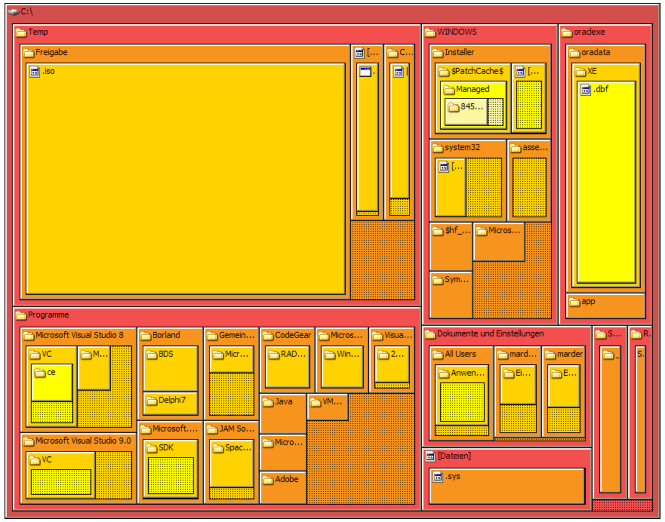
Trees

Jordi Cortadella and Jordi Petit
Department of Computer Science

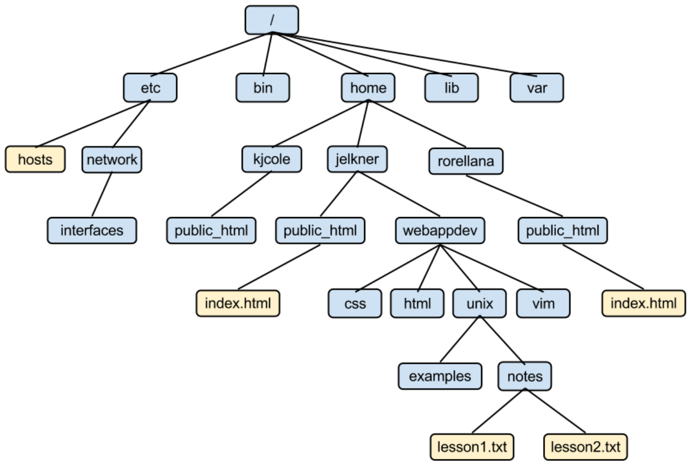
Trees

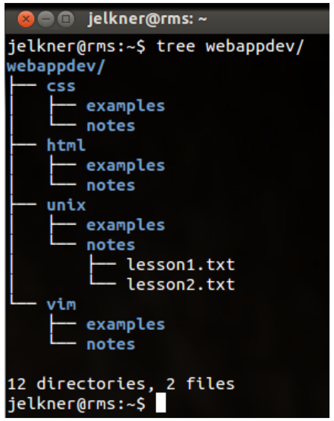
Data are often organized hierarchically



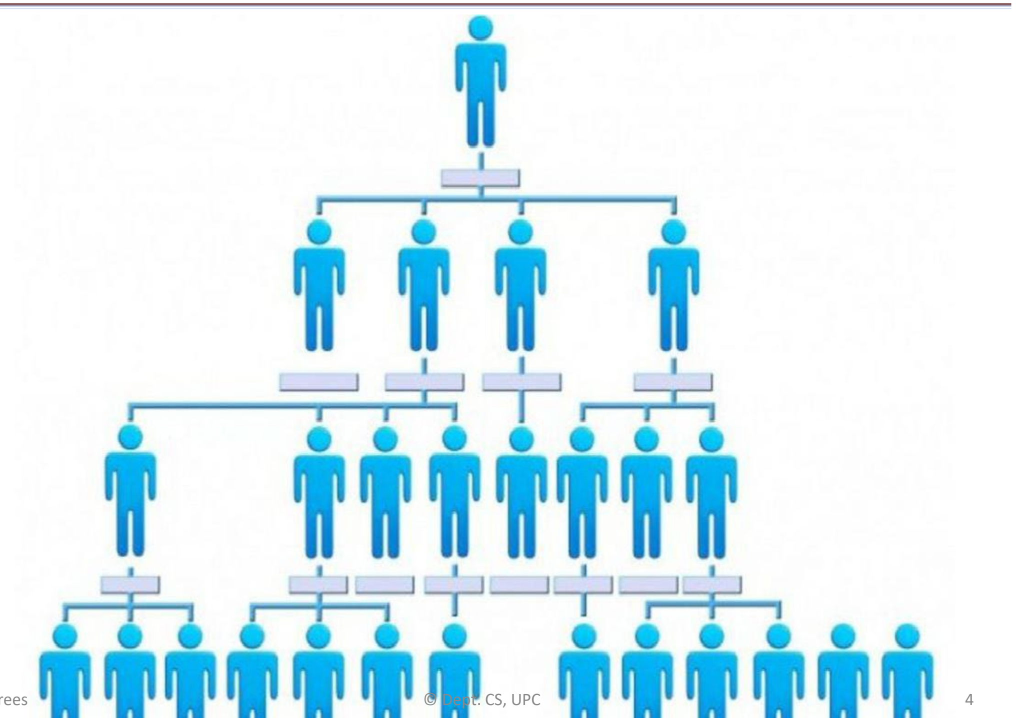
source: https://en.wikipedia.org/wiki/Tree_structure

Filesystems

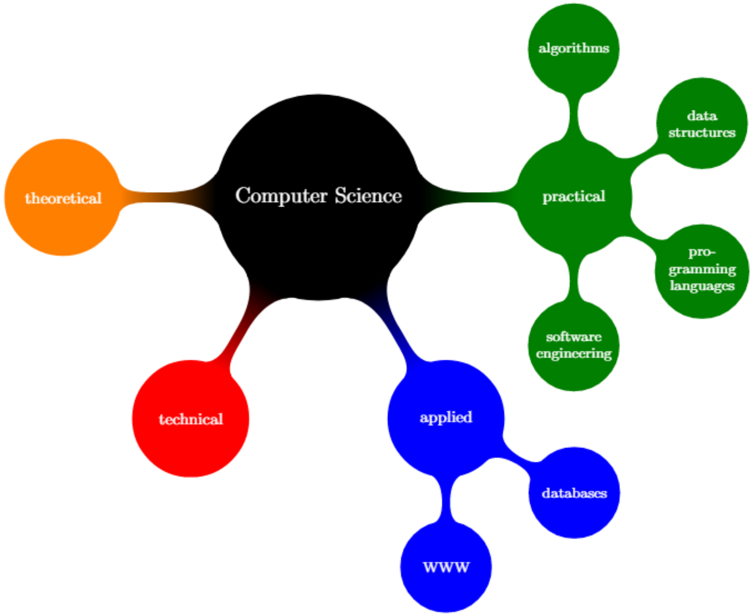




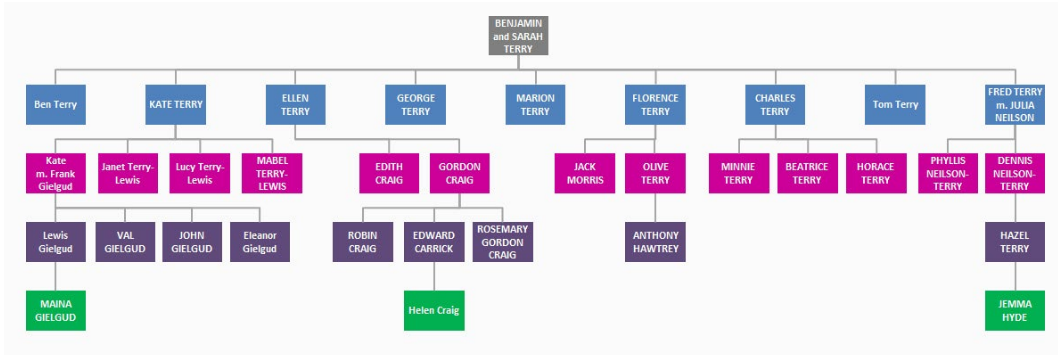
Company structure

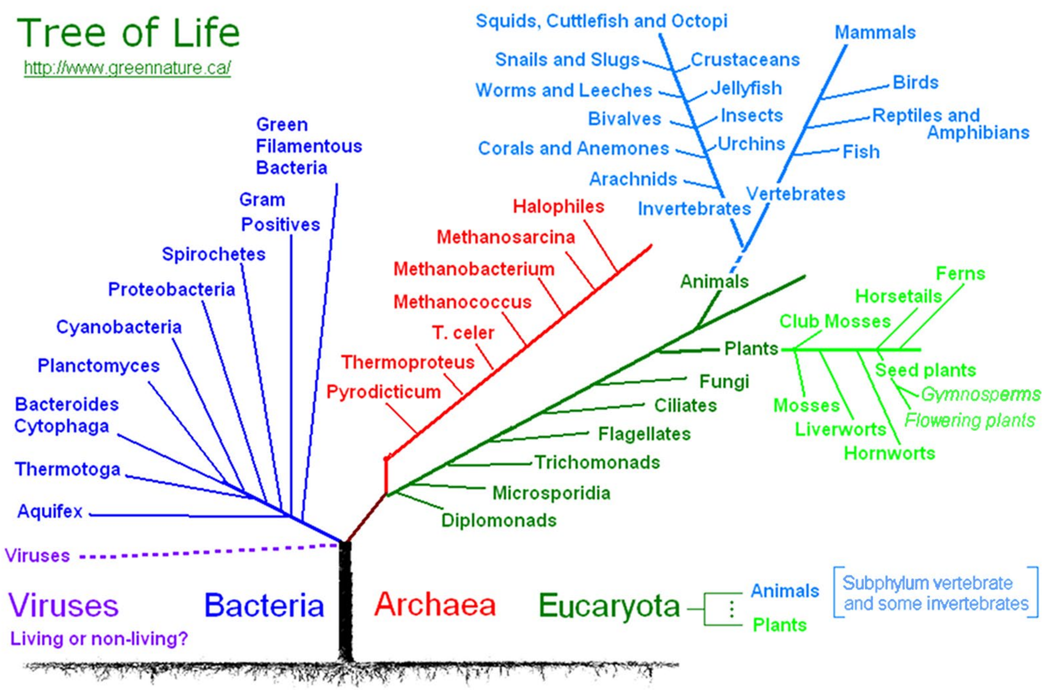


Mind maps

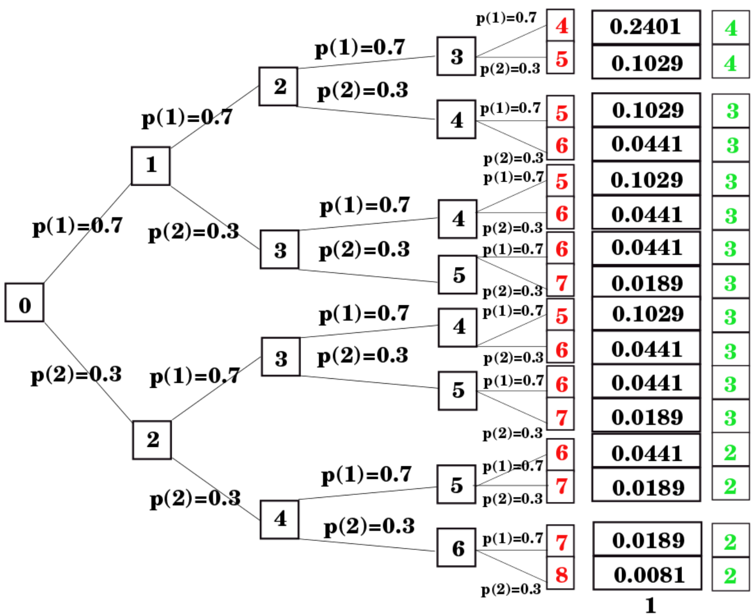


Genealogical trees





Probability trees



Parse trees

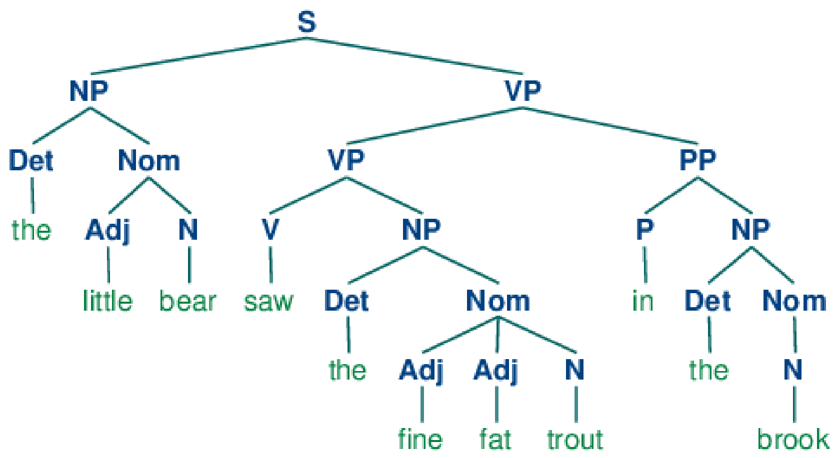
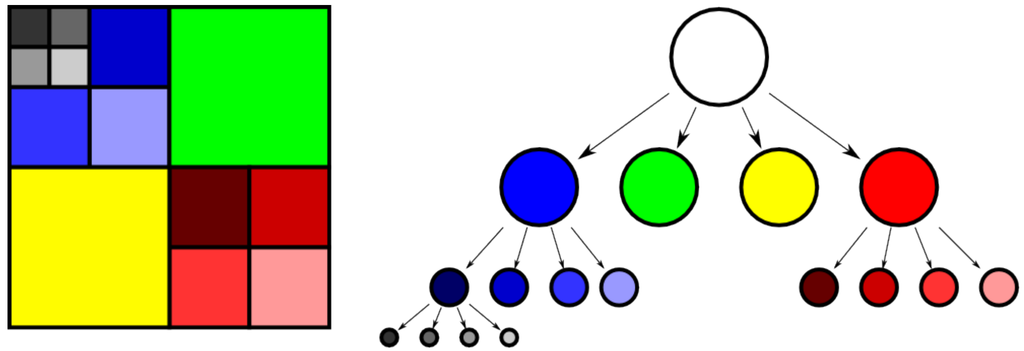
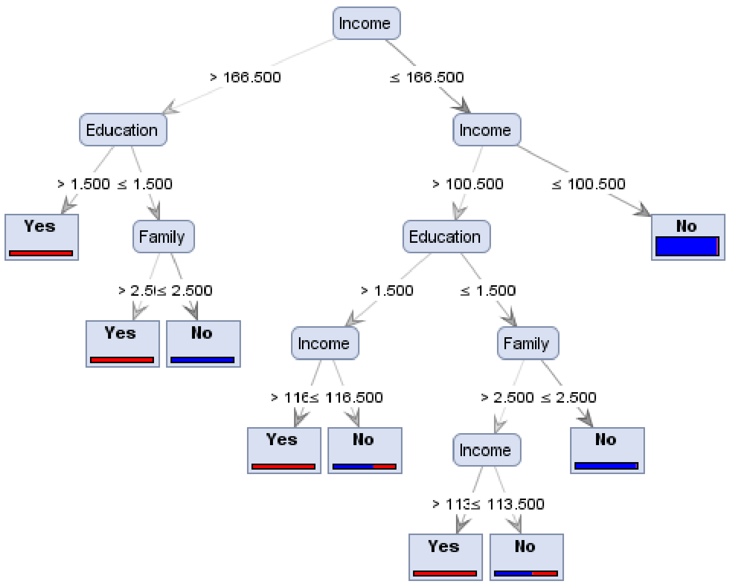


Image representation (quad-trees)



Decision trees

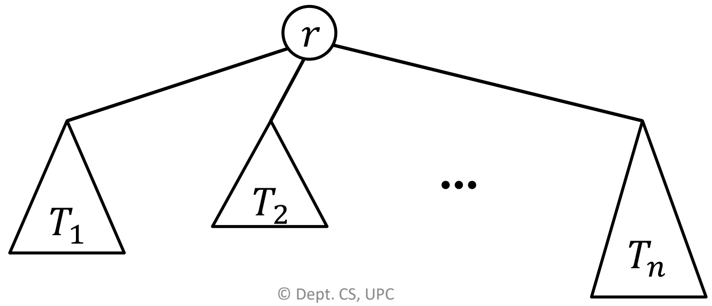


source: http://www.simafore.com/blog/bid/94454/A-simple-explanation-of-how-entropy-fuels-a-decision-tree-model

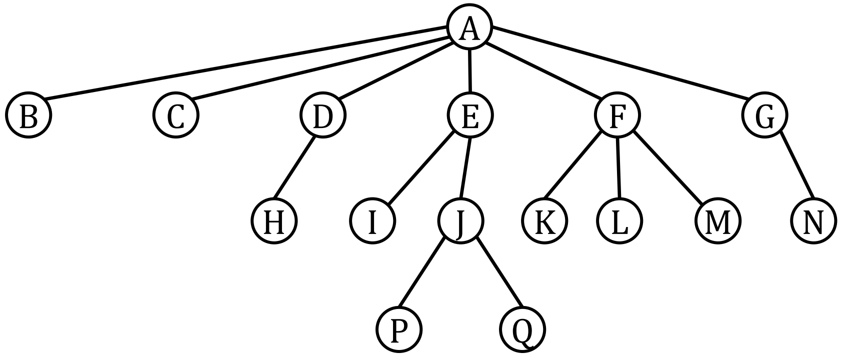
Tree: definition

- Graph theory: a tree is an undirected graph in which any two vertices are connected by exactly one path.

- Recursive definition (CS). A non-empty tree T consists of:
  - a root node r
  - a list of non-empty trees $T_1, T_2, \ldots, T_n$ that hierarchically depend on r. The list can be possibly empty ($n \geq 0$).



Tree: nomenclature



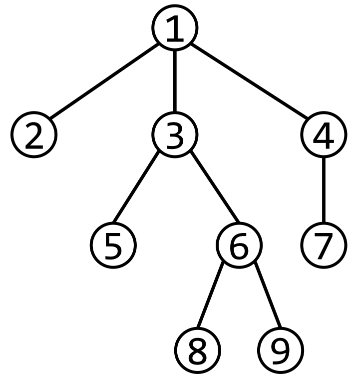
- A is the **root** node.
- Nodes with no children are **leaves** (e.g., B and P).
- Nodes with the same parent are **siblings** (e.g., K, L and M).
- The **depth** of a node is the length of the path from the root to the node. Examples: depth(A)=0, depth(L)=2, depth(Q)=3.

Tree: representation

There is a plethora of data structures that can be used to represent a tree, e.g., a hierarchical list.

[root, $child_1$, $child_2$, …, $child_n$]

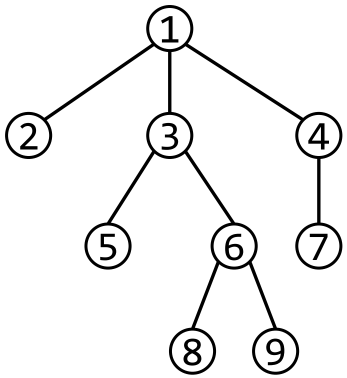
another tree



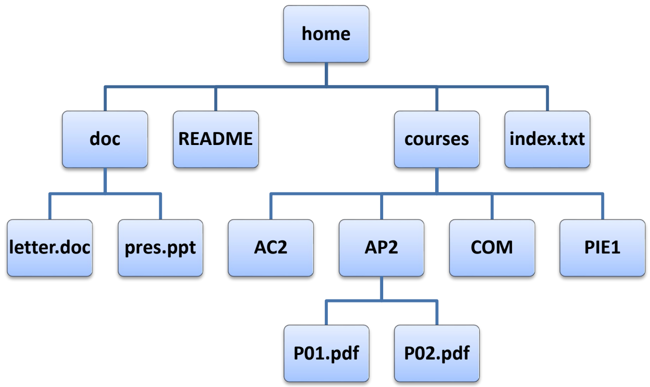

```
tree = [1, 2,
          [3, 5, [6, 8, 9]],
          [4, 7]
       ]
```


Tree: Abstract Data Type

```python
from dataclasses import dataclass
from typing import TypeVar, Generic

T = TypeVar('T')

@dataclass
class Tree(Generic[T]):
    """Class to represent a generic tree"""
    data: T
    children: list[Tree[T]]


def size(t: Tree) -> int:
    return 1 + sum(c.size() for c in t.children)


def num_levels(t: Tree) -> int:
    # implement it!
```



Write a tree




```
home
  doc
    letter.doc
    pres.ppt
  README
  courses
    AC2
    AP2
      P01.pdf
      P02.pdf
    COM
    PIE1
  index.txt
```

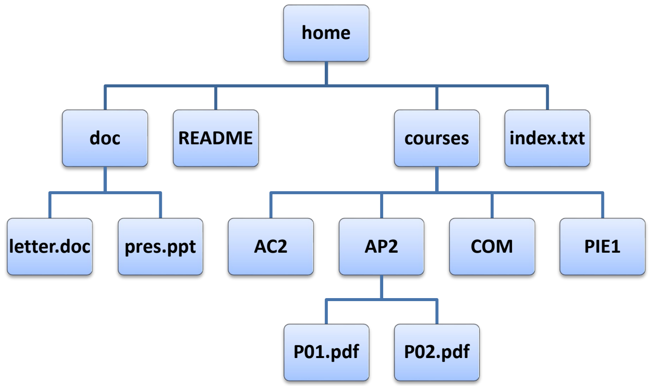
```python
def write(t: Tree[T], depth: int = 0) -> None:
    """Writes a tree indented according to the depth"""
```

Write a tree

```
def write(t: Tree[T], depth: int = 0) -> None:
    """Writes a tree indented according to the depth"""

    # print the root
    print(' '*2*depth, t.data, sep='')

    # print the children with depth + 1
    for c in t.children:
        write(c, depth + 1)
```

This function executes a ***preorder*** traversal of the tree: each node is processed *before* the children.

Write a tree (postorder traversal)




```
    letter.doc
    pres.ppt
  doc
  README
    AC2
      P01.pdf
      P02.pdf
    AP2
    COM
    PIE1
  courses
  index.txt
home
```

Postorder traversal: each node is processed after the children.

Write a tree (postordre traversal)

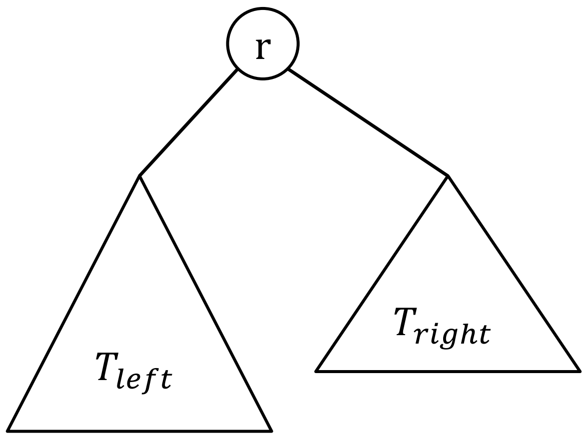
```
def write_postorder(t: Tree[T], depth: int = 0) -> None:
    """Writes a tree (in postorder) indented according
       to the depth"""

    # print the children with depth + 1
    for c in t.children:
        write_postorder(c, depth + 1)

    # print the root
    print(' '*2*depth, t.data, sep='')
```

This function executes a ***postorder*** traversal of the tree: each node is processed *after* the children.

Binary tree: definition

A binary tree is a finite set of nodes that either

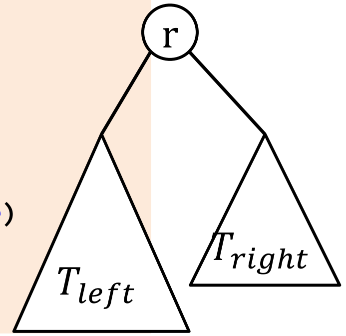
- is empty, or
- is comprised of three disjoint sets of nodes: a root node and two binary trees called its left and right subtrees



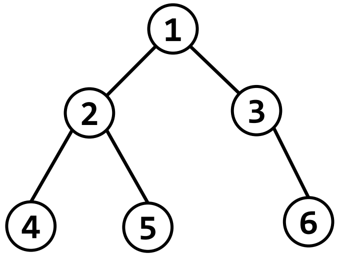
Binary tree: representation

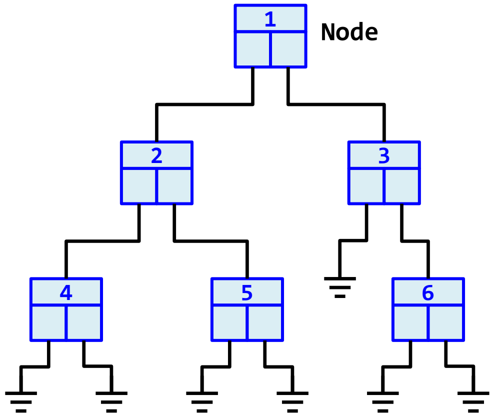
Data structures to represent binary trees are typically based on the definition of a node.

```python
from dataclasses import dataclass, field
from typing import TypeVar, Generic, Optional, Iterator

T = TypeVar('T')

@dataclass
class Node(Generic[T]):
    """Node of a bin tree"""
    data: T
    left: 'BinTree[T]' = field(default = None)
    right: 'BinTree[T]' = field(default = None)

BinTree = Optional[Node[T]]
NodeIter = Iterator[Node[T]]
```

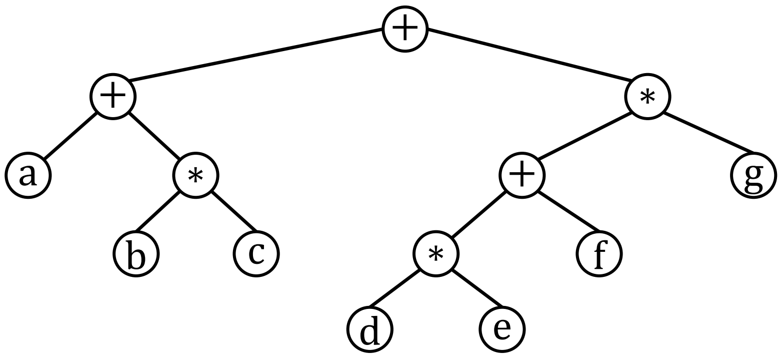


Binary tree: representation





Example: expression trees



Expression tree for: **a + b*c + (d*e + f) * g**

Postfix representation: **a b c * + d e * f + g * +**

How can the postfix representation be obtained?

Example: expression trees

Expressions are represented by strings in postfix notation in which `'a'`…`'z'` represent operands and `'+'` and `'*'` represent operators.

```python
Exprtree: TypeAlias = BinTree[str]

def build_expr(expr: str) -> Exprtree:
    """Builds an expression tree from a correct
       expression represented in postfix notation"""

def infix_expr(t: Exprtree) -> str:
    """Generates a string with the expression in
       infix notation"""

def eval_expr(t: Exprtree, v: dict[str, int]) -> int:
    """Evaluates an expression taking v as the value of the
       variables (e.g., v['a'] contains the value of a)"""
```

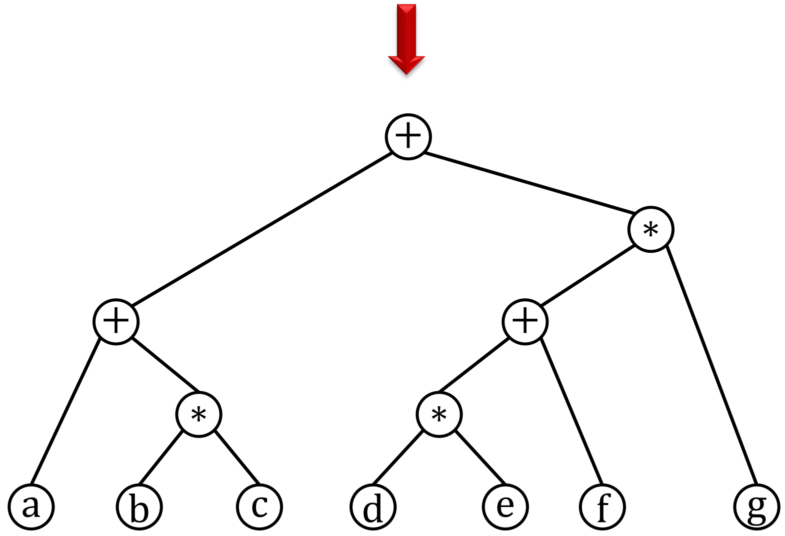
Example: expression trees

```python
def main():
    t = build_expr('a b c * + d e * f + g * +')
    print(infix_expr(t))
    print(eval_expr(t, {'a':3, 'b':1, 'c':0, 'd':5,
                        'e':2, 'f':1, 'g':6}))
```

```
Output:

((a+(b*c))+(((d*e)+f)*g))
69
```

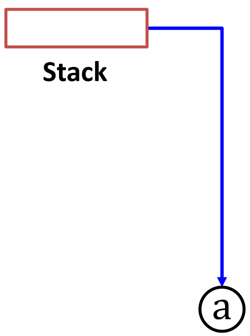

How to build an expression tree

a b c * + d e * f + g * +



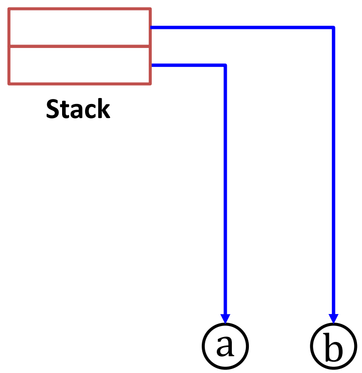
How to build an expression tree

a **b c * + d e * f + g * +**



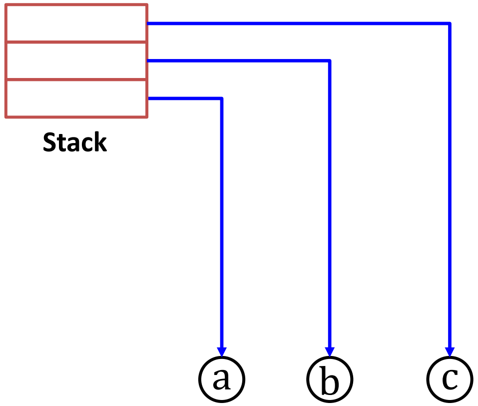
How to build an expression tree

a b **c * + d e * f + g * +**



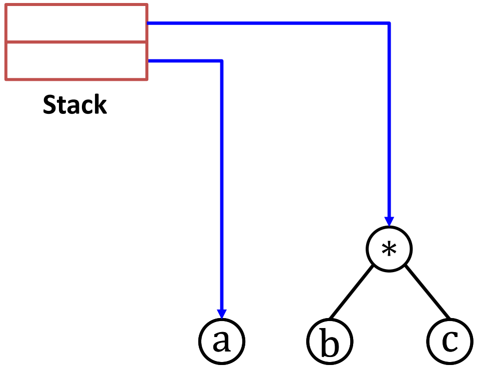
How to build an expression tree

a b c * + d e * f + g * +



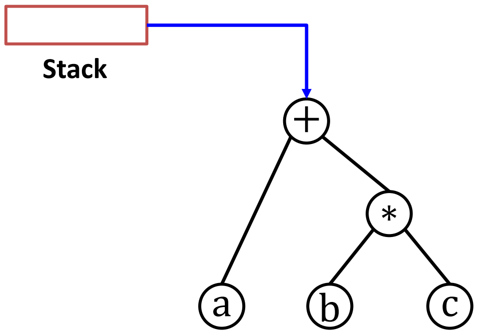
How to build an expression tree

a b c * + d e * f + g * +



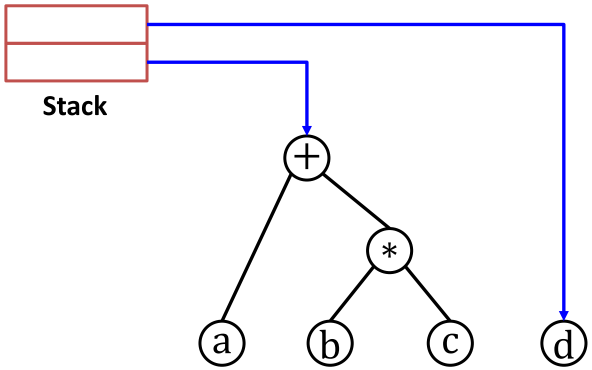
How to build an expression tree

a b c * + d e * f + g * +



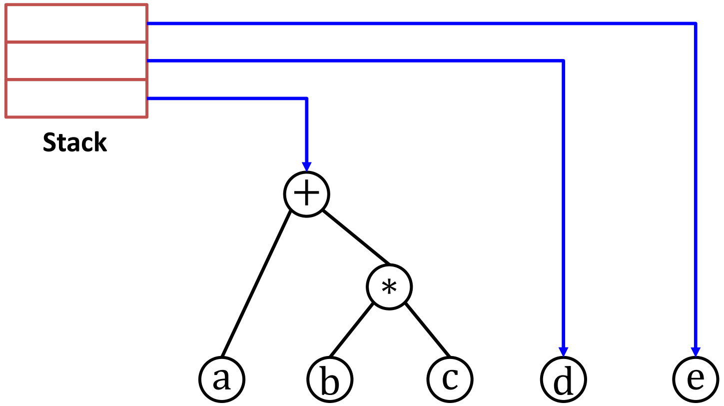
How to build an expression tree

a b c * + d e * f + g * +



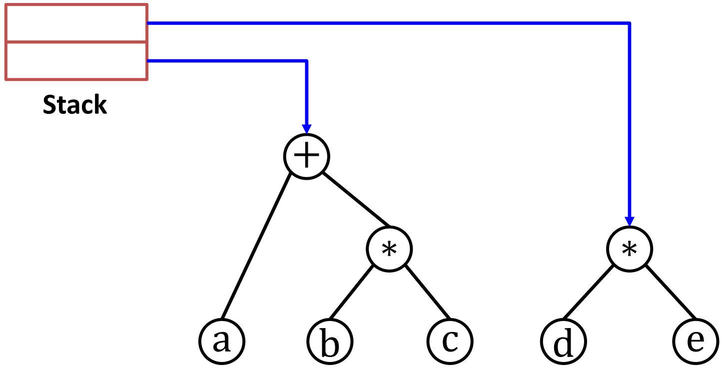
How to build an expression tree

a b c * + d e * f + g * +



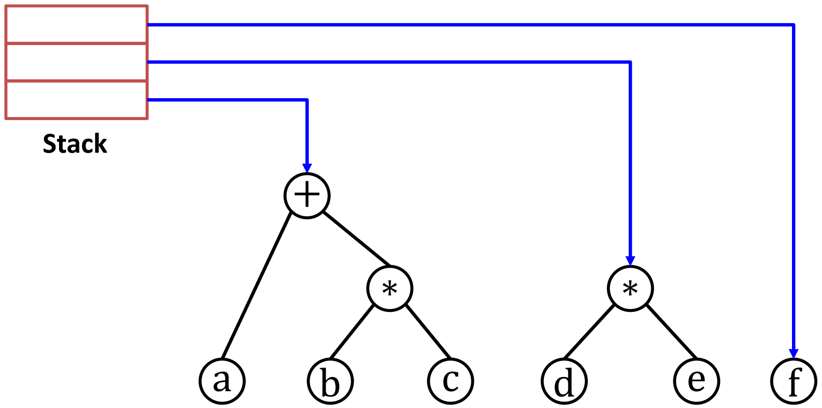
How to build an expression tree

a b c * + d e * f + g * +



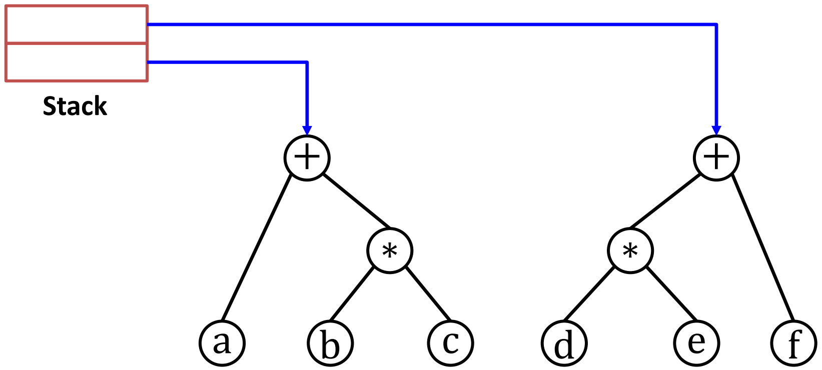
How to build an expression tree

a b c * + d e * f + g * +



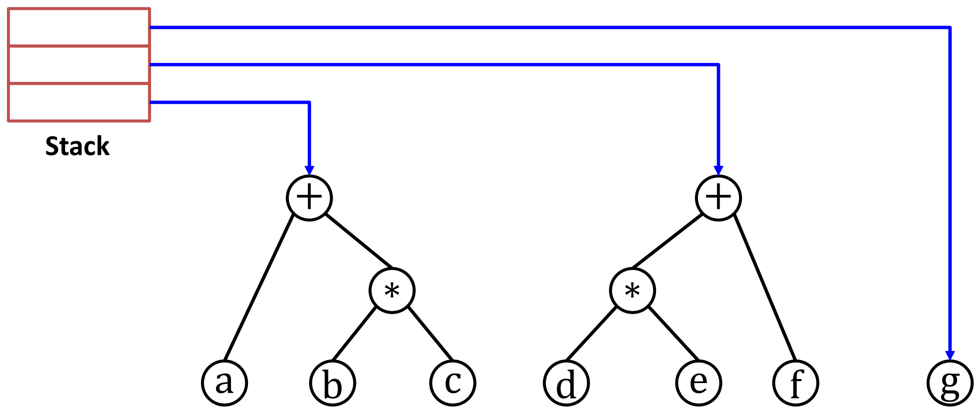
How to build an expression tree

a b c * + d e * f + g * +



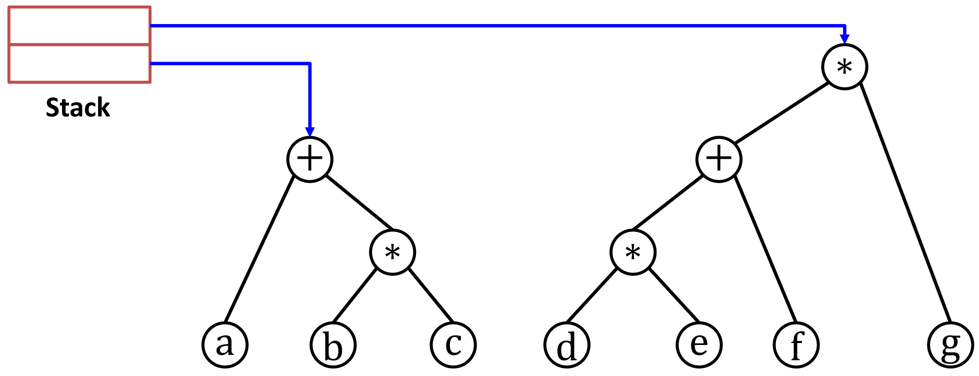
How to build an expression tree

a b c * + d e * f + g * +



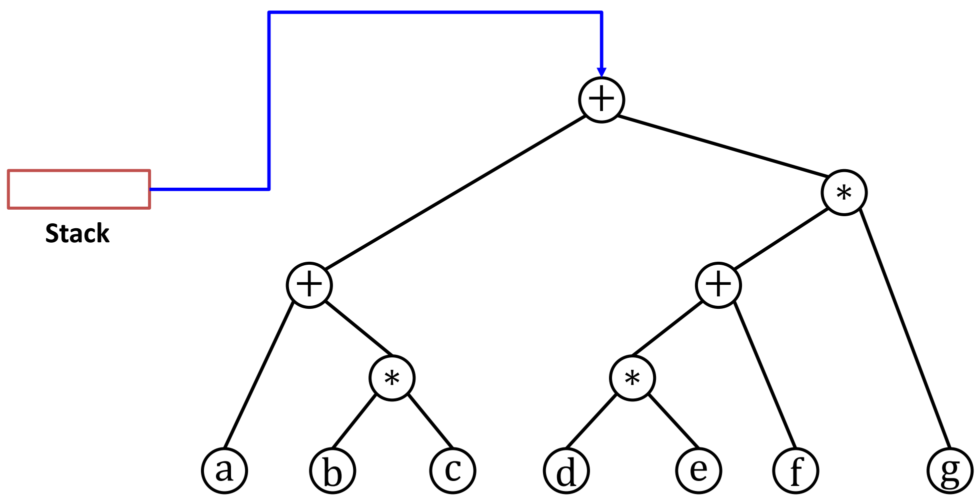
How to build an expression tree

a b c * + d e * f + g * +



How to build an expression tree

a b c * + d e * f + g * +



Example: expression trees

```python
def build_expr(expr: str) -> Exprtree:
    """Builds an expression tree from a correct
       expression represented in postfix notation"""
    # Create a list of all characters (without spaces)
    expr_char = [x for x in expr if not x.isspace()]
    stack: list[Node[str]] = []
    for c in expr_char:
        if c.isalpha():
            # We have an operand. Create a leaf node
            stack.append(Node(c))
        else:
            # We have an operator (+ or *)
            right = stack.pop()
            left = stack.pop()
            stack.append(Node(c, left, right))
    # The stack has only one element: the root of the expression
    return stack.pop()
```

Example: expression trees

```python
def infix_expr(t: Exprtree) -> str:
    """Generates a string with the expression in
       infix notation"""

    if not t.left:  # it is a leaf node (operand)
        return t.data

    # We have an operator. Add enclosing parenthesis (for safety)
    return '(' + infix_expr(t.left) + t.data +
           infix_expr(t.right) + ')'
```

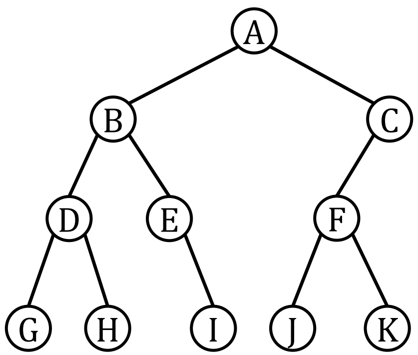
Inorder traversal: node is visited *between* the left and right children.

Exercise: redesign infix_expr to minimize the number of parenthesis.

Example: expression trees

```python
def eval_expr(t: Exprtree, v: dict[str, int]) -> int:
    """Evaluates an expression taking v as the value of the
       variables (e.g., v['a'] contains the value of a)"""

    if not t.left:  # it is a leaf node: return the value
        return v[t.data]

    # We have an operator: evaluate subtrees and operate
    left = eval_expr(t.left, v)
    right = eval_expr(t.right, v)
    return left + right if t.data == '+' else left * right
```


Tree traversals



Let us consider generators to visit the nodes of the tree in some specific order.

```python
t: BinTree[str] = … # some tree constructor

Lpreorder  = [n.data for n in preorder(t)]
Lpostorder = [n.data for n in postorder(t)]
Linorder   = [n.data for n in inorder(t)]
Llevels    = [n.data for n in level_order(t)]
```

```
Lpreorder:  ['A', 'B', 'D', 'G', 'H', 'E', 'I', 'C', 'F', 'J', 'K']

Lpostorder: ['G', 'H', 'D', 'I', 'E', 'B', 'J', 'K', 'F', 'C', 'A']

Linorder:   ['G', 'D', 'H', 'B', 'E', 'I', 'A', 'J', 'F', 'K', 'C']

Llevels:    ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K']
```

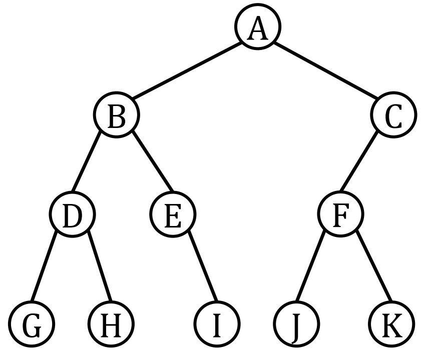

Tree traversals

```python
# Remember:
#     BinTree = Optional[Node[T]]
#     NodeIter = Iterator[Node[T]]

def preorder(t: BinTree) -> NodeIter:
    """Iterator to visit the nodes in preorder"""
    if t:
        yield t.data
        yield from preorder(t.left)
        yield from preorder(t.right)


def postorder(t: BinTree) -> NodeIter:
    """Iterator to visit the nodes in postorder"""
    if t:
        yield from postorder(t.left)
        yield from postorder(t.right)
        yield t.data
```

Tree traversals

```python
def inorder(t: BinTree) -> NodeIter:
    """Iterator to visit the nodes in inorder"""
    if t:
        yield from inorder(t.left)
        yield t.data
        yield from inorder(t.right)


def level_order(t: BinTree) -> NodeIter:
    """Iterator to visit the nodes by levels"""
    if not t:
        return
    q: deque[Node] = deque([t])
    while q:
        n = q.popleft()
        yield n
        if n.left:
            q.append(n.left)
        if n.right:
            q.append(n.right)
```


Tree visitors



A visitor is a function that is applied to all nodes of a tree.

Similar to the **map** function applied to iterables (e.g., lists)

```python
def visit_preorder(t: BinTree[T], f: Callable[[T], T]) -> None:
    """Visits all the nodes of the tree in preorder and applies
       f() to the data. The result is reassigned to the data"""
```

Type: `Callable[[T1,…Tn], Tr]`.
A function with parameters `[T1,…,Tn]` and result `Tr`.

Tree visitors

```python
def visit_preorder(t: BinTree[T], f: Callable[[T], T]) -> None:
    """Applies f to all data in preorder"""
    if t:
        t.data = f(t.data)
        visit_preorder(t.left, f)
        visit_preorder(t.right, f)

# Example
def square(x: int) -> int:
    return x*x

t: Bintree[int] = …  # some tree constructor
visit_preorder(t, square) # squares all data in the tree

# equivalent with lambda: visit_preorder(t, lambda x: x*x)
```


EXERCISES

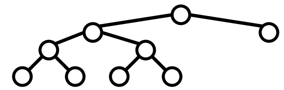
Expression tree

- Modify `infixExpr` for a nicer printing:
  - Minimize number of parenthesis.
  - Add spaces around + (but not around ∗).
- Extend the functions to support other operands, including the unary – (e.g., –a/b).

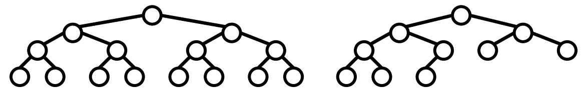
Binary tree types

Design the function "`def check_type(t: BinTree) -> bool:`" for each type tree.

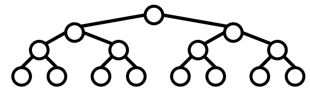
- **Full Binary Tree**: each node has 0 or 2 children.



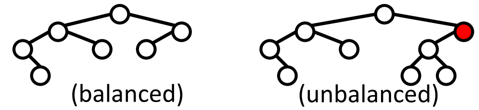
- **Complete Binary Tree:** all levels are filled entirely with nodes, except the lowest level. In the lowest level, all nodes reside on the left side.



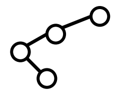
- **Perfect Binary Tree:** all the internal nodes have exactly two children and all leaves are at the same level.



- **Balanced Binary Tree:** the tree height is $O(\log n)$, where n is the number of nodes. The height of the left and right subtrees of each node should vary by at most one.



- **Degenerated Binary Tree:** every internal node has a single child.



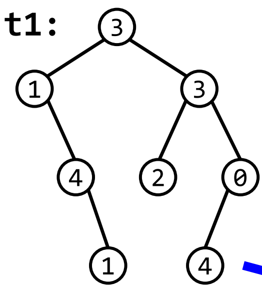
Intersection of binary trees

Design the function

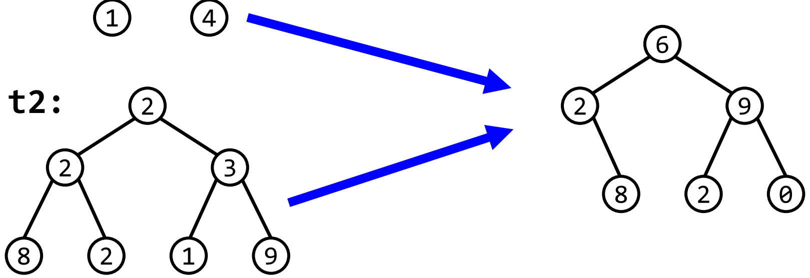
```
def intersection(t1: BinTree[T], t2: BinTree[T],
                 f: Callable[[T, T], T]) -> BinTree[T]:
```

that returns the common structure of both trees and combines the values of the common nodes with the function **f**.

t1:




```
intersection(t1, t2, lambda x, y: x*y)
```



Traversals: Full Binary Trees

- A Full Binary Tree is a binary tree where each node has 0 or 2 children.
- Draw the full binary trees corresponding to the following tree traversals:
  - Preorder: 2 7 3 6 1 4 5; Postorder: 3 6 7 4 5 1 2
  - Preorder: 3 1 7 4 9 5 2 6 8; Postorder: 1 9 5 4 6 8 2 7 3
- Given the pre- and post-order traversals of a binary tree (not necessarily full), can we uniquely determine the tree?
  - If yes, prove it.
  - If not, show a counterexample.

Traversals: Binary Trees

- Draw the binary trees corresponding the following traversals:
  - Preorder: 3 6 1 8 5 2 4 7 9; Inorder: 1 6 3 5 2 8 7 4 9
  - Level-order: 4 8 3 1 2 7 5 6 9; Inorder: 1 8 5 2 4 6 7 9 3
  - Postorder: 4 3 2 5 9 6 8 7 1; Inorder: 4 3 9 2 5 1 7 8 6

- Describe an algorithm that builds a binary tree from the preorder and inorder traversals.

Drawing binary trees

We want to draw the skeleton of a binary tree as it is shown in the figure. For that, we need to assign (x, y) coordinates to each tree node. The layout must fit in a pre-defined bounding box of size $W \times H$, with the origin located in the top-left corner. Design the function:

```
T = TypeVar('T')
Coordinate = tuple[float, float]
Coordinates = dict[Bintree, Coordinate]

def draw(t: Bintree, w: float, h: float) -> Coordinates:
```

that returns a dictionary with the coordinates of all tree nodes in such a way that the lines that connect the nodes do not cross.