LECTURE 16

TREES

MCS 275 Spring 2023
David Dumas

RECENTLY

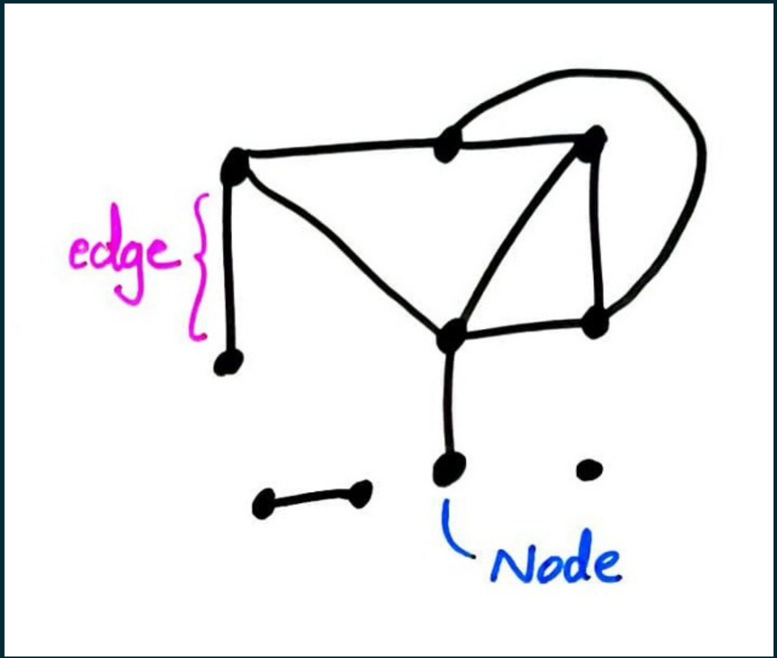
We've talked about some recursive algorithms

NEXT

Recursive **data structures**!

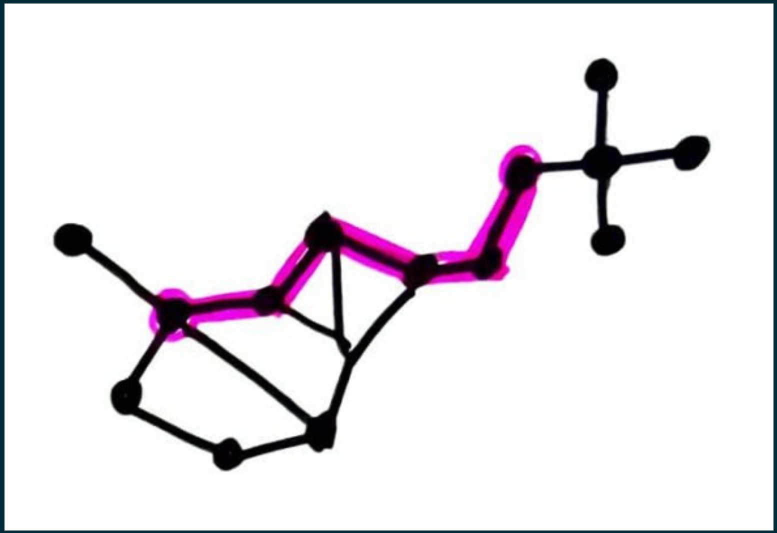
GRAPHS

In mathematics, a graph is a collection of **nodes** (or vertices) and **edges** (which join pairs of nodes).



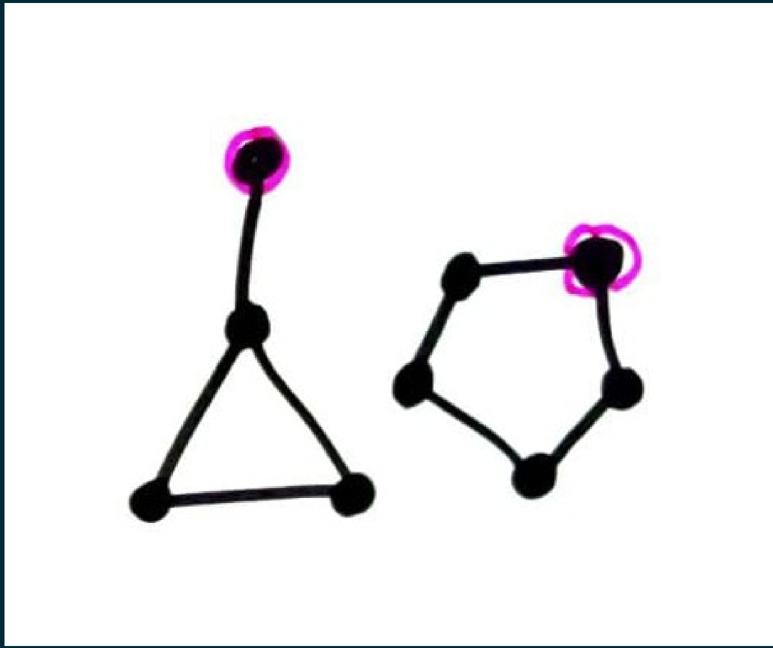
CONNECTIVITY

A graph is **connected** if every pair of nodes can be joined by *at least* one path.



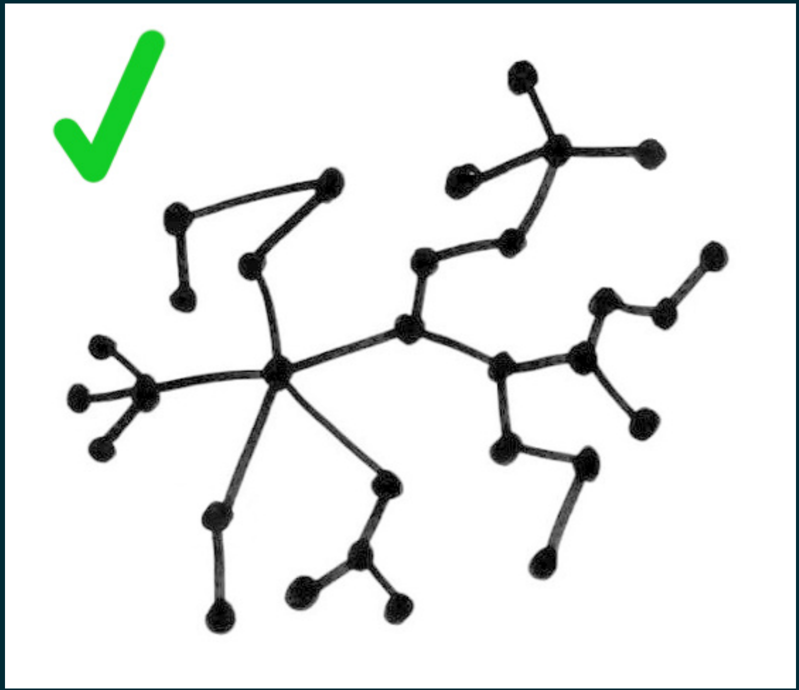
CONNECTIVITY

A graph is **connected** if every pair of nodes can be joined by *at least* one path.



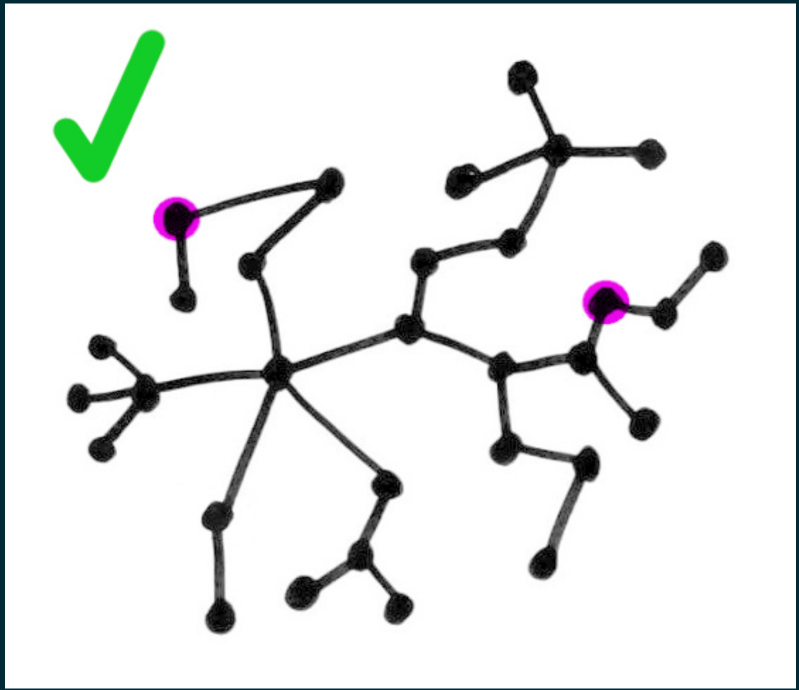
TREE

A **tree** is a graph where every pair of nodes can be joined by *exactly* one path (no more, no less).



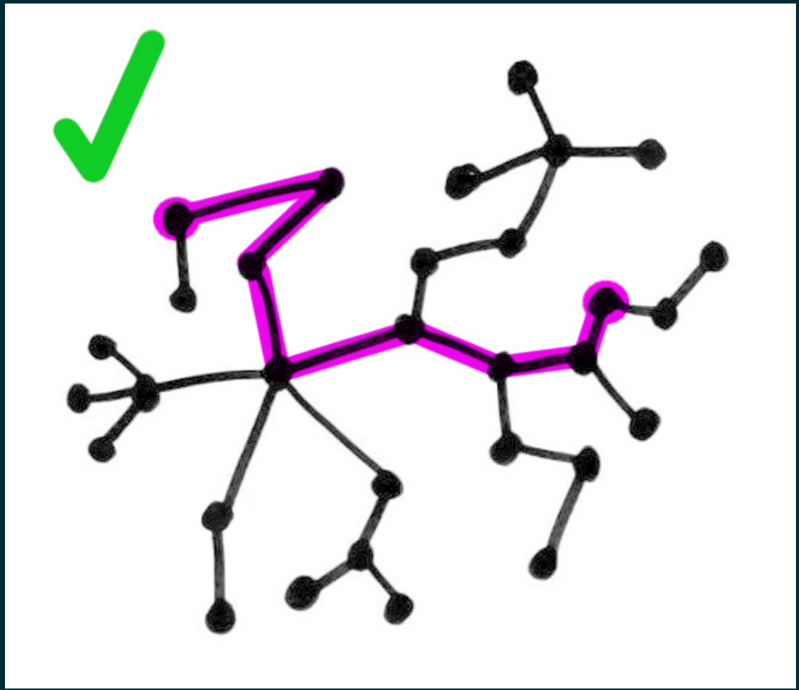
TREE

A **tree** is a graph where every pair of nodes can be joined by *exactly* one path (no more, no less).



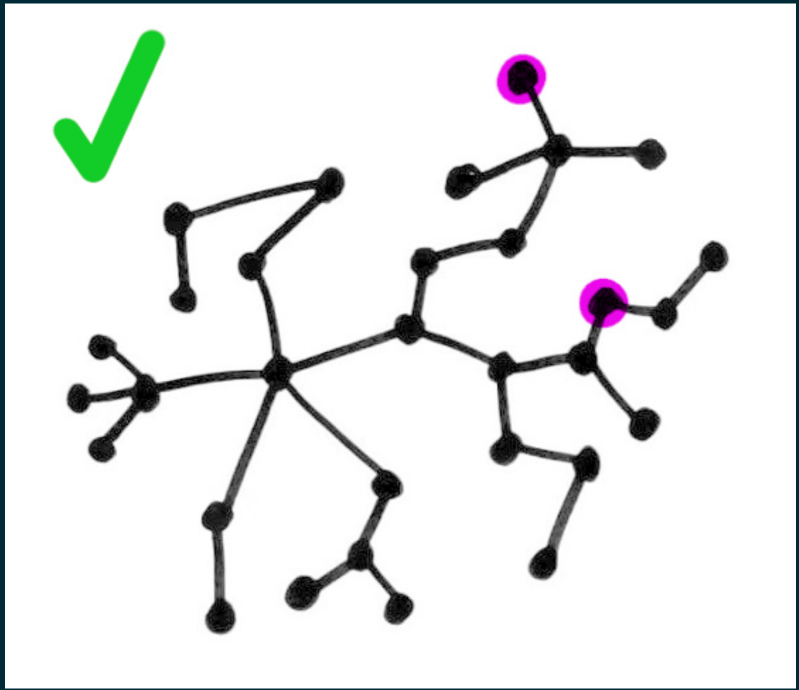
TREE

A **tree** is a graph where every pair of nodes can be joined by *exactly* one path (no more, no less).



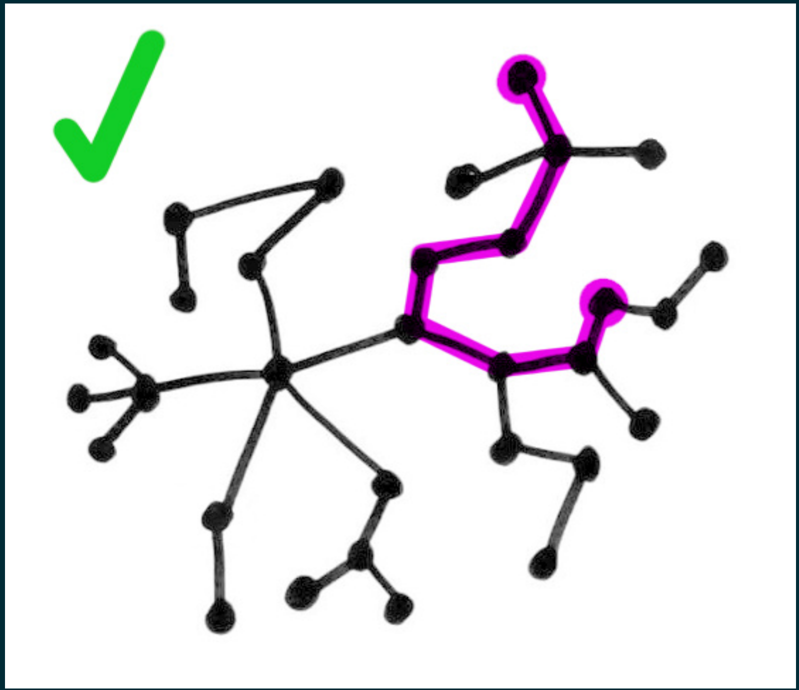
TREE

A **tree** is a graph where every pair of nodes can be joined by *exactly* one path (no more, no less).



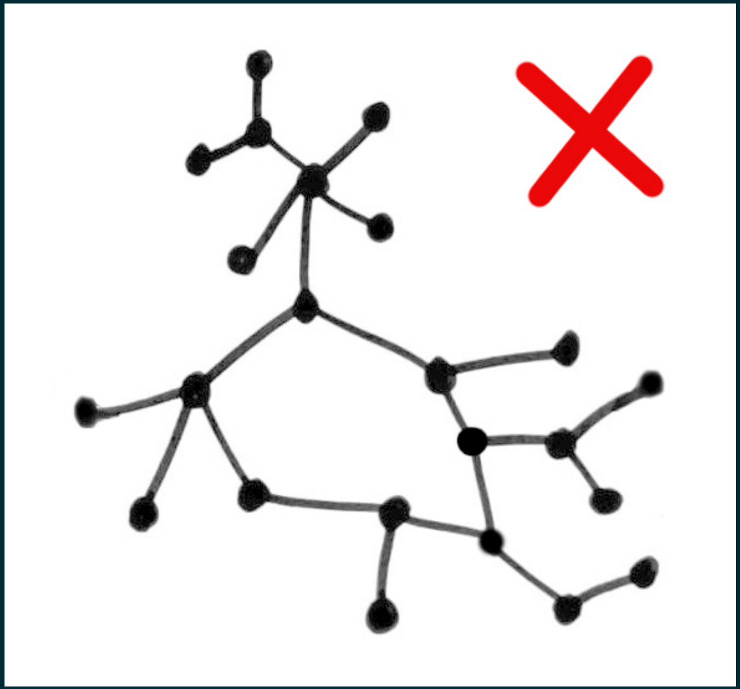
TREE

A **tree** is a graph where every pair of nodes can be joined by *exactly* one path (no more, no less).



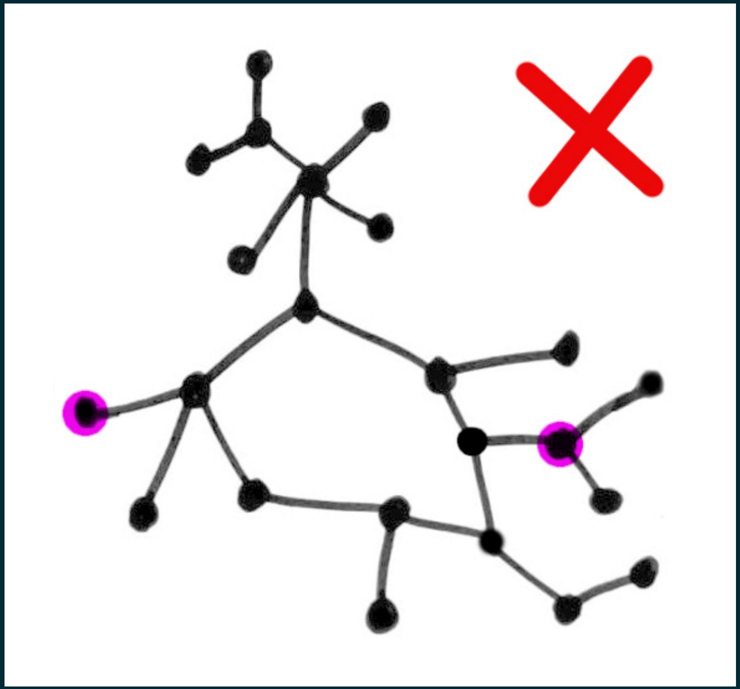
TREE

A **tree** is a graph where every pair of nodes can be joined by *exactly* one path (no more, no less).



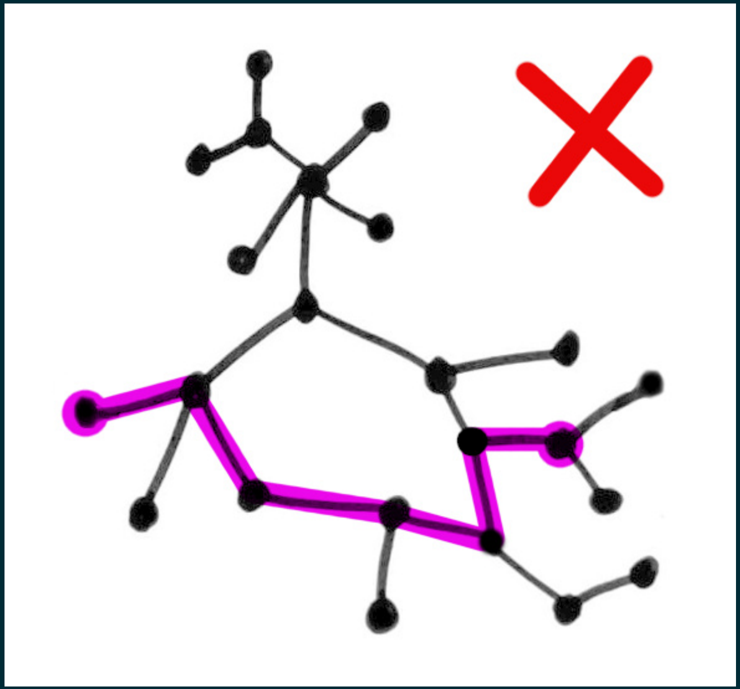
TREE

A **tree** is a graph where every pair of nodes can be joined by *exactly* one path (no more, no less).



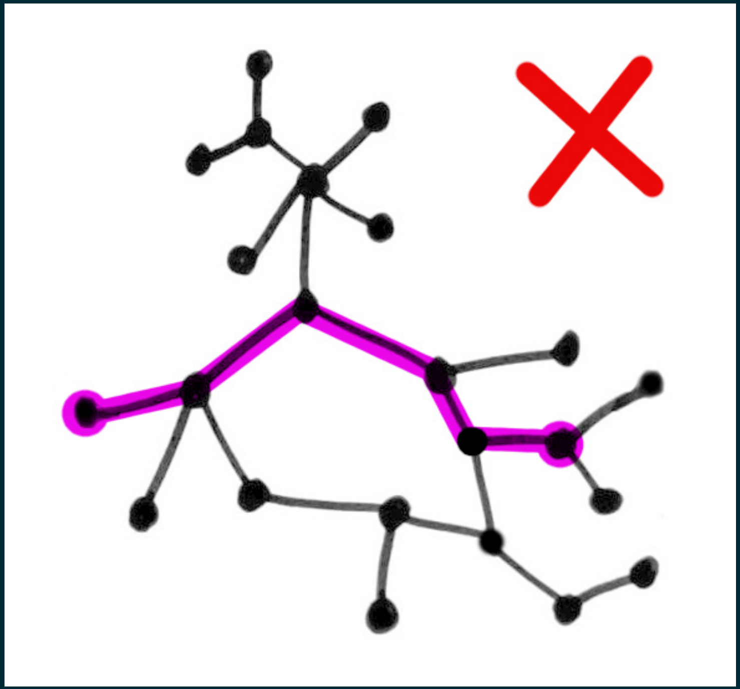
TREE

A **tree** is a graph where every pair of nodes can be joined by *exactly* one path (no more, no less).

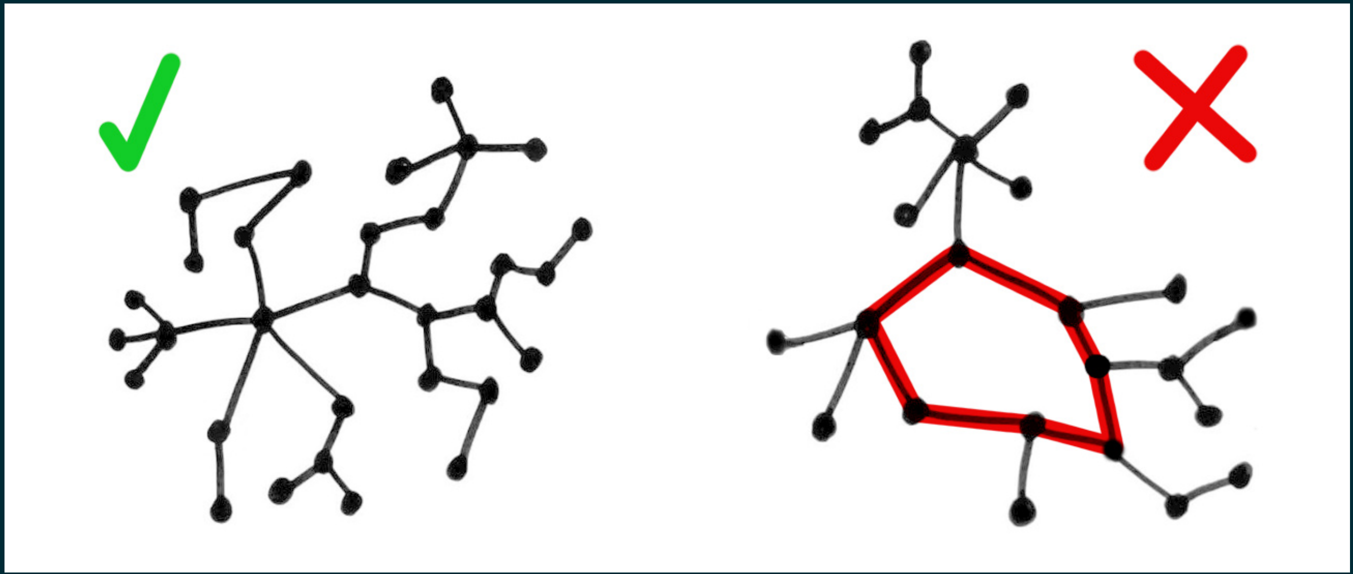


TREE

A **tree** is a graph where every pair of nodes can be joined by *exactly* one path (no more, no less).



Equivalently, a tree is a **connected graph** with **no loops**.



Equivalently, a tree is a connected graph that becomes disconnected if any edge is removed.

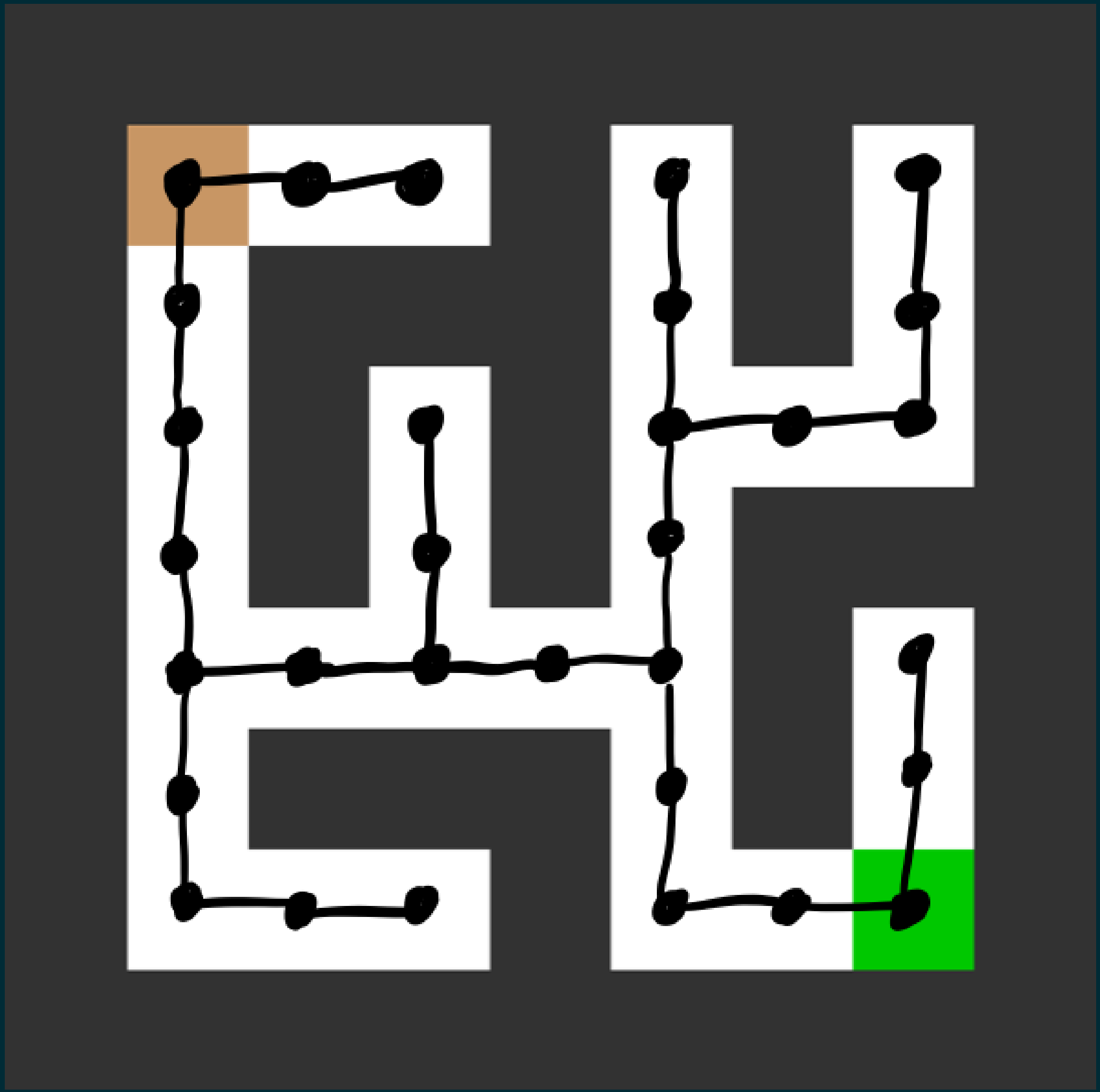
(Exercise: Prove this is an equivalent definition!)

EXAMPLE

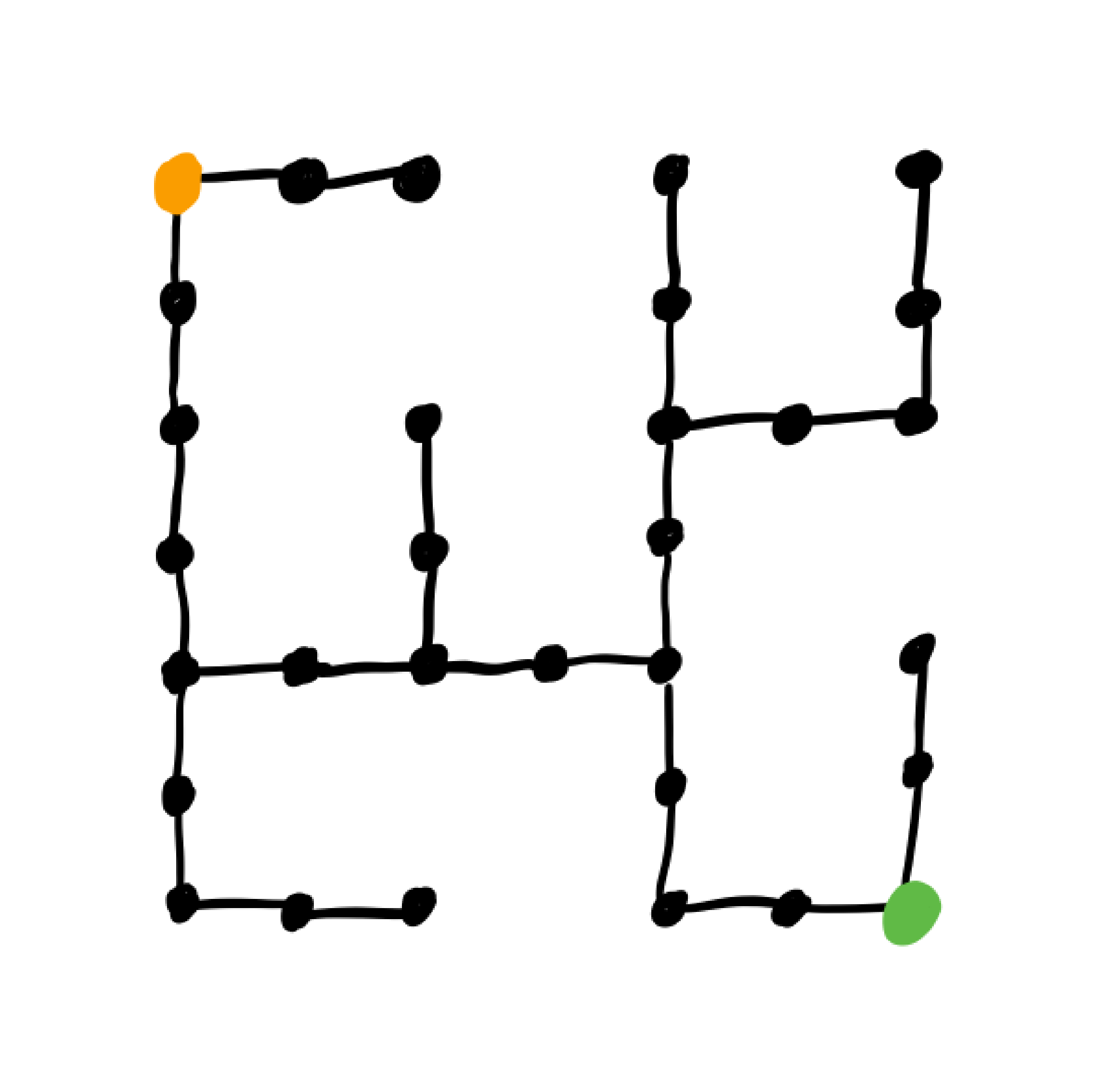
The random mazes produced by `maze.PrimRandomMaze(...)` can be seen as trees, with the nodes being the open squares and edges between nodes if the corresponding squares share an edge.

EXAMPLE

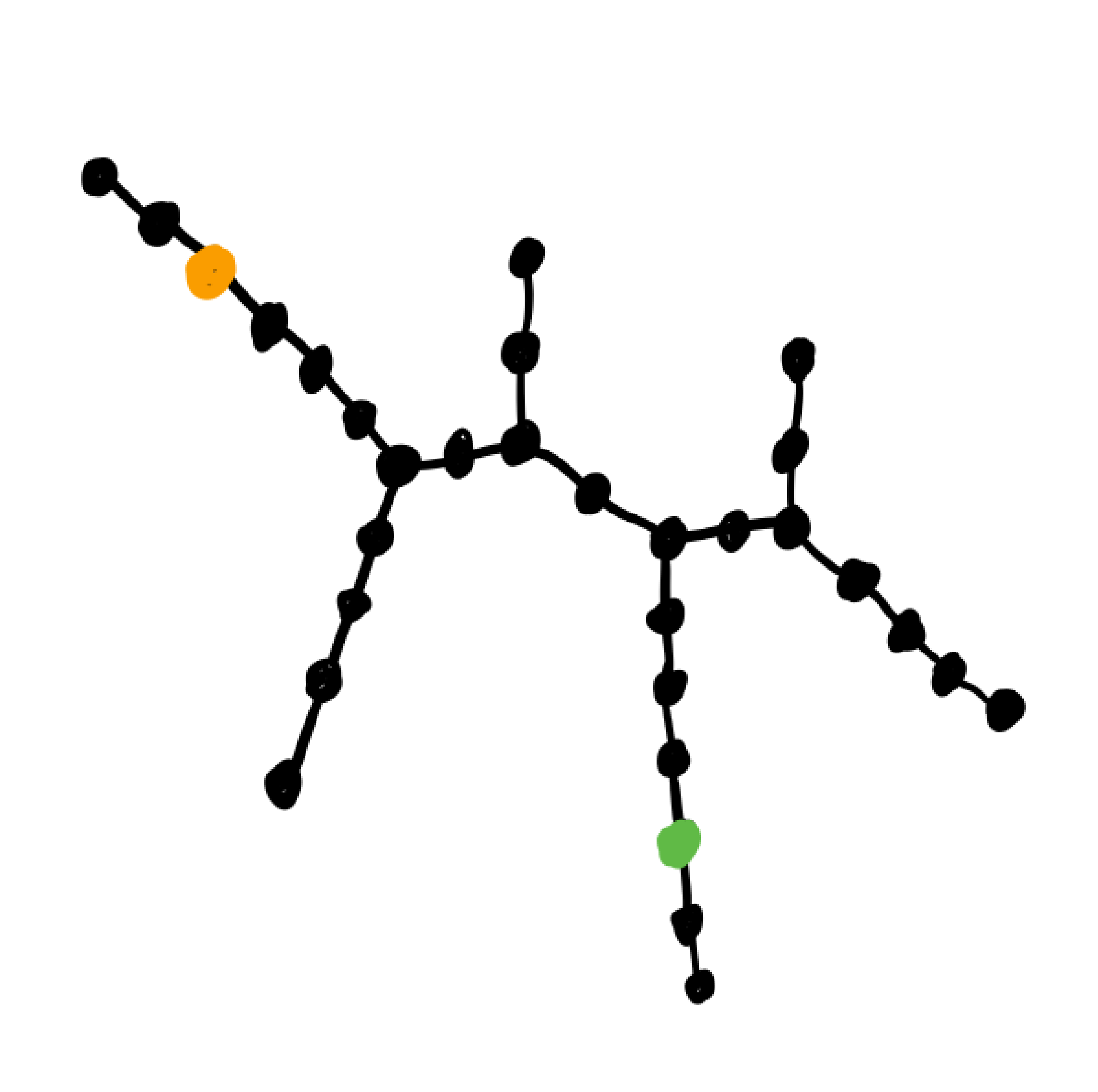
EXAMPLE



EXAMPLE

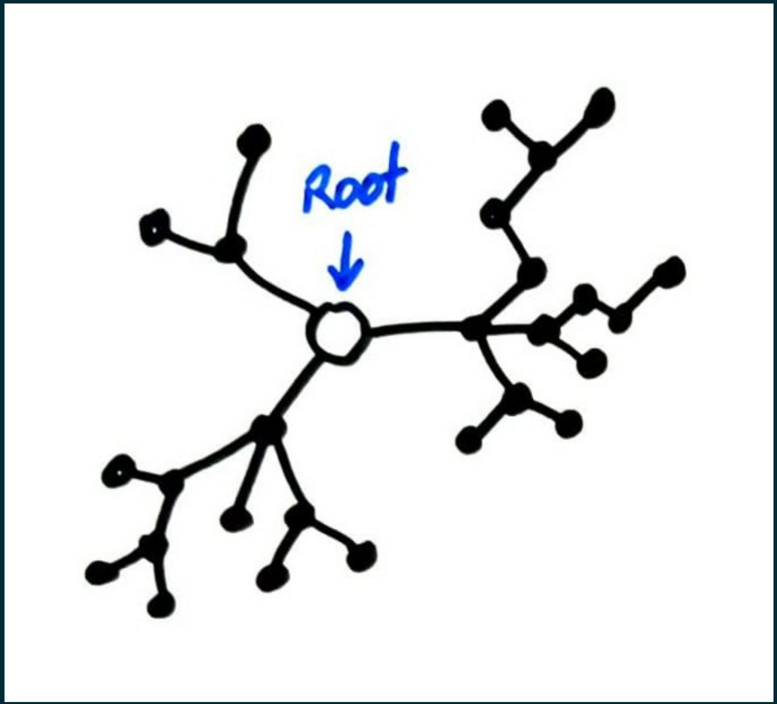


EXAMPLE



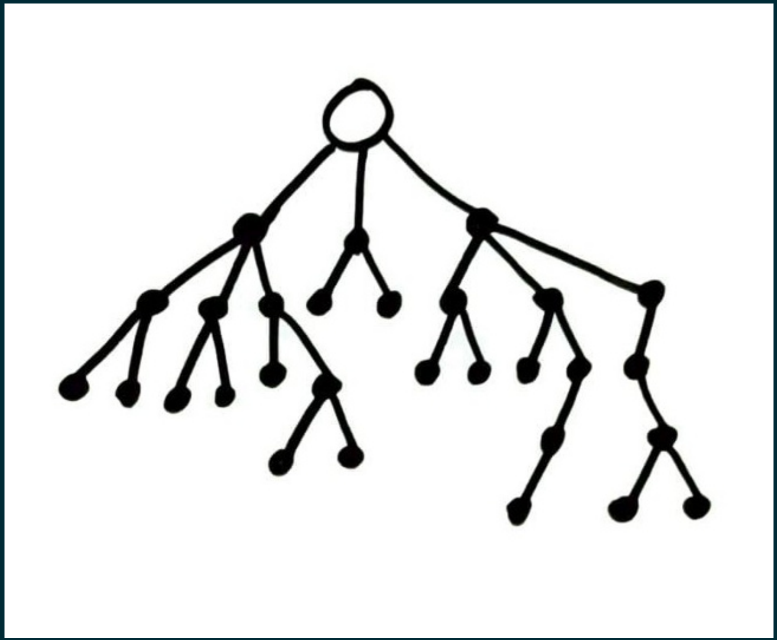
ROOTS AND DIRECTIONS

The trees considered in CS usually have one node distinguished, called the **root**.



There's nothing special about the root except that it is labeled as such. Any node of a tree could be chosen to be its root node.

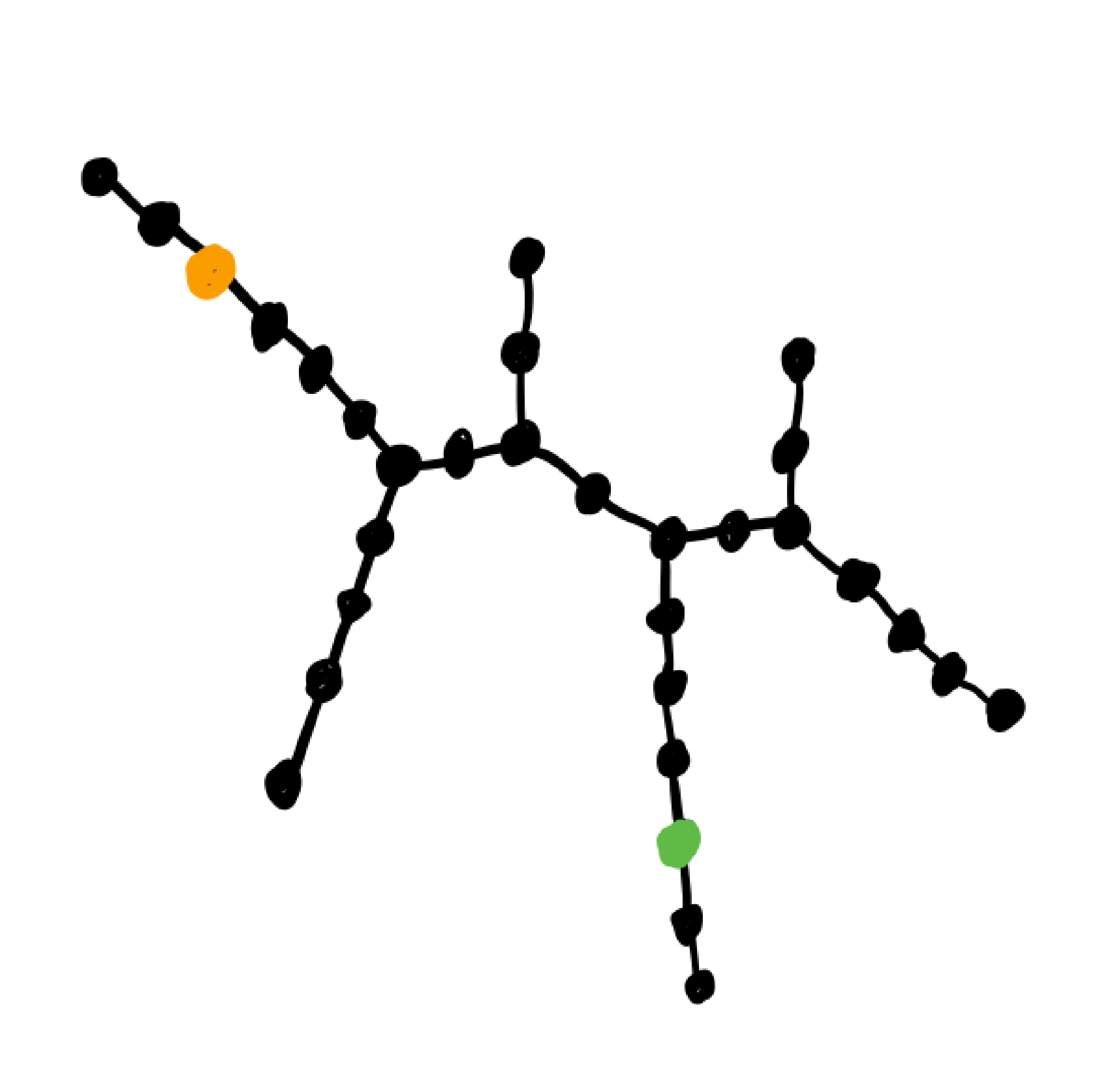
Such *rooted trees* are usually drawn with the root at top



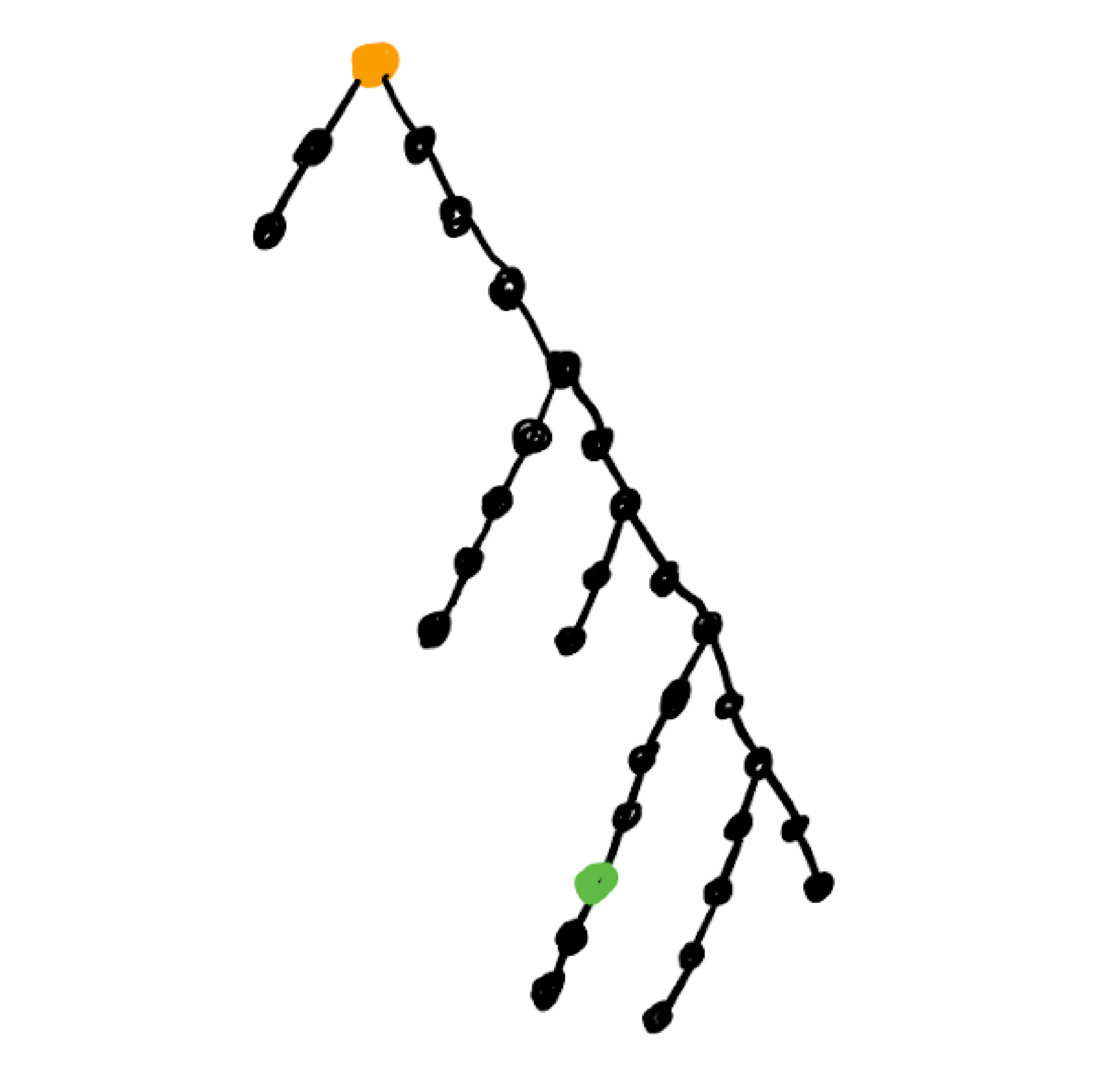
and vertices farther from the root successively lower.

Such diagrams look like trees in the natural world.

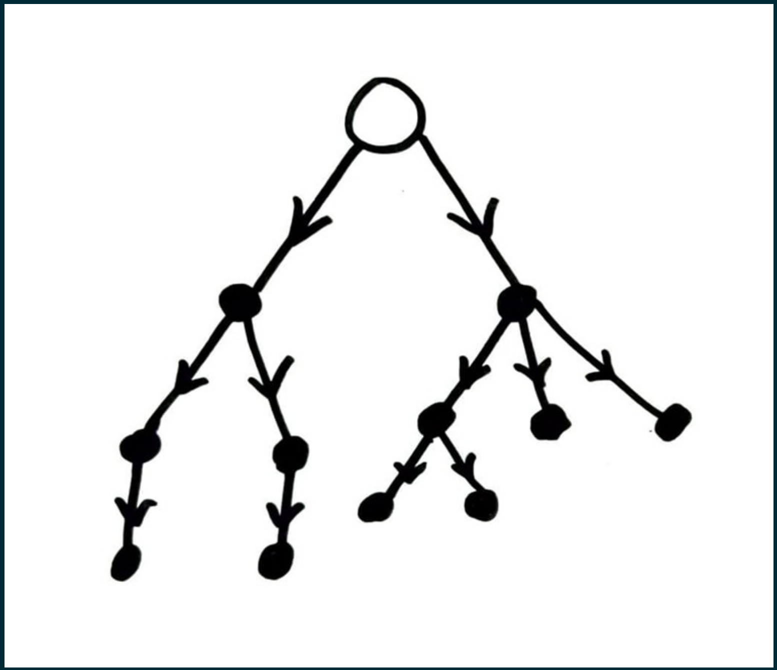
THAT MAZE AGAIN



THAT MAZE AGAIN

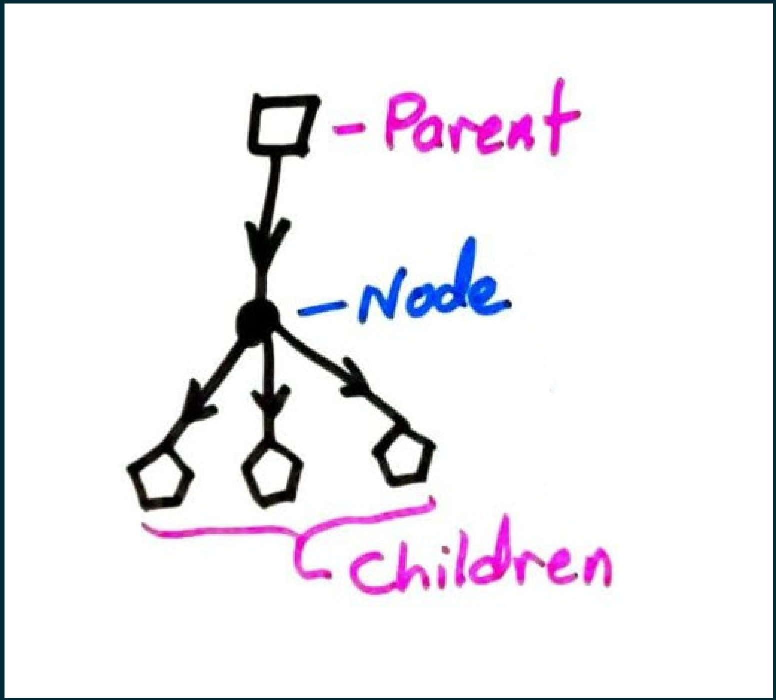


Choosing a root lets us orient all of the edges so they point away from it.



Hence the usual way of drawing a tree will have these arrows pointing downward.

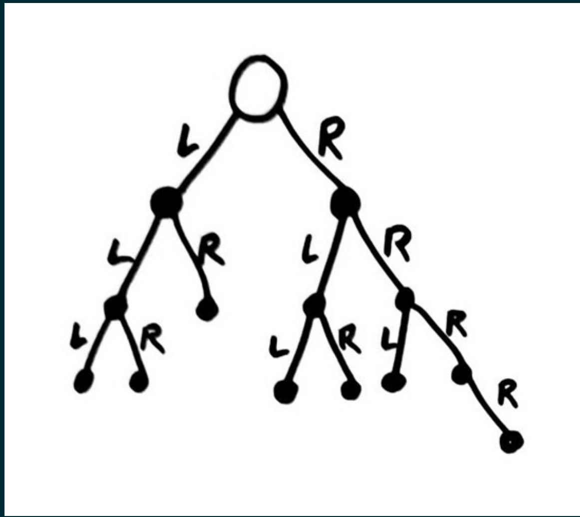
Each node (except the root) has an incoming edge, from its **parent** (closer to the root).



Each node may have one or more outgoing edges, to its **children** (farther from the root).

BINARY TREES

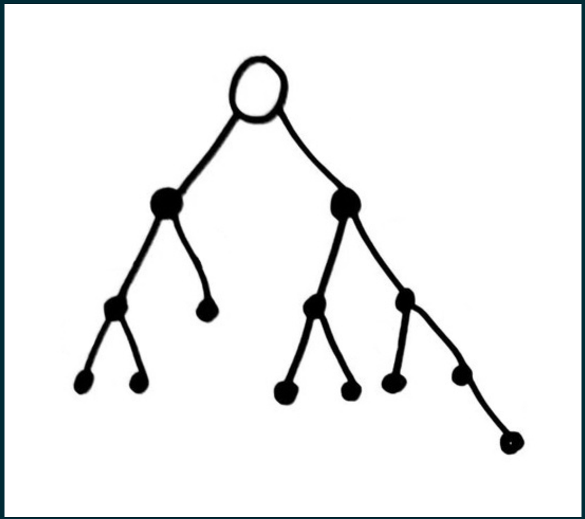
In CS, a binary tree is a (rooted) tree in which every node has ≤ 2 children, labeled "left" and "right".



Horizontal relative position is used to indicate this labeling, rather than explicitly writing it on the edges.

BINARY TREES

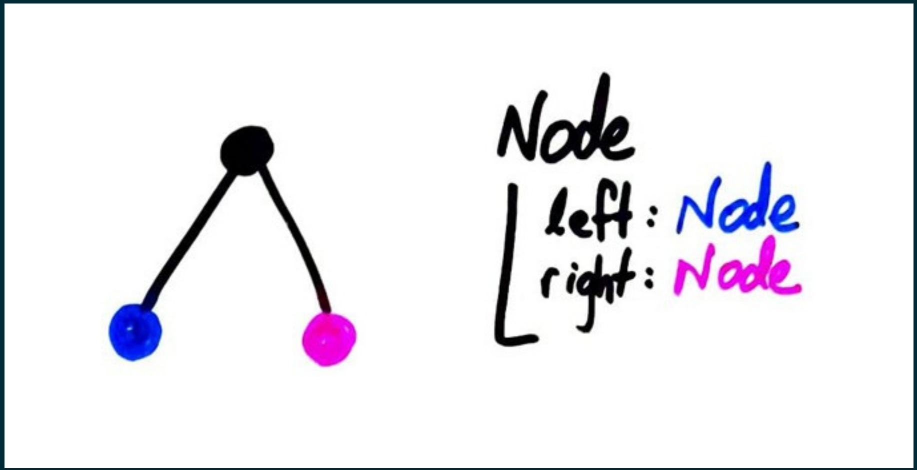
In CS, a binary tree is a (rooted) tree in which every node has ≤ 2 children, labeled "left" and "right".



Horizontal relative position is used to indicate this labeling, rather than explicitly writing it on the edges.

REPRESENTATION

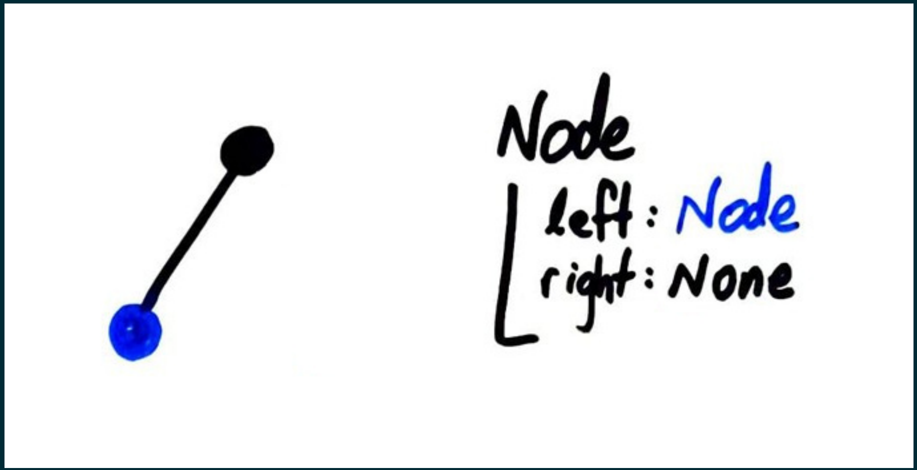
How can we store a tree in Python?



Make a class `Node`, with attribues `left` and `right` that can be `None` or other `Node` objects.

REPRESENTATION

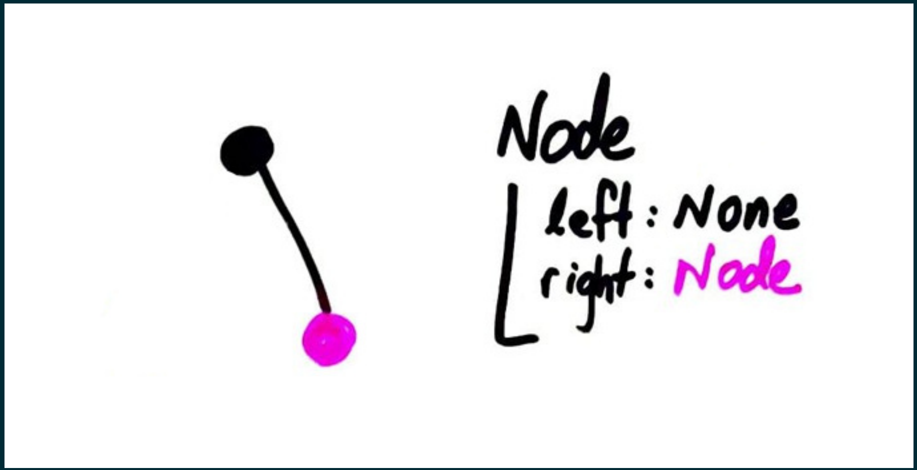
How can we store a tree in Python?



Make a class `Node`, with attribues `left` and `right` that can be `None` or other `Node` objects.

REPRESENTATION

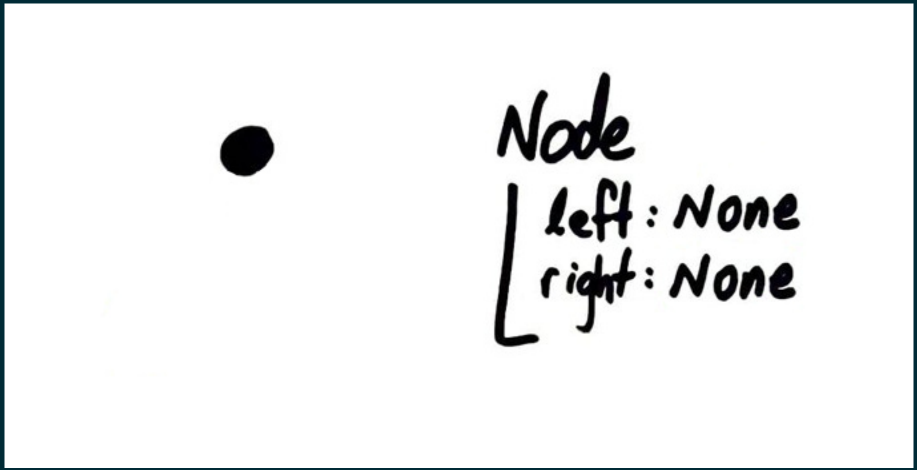
How can we store a tree in Python?



Make a class `Node`, with attribues `left` and `right` that can be `None` or other `Node` objects.

REPRESENTATION

How can we store a tree in Python?



Make a class `Node`, with attribues `left` and `right` that can be `None` or other `Node` objects.

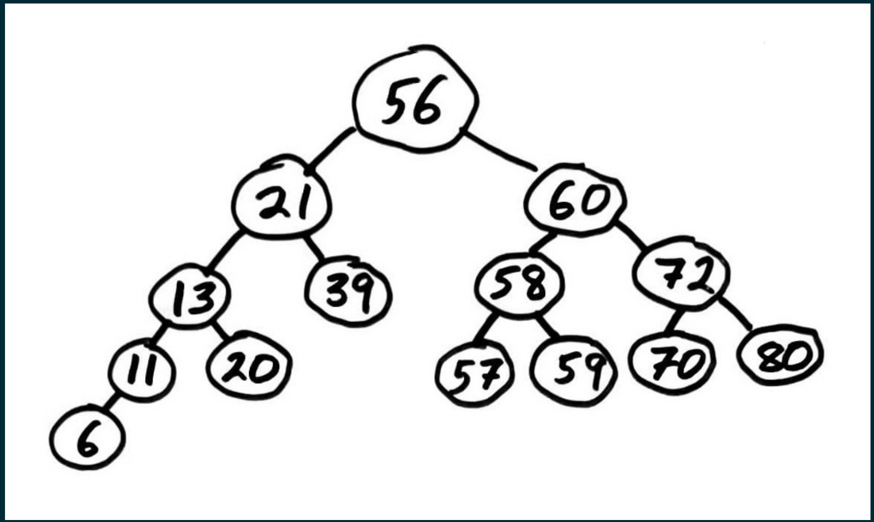
WHY?

We can also store additional information in the nodes of a binary tree. If present, this is called the **key** or *value* or *cargo* of a node.

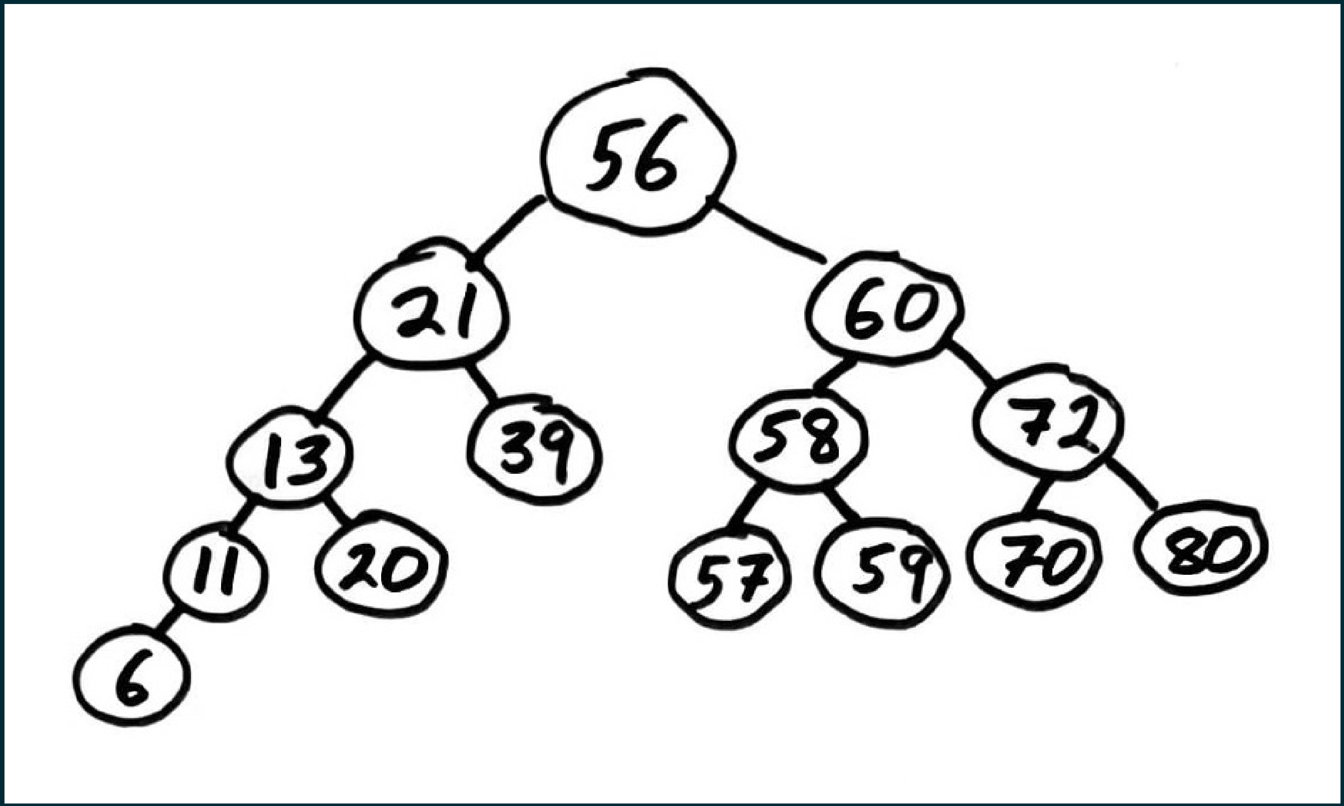
This turns out to be a very efficient data structure for many purposes. A lot of data can be accessed in a few steps from the root node.

BINARY SEARCH TREE

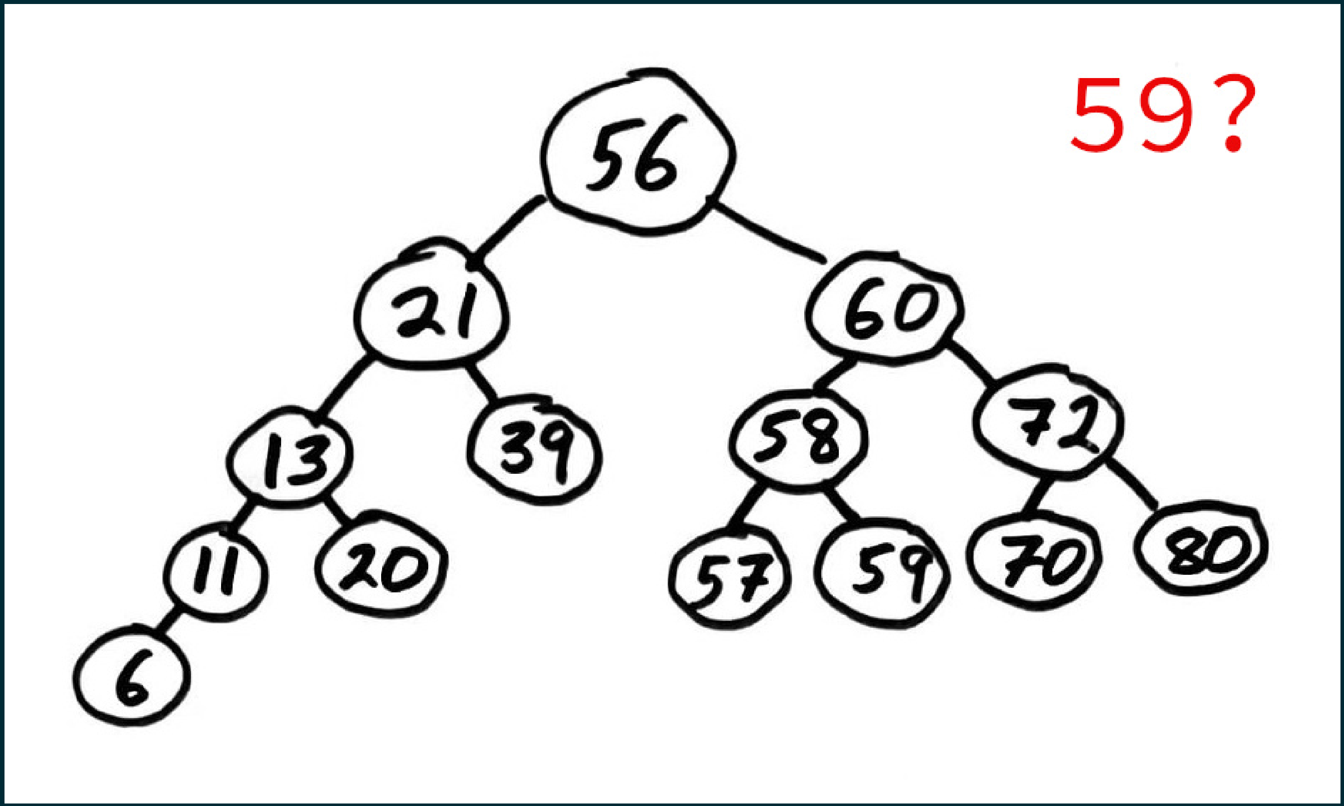
Stores numbers or other objects allowing comparison as node values. Enforce the rule: Anything in the subtree to the left is smaller or equal. Anything in the subtree to the right is greater.



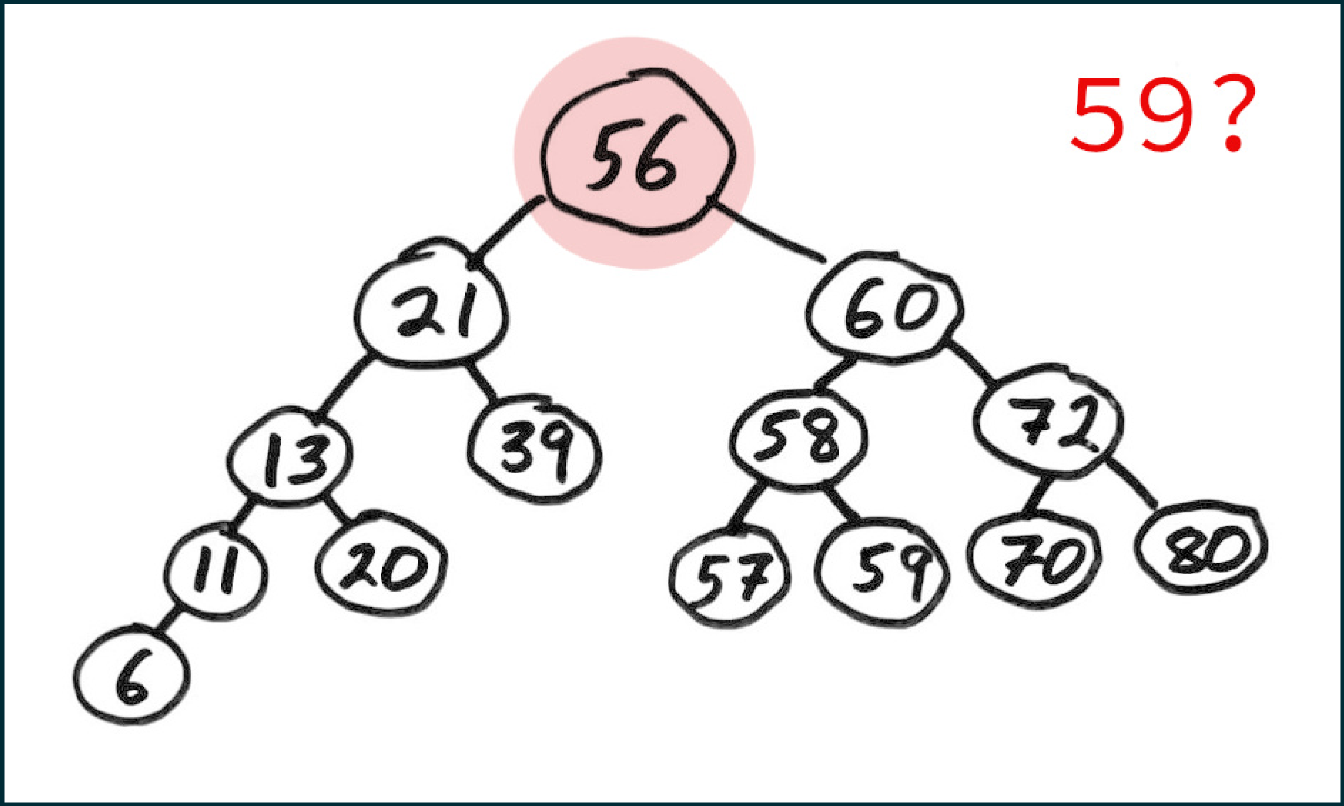
This allows a natural way to check if a value is present with a game of "too high / too low".



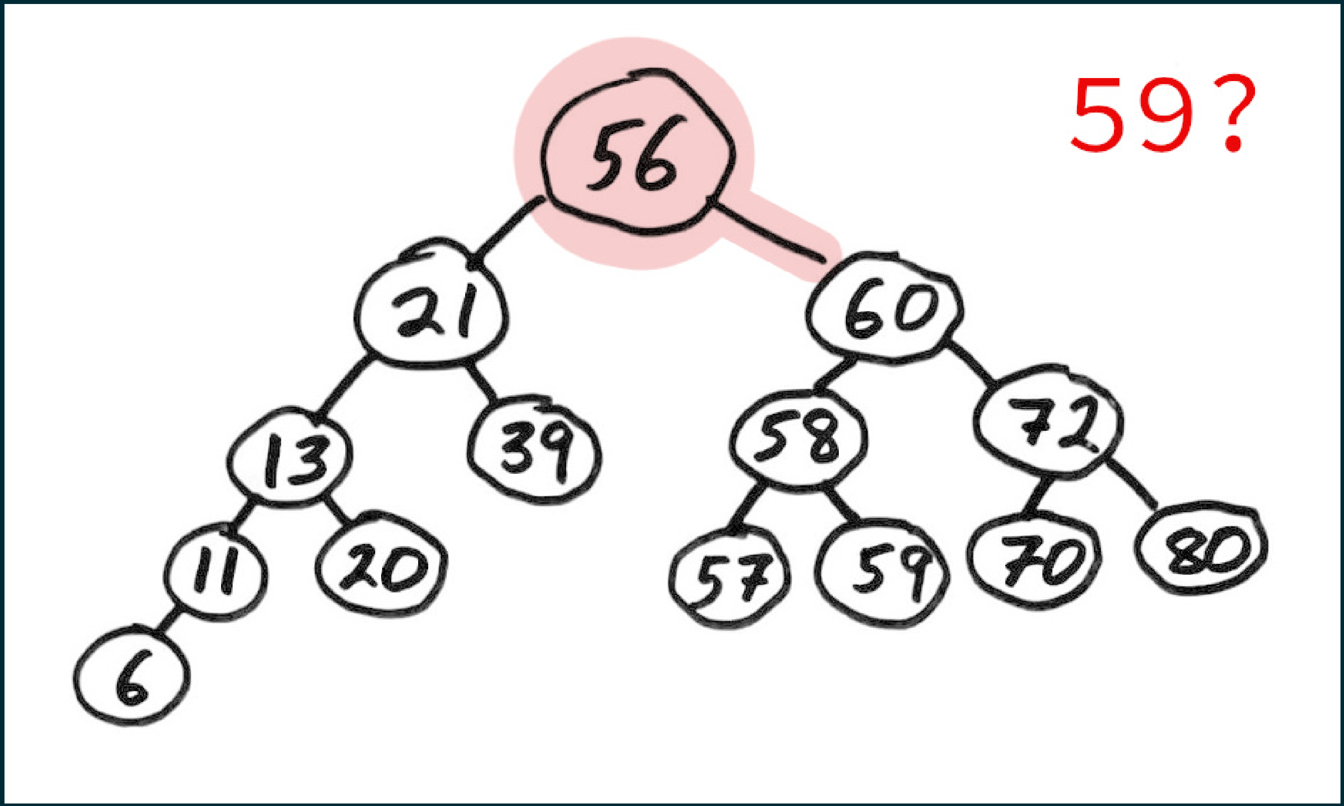
This allows a natural way to check if a value is present with a game of "too high / too low".



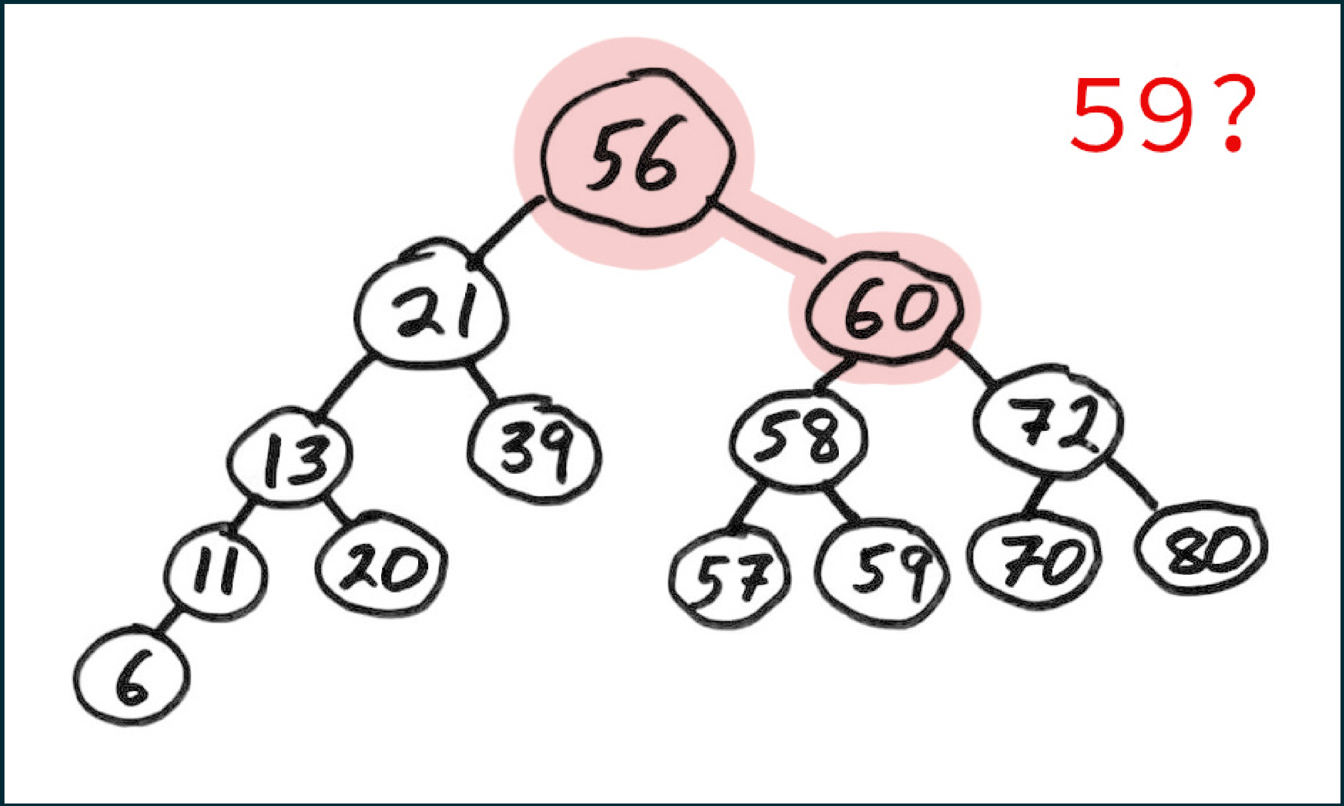
This allows a natural way to check if a value is present with a game of "too high / too low".



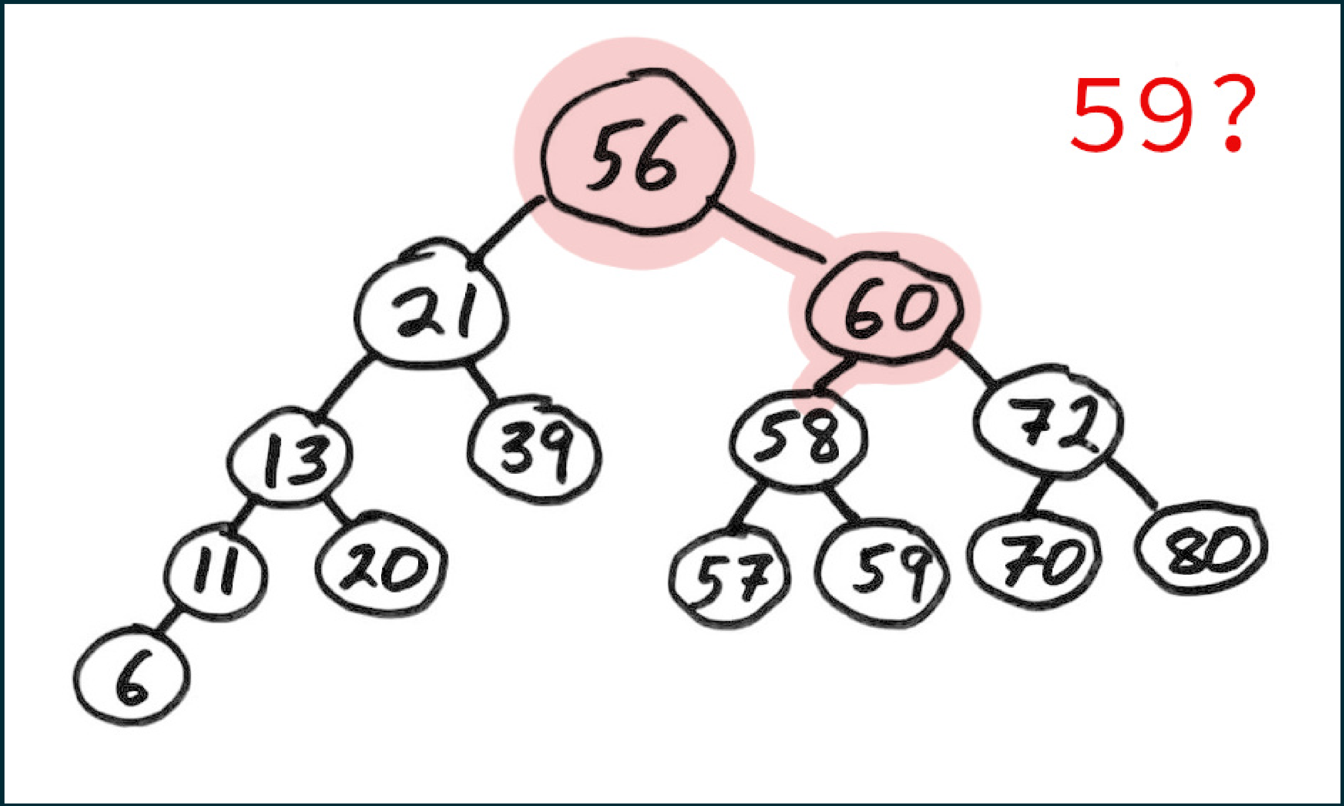
This allows a natural way to check if a value is present with a game of "too high / too low".



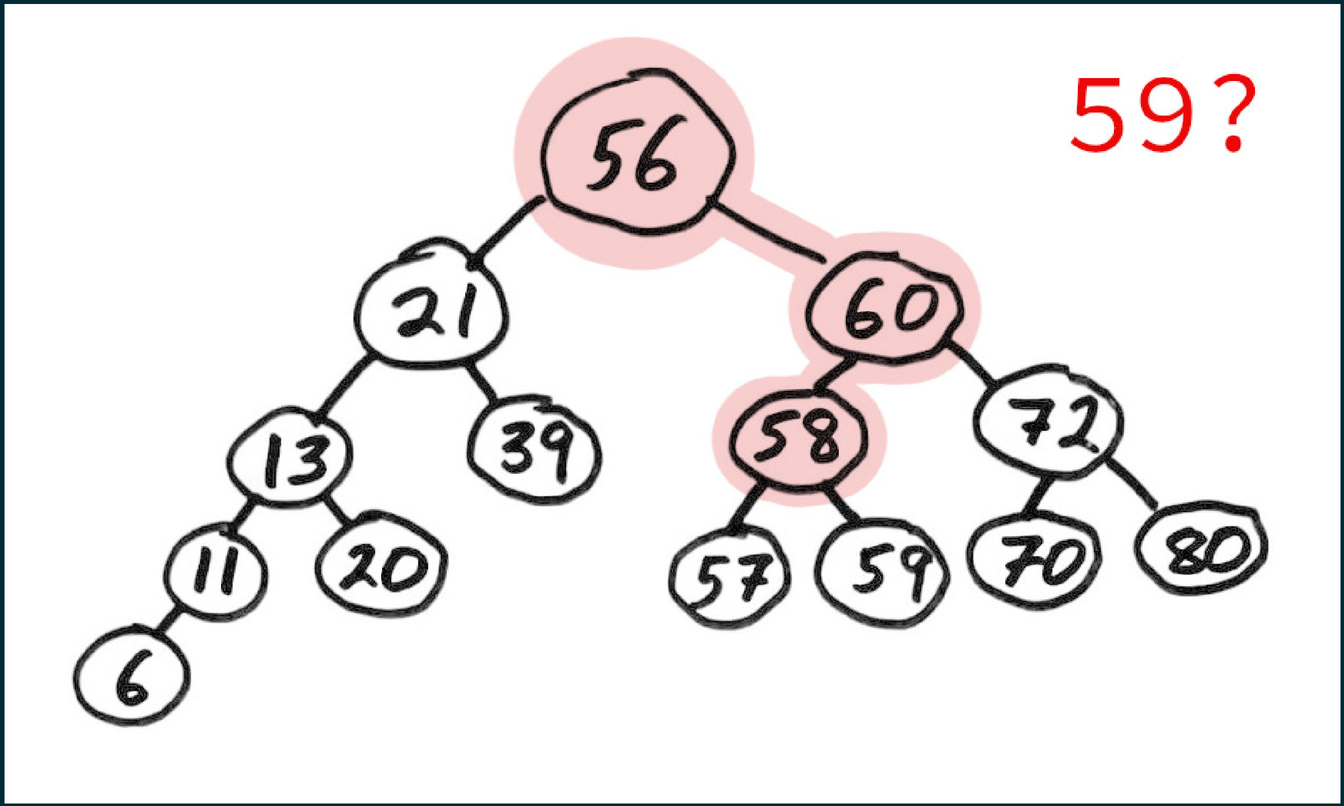
This allows a natural way to check if a value is present with a game of "too high / too low".



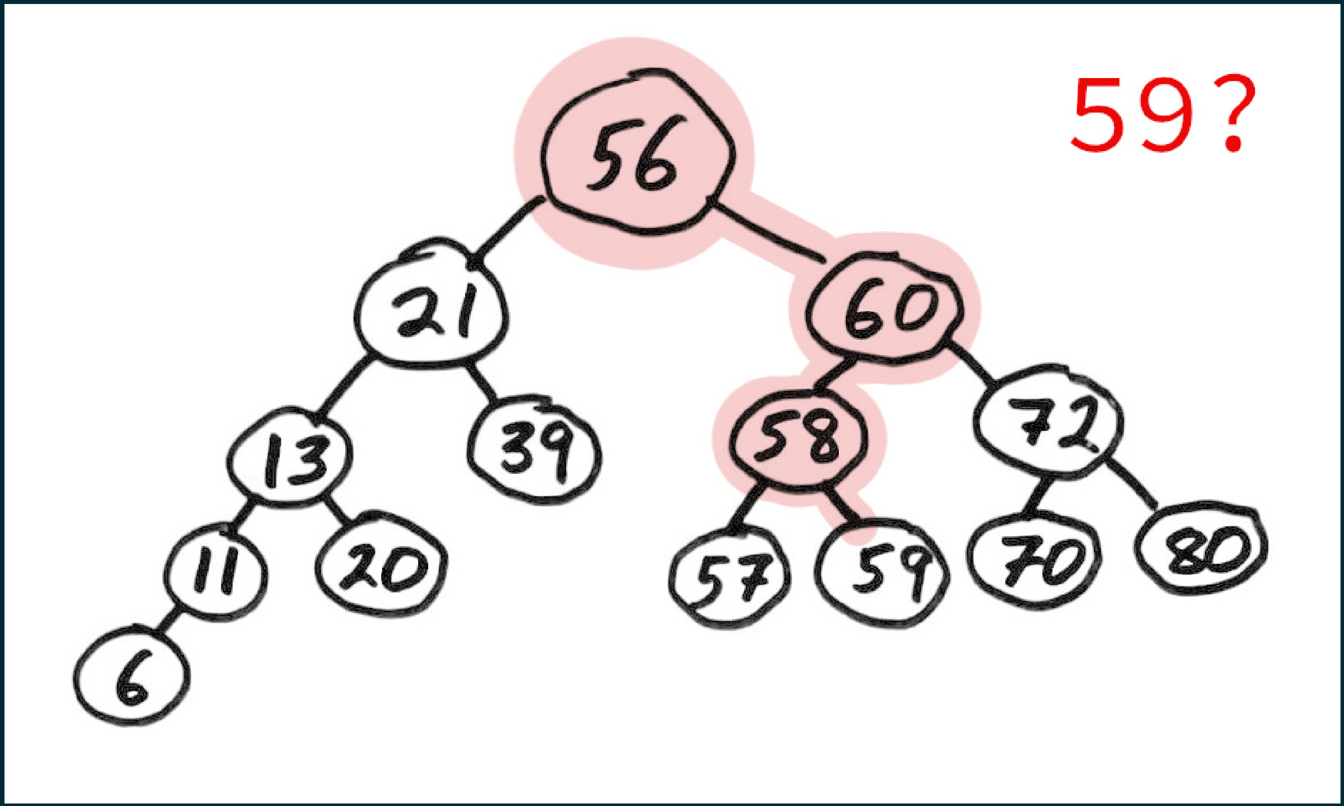
This allows a natural way to check if a value is present with a game of "too high / too low".



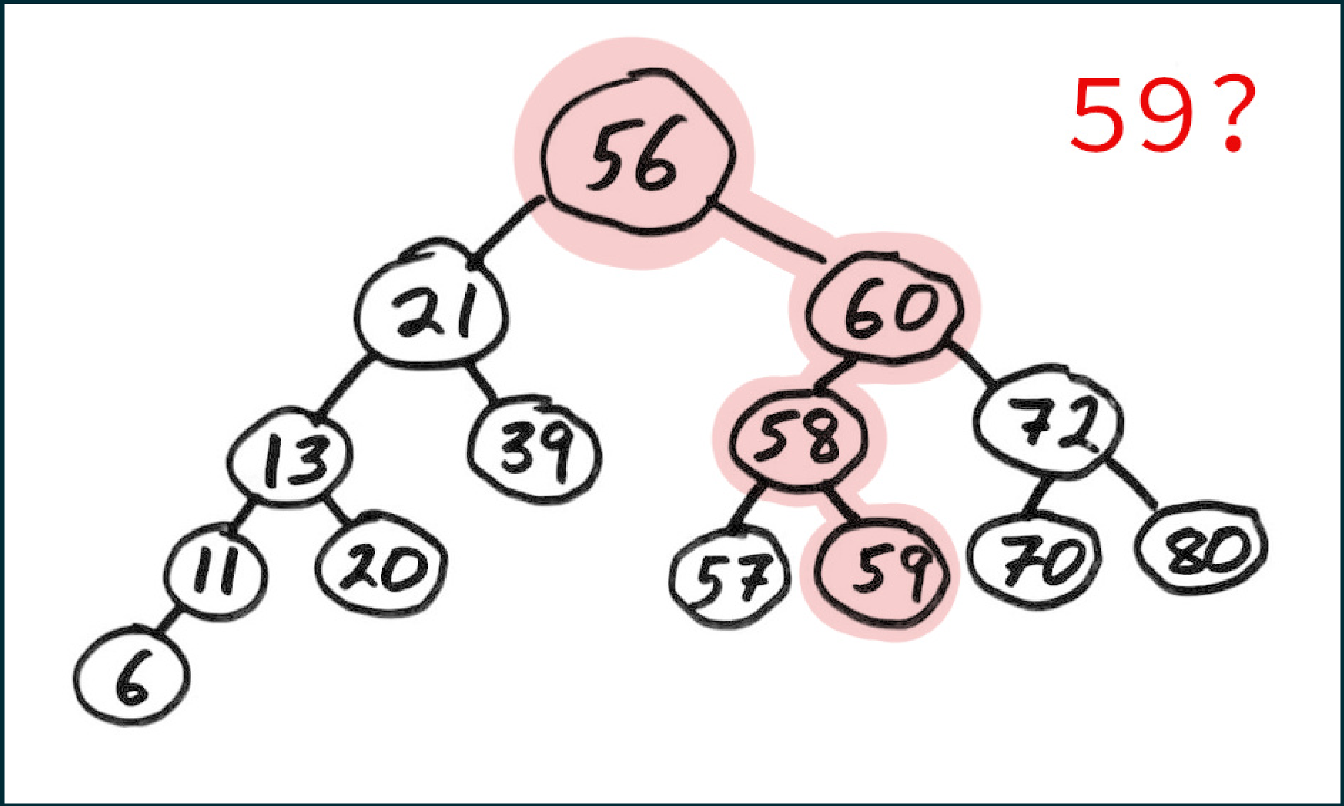
This allows a natural way to check if a value is present with a game of "too high / too low".



This allows a natural way to check if a value is present with a game of "too high / too low".



This allows a natural way to check if a value is present with a game of "too high / too low".



REFERENCES

- In optional course texts:
  - *Problem Solving with Algorithms and Data Structures using Python* by Miller and Ranum, discusses binary trees in Chapter 7.
- Elsewhere:
  - Cormen, Leiserson, Rivest, and Stein discusses graph theory and trees in Appendices B.4 and B.5, and binary search trees in Chapter 12.

REVISION HISTORY

- 2022-02-23 Last year's lecture on this topic finalized
- 2023-02-16 Updated for 2023