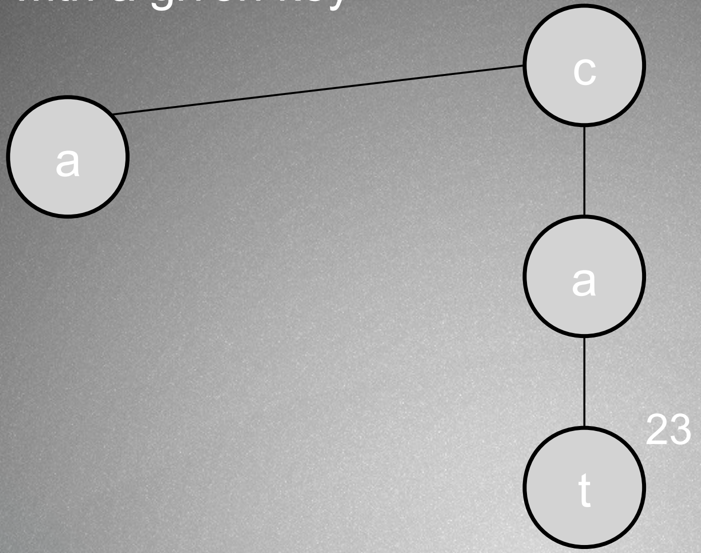
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„apple”,46)



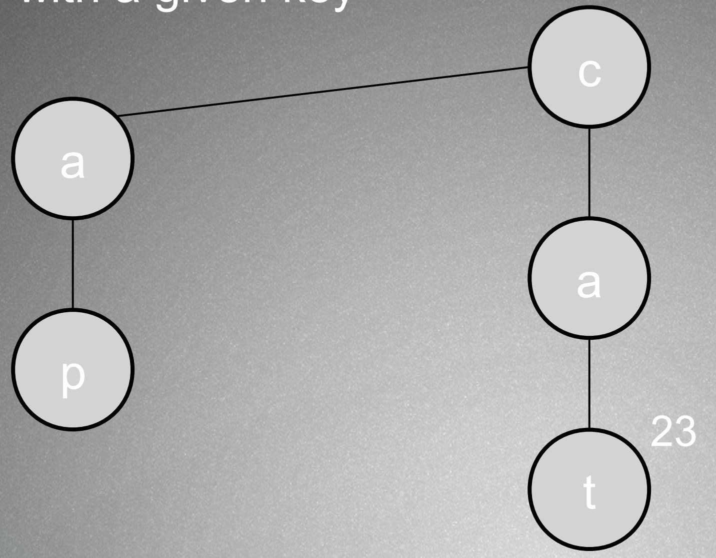
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„apple”,46)



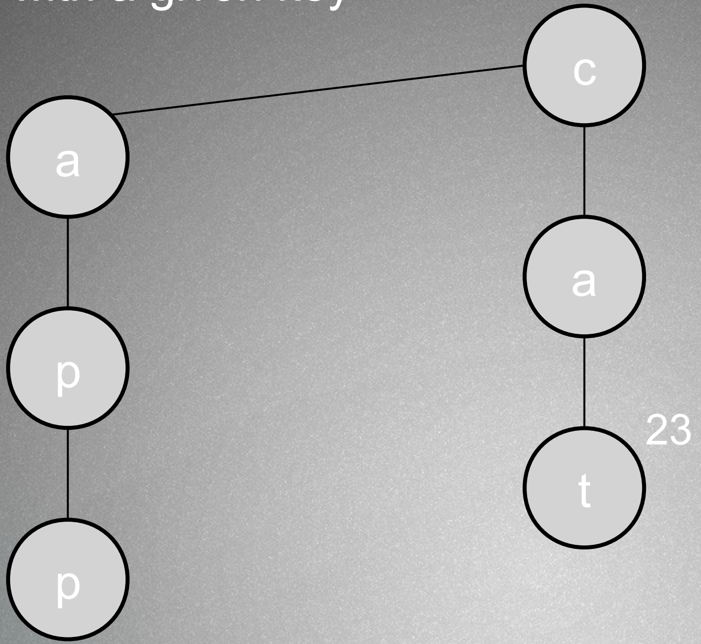
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„apple”,46)



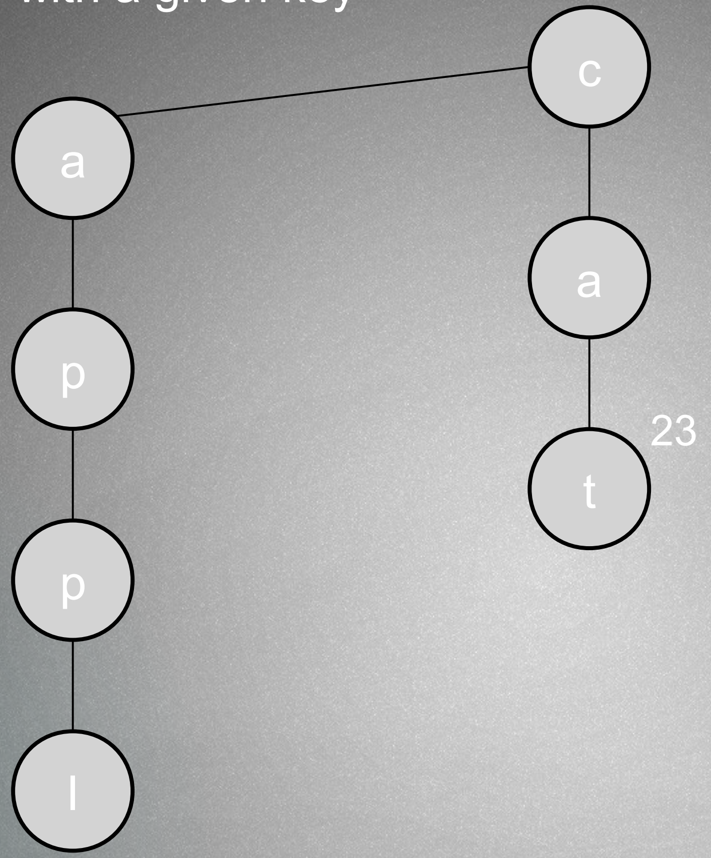
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„apple”,46)



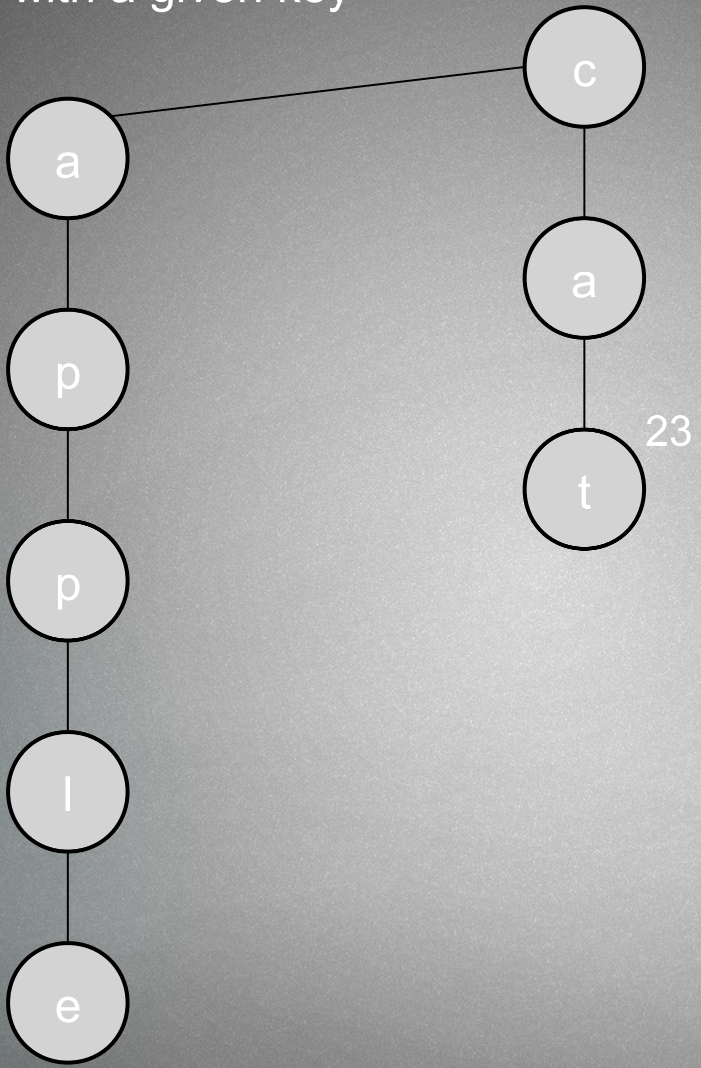
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„apple”,46)

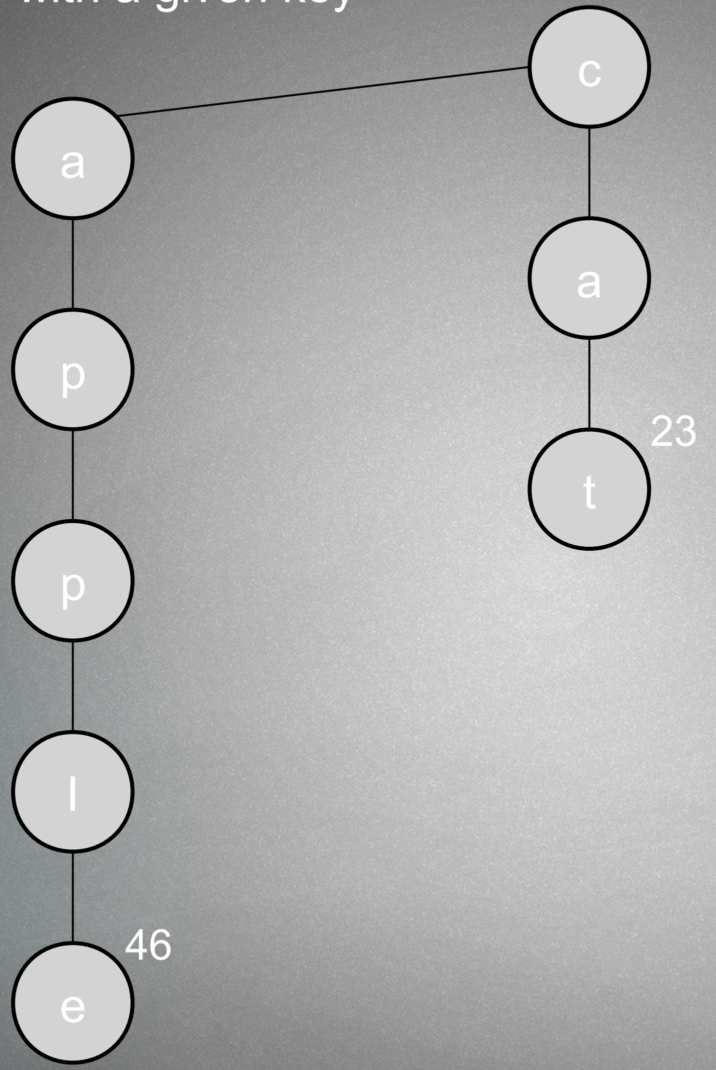


put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„apple”,46)

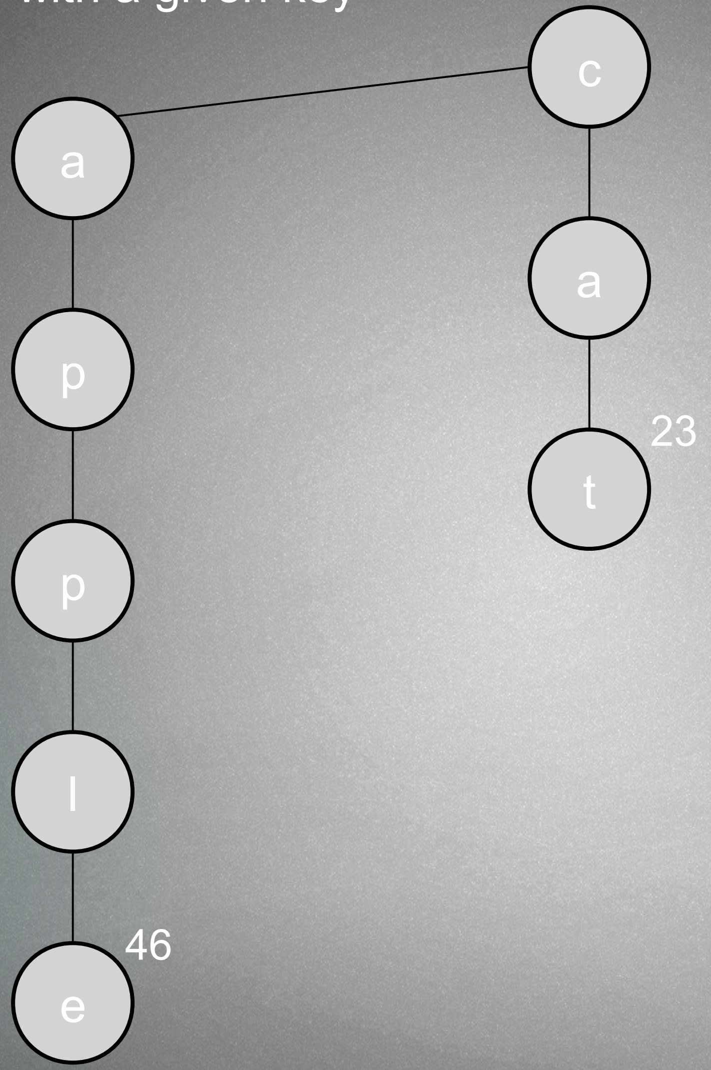


put: with this operation we would like to insert a new element to the ternary search tree with a given key



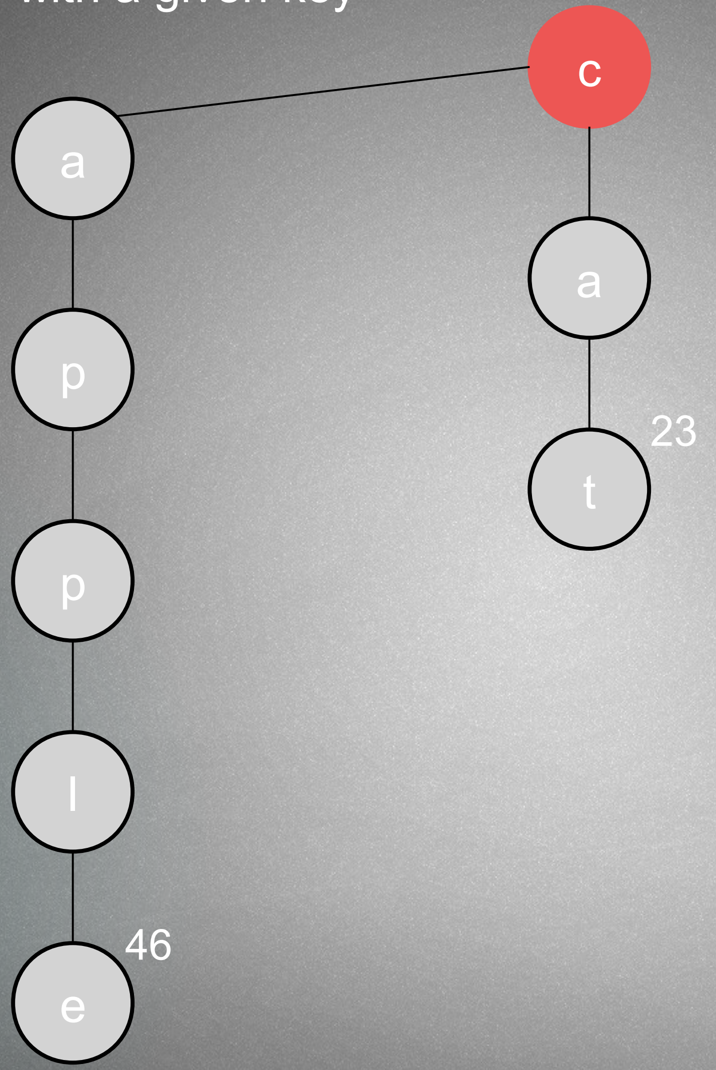
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„car”,6)



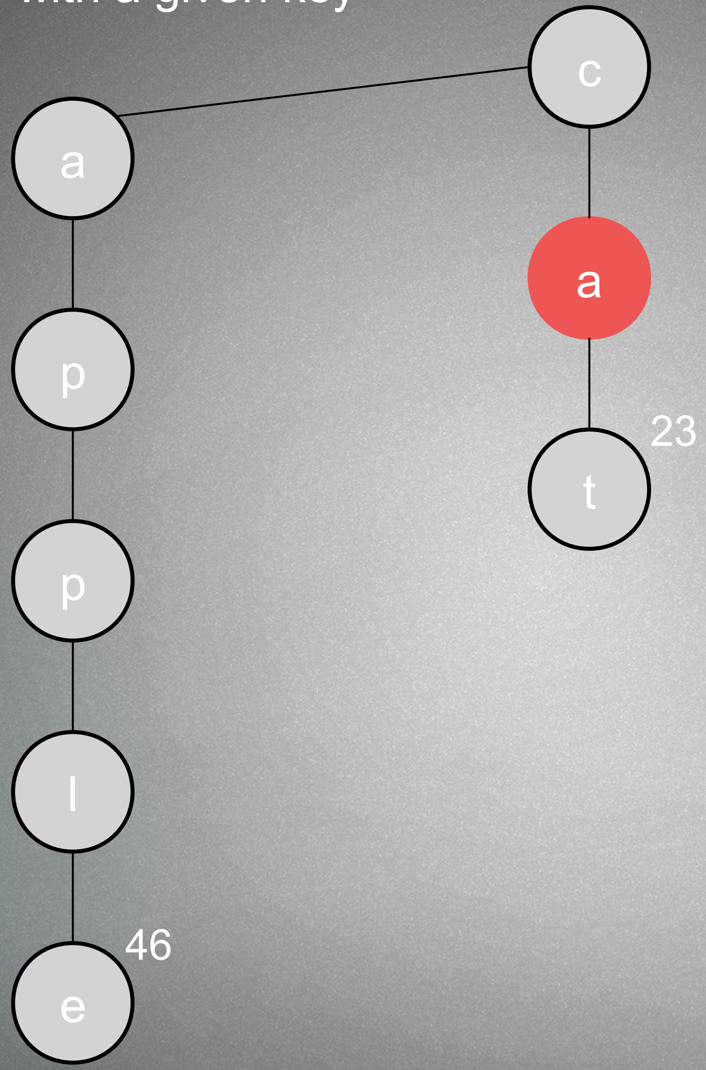
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„**car**”,6)



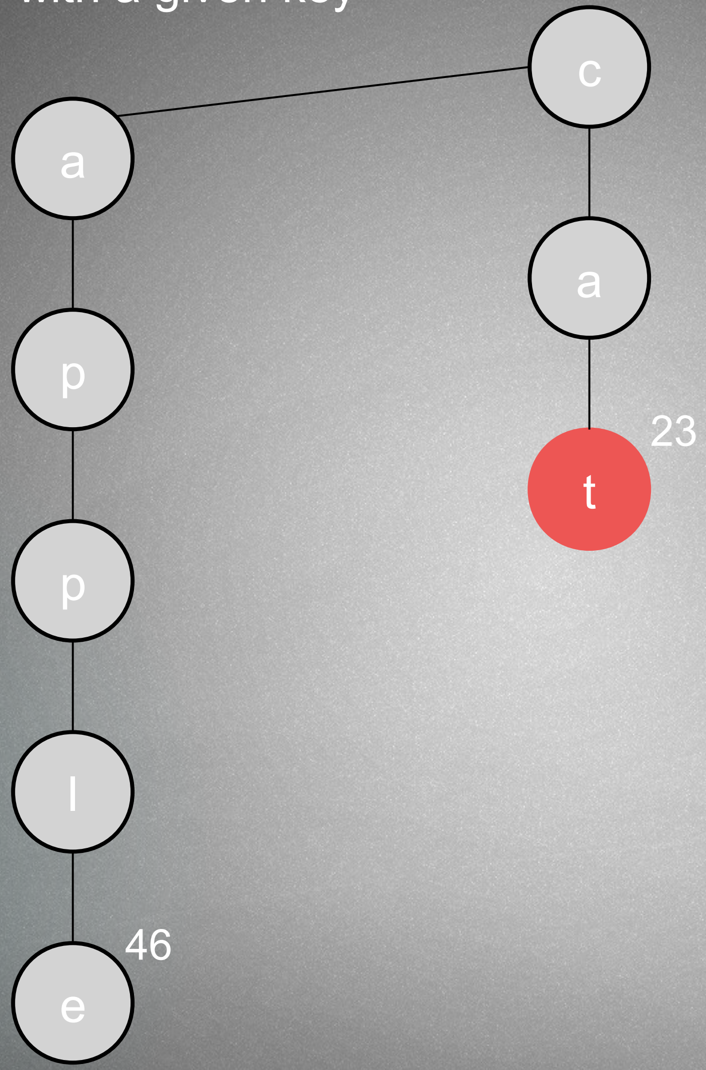
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„car”,6)



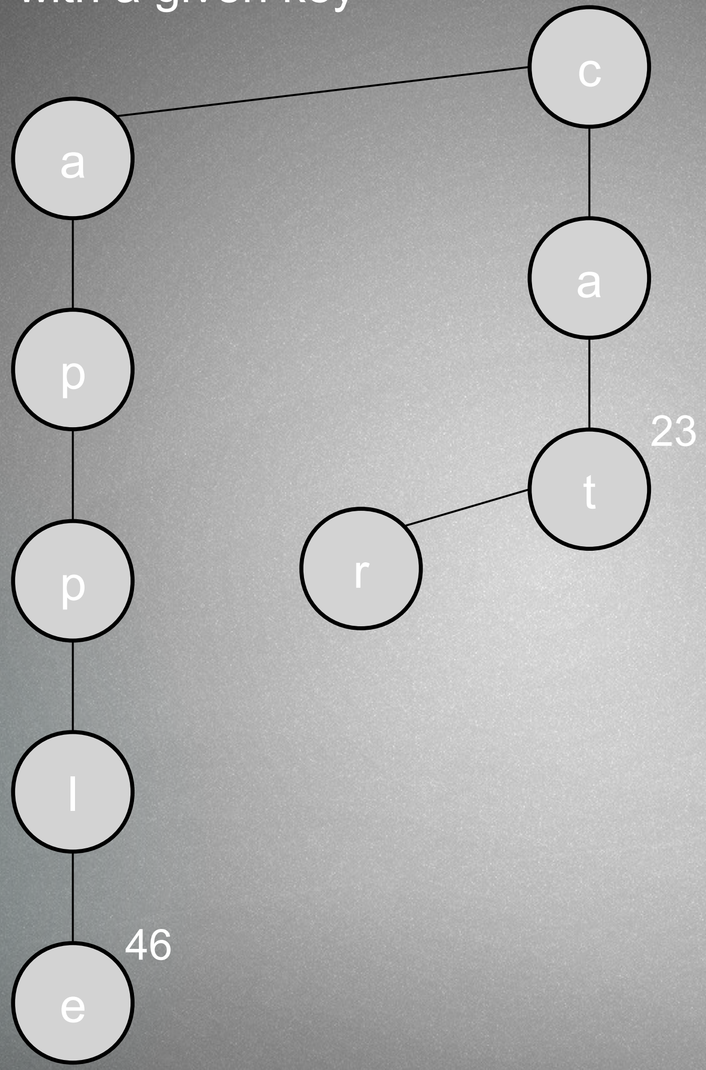
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„car”,6)

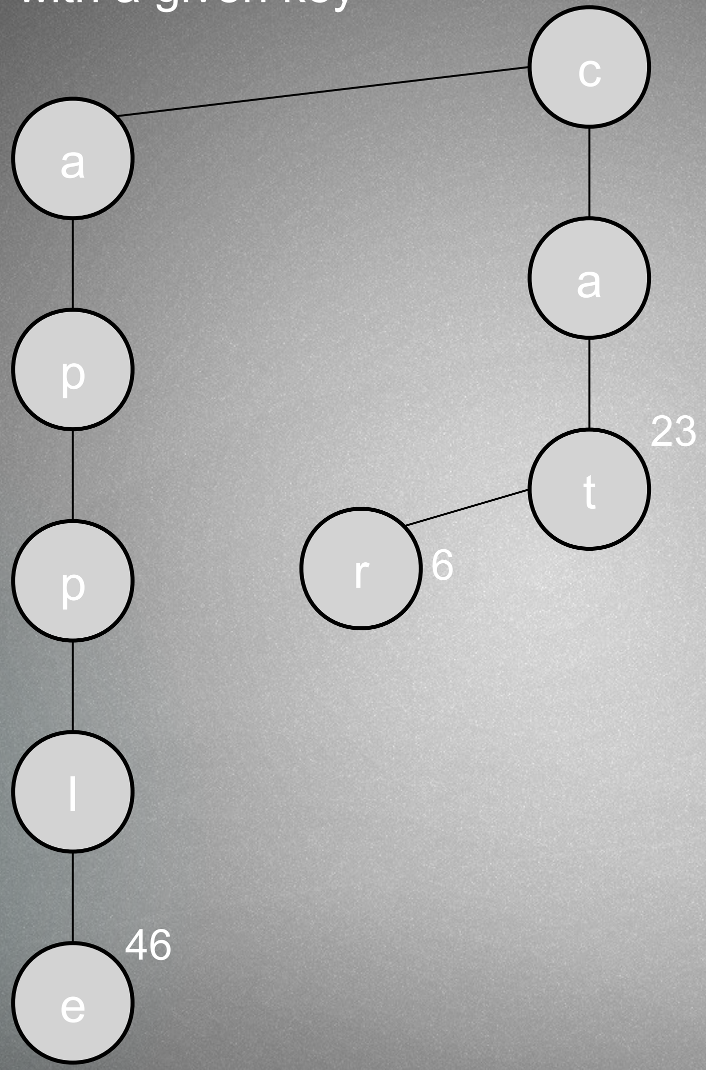


put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„car”,6)

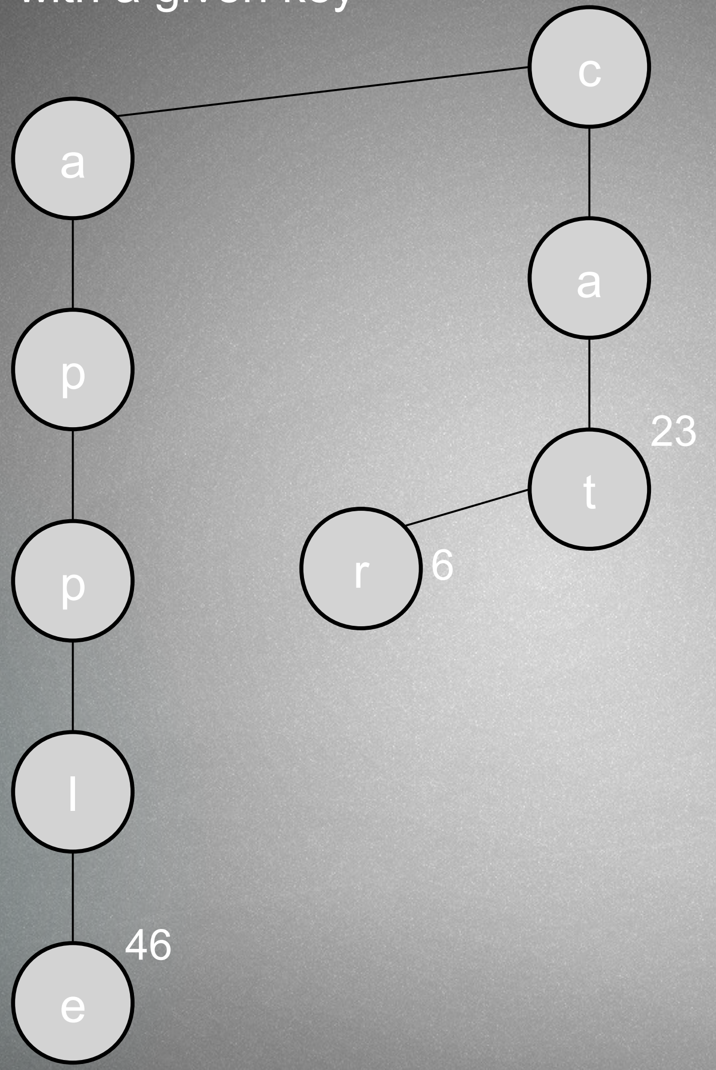


put: with this operation we would like to insert a new element to the ternary search tree with a given key



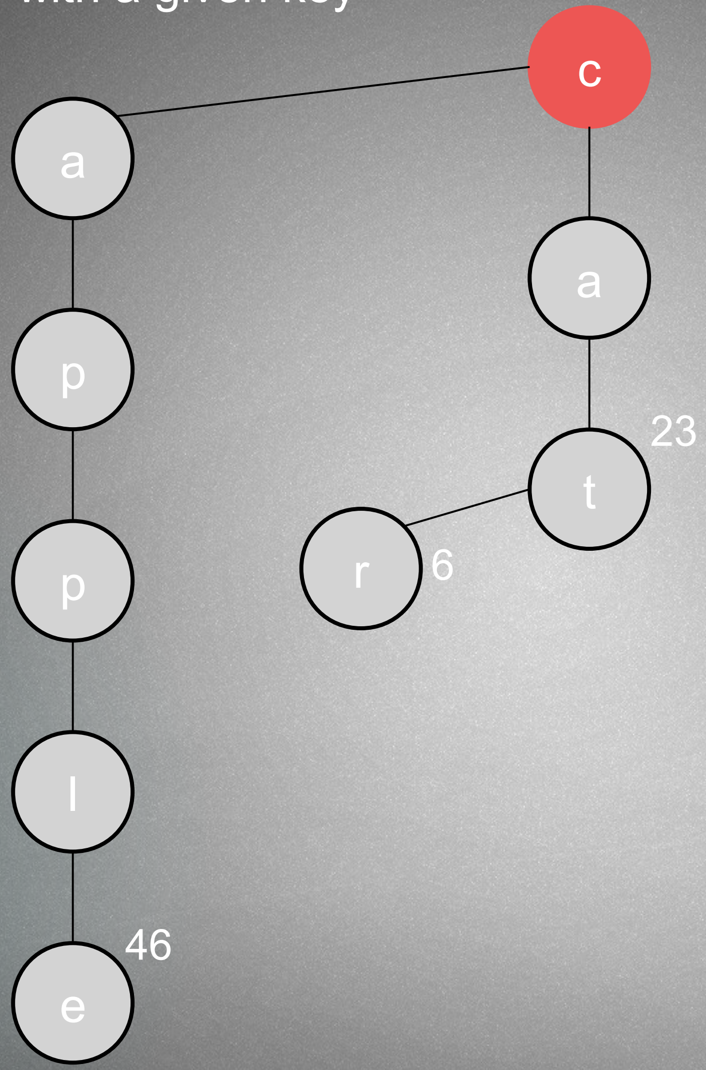
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„carrot”,68)



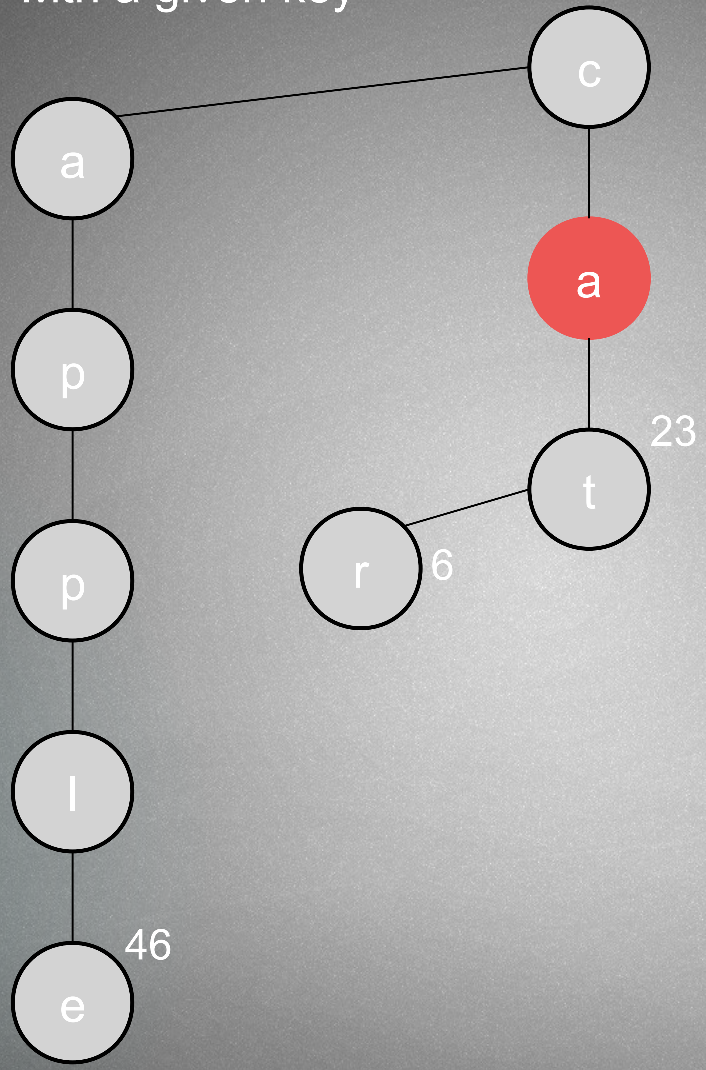
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„**c**arrot”,68)



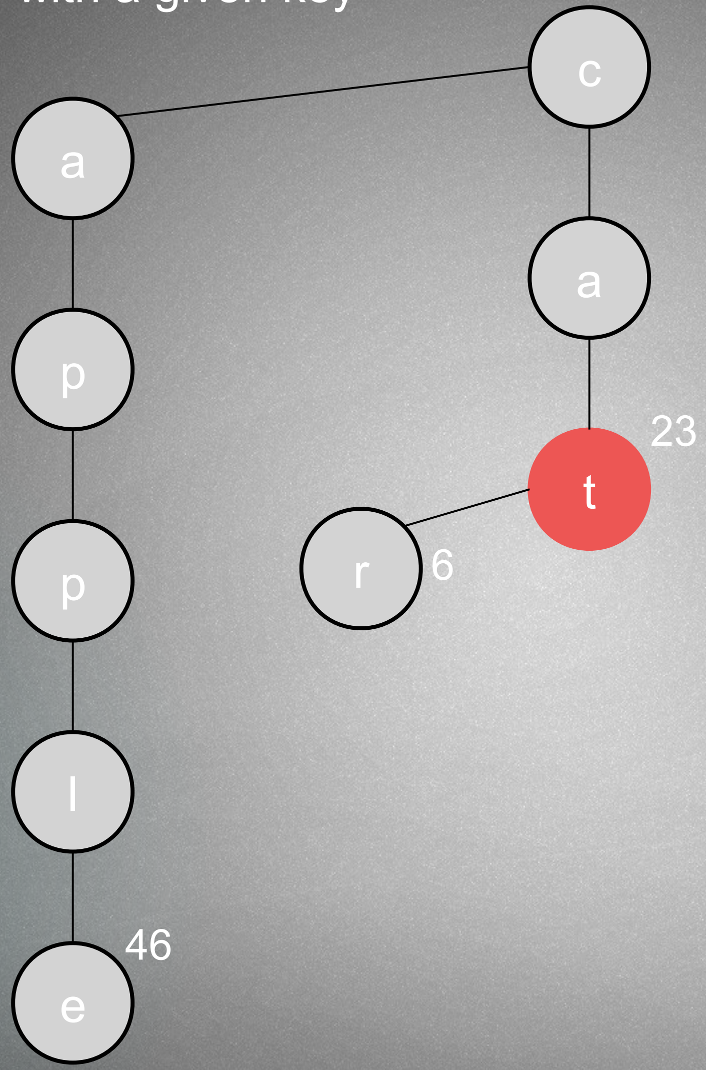
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„carrot”,68)



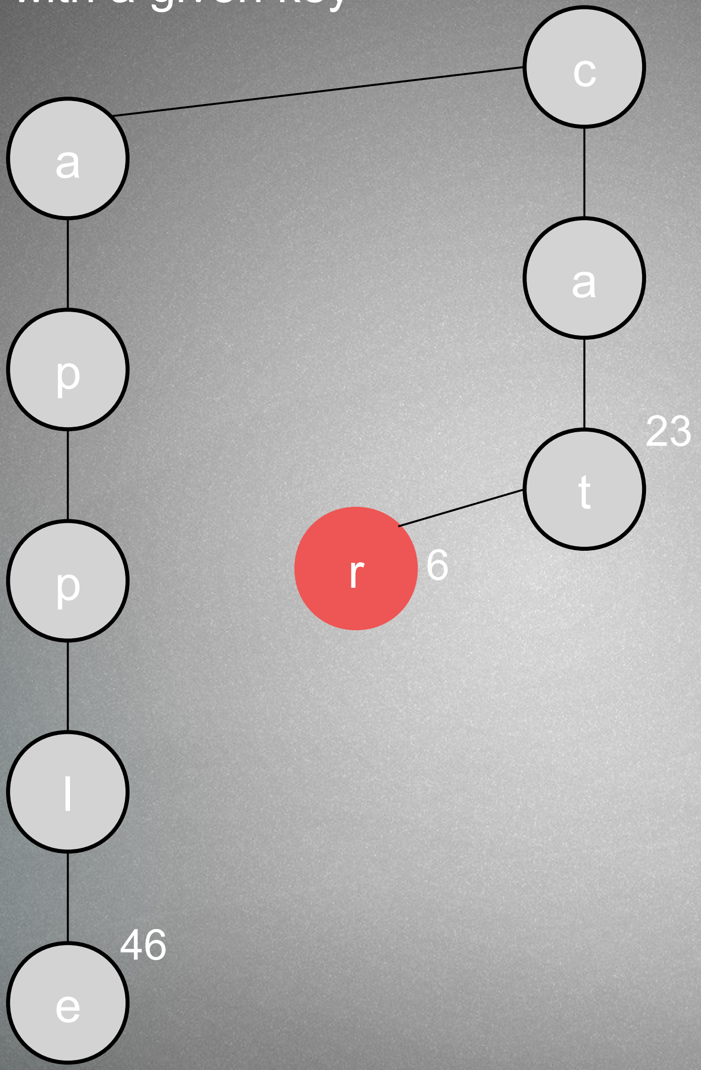
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„carrot”,68)



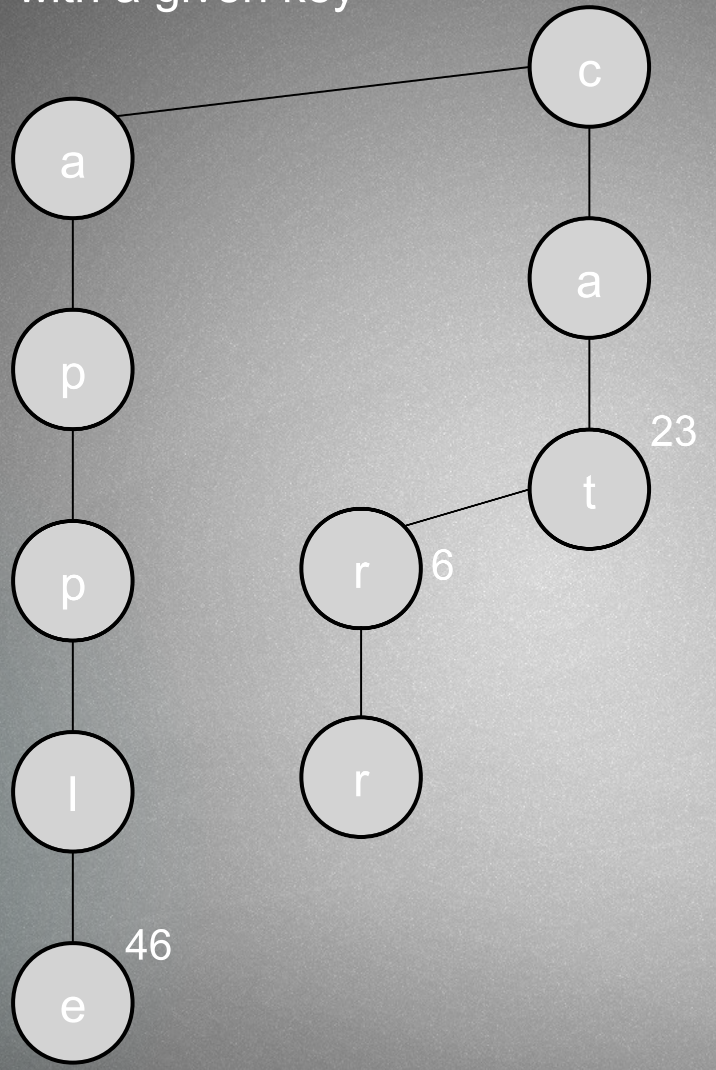
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„carrot”,68)



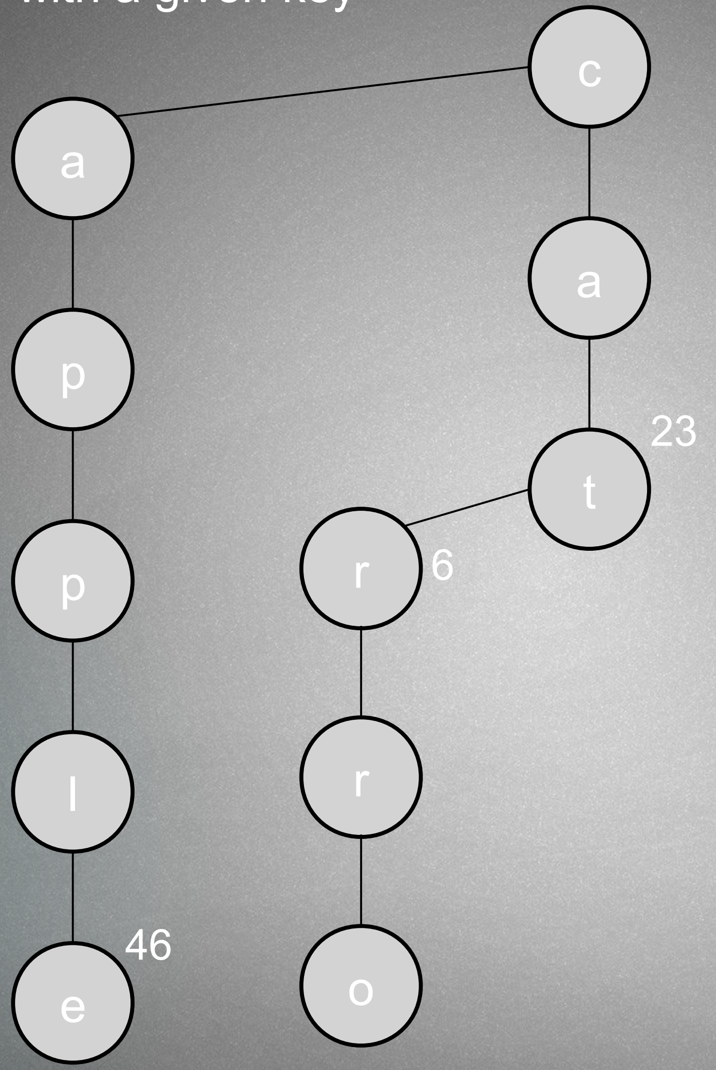
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„carrot”,68)



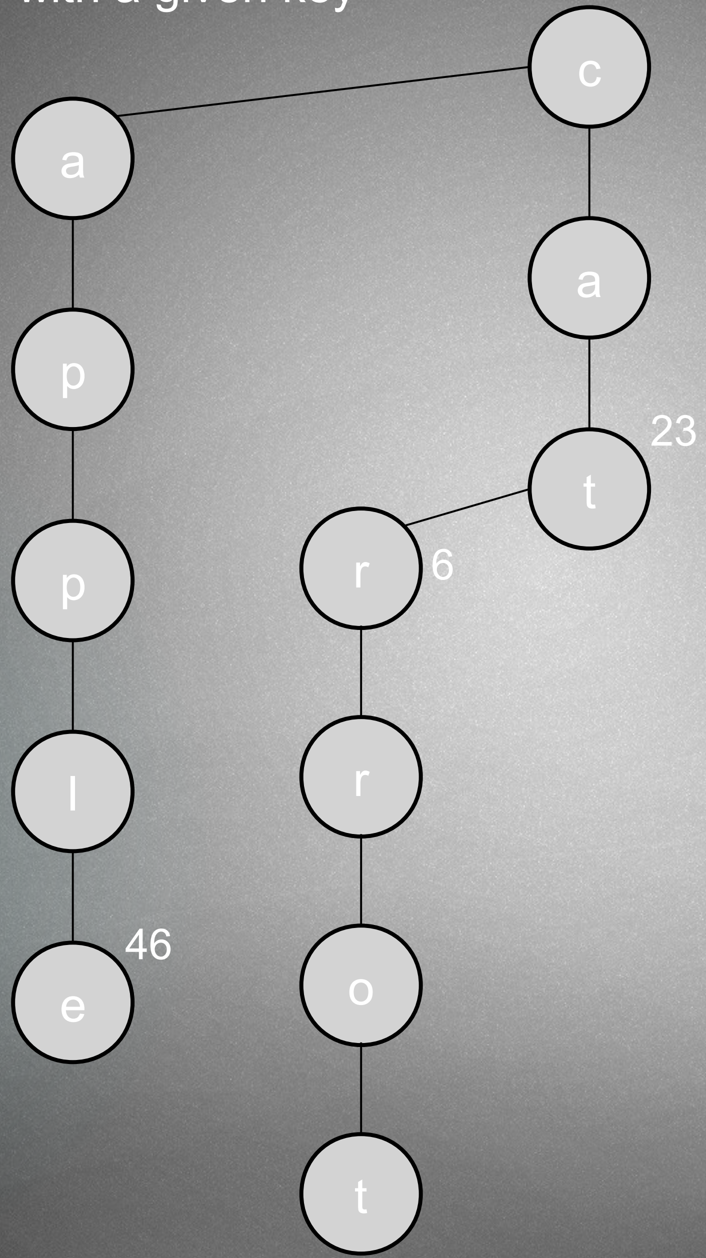
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„carrot”,68)

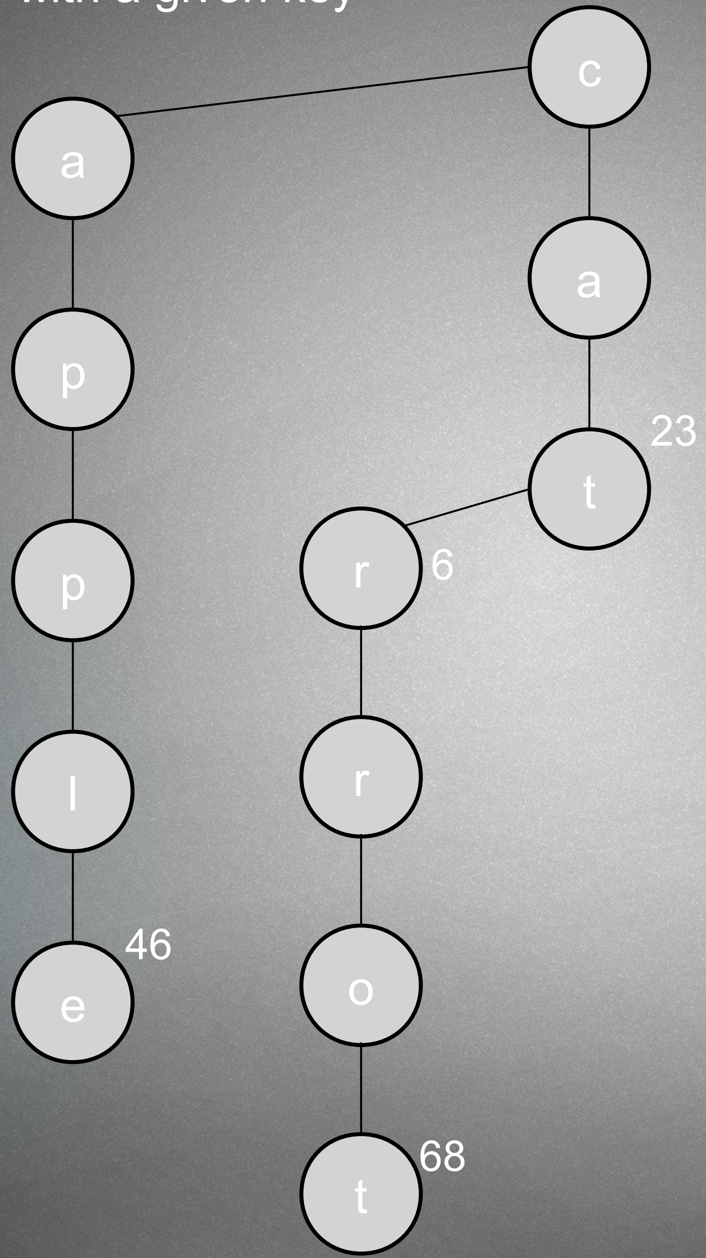


put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„carrot”,68)

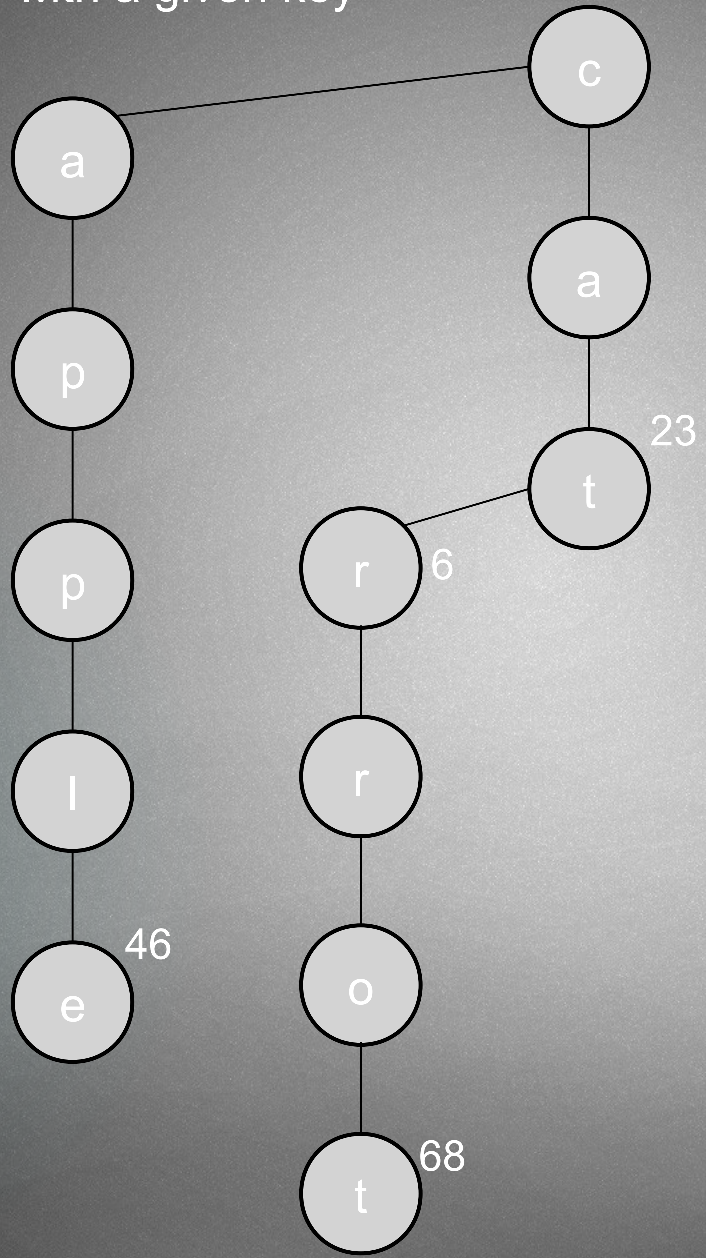


put: with this operation we would like to insert a new element to the ternary search tree with a given key



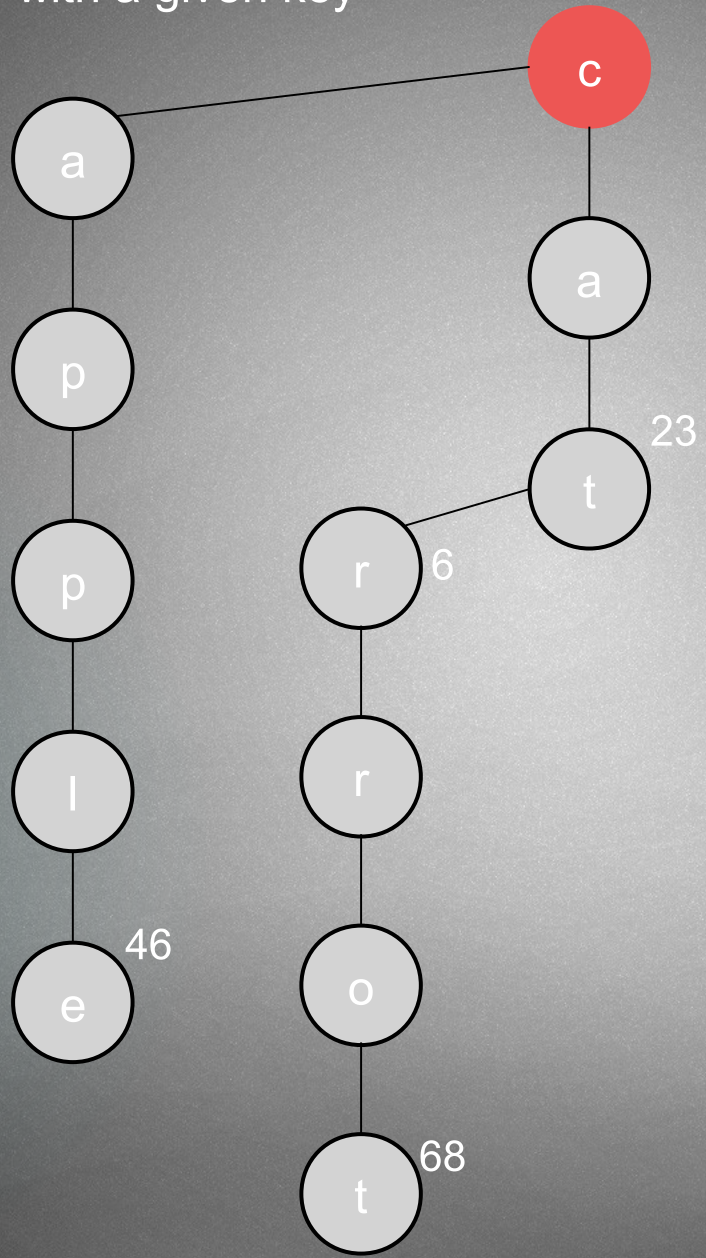
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„cow”,112)



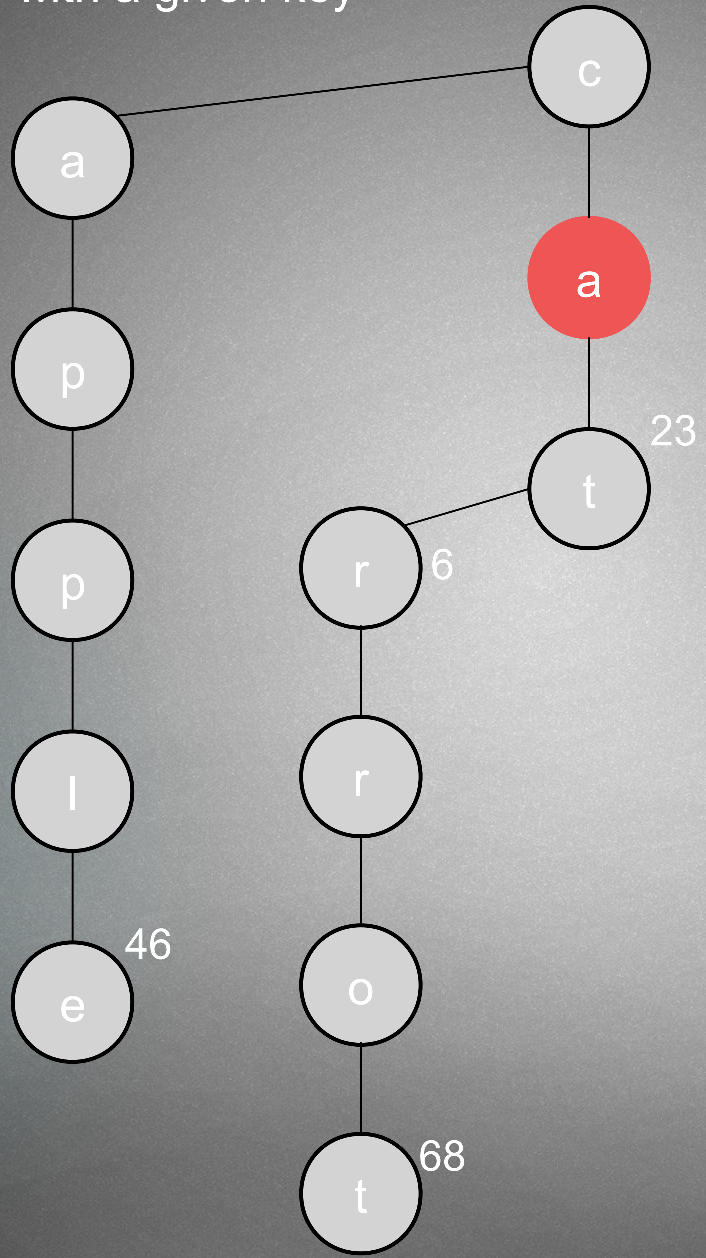
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„**c**ow”,112)



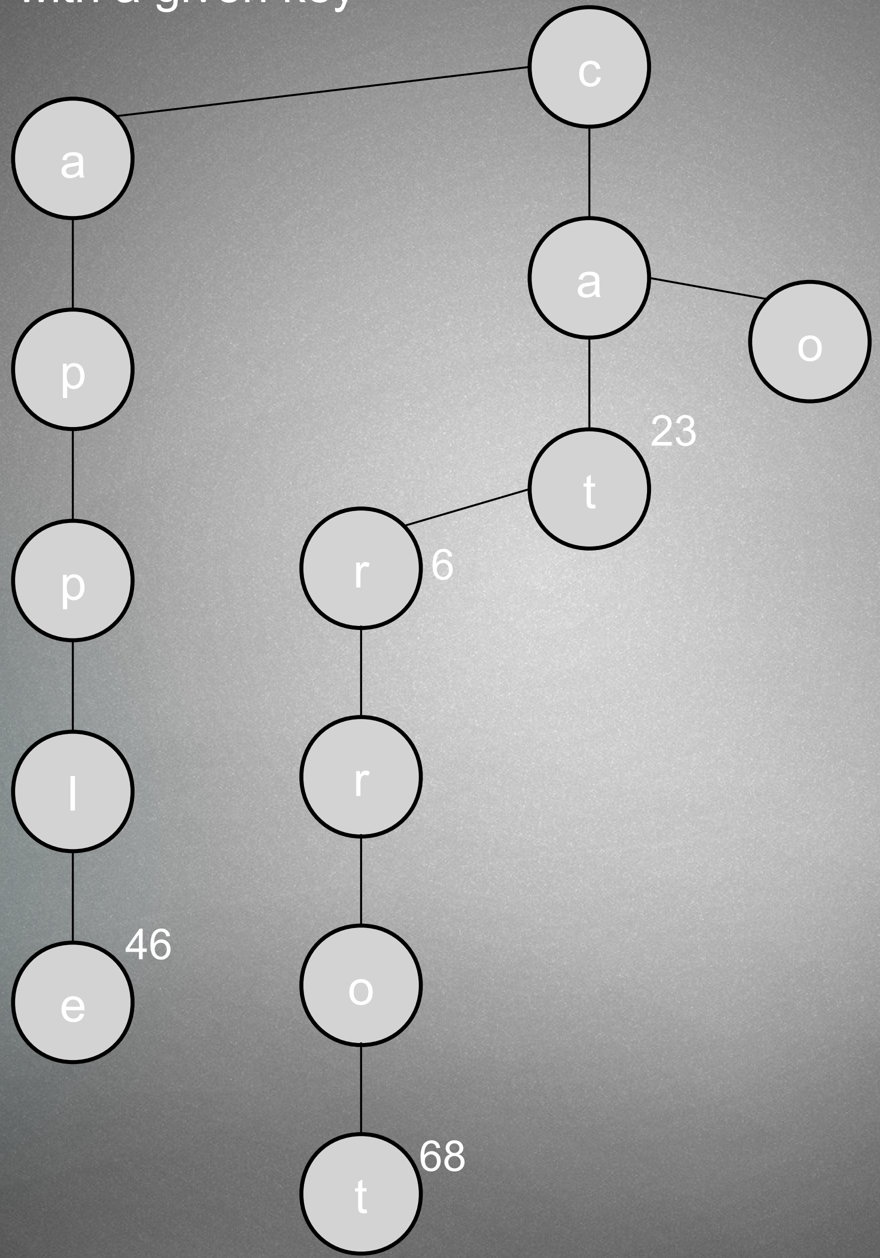
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„c**o**w”,112)



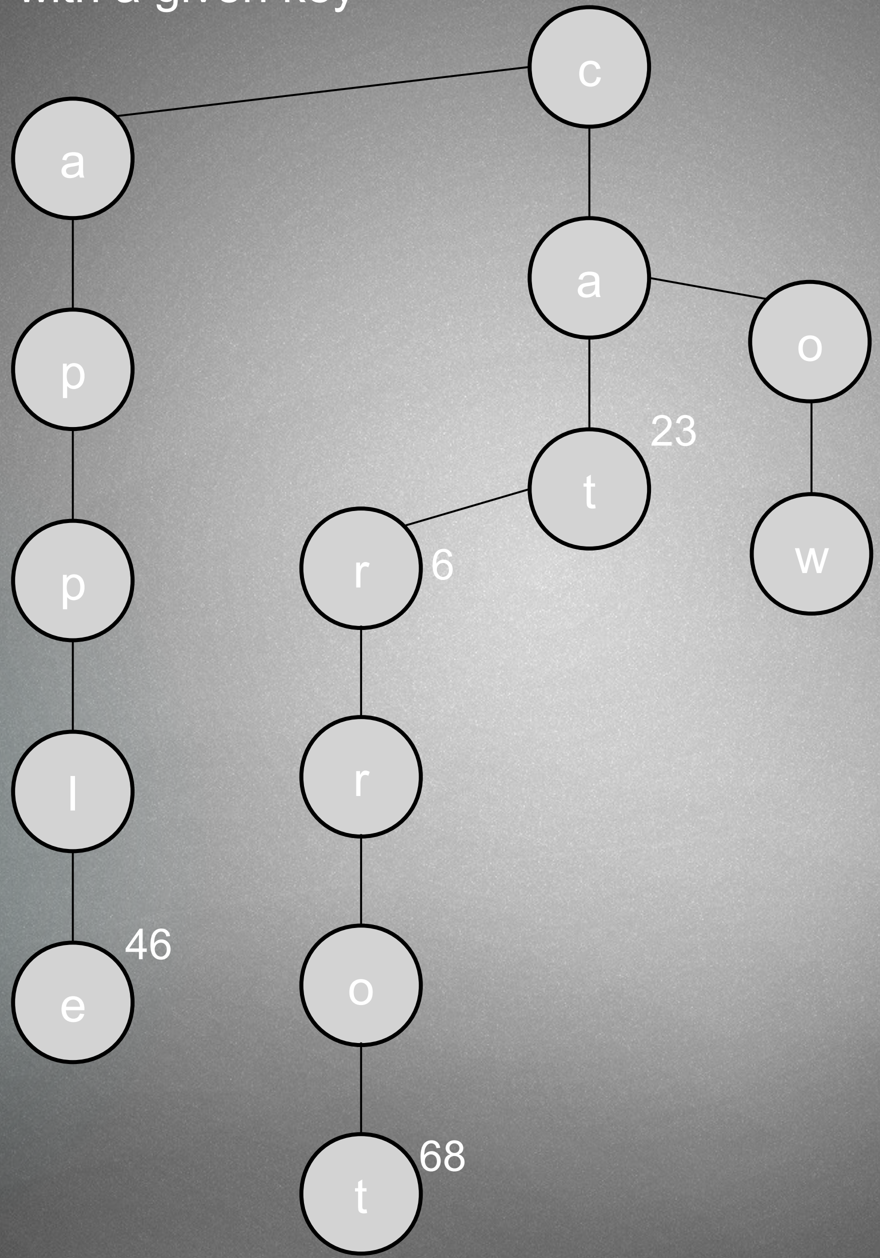
put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„cow”,112)

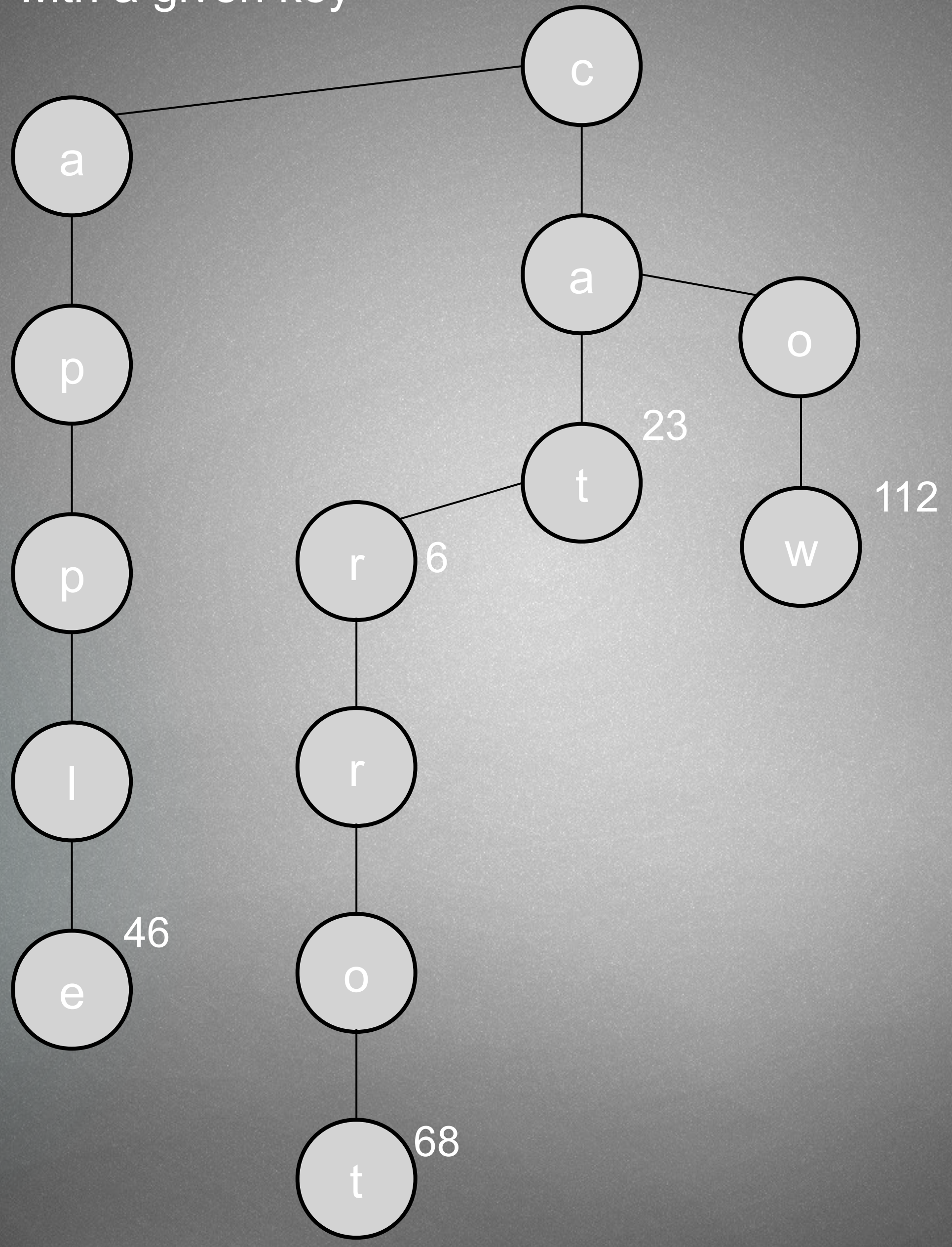


put: with this operation we would like to insert a new element to the ternary search tree with a given key

put(„cow",112)



put: with this operation we would like to insert a new element to the ternary search tree with a given key



get: with this operation we would like to get an item from the ternary search tree with a given key

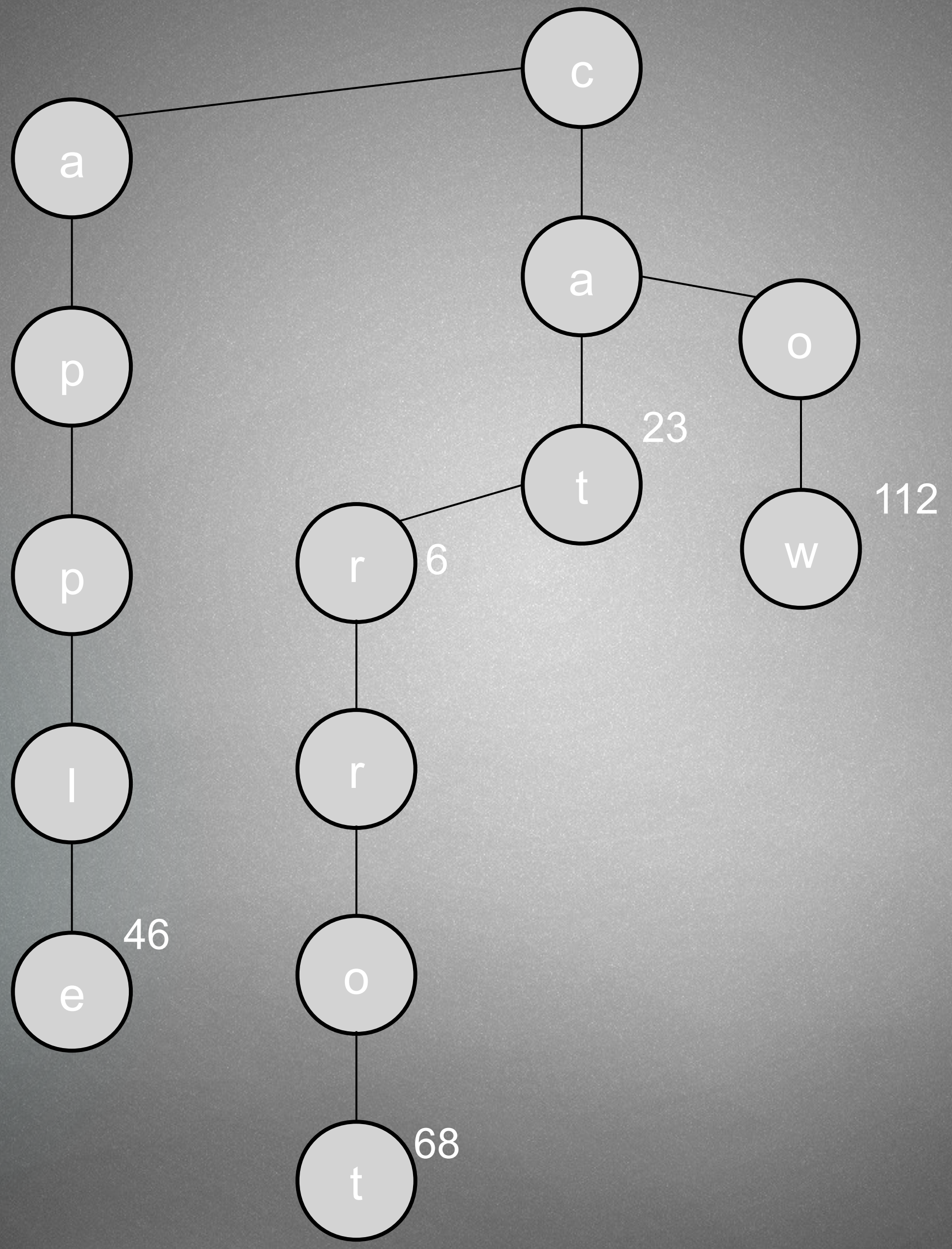
IMPORTANT:

- hashmap: we generate an index from the key with the hashfunction.
  We use every single character of the key

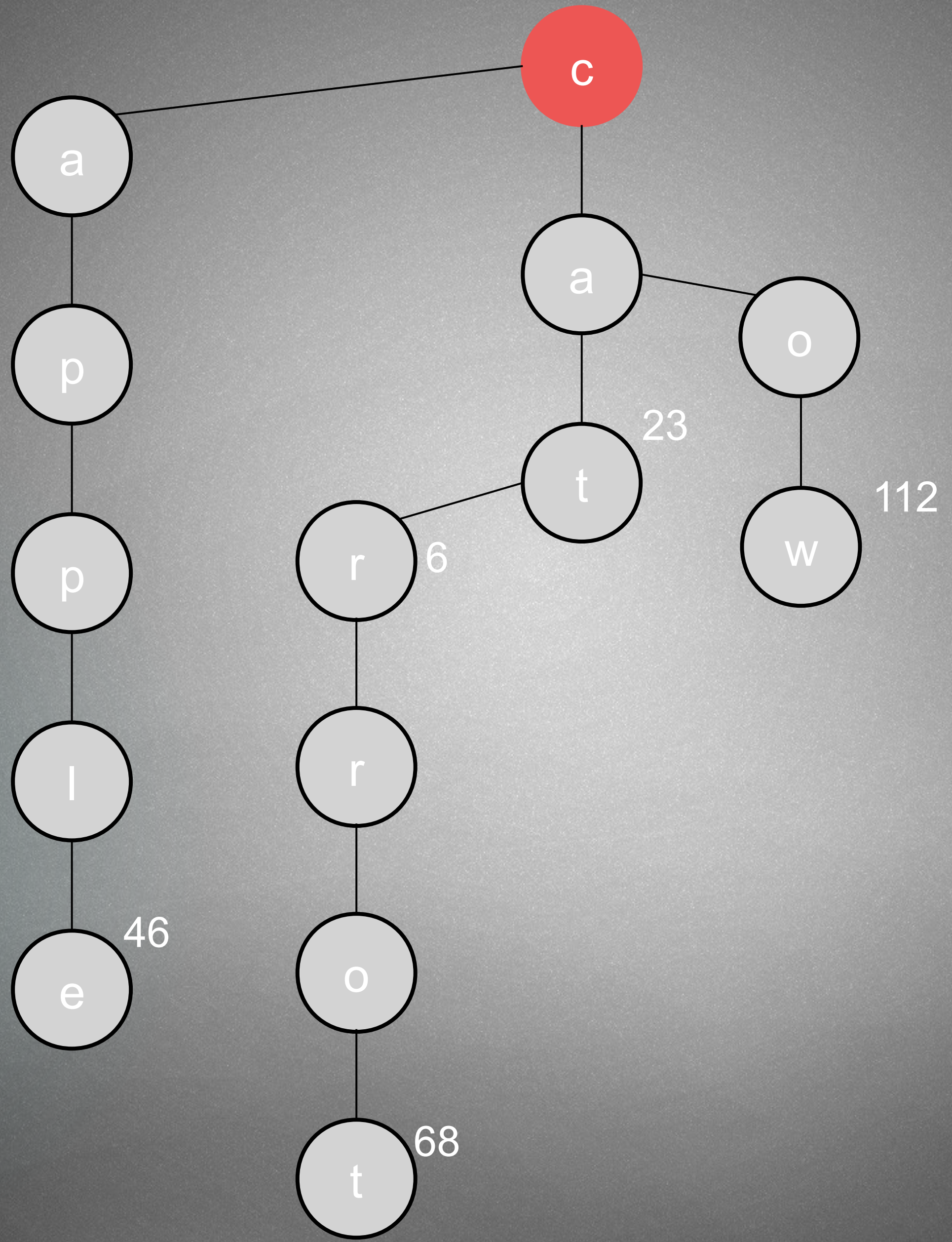
- TST: we may come to the conclusion that there is no value with a given key
  without considering every character
  For example: we may return after the second character

CONCLUSION: for mismatch → TST is faster !!!

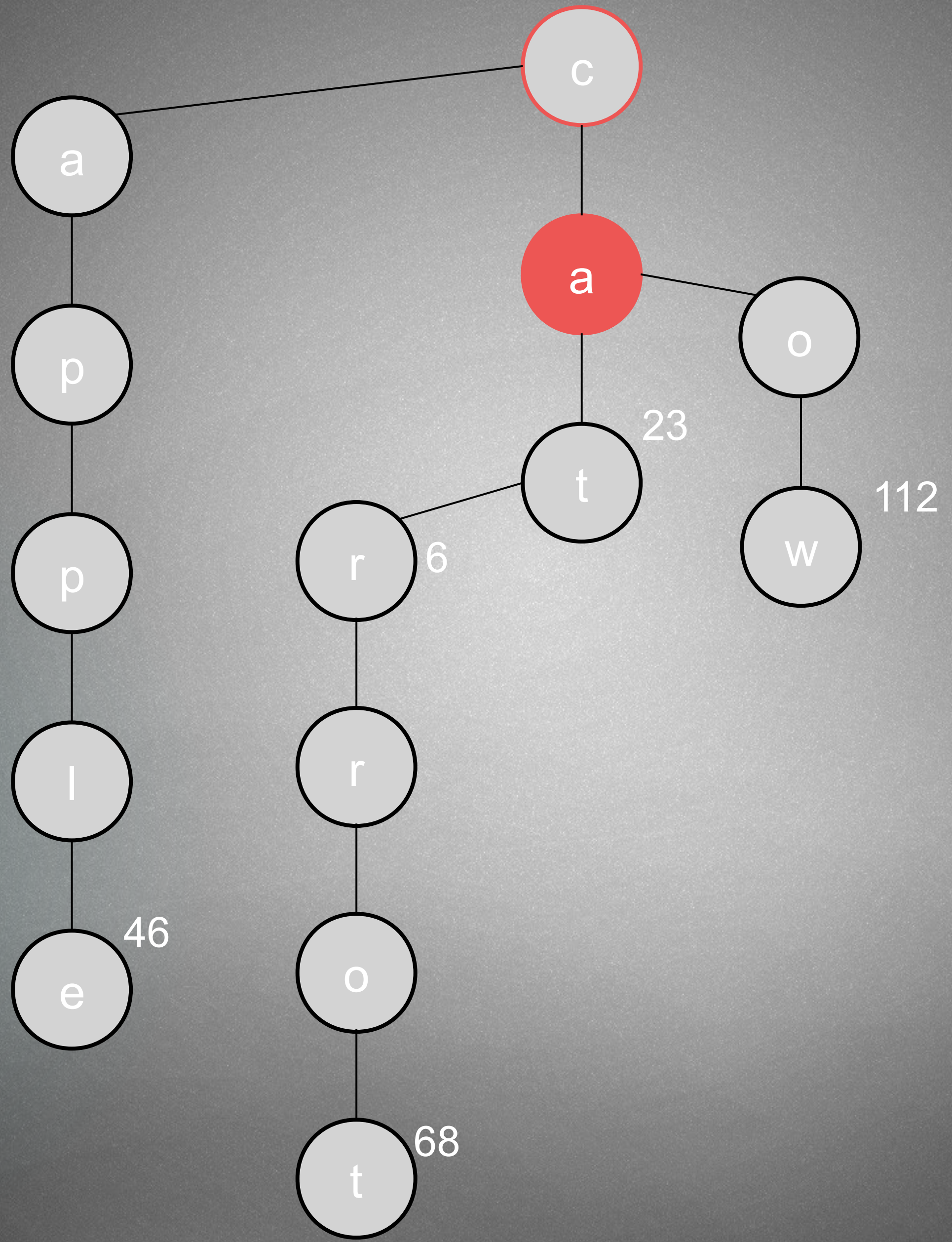
get(„car”)



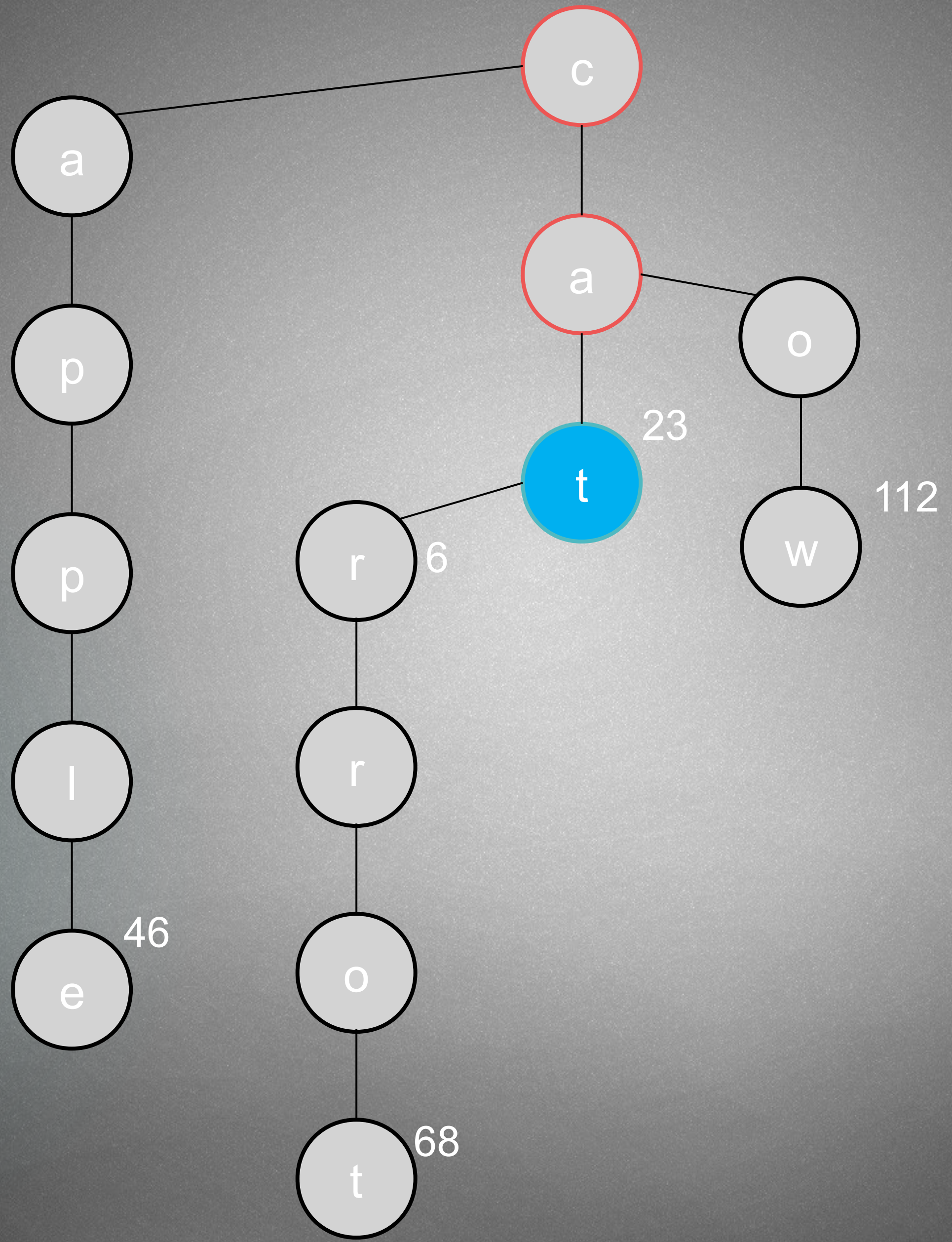
get(„**car**")



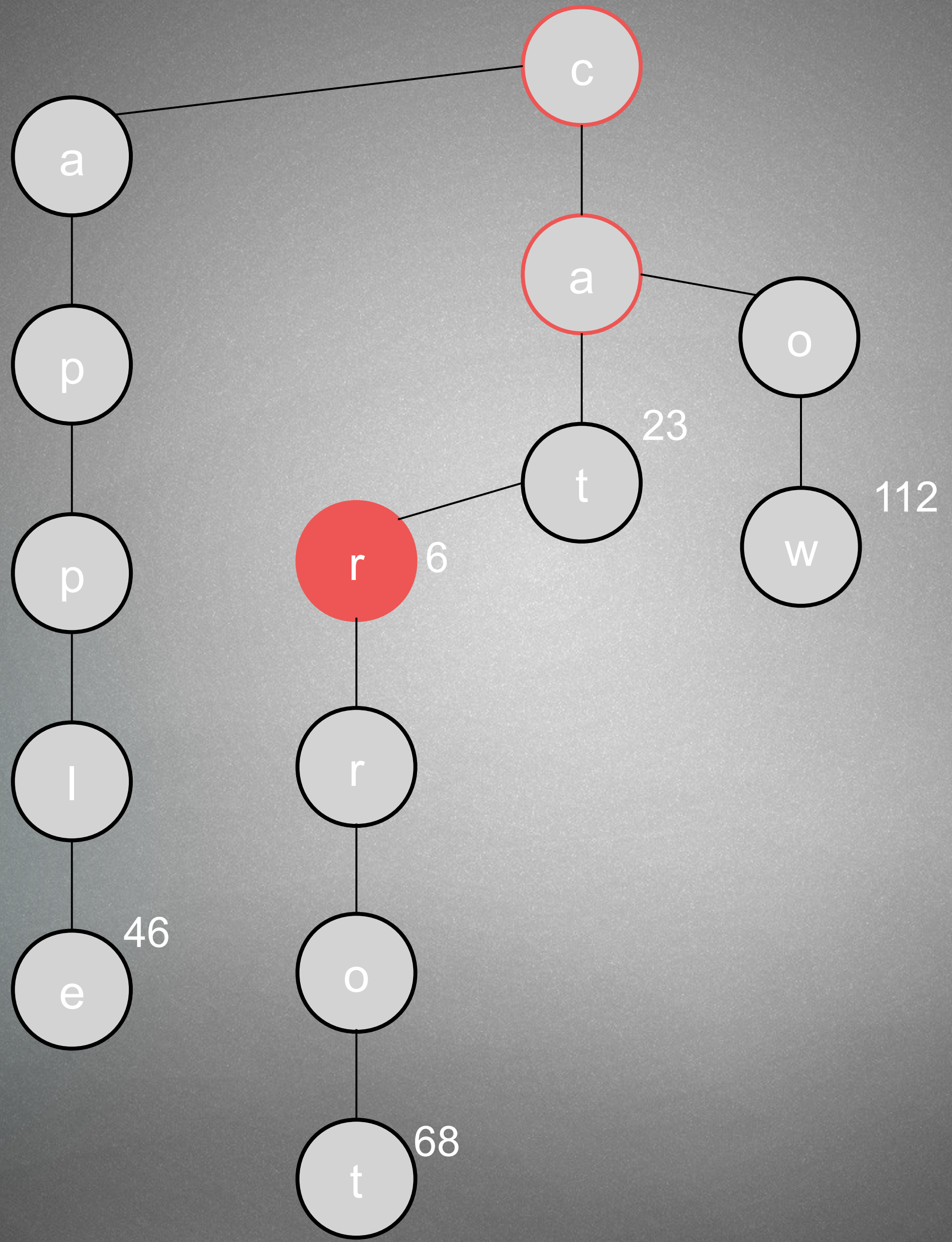
get(„car”)



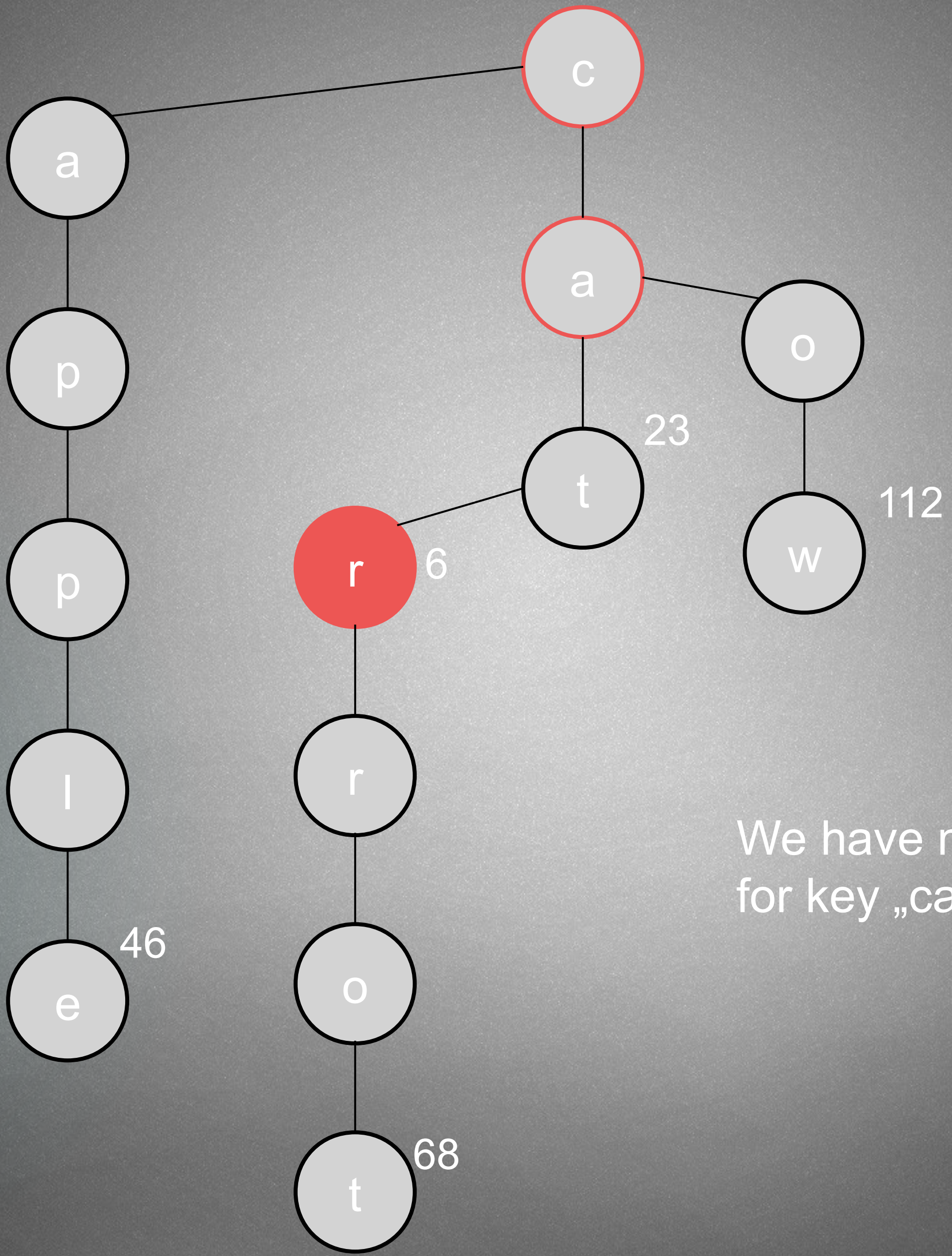
get(„car”)



get(„car”)

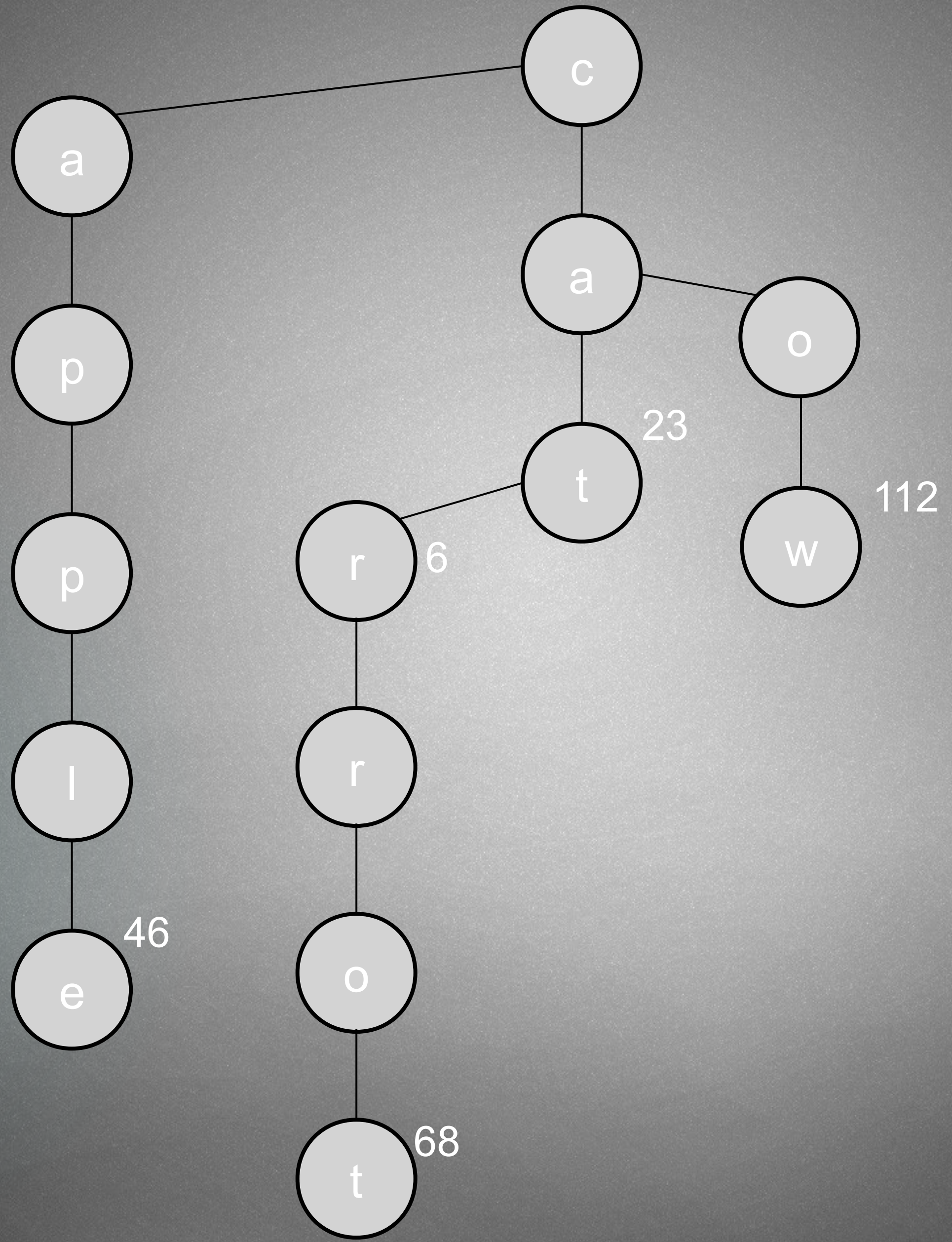


get(„car”)

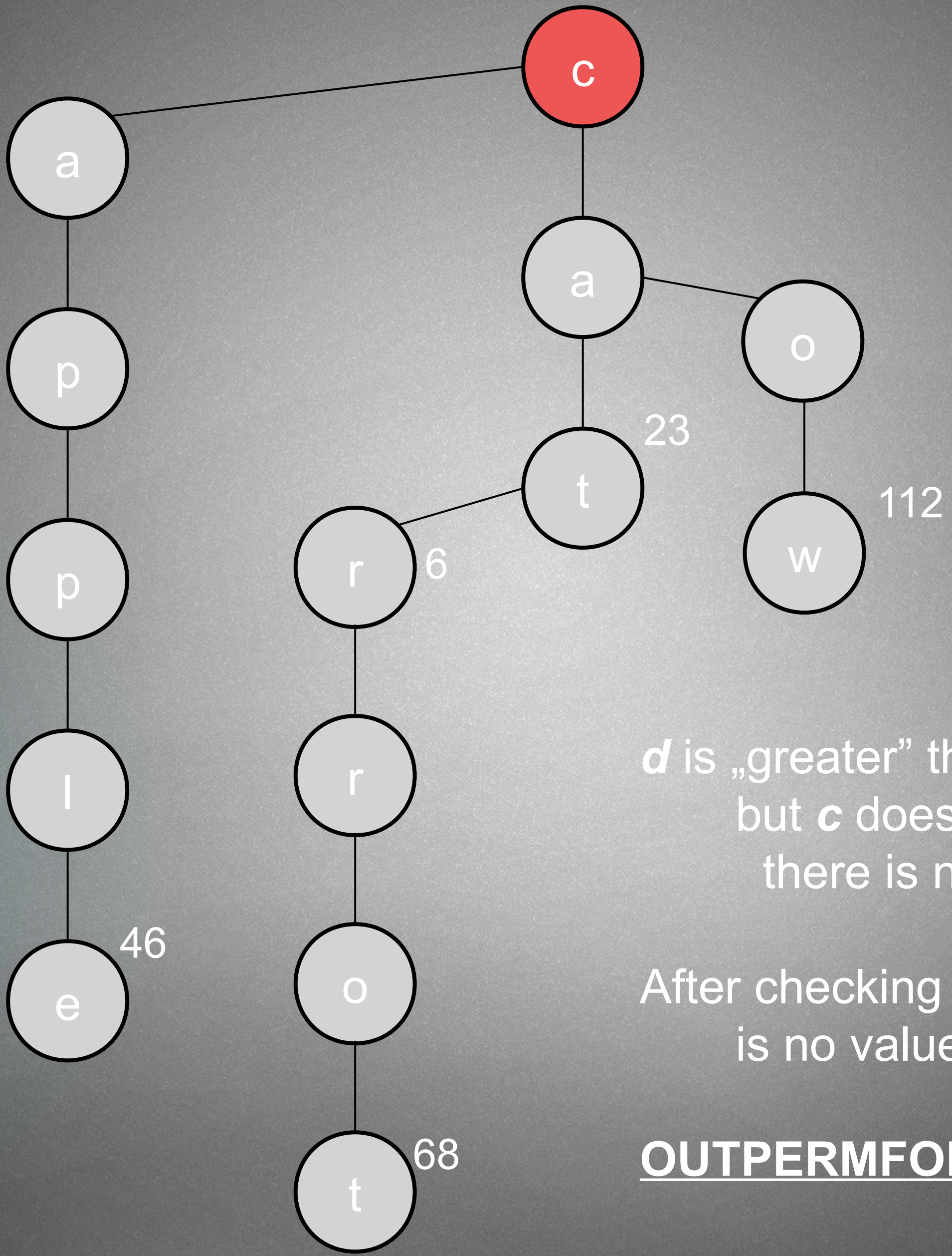


We have managed to find the value 6 for key „car” !!!

get(„dog”)



get(„**dog**”)



d is „greater” than ***c*** in the alphabetical order but ***c*** does not have any right child: it means there is no value with key „dog” in the TST

After checking the first character: we are sure there is no value with this key !!!

__**OUTPERMFORMS HASHMAP**__ !!!

Important notes

- We should combine tries with TST
- At the root: it is a trie with many many children
- At lower levels it becomes a TST with 3 children only
- This combination is quite efficient !!!

TST vs hashing

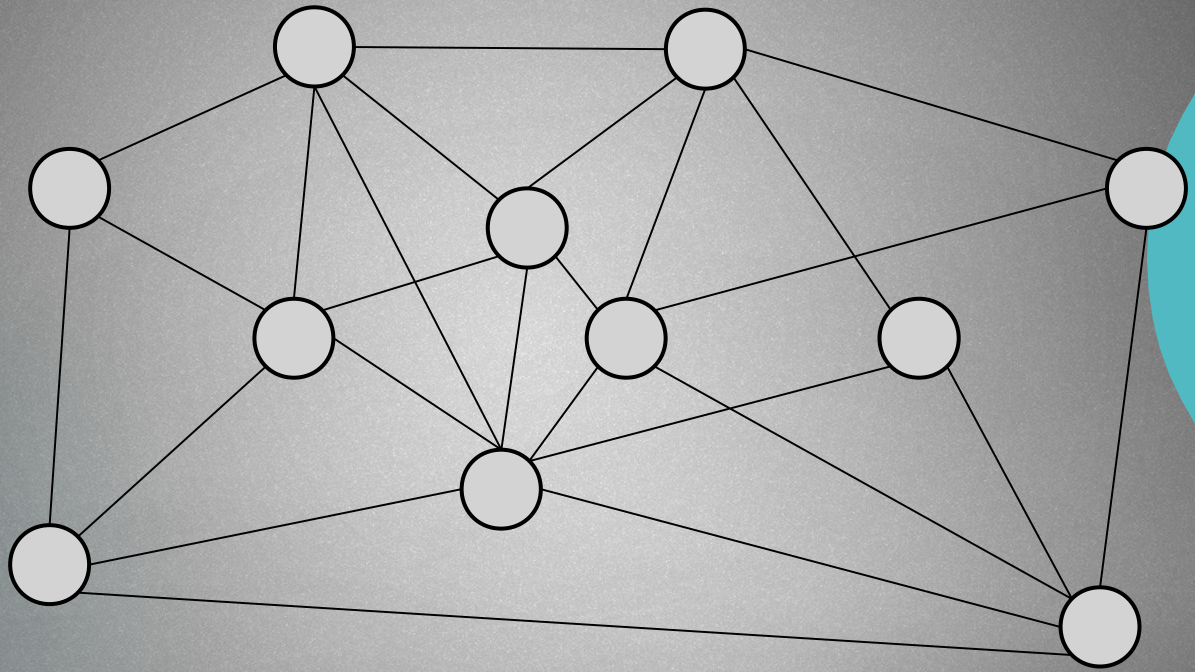
- **Hashing**
  - Need to examine the entire key (because that is the way the hash function works)
  - Search hits and misses cost the same
  - The running time and performance relies heavily on the hashfunction
  - Does not support as much operations than TST (sorting)
- **TST**
  - Works only for strings
  - Only examines just enough key characters
  - Search miss may only involve a few characters
  - Support more operations (sorting)
  - Faster than hashing (for misses especially) and more flexible than BST

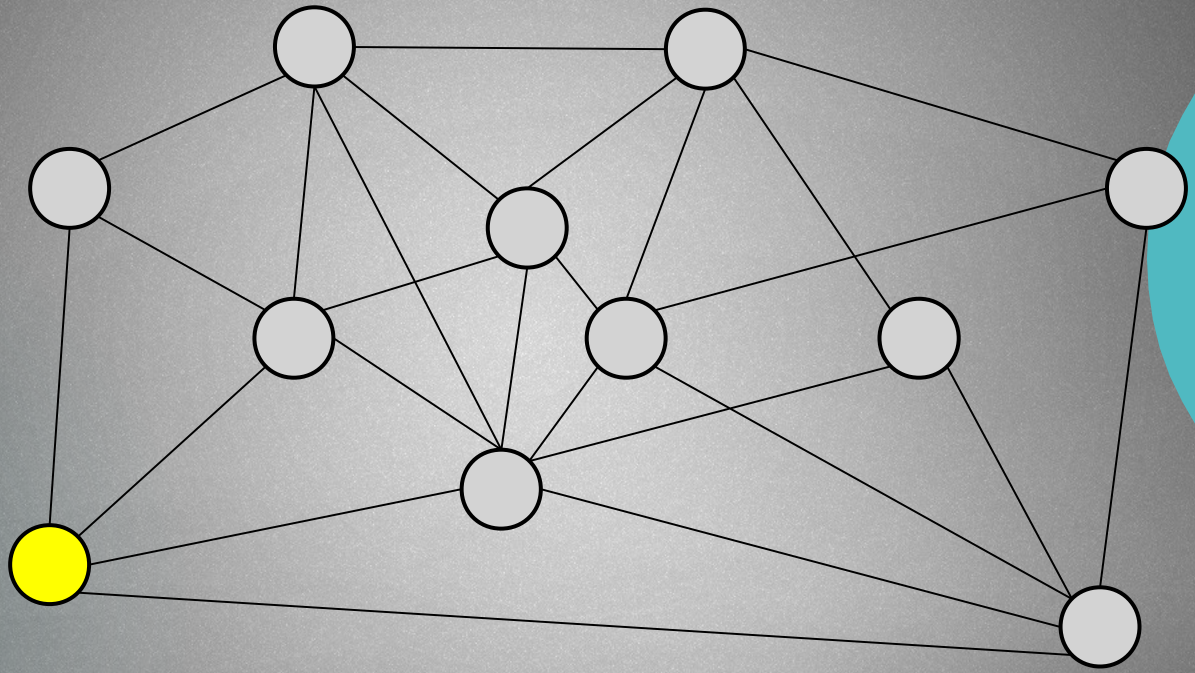
Applications

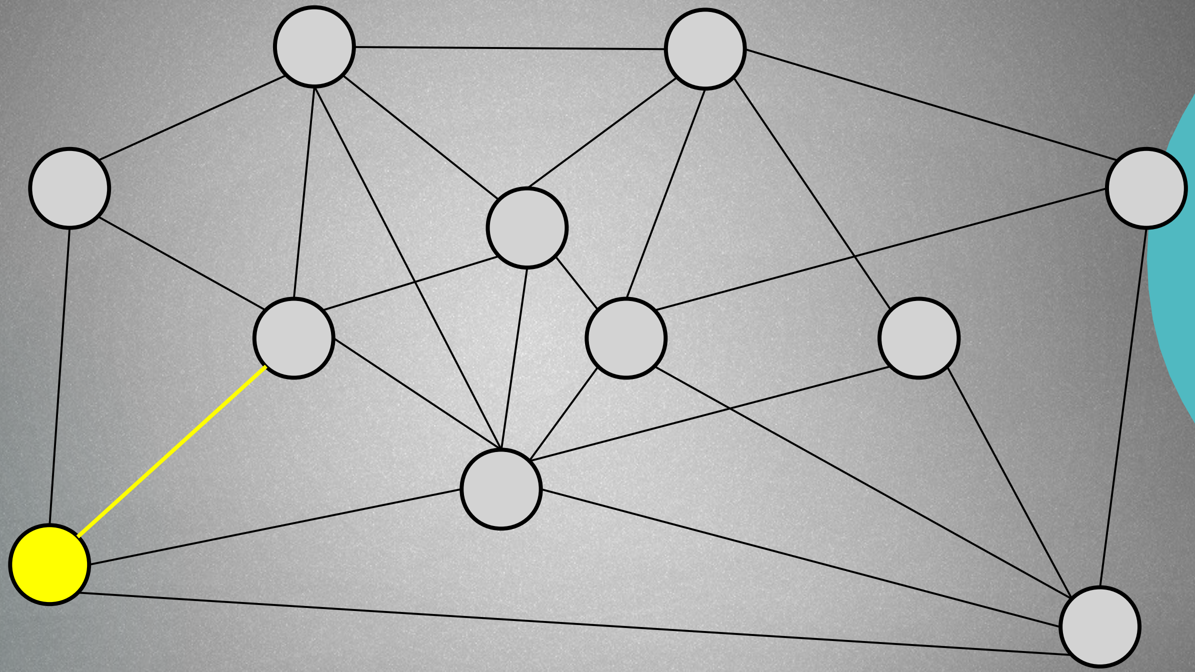
- It can be used to implement the auto-complete feature very very efficiently
- Can be used for spell-checkers
- Near-neighbor searching (of which a spell-check is a special case)
- For databases especially when indexing by several non-key fields is desirable
- Very important in package routing on WWW → the router direct the packages in the direction of the longest prefix. It can be found very quickly with the help of TST-s
- Prefix matching ~ google search
  - We can use DFS instead usually

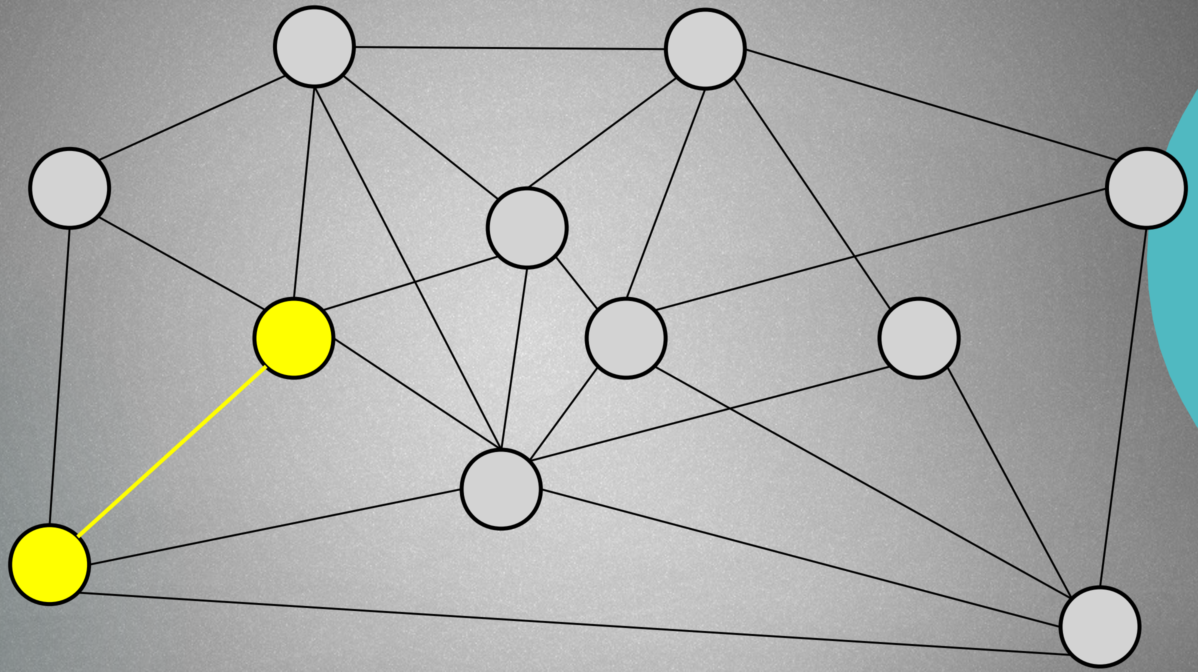
TRIES

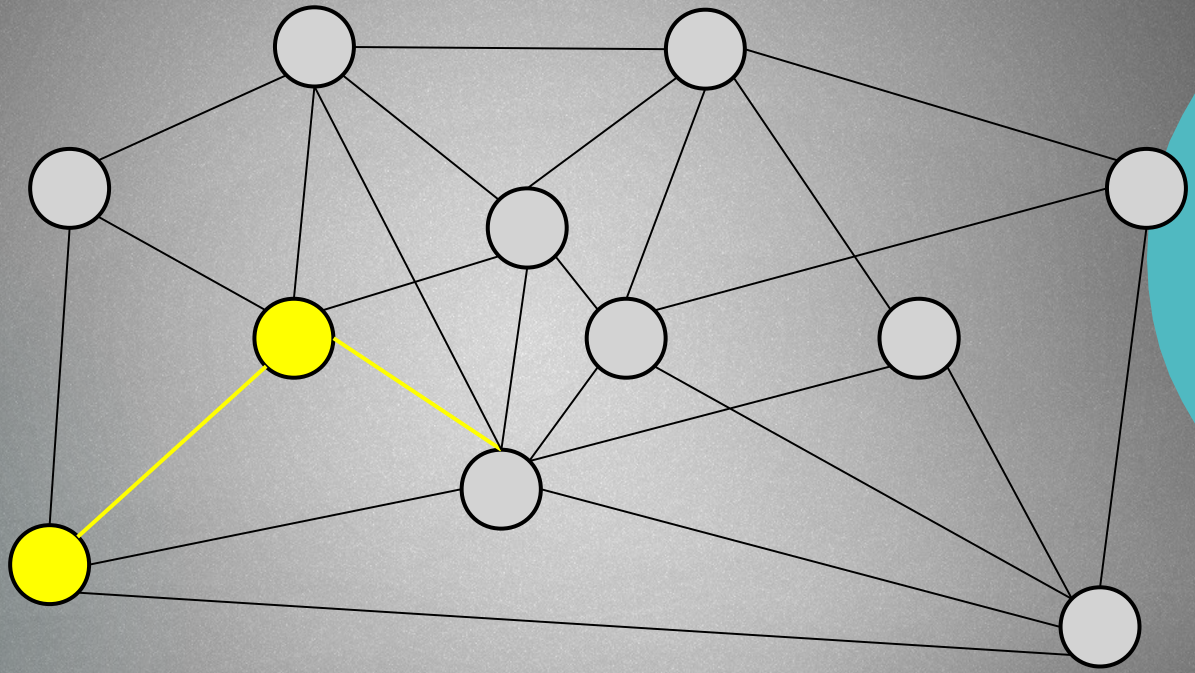
IP routing with trie data structures

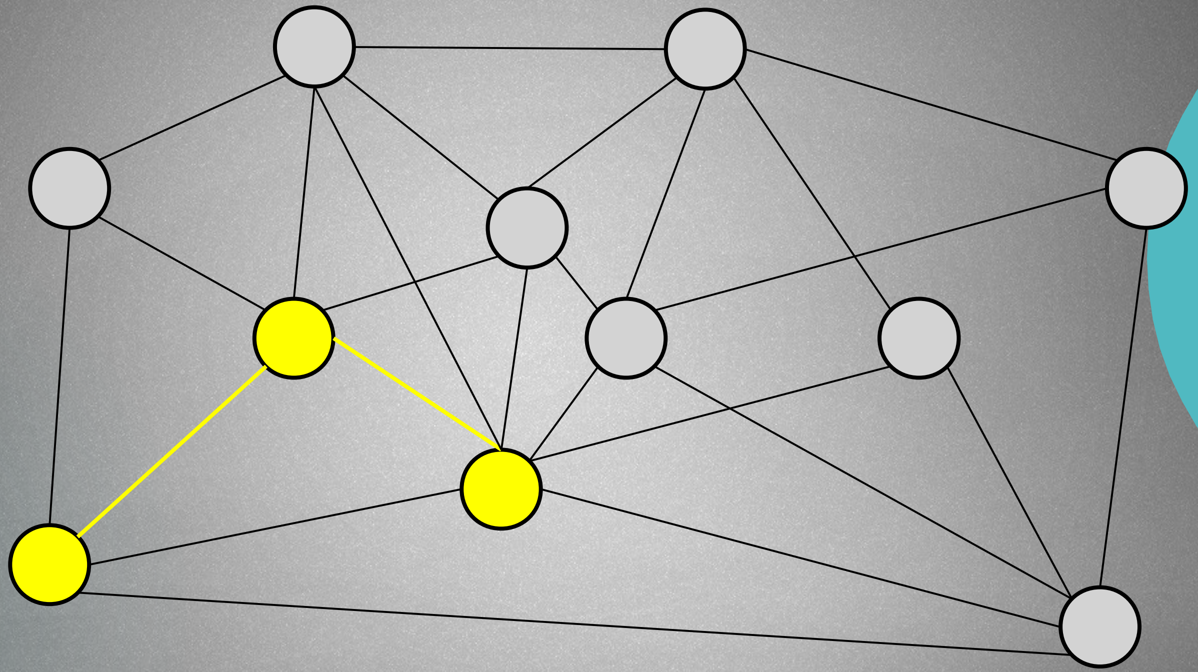


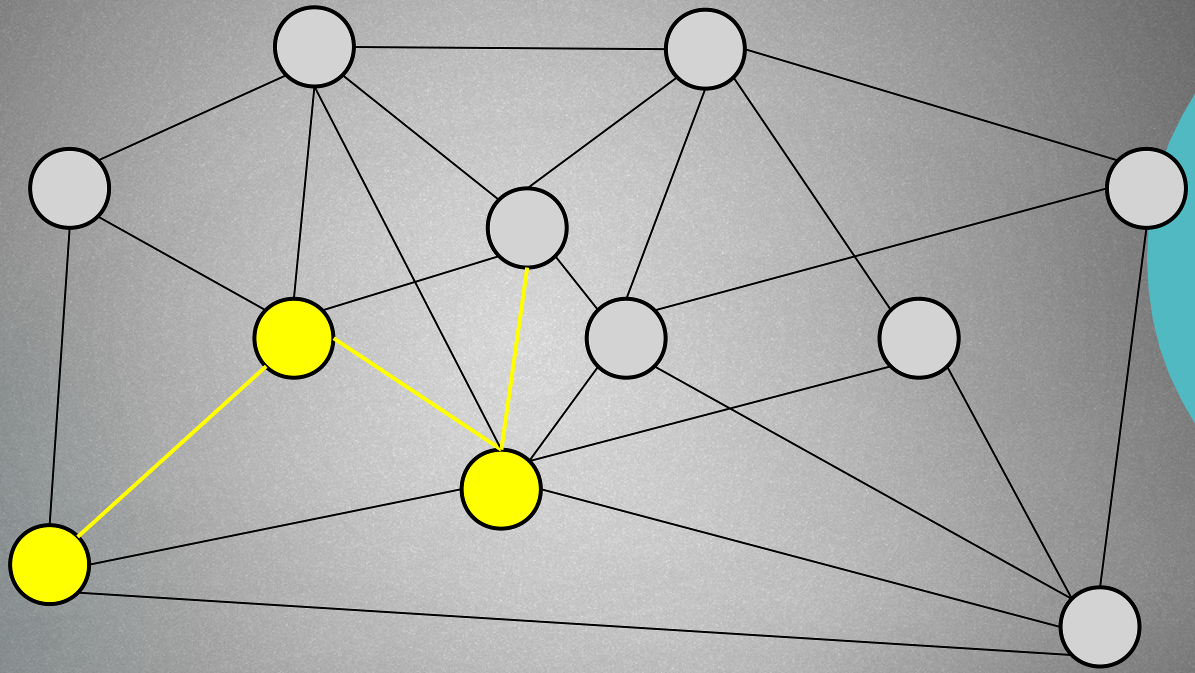


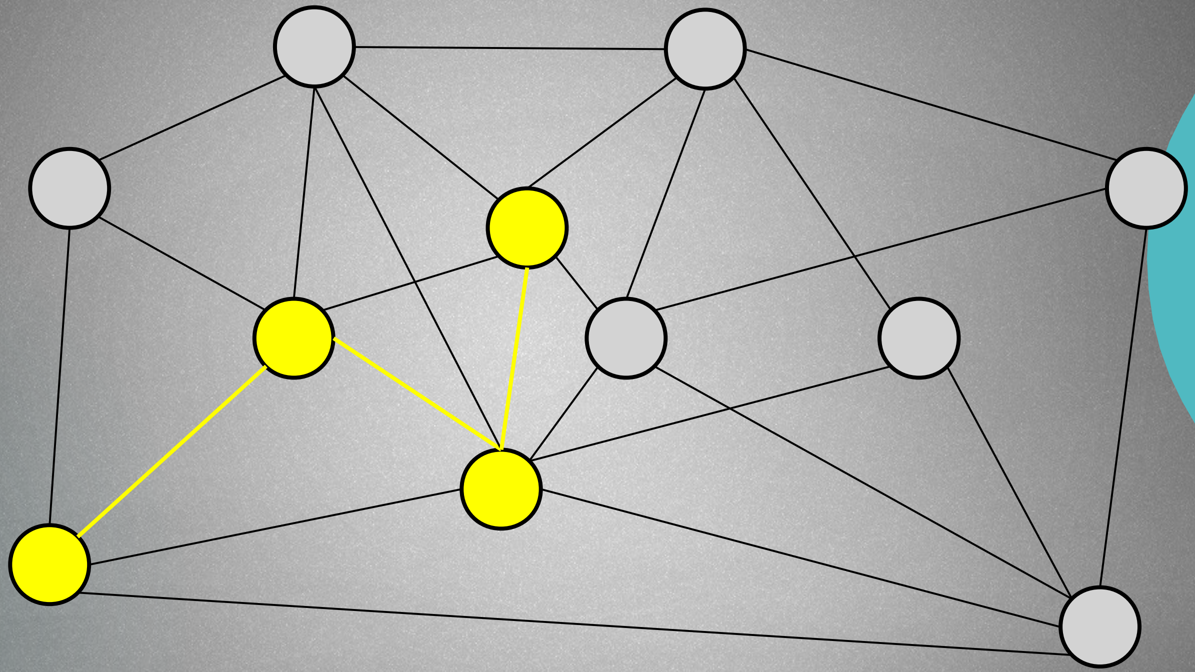


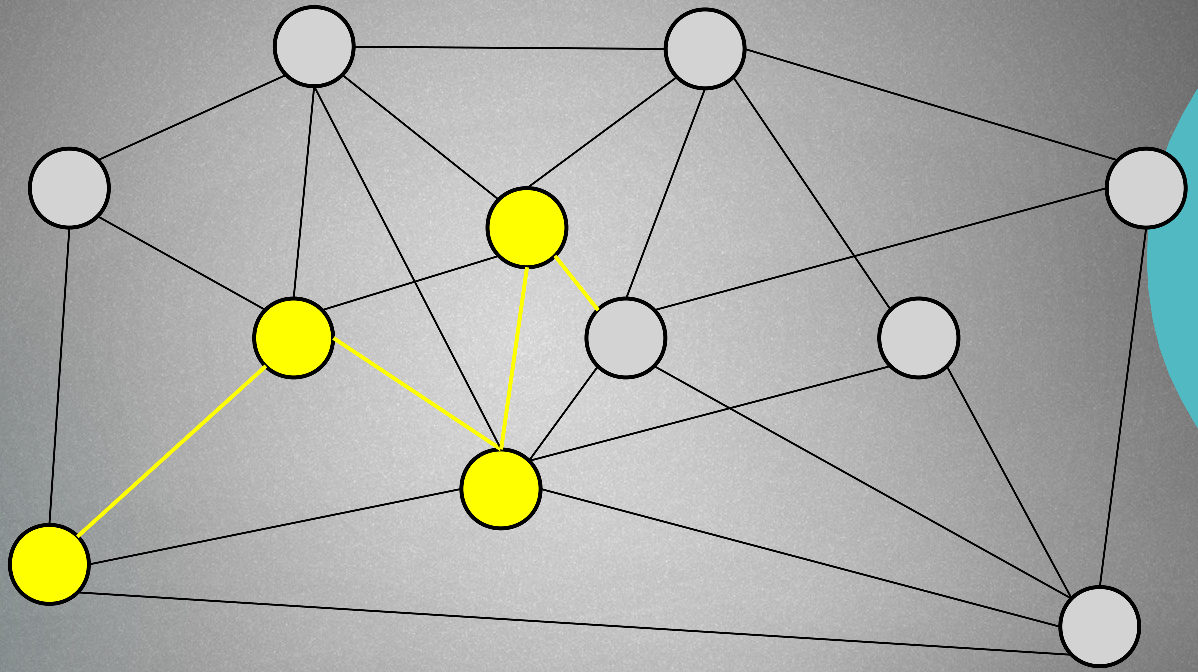


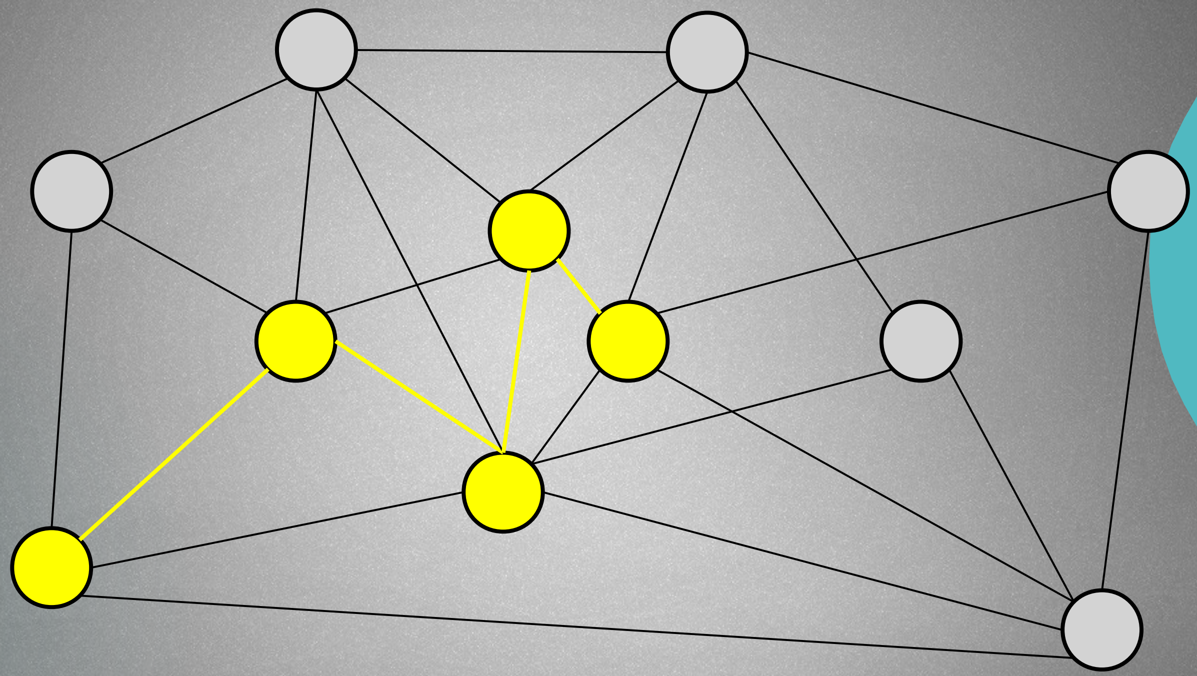


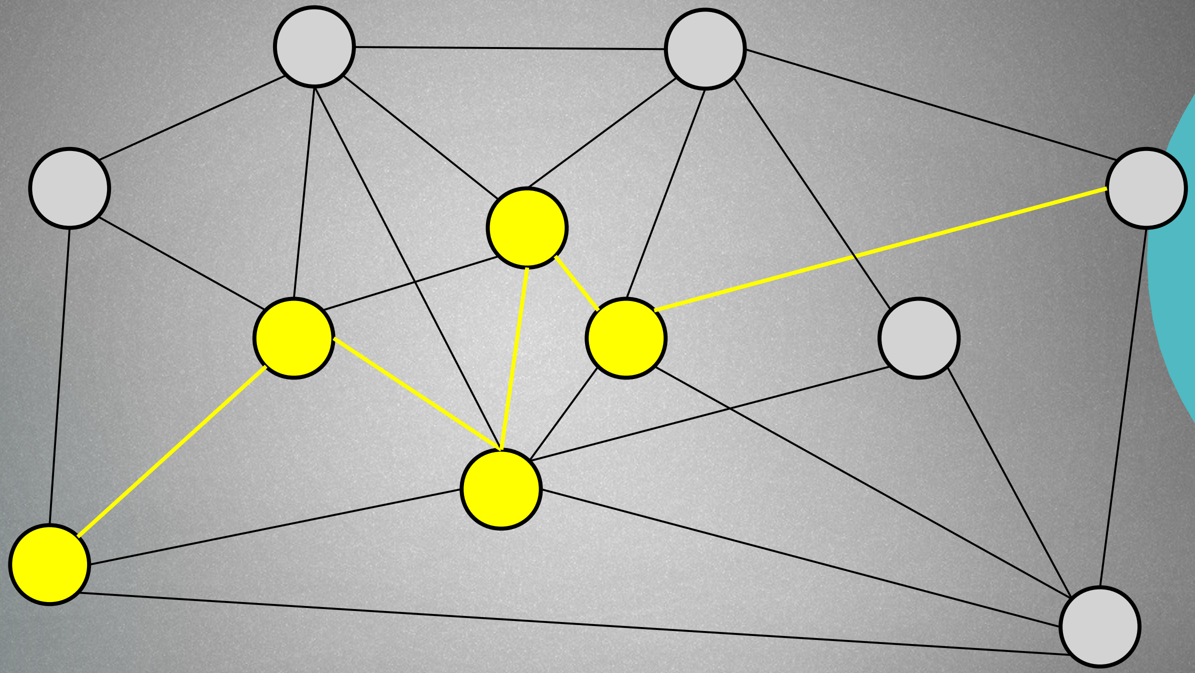


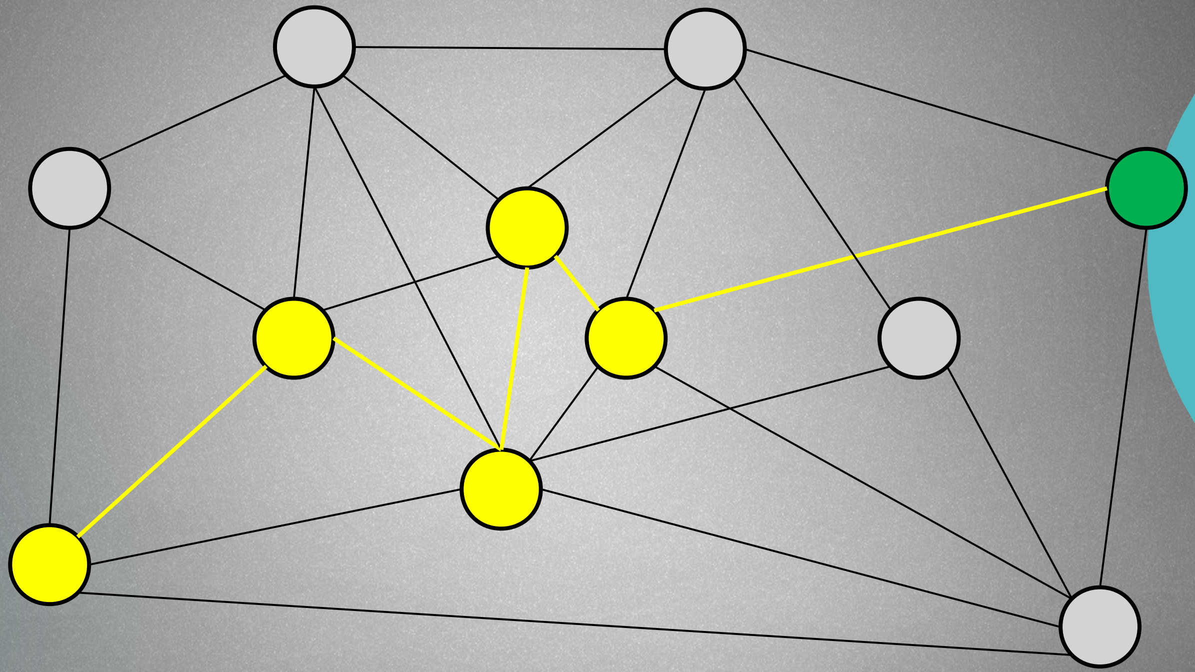












IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send the packages approximately to the right direction

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send the packages approximately to the right direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.189.345.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.189.345.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.189.345.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.189.345.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.189.345.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.189.345.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.145.667.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.145.667.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.145.667.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.145.667.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

243.145.111.173

243.189.345.763

243.145.667.221

IP routing:

Routing IP packages on the web relies heavily on graph algorithms and data structures !!!

- the router sends the packages toward the destination IP address
- it calculates the longest common prefix
- just send it approximately to the good direction

243.189.345.123

243.189.562.173

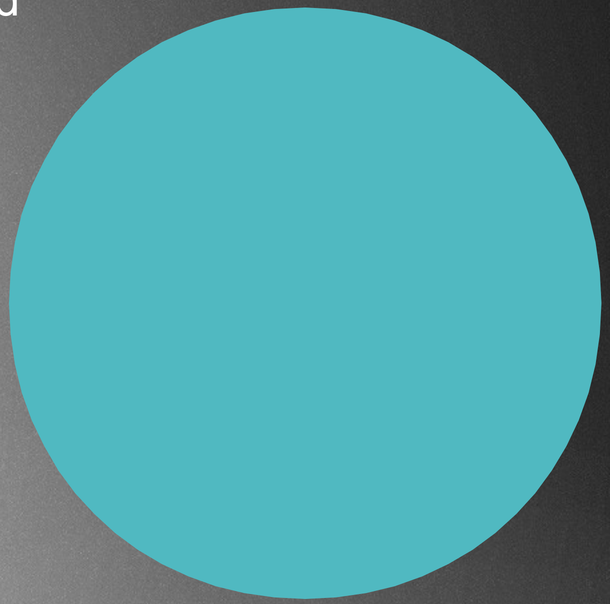
243.145.111.173

243.189.345.763

243.145.667.221

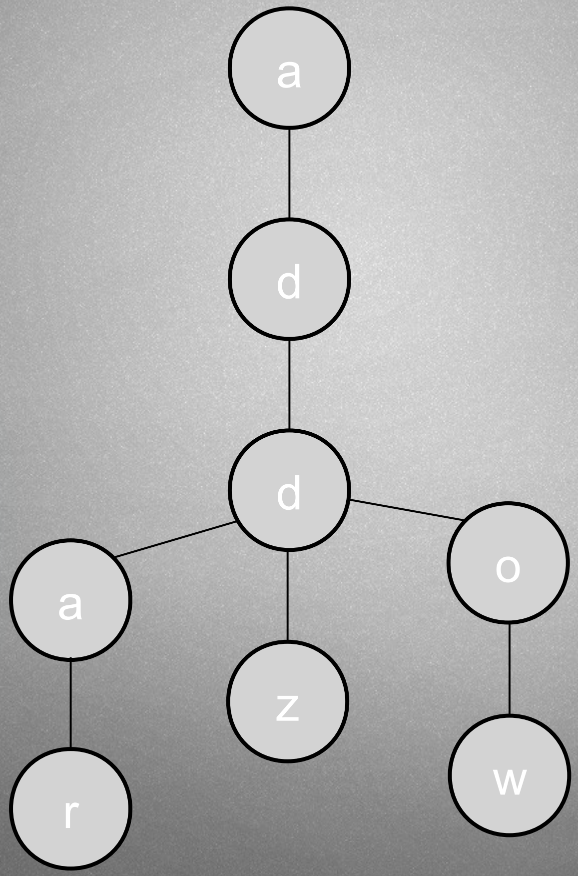
Algorithm

We have to consider the nodes where just a single node is valid
~ so a single node is not null !!!



Algorithm

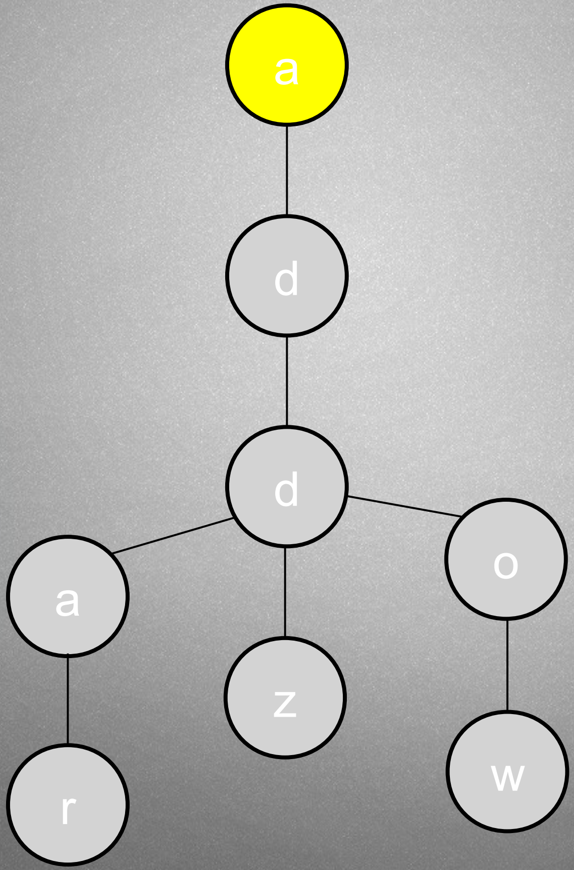
We have to consider the nodes where just a single node is valid
~ so a single node is not null !!!



Algorithm

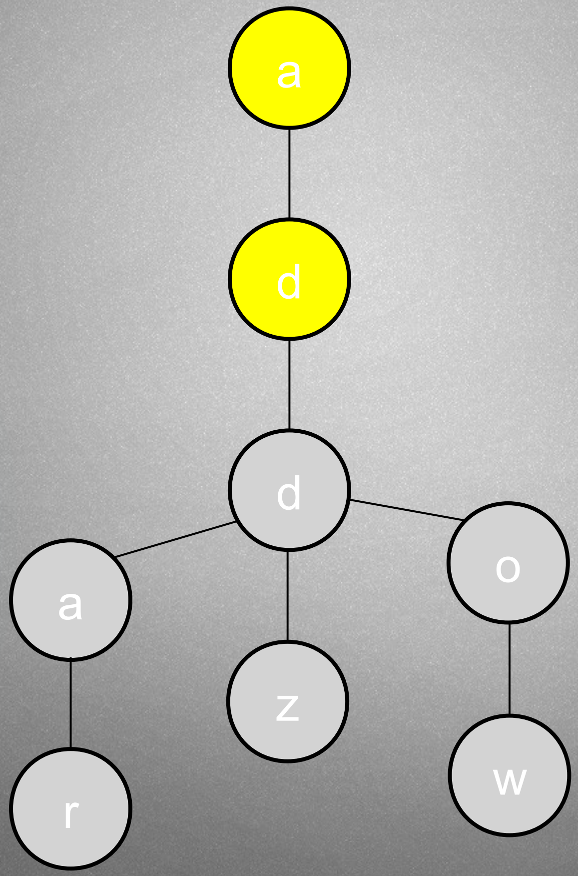
We have to consider the nodes where just a single node is valid

~ so a single node is not null !!!



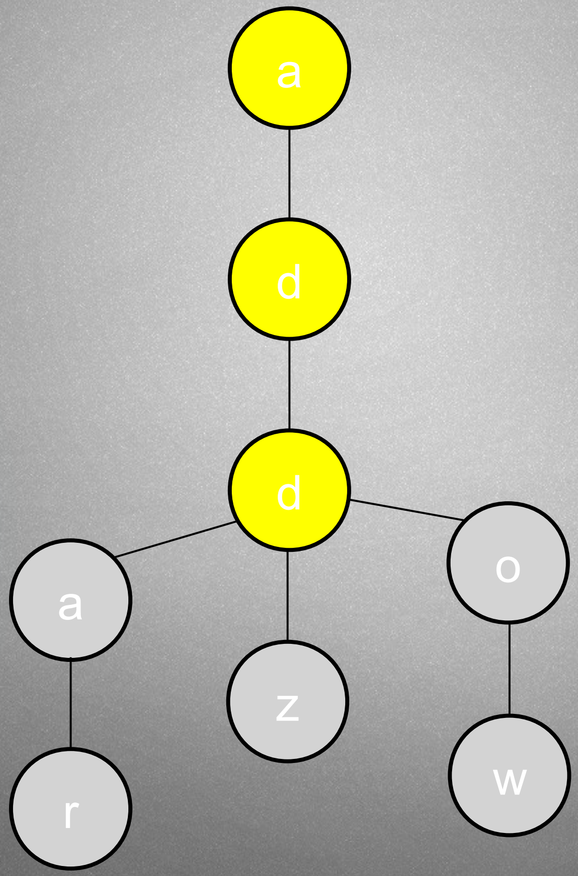
Algorithm

We have to consider the nodes where just a single node is valid
~ so a single node is not null !!!



Algorithm

We have to consider the nodes where just a single node is valid
~ so a single node is not null !!!



Algorithm

We have to consider the nodes where just a single node is valid
~ so a single node is not null !!!