Trees

Recap of Lists

Recall the standard definition of lists:

```
data [] a = [] | a : [a]
```

This is great, but uses things like infix notation that are slightly unusual

How would we defined lists ‘from scratch’?

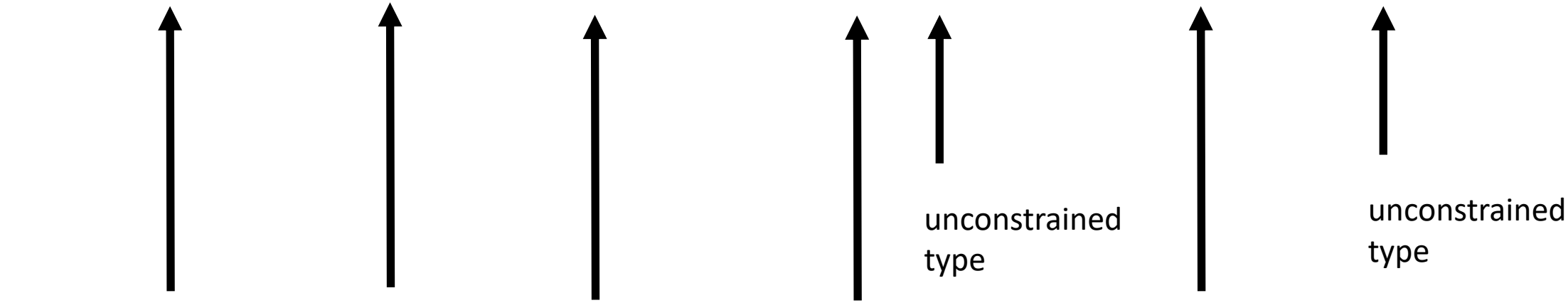
Defining our own Lists

- Lists are polymorphic, so the definition should include a type variable
- Each list is either
  - Empty; or
  - The product of an element with a list (recursion!)

This suggests the definition on the next slide…

User-defined Polymorphic Type for List

```
data CustomList a = Empty | Cons a (CustomList a)
```



Name of the datatype we are defining

Unconstrained type variable

Constructor for an empty list

Constructor for a non-empty list

Recursion

Defining our own Lists

The definition

```
data CustomList a = Empty | Cons a (CustomList a)
```

is not as nice as the built-in lists, e.g. we must write `[1,2,3]` as

```
Cons 1 (Cons 2 (Cons 3 Empty))
```

but the principle is no different.

From Lists to Trees

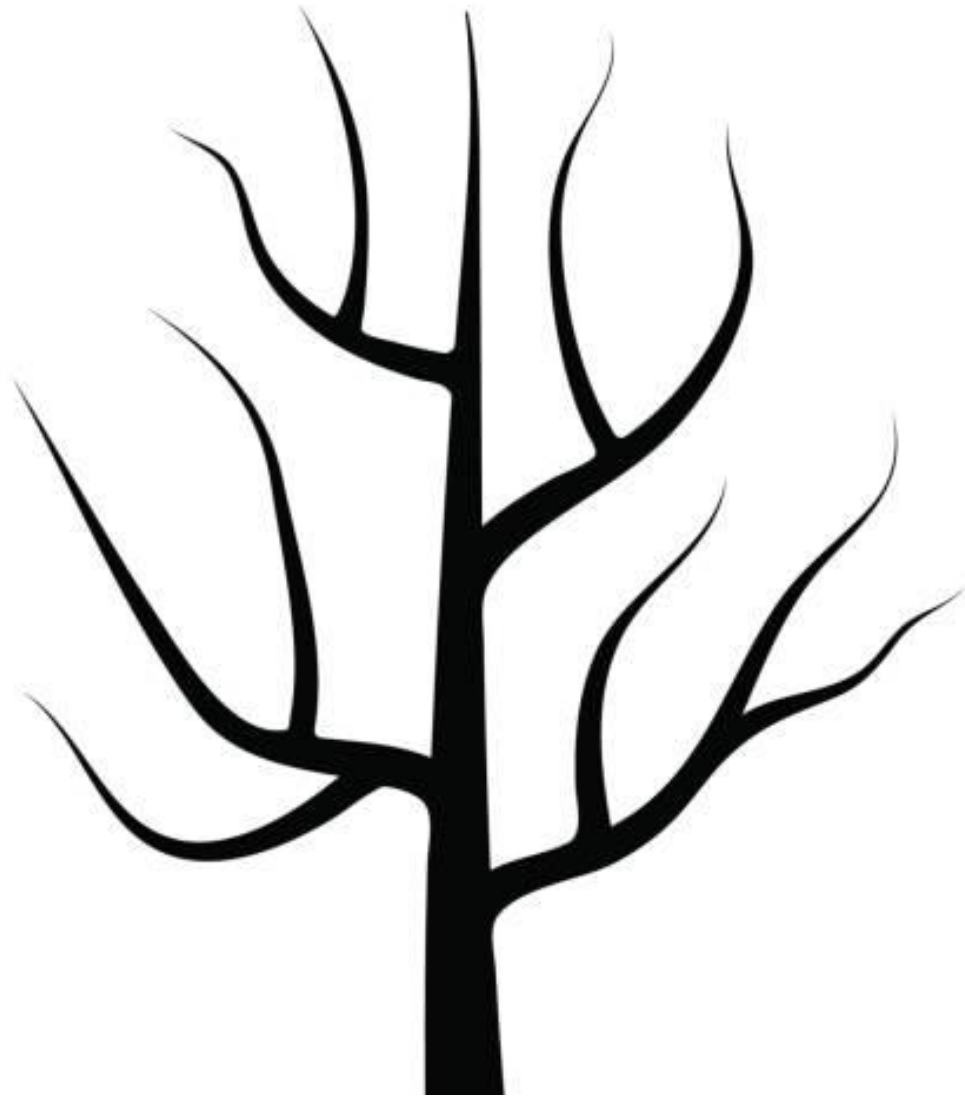
```
data CustomList a = Empty | Cons a (CustomList a)
```

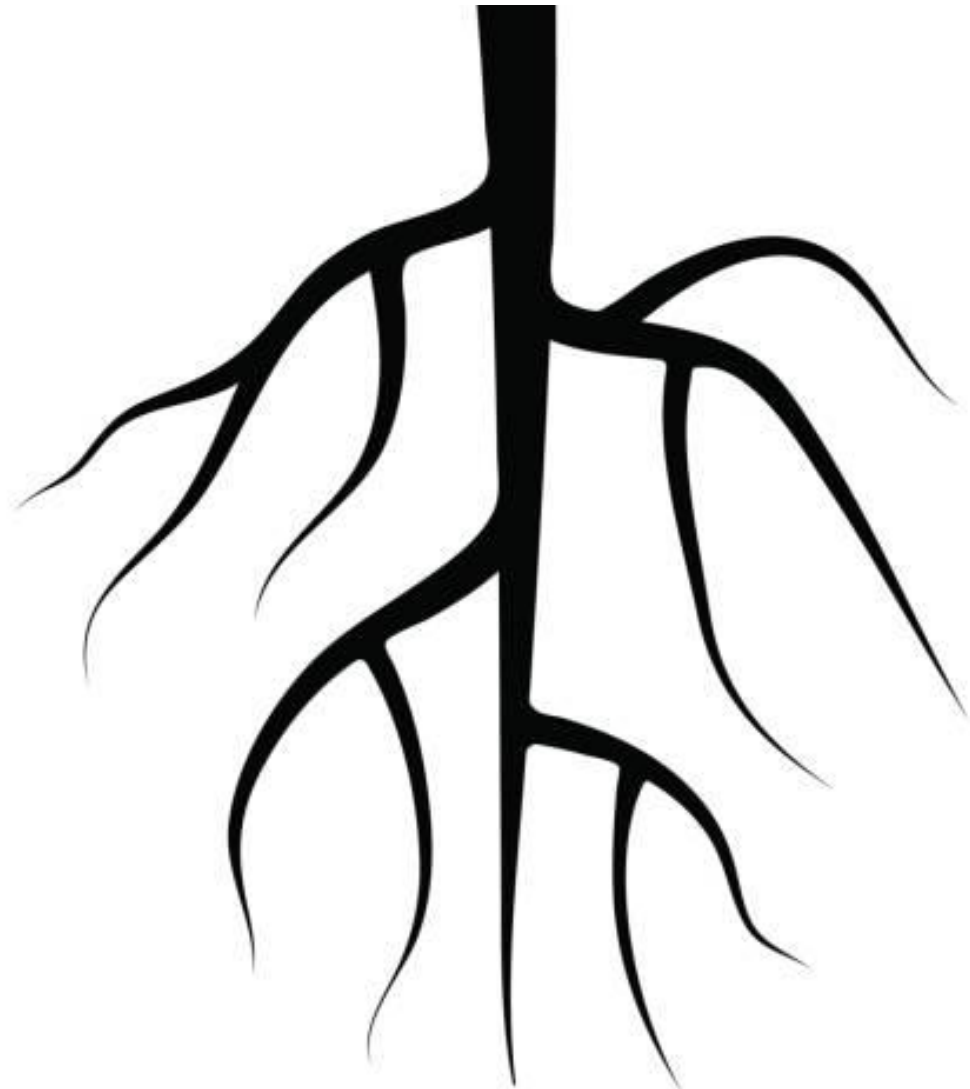
Lists have **one** recursive construction attached to the non-empty case.

Trees are just the same, except with more than one, for example:

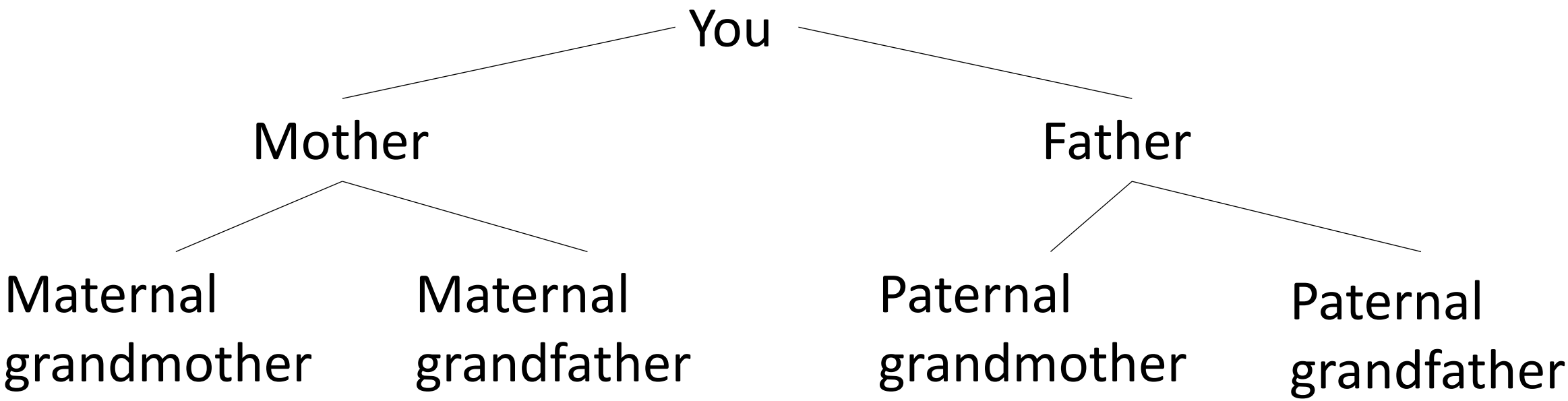
```
data BinaryTree a = Null |
                    Node (BinaryTree a) a (BinaryTree a)
```

But.. why?

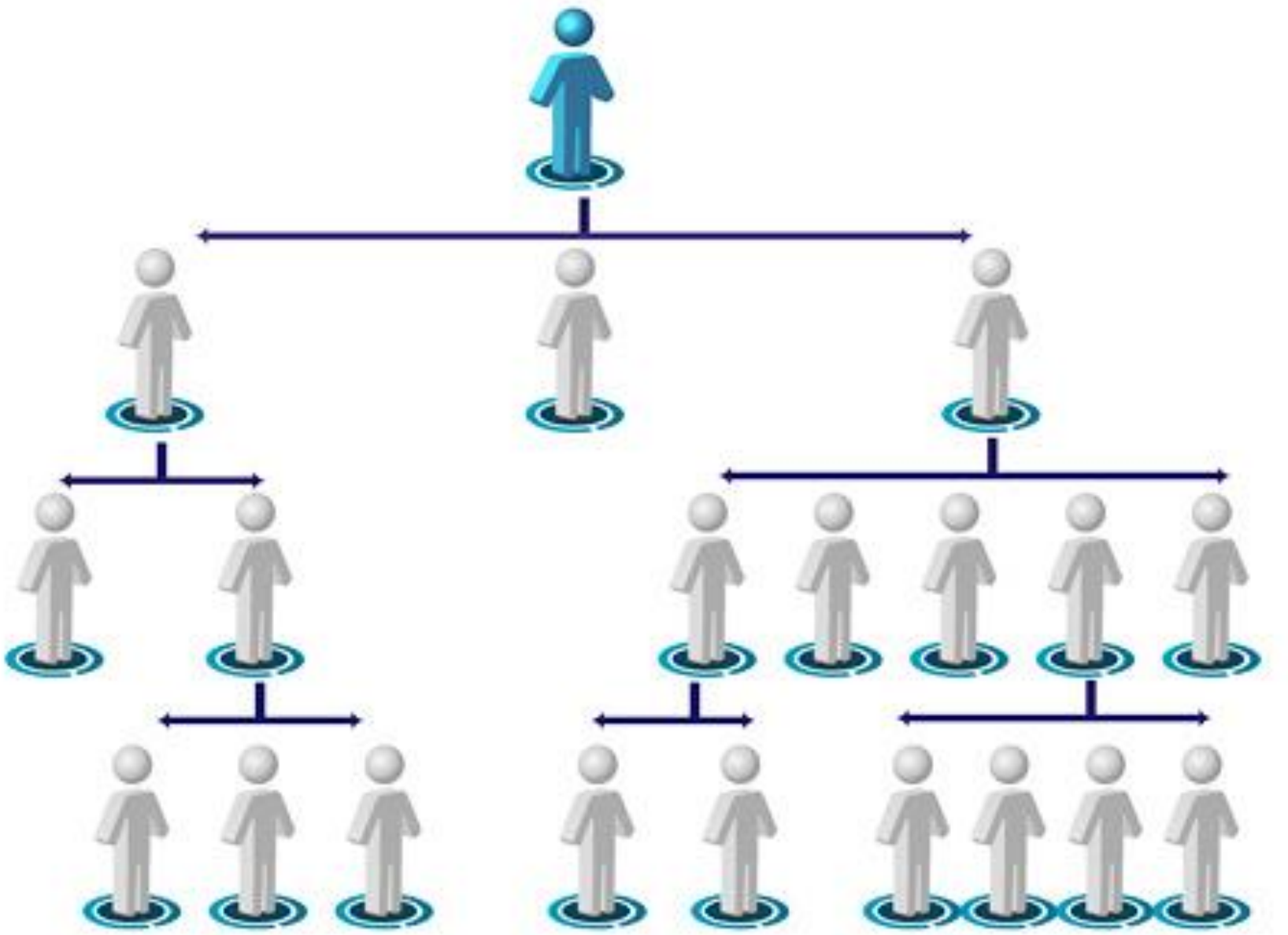




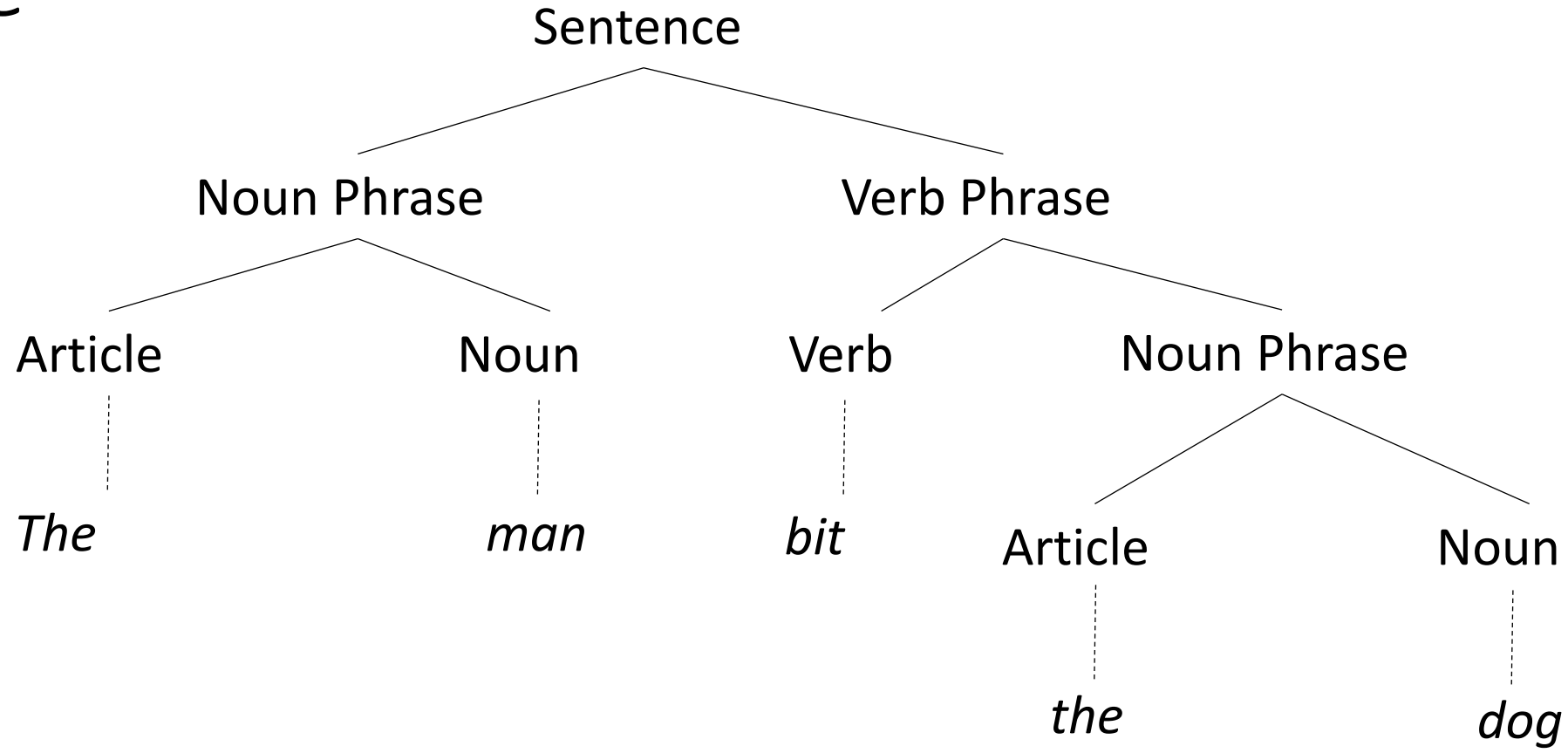
Family (Binary) Tree



Organisation Hierarchy Tree



Parse Tree



The man bit the dog

List vs Tree

List is a linear structure

- One item follows the other

Trees are non-linear structures

- More than one item can follow another.
- In some cases, the number of items that follow can vary from one item to another.

Why Trees?

- More structure
- Faster traversing
- Fits problems like
  - ”Keep it sorted”
  - “Find fast”
  - “Express hierarchies”
  - “Build decision paths”

Applications of Tree Data Structures

- Parsing a natural language sentence or a programming language expression
- Storing the boards in e.g. a Chess or Backgammon program which emulates many potential move combinations
- Index management of databases
- …

Terminology

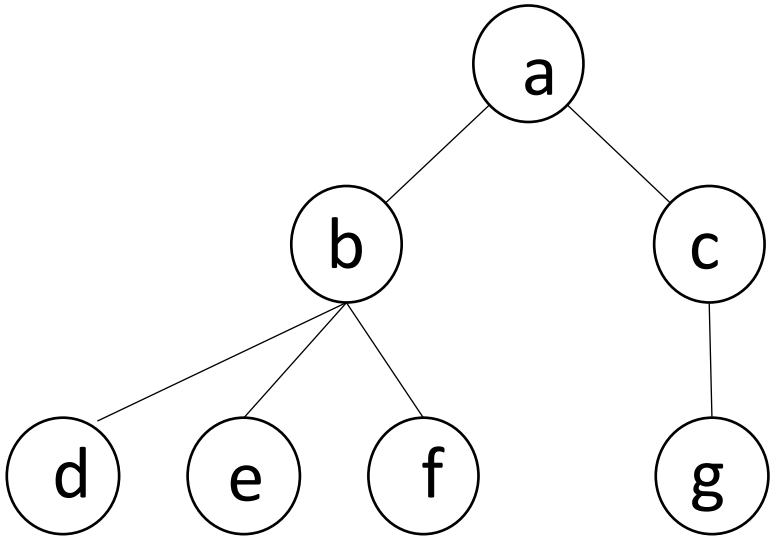
Nodes

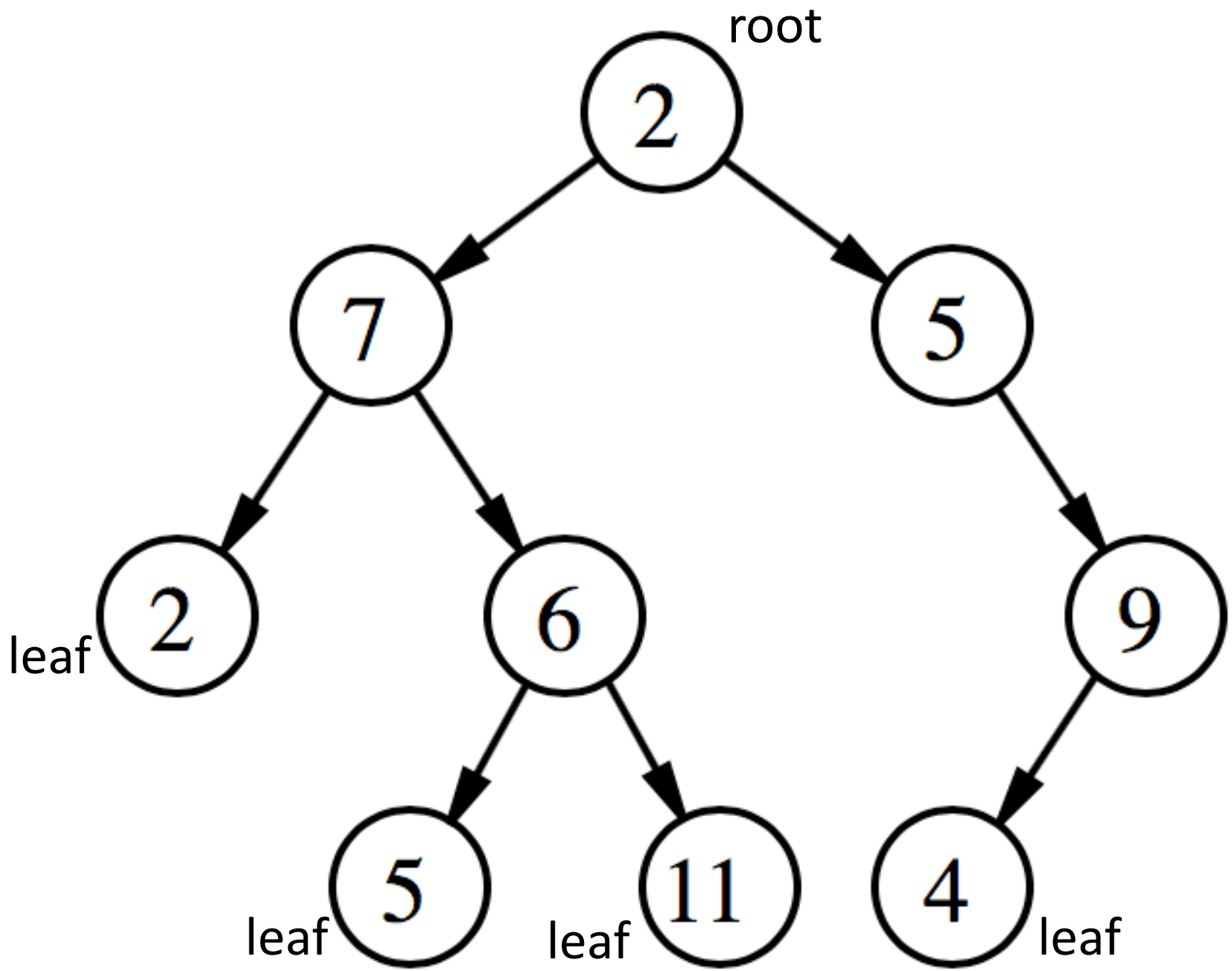
- root (the topmost node, with no incoming edges)
- leaves (the bottom nodes, with no outgoing edges)
- parent node
- child nodes

Edges

Any node is the root of a **subtree** consisting of it and the nodes below it.

Any subtree is a tree.




root
2
7
5
2
6
9
leaf
5
11
4
leaf
leaf
leaf

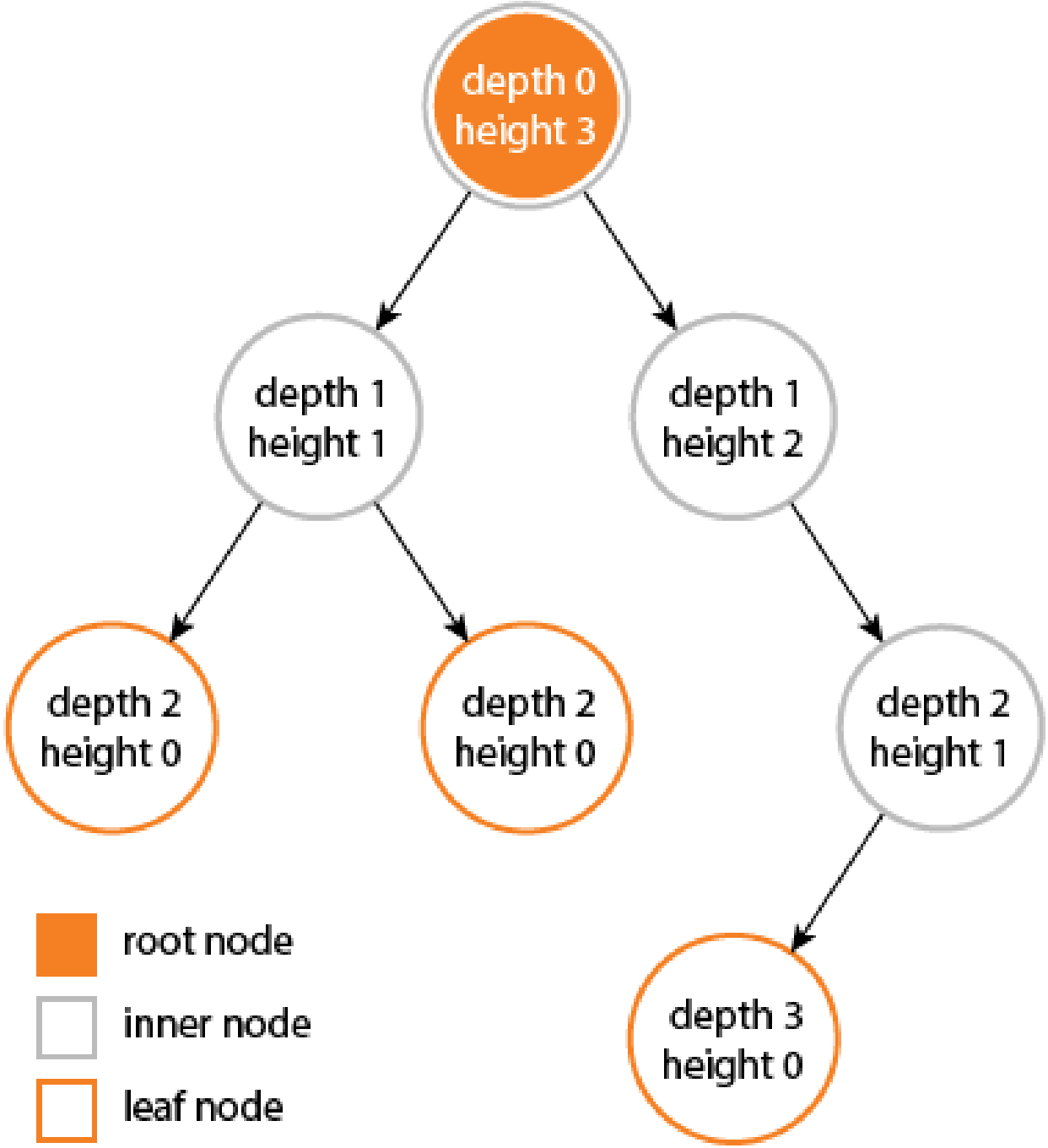
Node

- The **depth of a node** is the number of edges from the node to the tree's root node. A root node has a depth of 0.

- The **height of a node** is the number of edges on the *longest path* from the node to a leaf. Any leaf node has a height of 0.

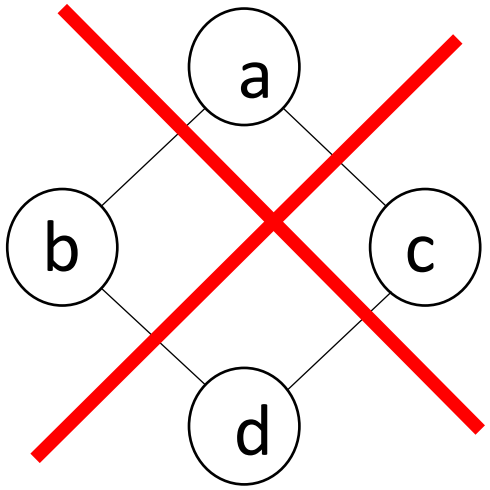
Tree

- The **height of a tree** is the height of its root node
- The **height of a tree** is equal to the depth of its deepest node


depth 0
height 3
depth 1
height 1
depth 1
height 2
depth 2
height 0
depth 2
height 0
depth 2
height 1
depth 3
height 0
root node
inner node
leaf node

Properties of Trees

- There is exactly one path connecting any two nodes in a tree
- Every node, except the root, has a unique parent



- A tree with N nodes has (N – 1) edges

Trees can be classified

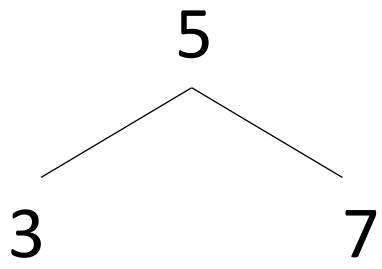
according to

- the precise form of the branching structure
- the location of information within the tree
- the nature of the relationships between the information in different parts of the tree

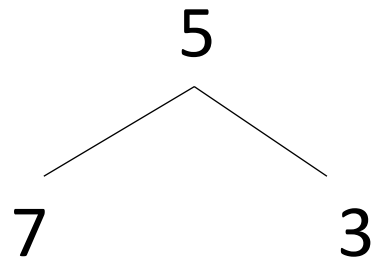
Binary Trees

A binary tree is a tree data structure in which each node has two children, which are referred to as the **left child** and the **right child**.

A binary tree is an **ordered** data structure!



is different from



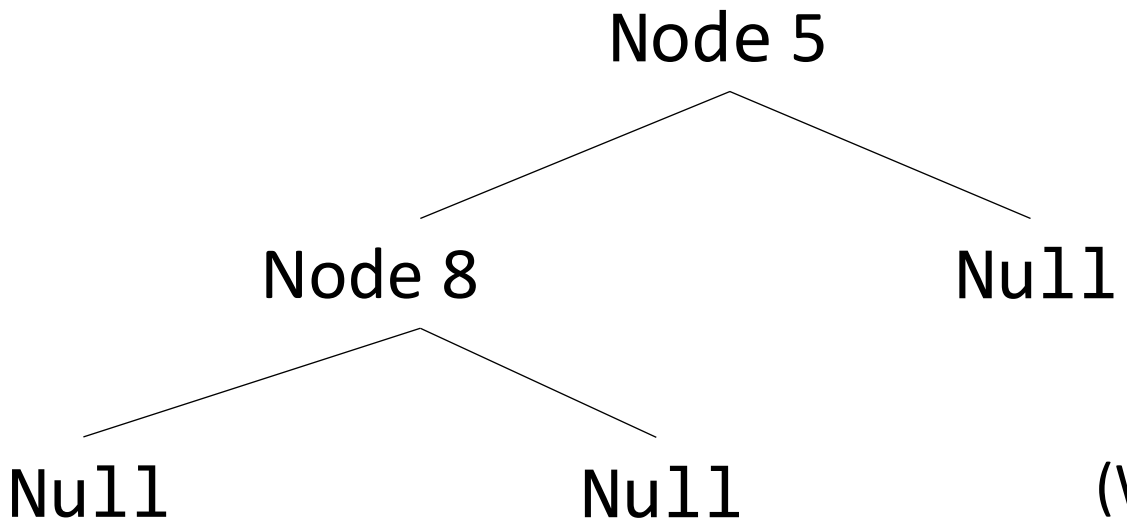
Binary Trees in Haskell

An integer binary tree is either a Null or is given by combining a value and two sub-trees:

```
data BinaryTree a = Null |
                    Node (BinaryTree a) a (BinaryTree a)
```

```
data BinaryTree a = Null |
                    Node (BinaryTree a) a (BinaryTree a)
```

```
Node (Node Null 8 Null) 5 Null
```



(We would usually draw the tree without the `Null`s)

Binary Search Trees

The type of Binary Search Trees are no different from standard Binary Trees:

```
type BSTree a = BinaryTree a
```

But we will use them differently by requiring that **elements** are **sorted**.

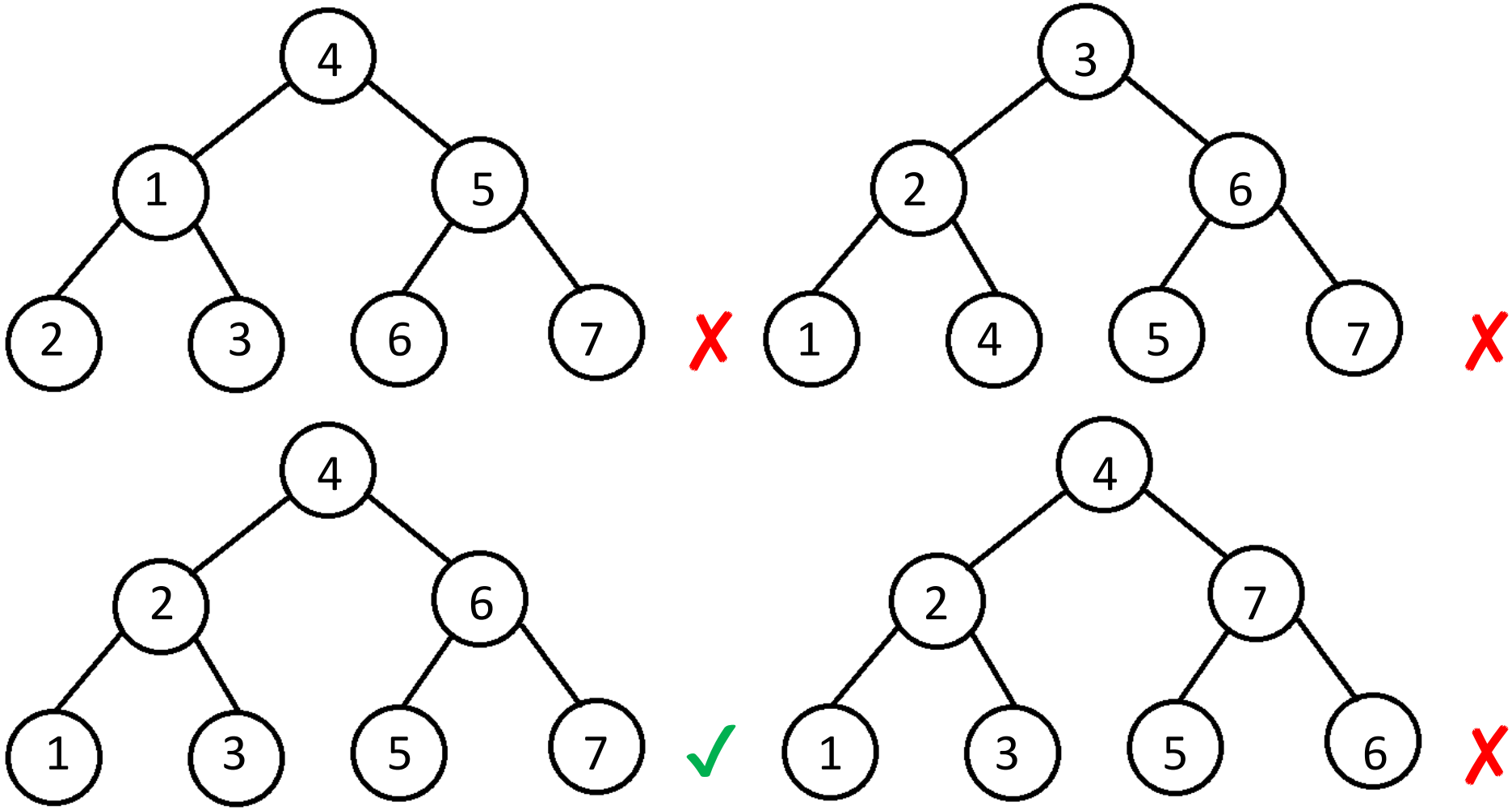
Binary Search Trees

A binary tree is **a binary search tree** if

- It is `Null`, or
- It has form `Node left val right` and
  - all values in `left` are smaller than or equal to `val`,
  - all values in `right` are larger than or equal to `val`, and
  - the trees `left` and `right` are also **binary search trees**

We need to maintain the sorting!

Binary Search Trees – Examples and Non-Examples



The value of Binary Search Trees

Is an element in a Binary Tree?

- The only way to know is look at every node

Is an element in a Binary **Search** Tree?

- At each step we can ignore half our tree!
  - (We lose this advantage if the tree is not **balanced**)

Nothing in life is free of course – **inserting** elements correctly is now more complicated, and slower.

Rose Trees

```
data Rose a = Rose a [Rose a]
```

This is a recursive data type which uses another recursive data type in its definition!

example:

```
Rose 5 [Rose 3 [Rose 1 [], Rose 4 []], Rose 7 [], Rose 4 []]
```

Rose tree can have an arbitrary and internally varying branching factor (0,1,2, or more).

```
Rose 5 [Rose 3 [Rose 1 [], Rose 4 []], Rose 7 [], Rose 4 []]
```