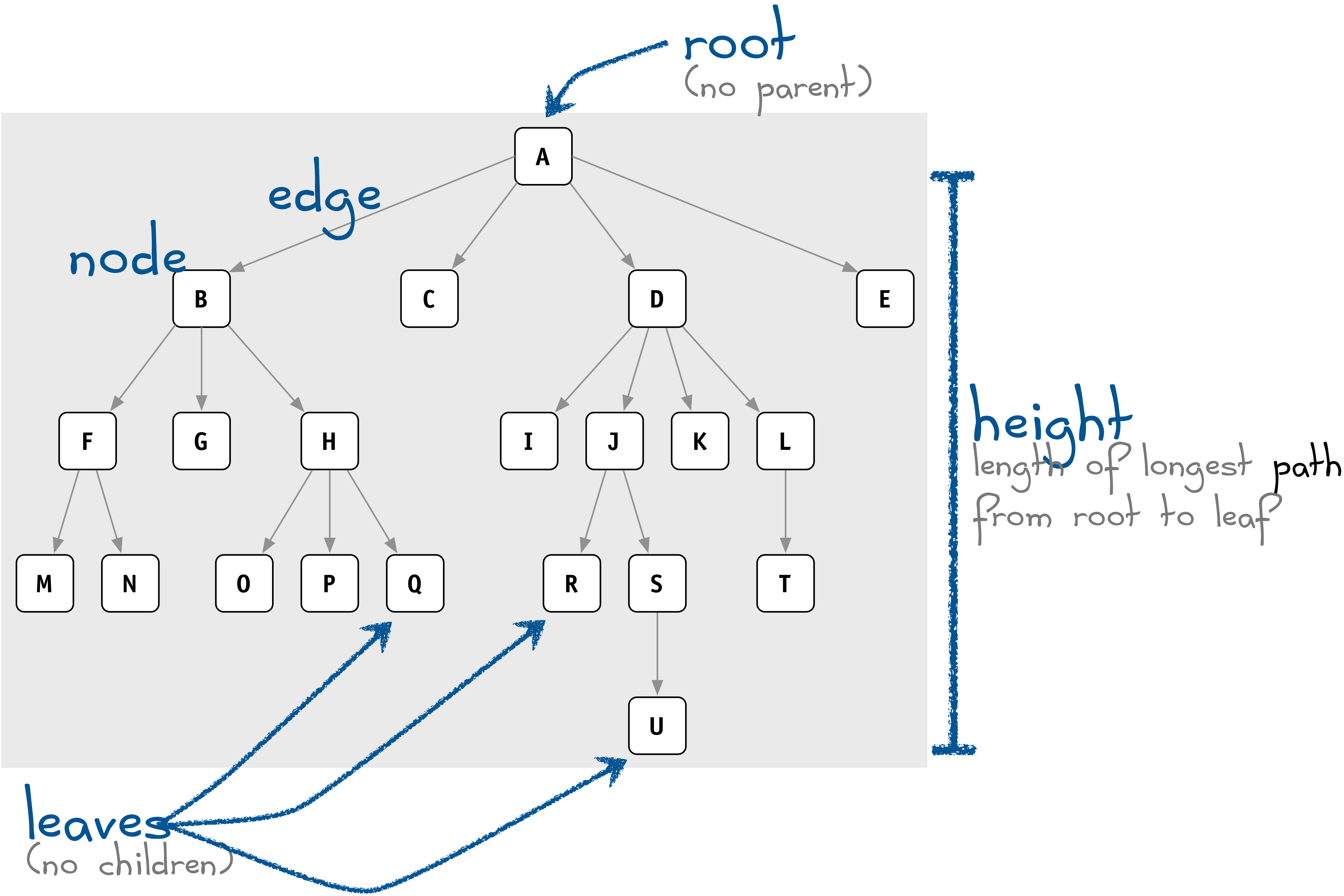
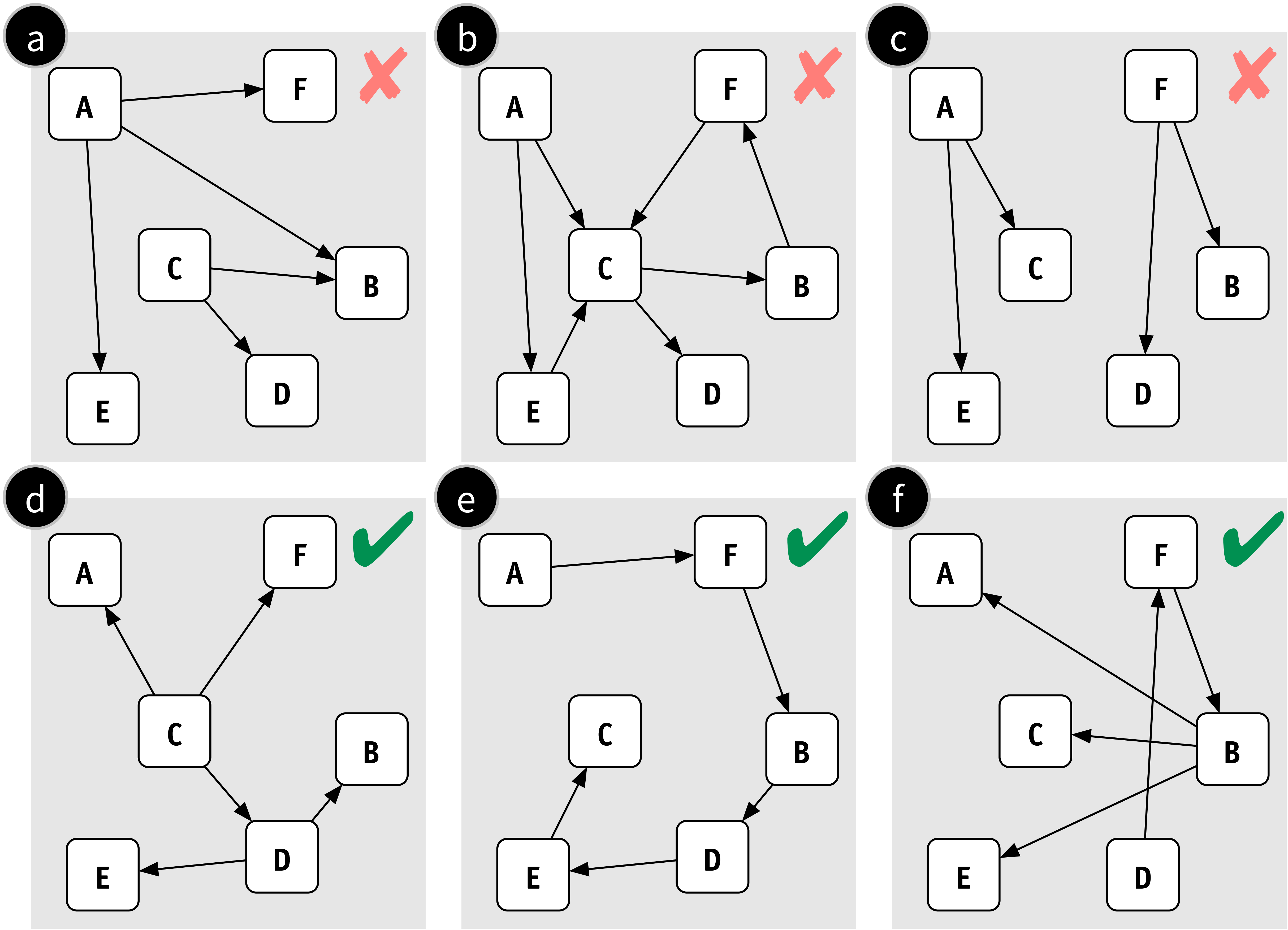
# Trees

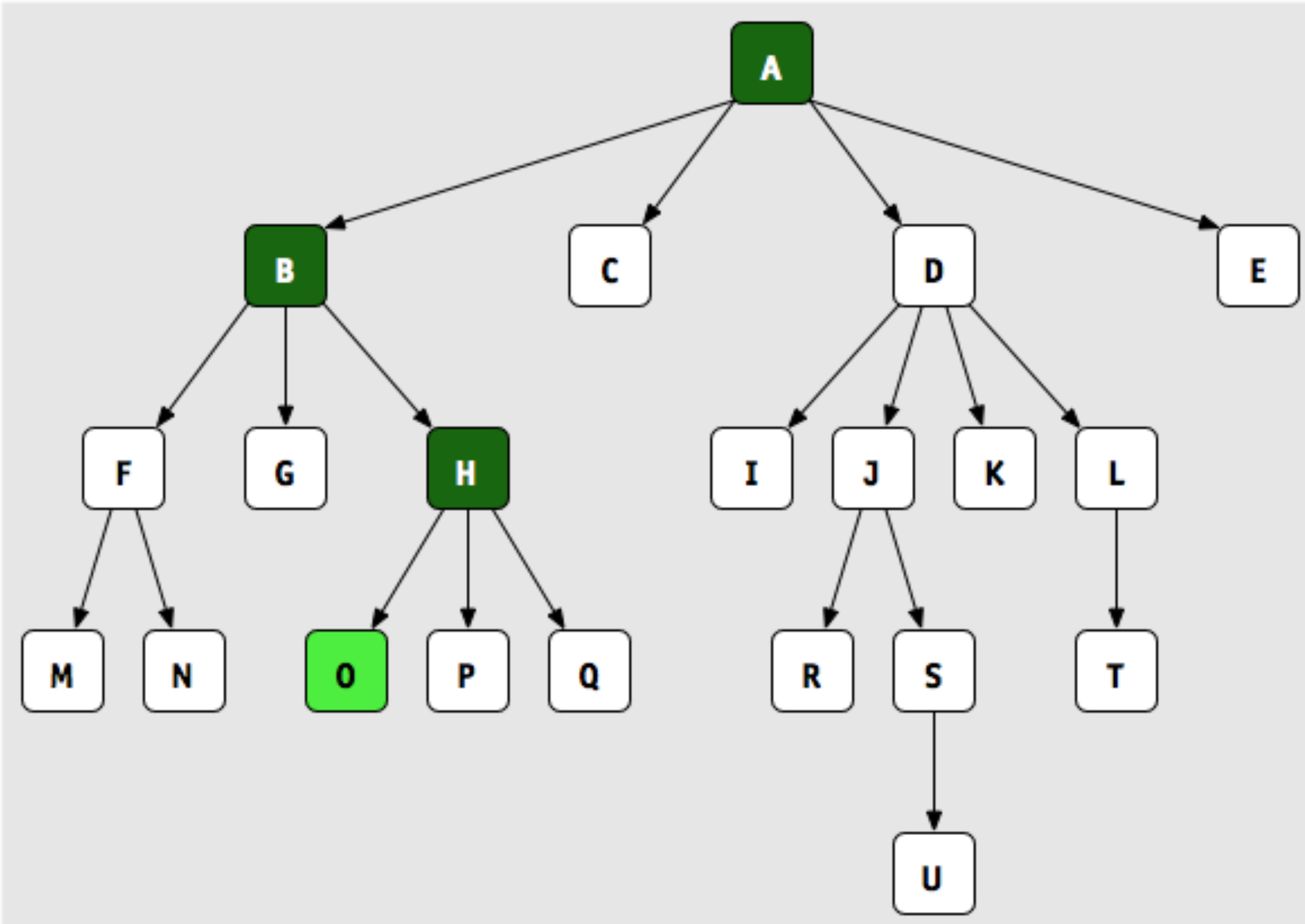
a unique path from root to every element



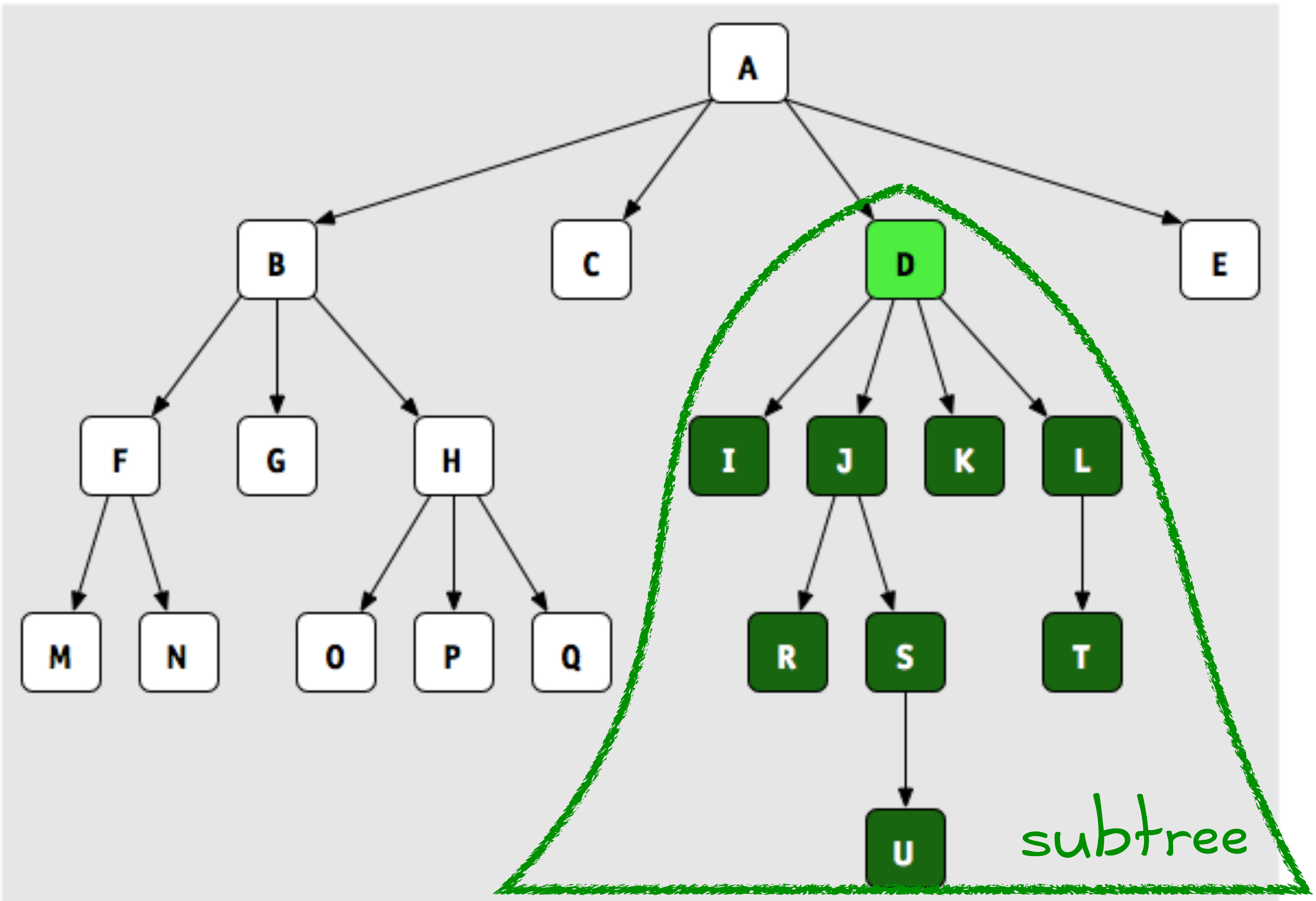
# Which of these are (not) trees? (And why?)



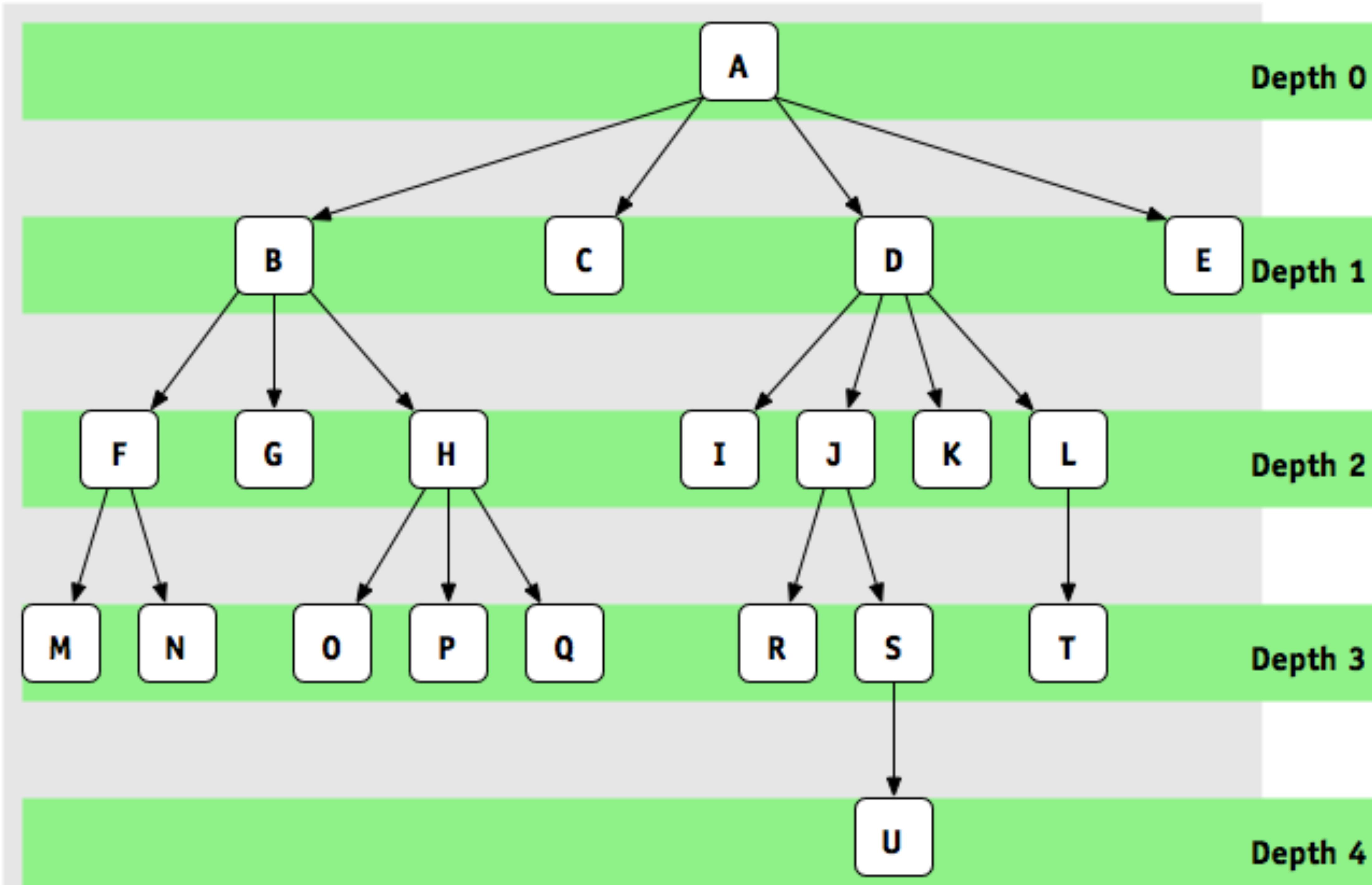
# Ancestors



# Descendants

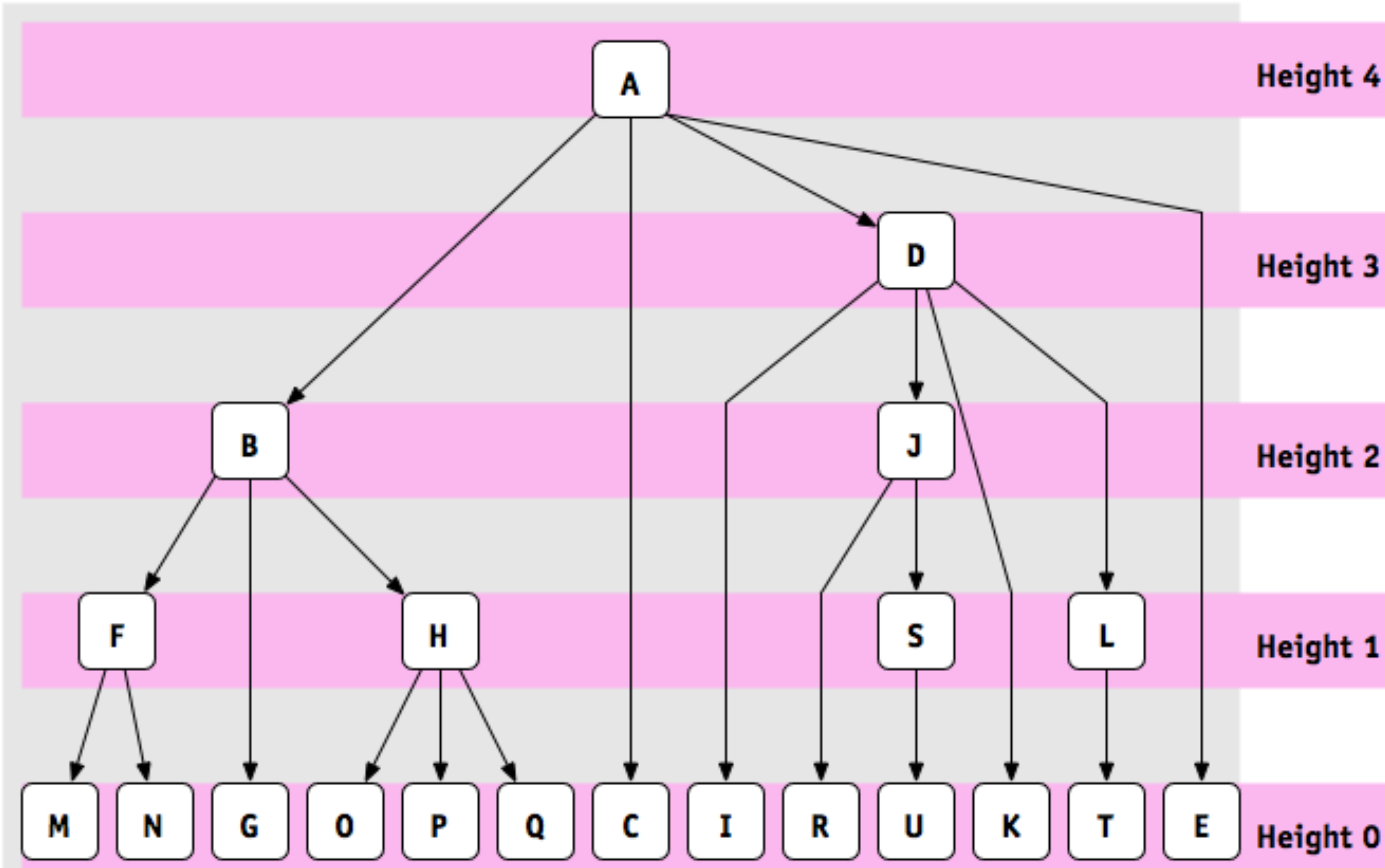


# Depth



A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
Depth 0
Depth 1
Depth 2
Depth 3
Depth 4

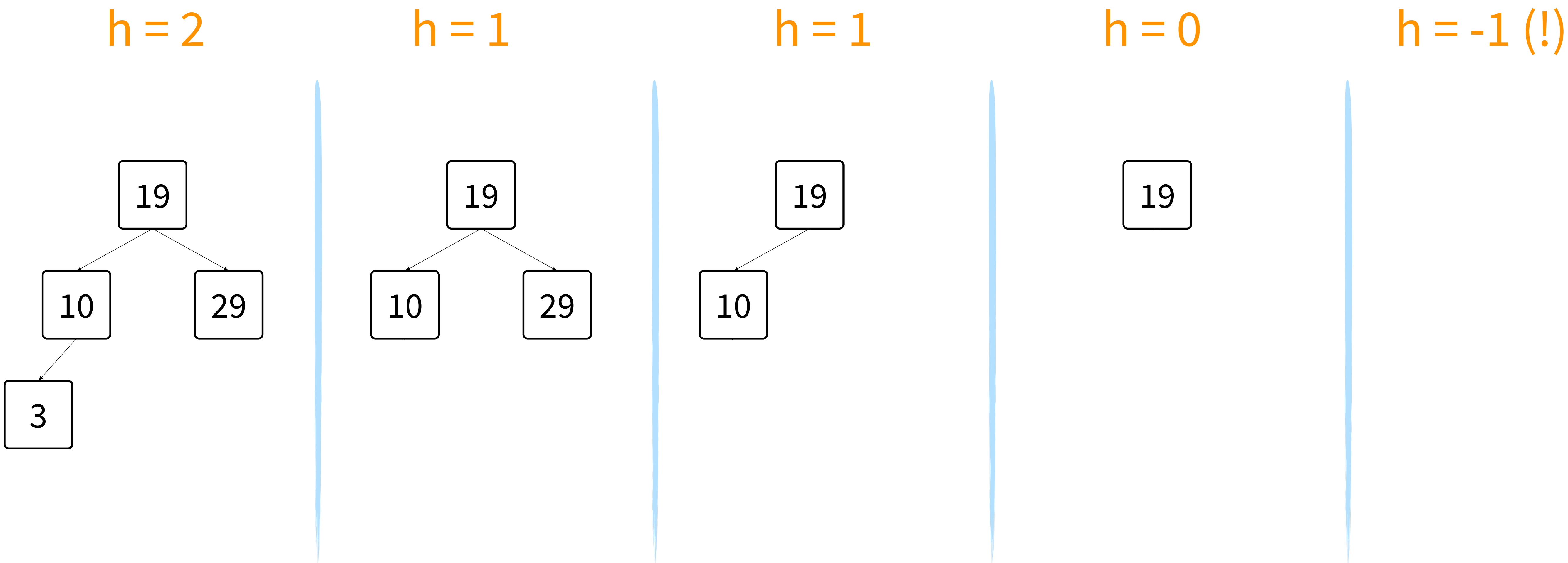
# Height



A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
Height 4
Height 3
Height 2
Height 1
Height 0

# Tree height

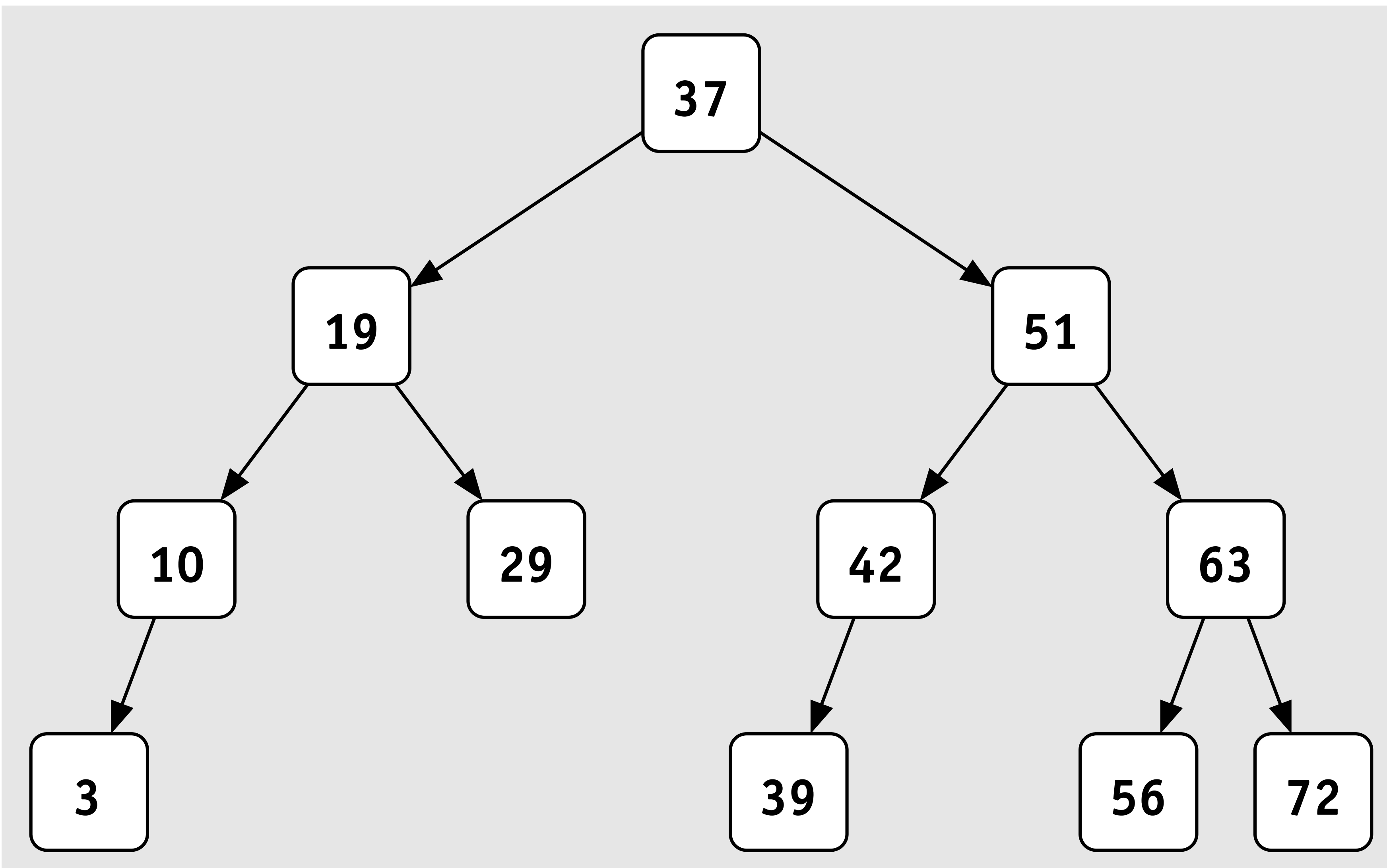
the length of the longest path from the root to a leaf




The height of an empty tree is -1.

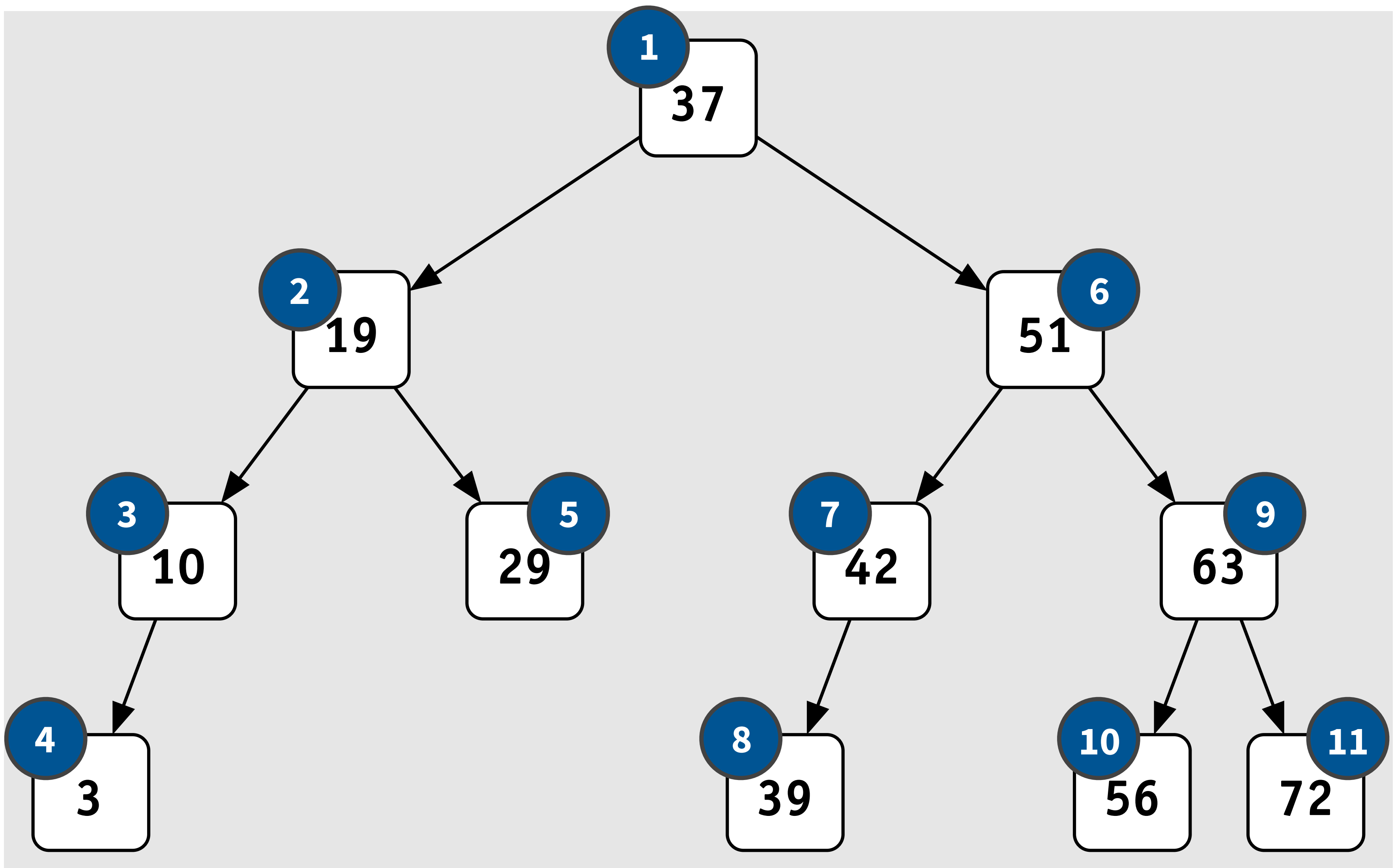
# Binary trees

**structure constraint**: every node has at most two children



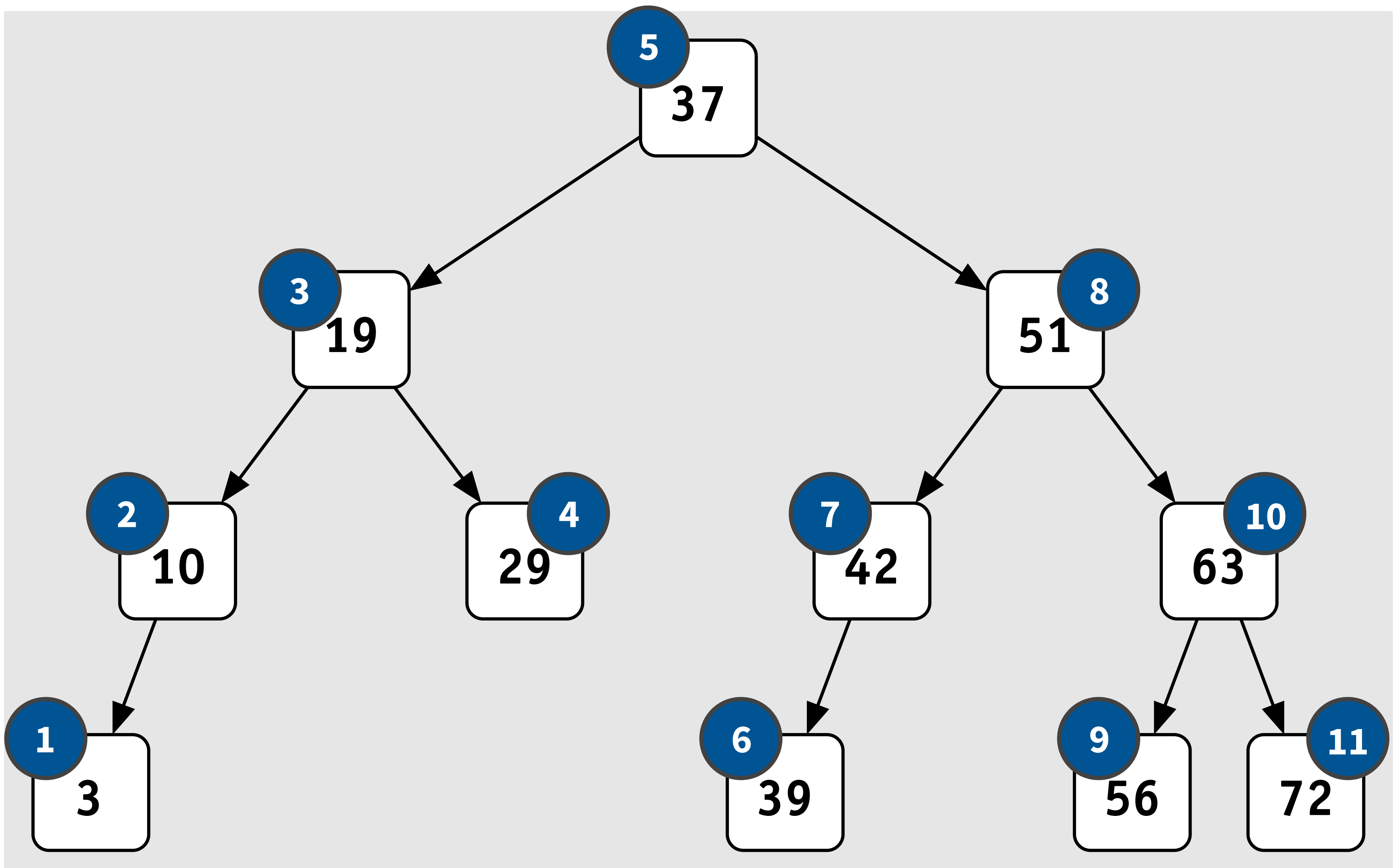
# Preorder traversal

Visit the root, then preorder traverse the left subtree, then preorder traverse the right subtree



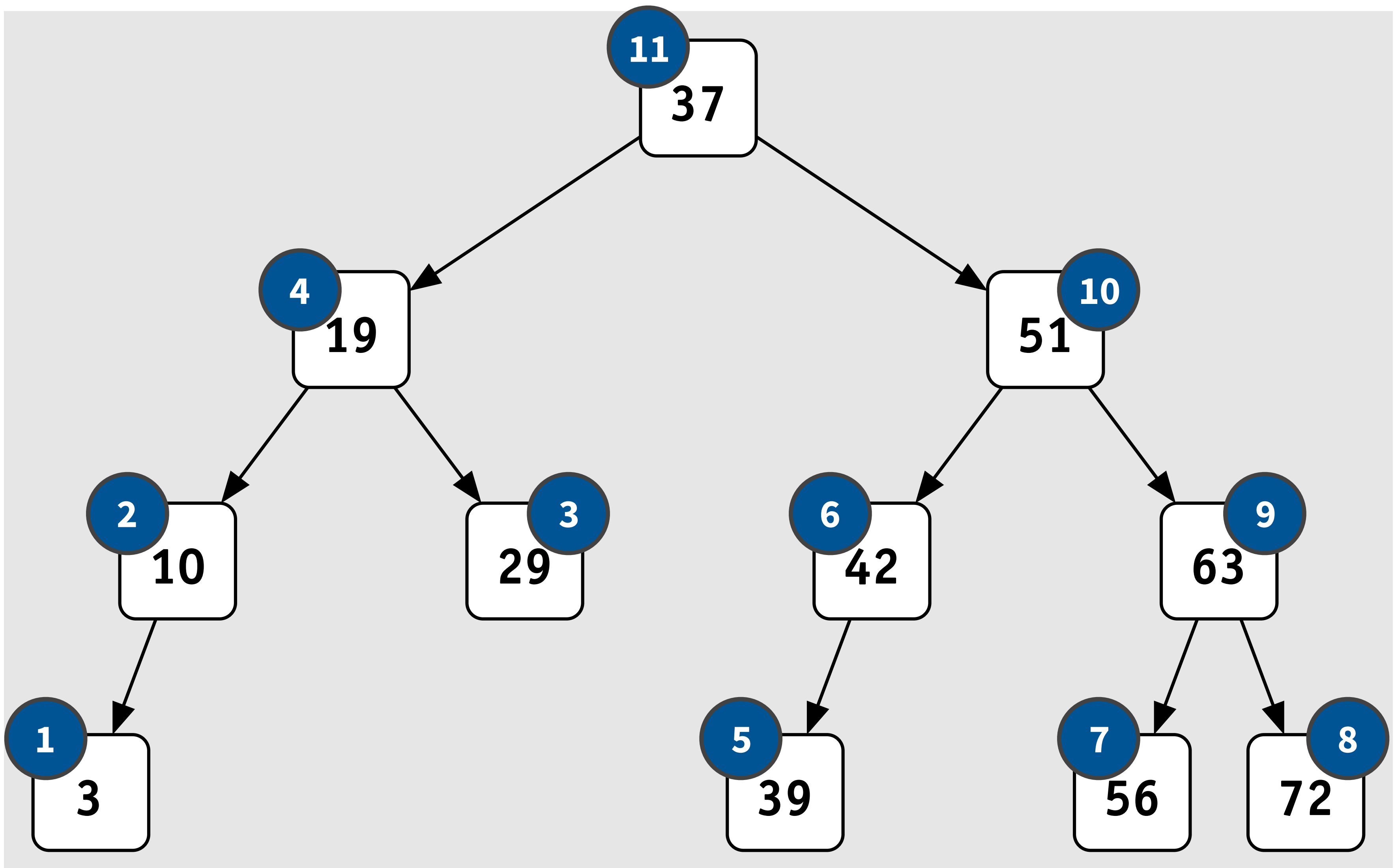
# Inorder traversal

Inorder traverse the left subtree, then visit the root, then inorder traverse the right subtree



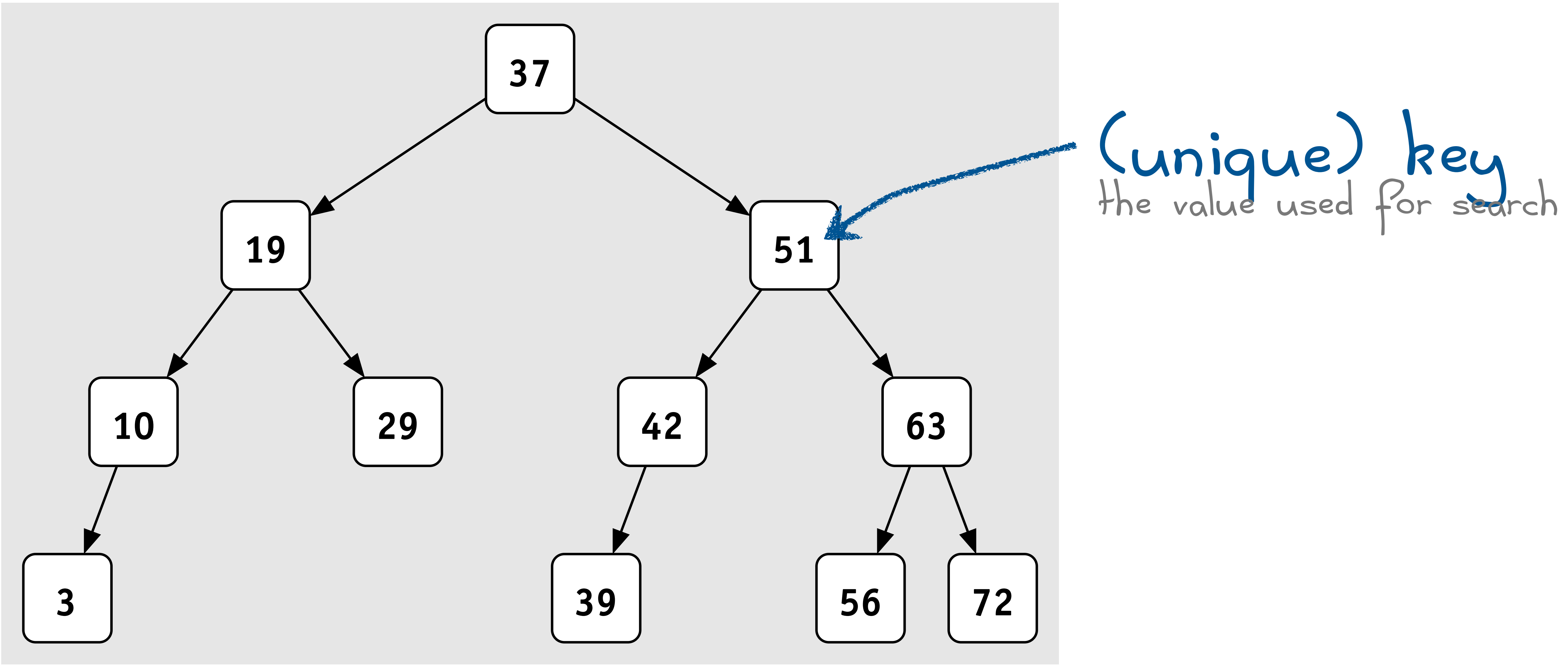
# Postorder traversal

Postorder traverse the left subtree, then postorder traverse the right subtree, then visit the root



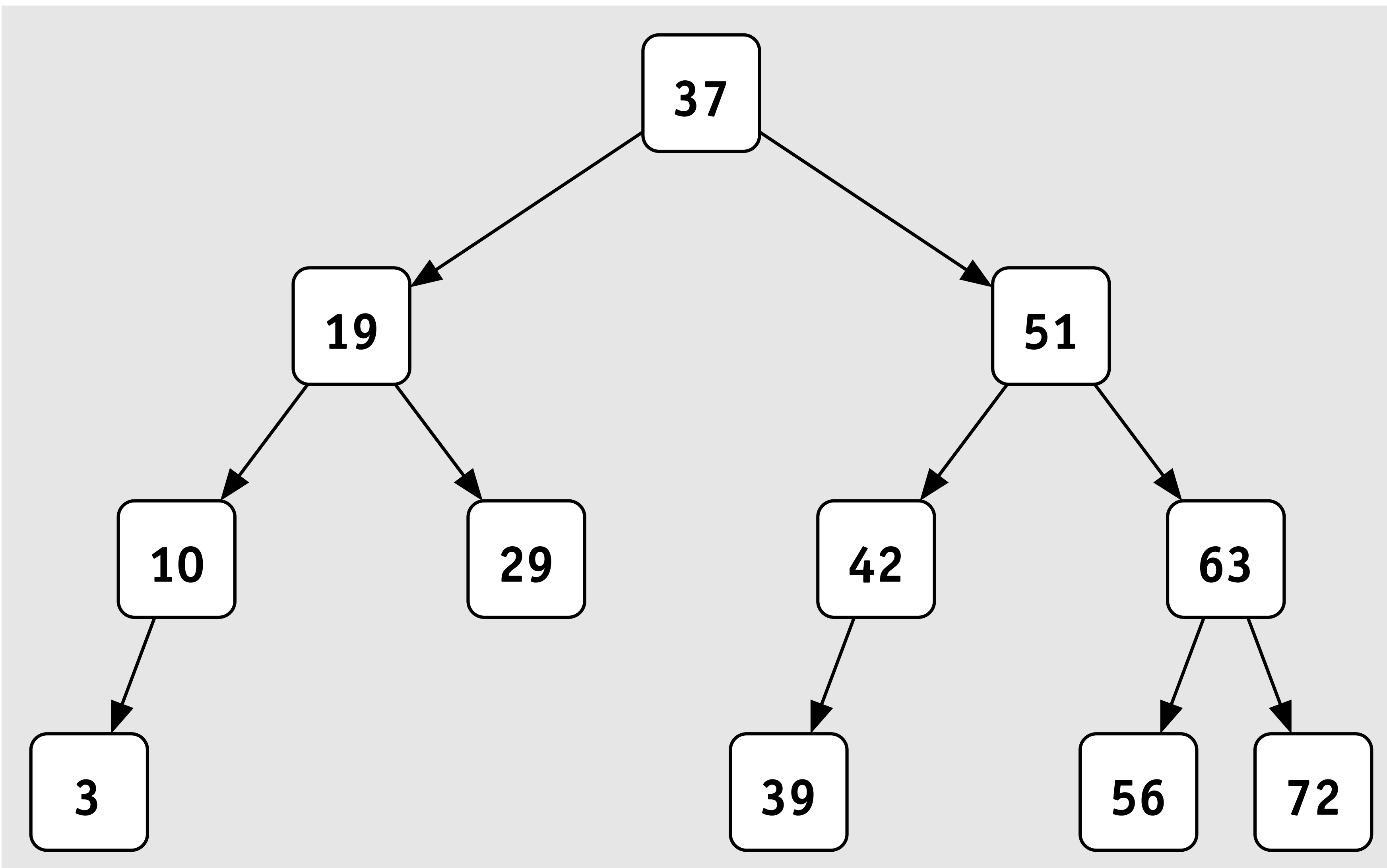
# Binary *search* trees (BSTs)

**order constraint**: every parent is greater than all the nodes in its left subtree and less than all the nodes in the right



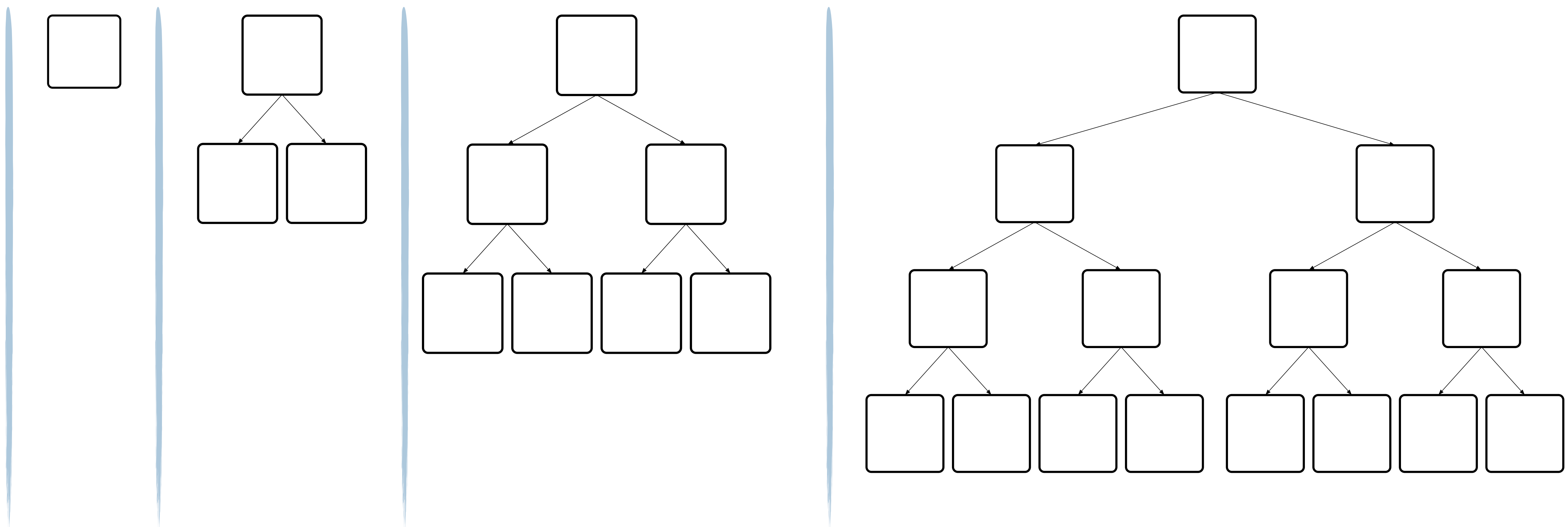
# *Balanced* binary search trees

**structure constraint**: every subtree is about the same size as its sibling



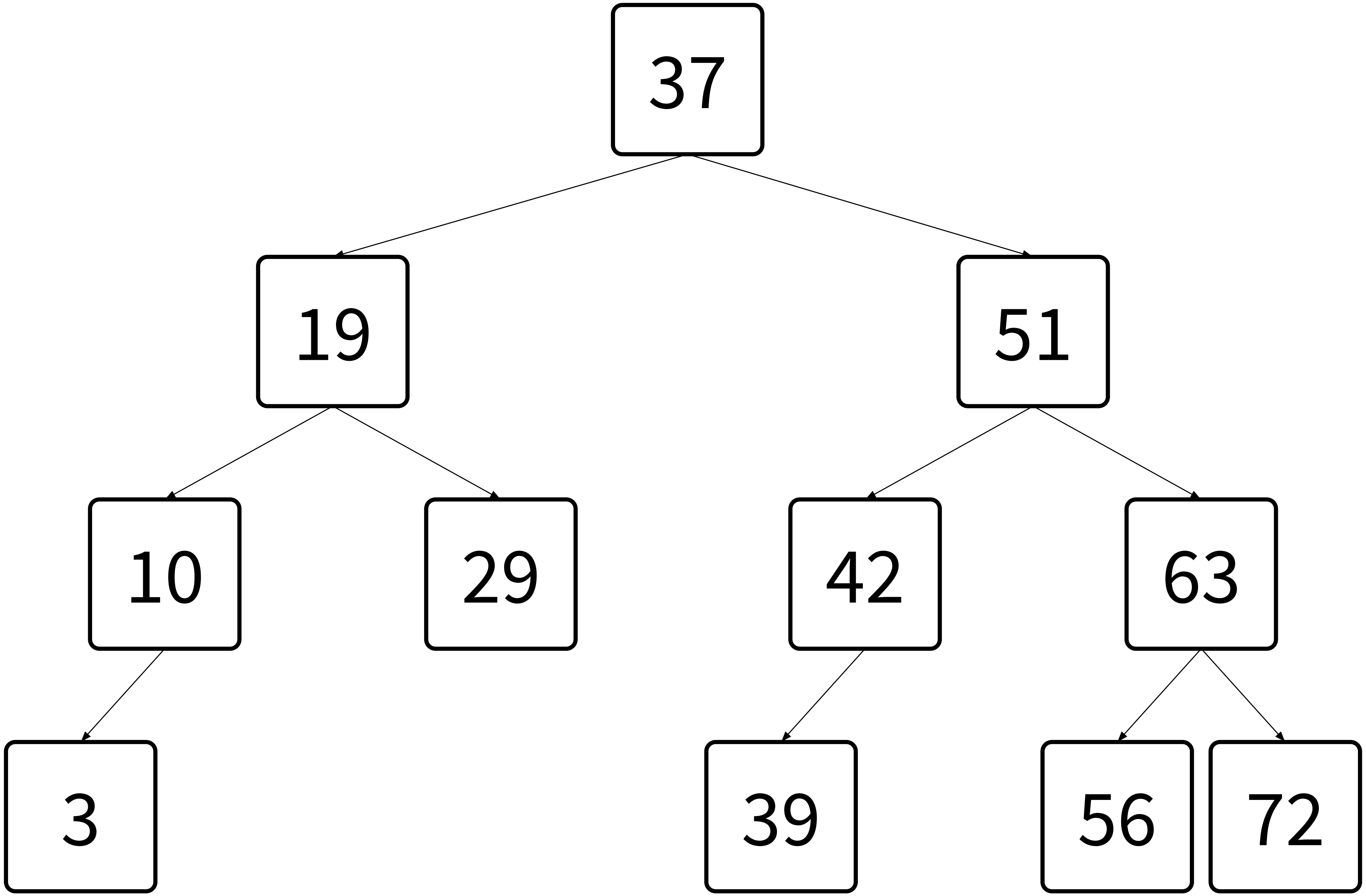
# Perfect trees

**structure:** all leaves are at the same level and every level is full



- Most trees aren't perfect (why not?)
- But perfect trees are useful for analyzing balanced trees.

# BST algorithm: find



# BST algorithm: find

## Given a BST *values* and a number *i*:

find(i, values):

  If the tree is empty, return false.

  Let key be the value at the root of the tree.

  If key is i, return true.

  If i < key, call find on the left subtree.

  If i > key, call find on the right subtree.

# BST algorithm: insert

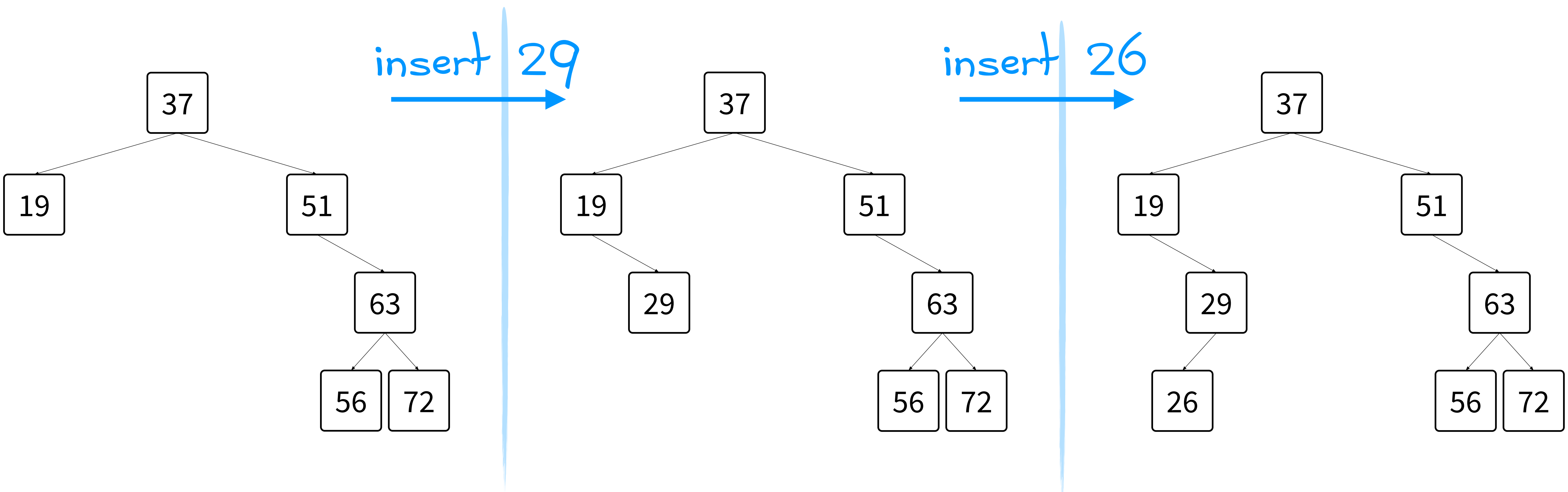
we'll assume that i is not in the tree

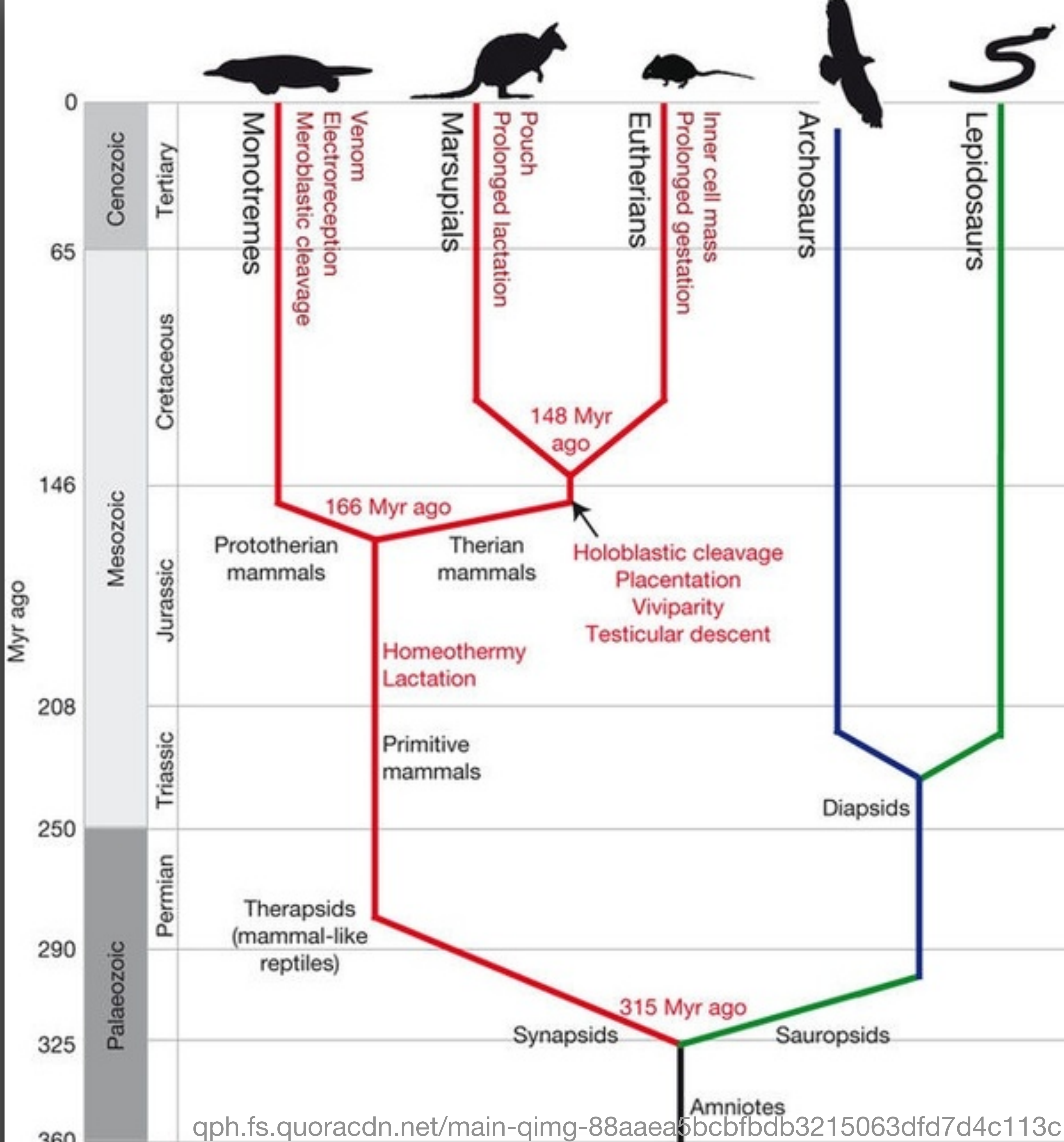
Given a BST *values* and a number *i*:

insert(i, values):

Look for i in values.

**Insert i as a leaf** where it should be.




Monotremes
Venom
Electroreception
Meroblastic cleavage
Marsupials
Pouch
Prolonged lactation
Eutherians
Inner cell mass
Prolonged gestation
Archosaurs
Lepidosaurs
0
65
146
208
250
290
325
360
Myr ago
Cenozoic
Mesozoic
Palaeozoic
Tertiary
Cretaceous
Jurassic
Triassic
Permian
148 Myr ago
166 Myr ago
Prototherian mammals
Therian mammals
Holoblastic cleavage
Placentation
Viviparity
Testicular descent
Homeothermy
Lactation
Primitive mammals
Diapsids
Therapsids (mammal-like reptiles)
315 Myr ago
Synapsids
Sauropsids
Amniotes

# Designing and implementing a new data structure

Interface and implementation

- Interface
  Answers: **what** can this data structure do
- Implementation: encoding
  Answers: **how** the structure is stored, using existing data structures
- Implementation: operations
  Answers: **how** the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.

We'll talk only about the interface for trees
(but you have access to the code for the implementation).

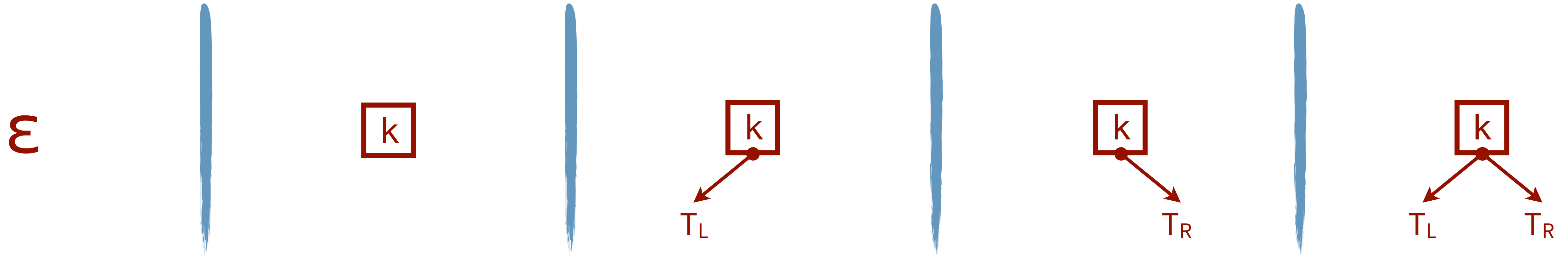
# Our Racket trees: Interface

Inductive data structure, manipulated via constructors, accessors, and operations

| **constructors**<br>*put together* | **accessors**<br>*take apart* | **operations**<br>*often recursive* |
|---|---|---|
| empty-tree | (empty-tree? <*tree*>) | (size <*tree*>) |
| | (leaf? <*tree*>) | (height <*tree*>) |
| (make-leaf <*key*>) | (root <*tree*>) | (find <*value*> <*tree*>) |
| | (left <*tree*>) | (insert <*value*> <*tree*>) |
| (make-tree <*key*> <*left*> <*right*>) | (right <*tree*>) | (traverse-inorder <*tree*>)<br>(traverse-preorder <*tree*>)<br>(traverse-postorder <*tree*>) |

Our BSTs won't be balanced.

# What are some good test cases for trees?



# size

Firstname Lastname Th. 10 / 18

```
; the number of nodes in the tree
(define (size tree)
```

(Your response)

# size

Firstname Lastname Th. 10 / 18

```
; the number of nodes in the tree
(define (size tree)
  (if (empty-tree? tree)
    0
    (+ 1
       (size (left tree))
       (size (right tree)))))
```

# Worst-case analysis

How bad can it get?

## Given a collection of size N and an operation:

What's the worst input for the operation?

How expensive is the operation, for that input? (cost = # of elements visited)

| | find | insert | min |
|---|---|---|---|
| **List** | | | |
| **Tree** | | | |
| **Binary search tree (BST)** | | | |

For trees (including unbalanced BSTs), the worst-case version of an N-element tree is a "stick" (i.e., a linked list).

# Worst-case analysis

How bad can it get?

## Given a collection of size N and an operation:

What's the worst input for the operation?

How expensive is the operation, for that input? (cost = # of elements visited)

| | find | insert | min | list elements in order |
|---|---|---|---|---|
| **List** | O(n) elements visited | O(1) elements visited | O(n) elements visited | O(n log n) elements visited |
| **Tree** | O(n) elements visited | O(1) elements visited | O(n) elements visited | O(n log n) elements visited |
| **Binary search tree (BST)** | O(n) elements visited | O(n) elements visited | O(n) elements visited | O(n) elements visited |
| ***balanced*** **Binary search tree (BST)** | O(log n) elements visited | O(log n) elements visited | O(log n) elements visited | O(n) elements visited |

For trees (including unbalanced BSTs), the worst-case version of an N-element tree is a "stick" (i.e., a linked list).

# Analyze `size`, using a recurrence relation

For a given cost metric: additions; on the worst-case input: a stick

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?
2. Translate the recursive case(s), using input size N
   Define T(N) recursively, in terms of smaller cost.

```
(define (size tree)
  (if (empty-tree? tree)
    0
    (+ 1 (size (left tree)) (size (right tree)))))
```

$T(0) = 0$

$T(N) = 1 + T(N-1) + 0$

$$
\begin{aligned}
T(N) &= 1 + T(N-1) + 0 & &= 1*1 + T(N-1) \\
&= 1 + 1 + T(N-2) & &= 2*1 + T(N-2) \\
&= 1 + 1 + 1 + T(N-3) & &= 3*1 + T(N-3) \\
&\ldots & &\ldots \\
&= 1 + 1 + 1 + \ldots 1 + T(N-N) & &= N*1 + T(N-N) = N \in O(N)
\end{aligned}
$$