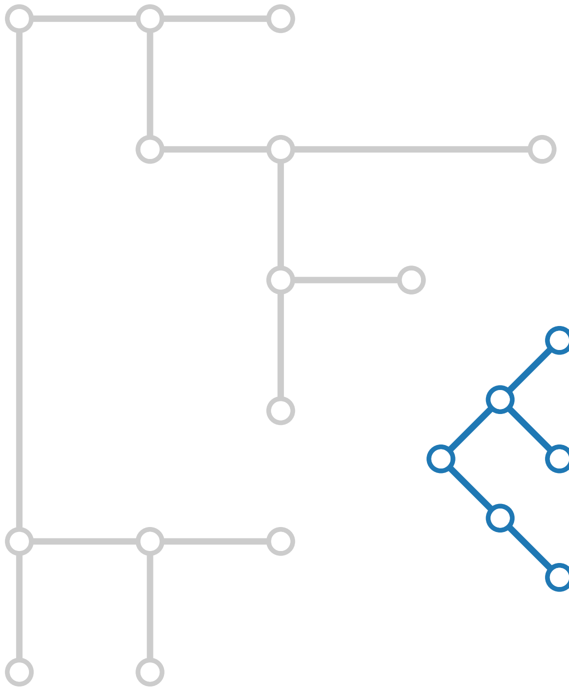
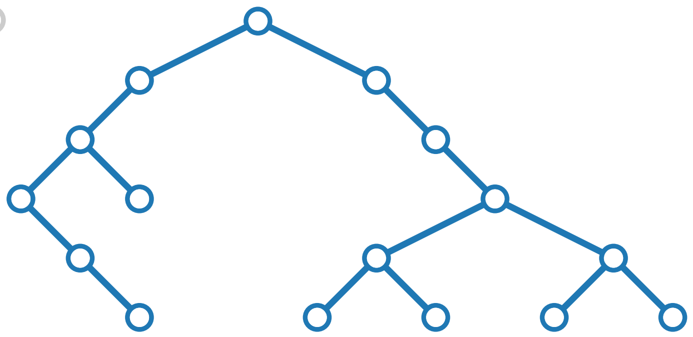
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Visualization of Graphs

Lecture 1b:
Drawing Trees

Part I:
Layered Drawings

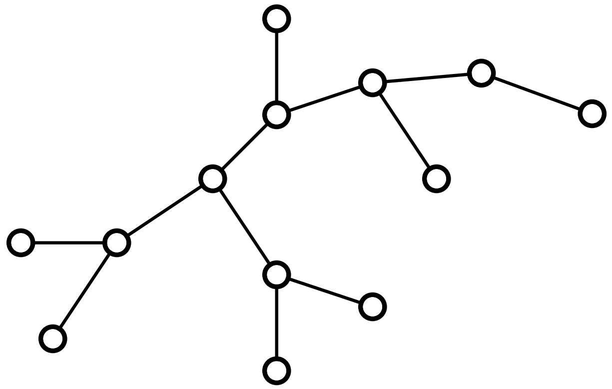




Johannes Zink

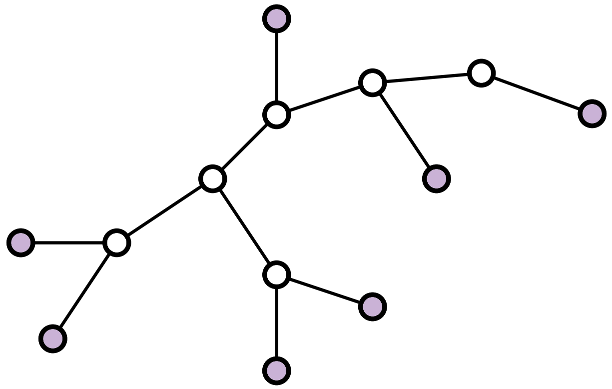
Summer semester 2024

(Rooted) Trees



(Rooted) Trees

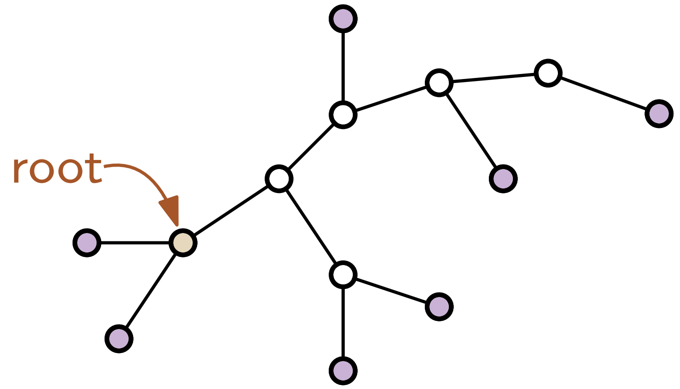
Leaf: vertex of degree 1



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

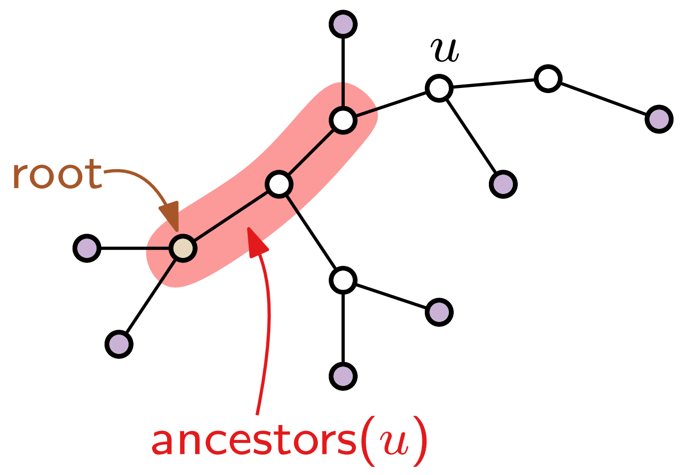


(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root



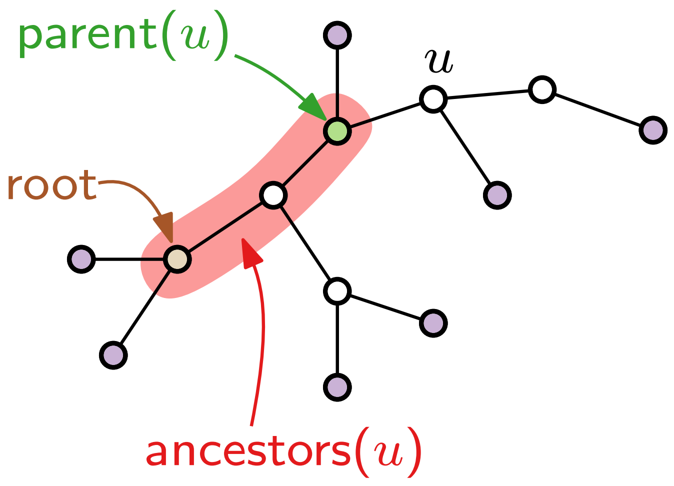
(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root



(Rooted) Trees

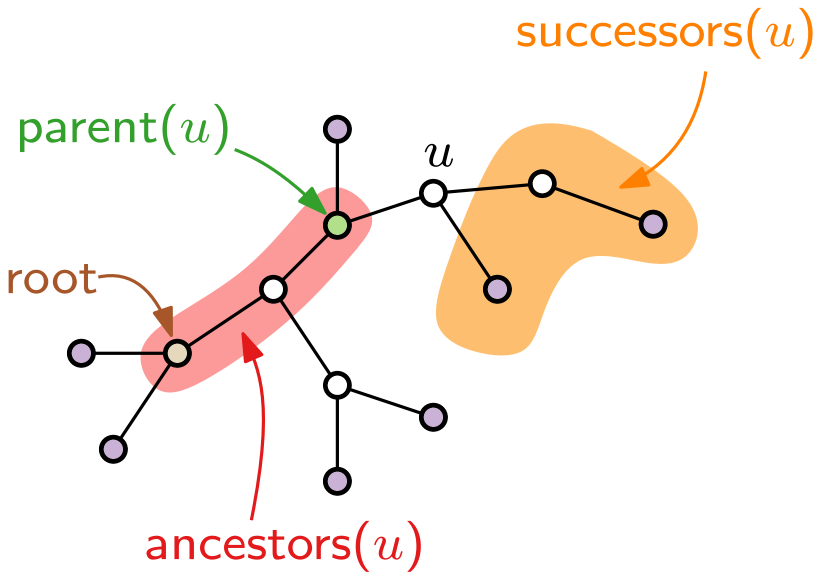
Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

Successor: vertex on the path away from the root



(Rooted) Trees

Leaf: vertex of degree 1

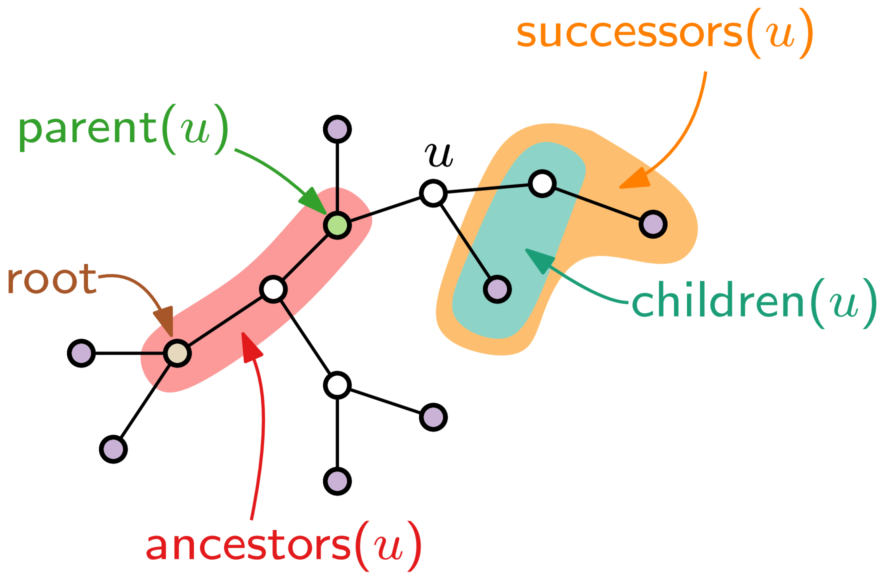
Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

Successor: vertex on the path away from the root

Child: neighbor not on the path to the root



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

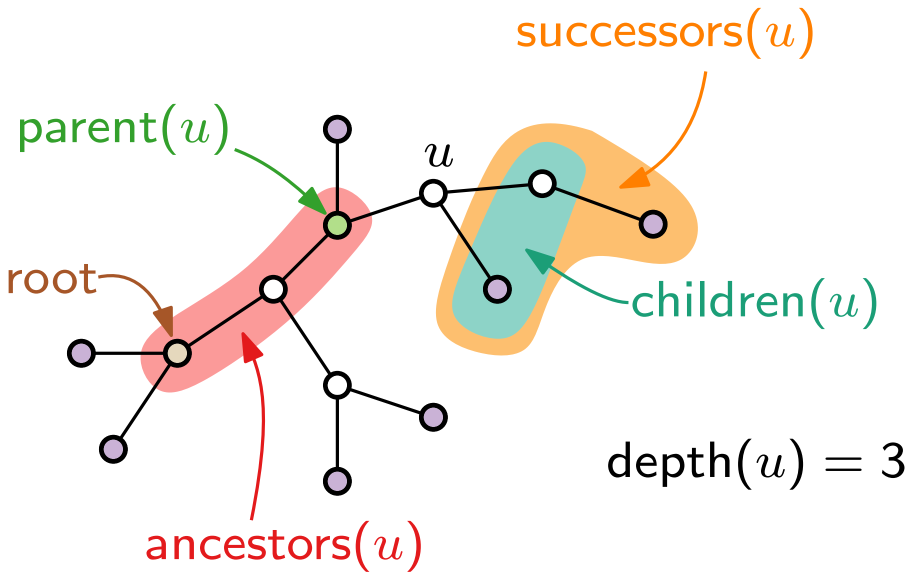
Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

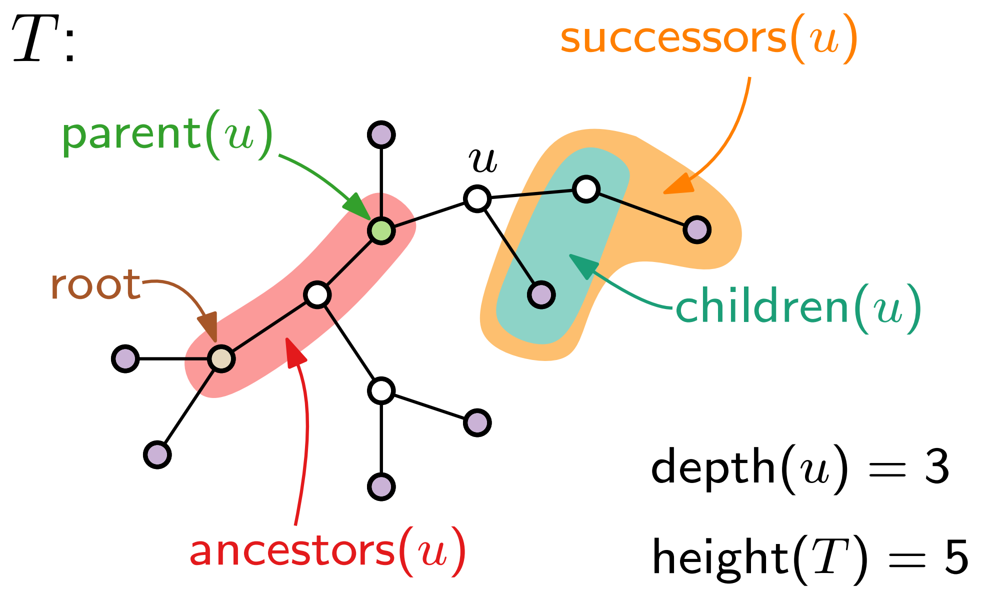
Parent: neighbor on the path to the root

Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

Height: maximum depth of a leaf



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

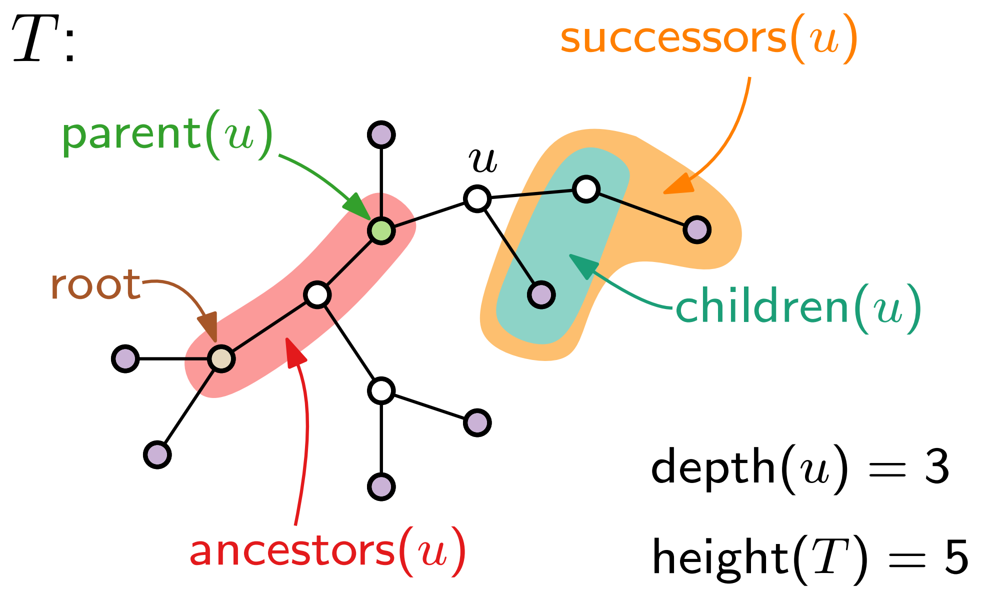
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

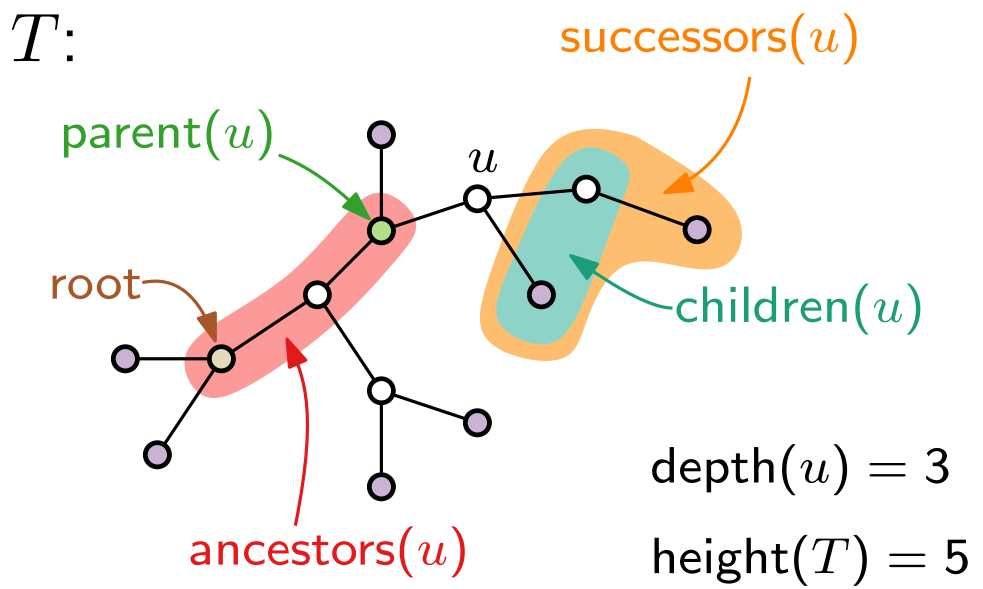
Successor: vertex on the path away from the root

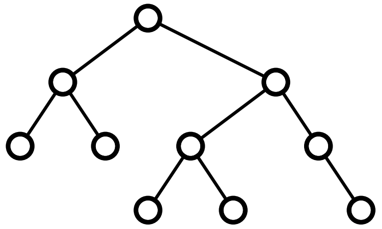
Child: neighbor not on the path to the root

Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)





(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

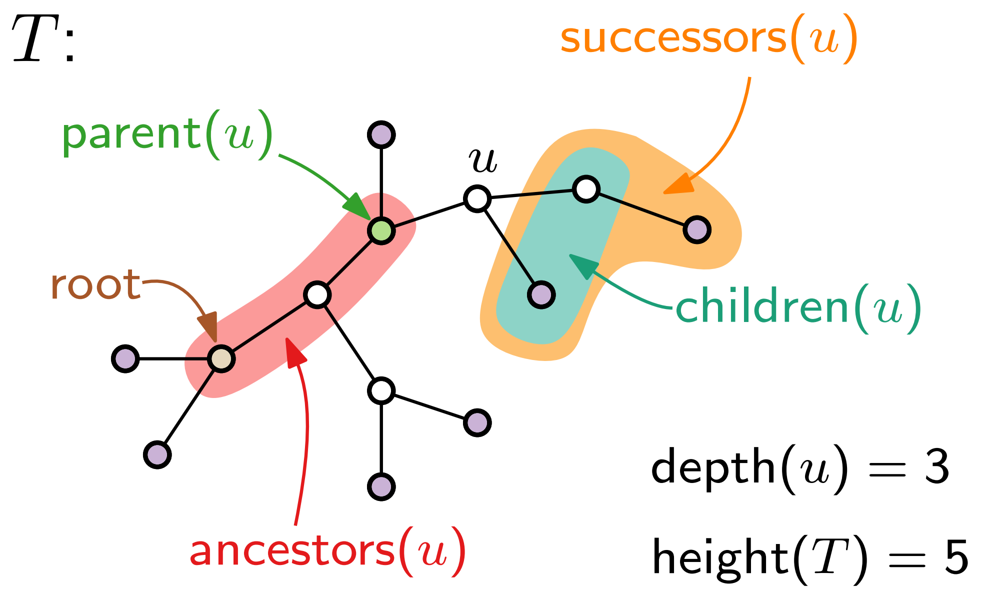
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

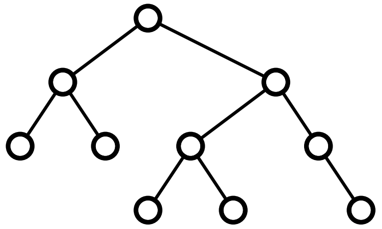
Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

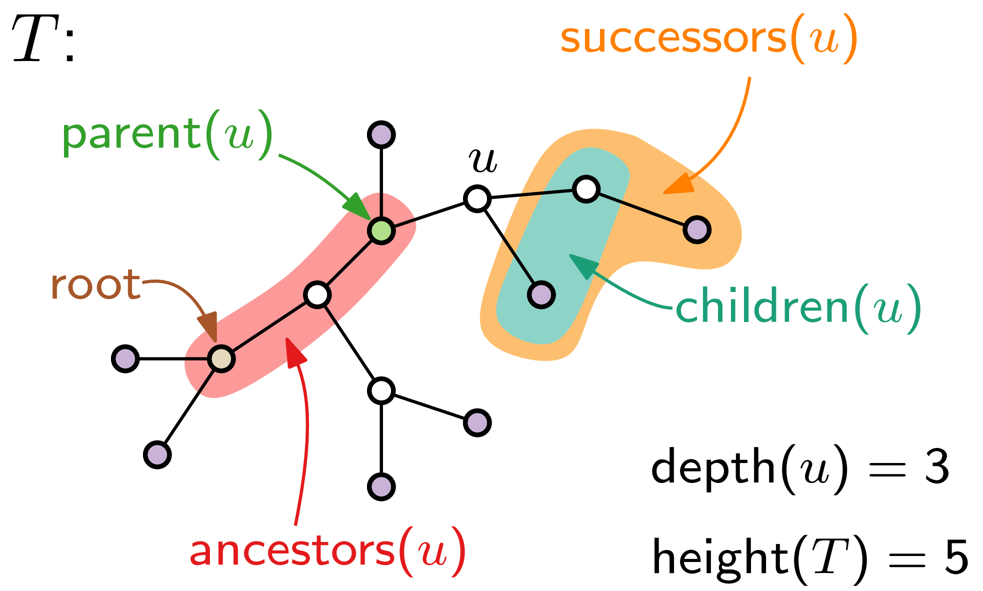
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

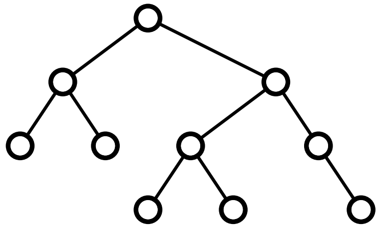
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

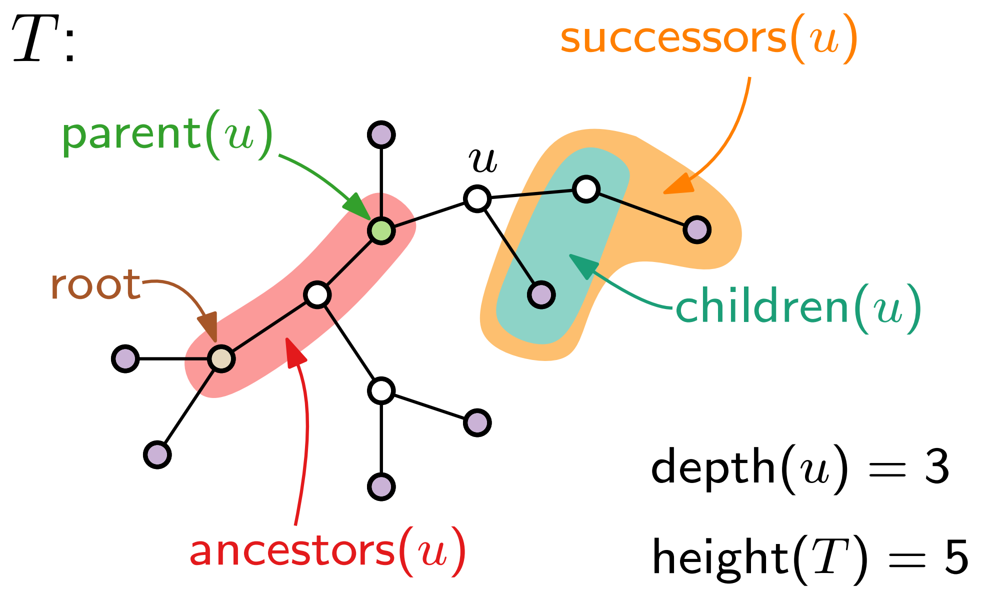
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

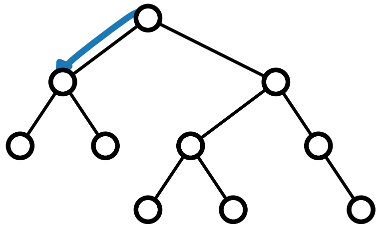
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

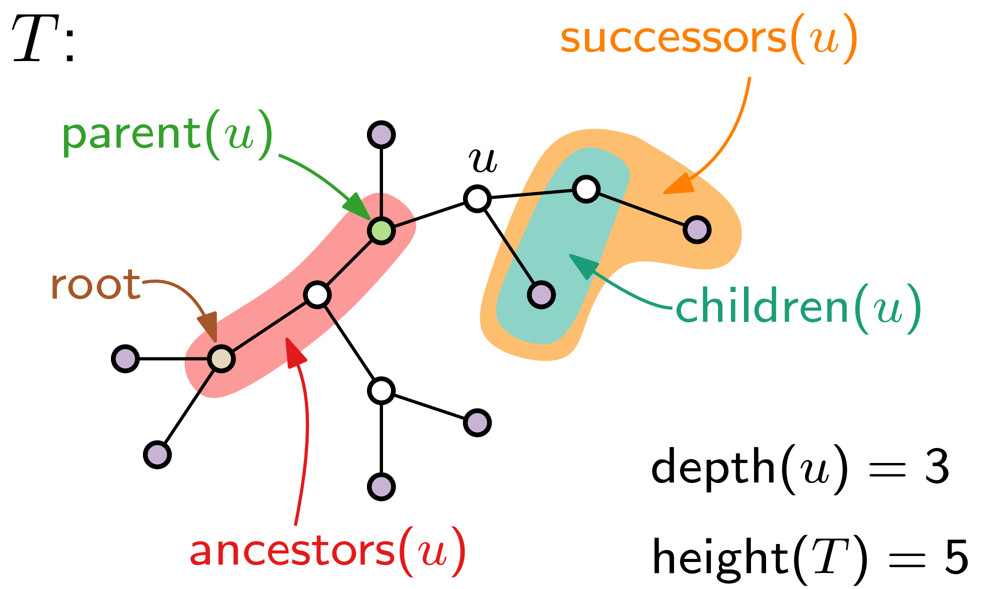
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

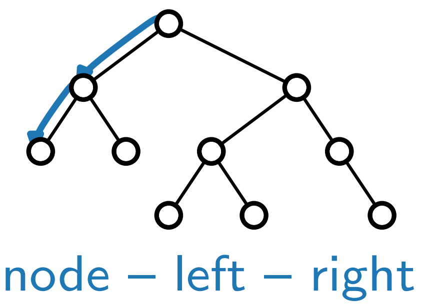
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:

preorder



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

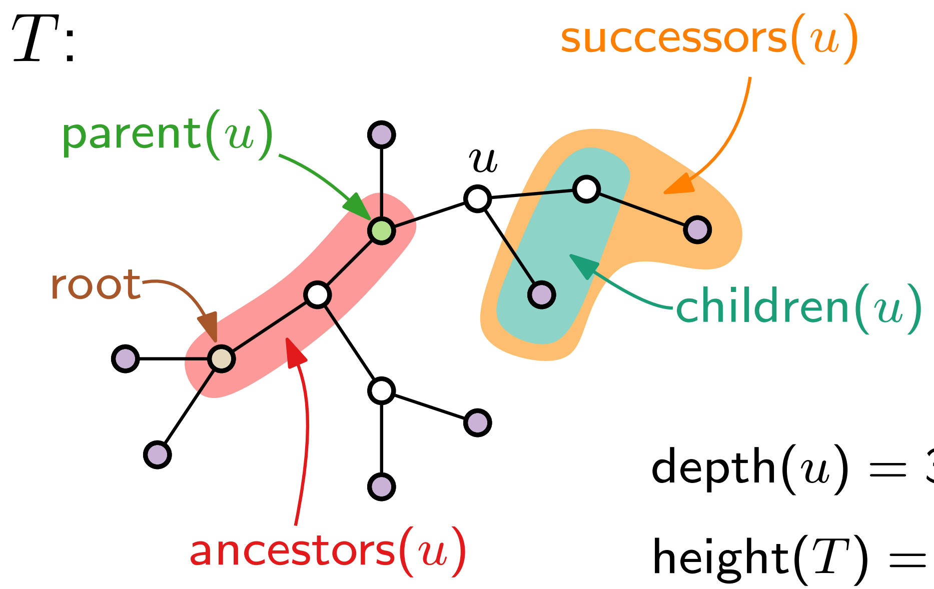
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)

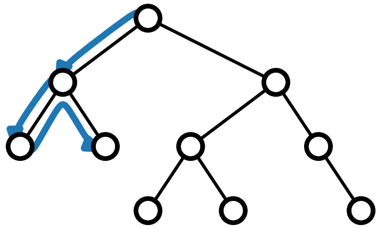


depth(u) $= 3$

height(T) $= 5$

3 types of tree traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

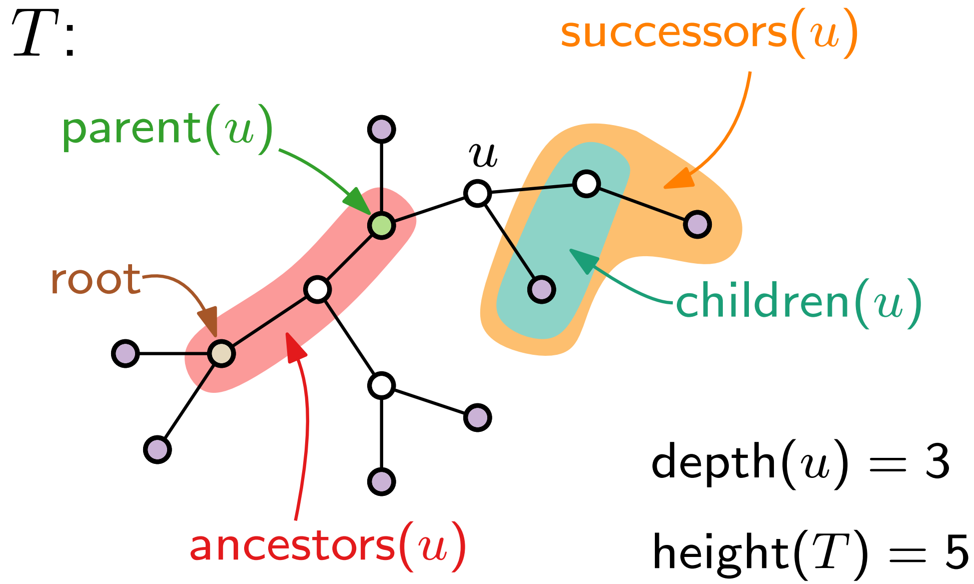
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

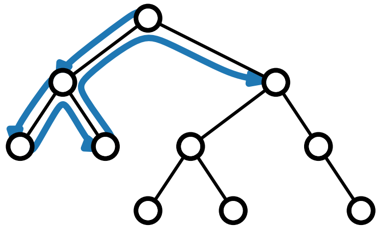
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:

preorder



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

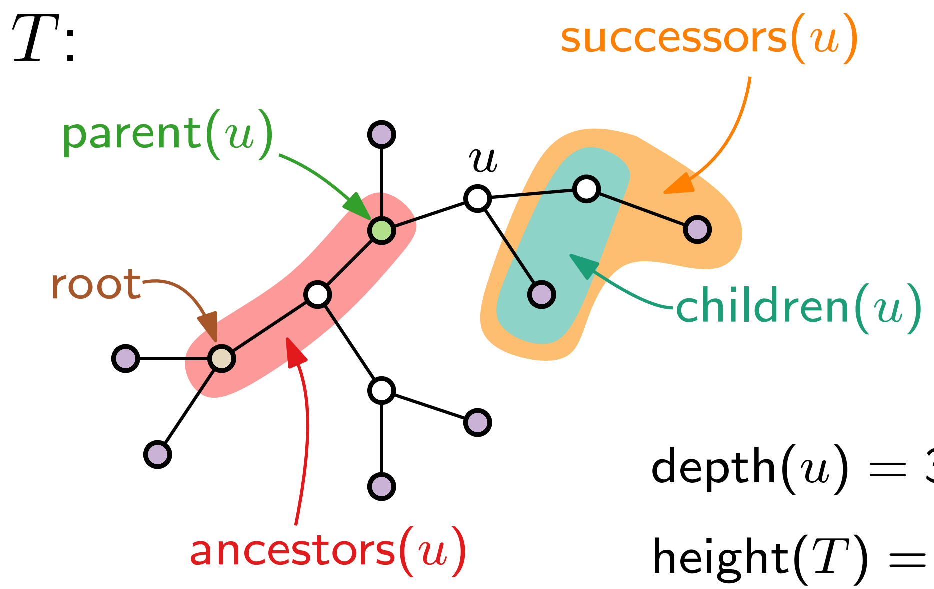
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)

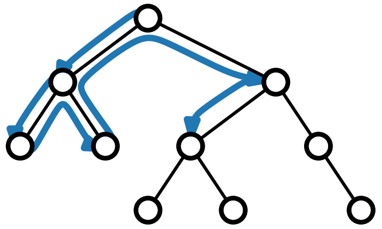


depth(u) $= 3$

height(T) $= 5$

3 types of tree traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

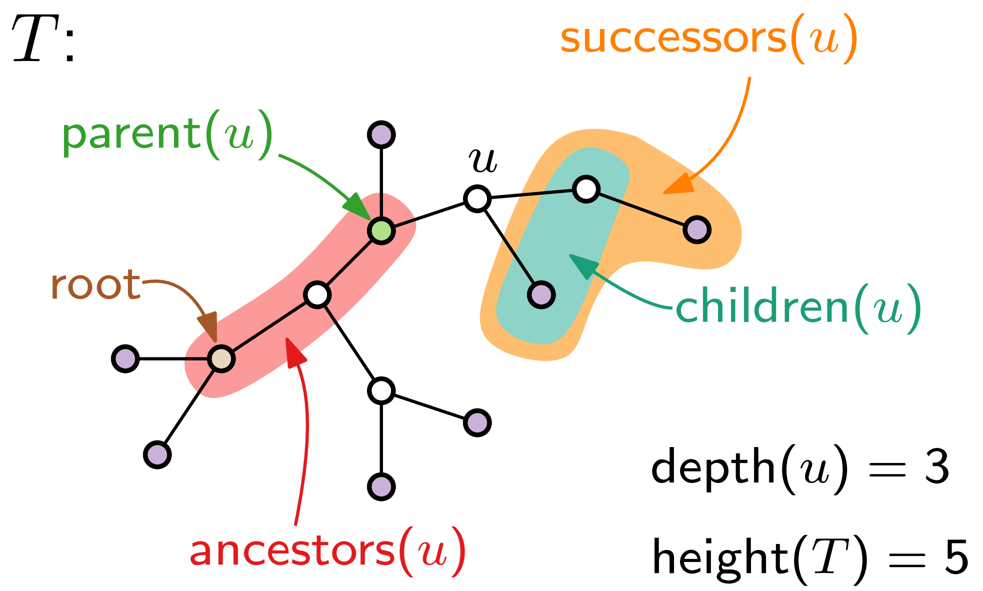
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

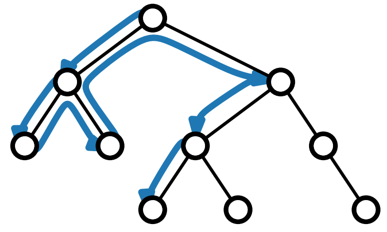
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

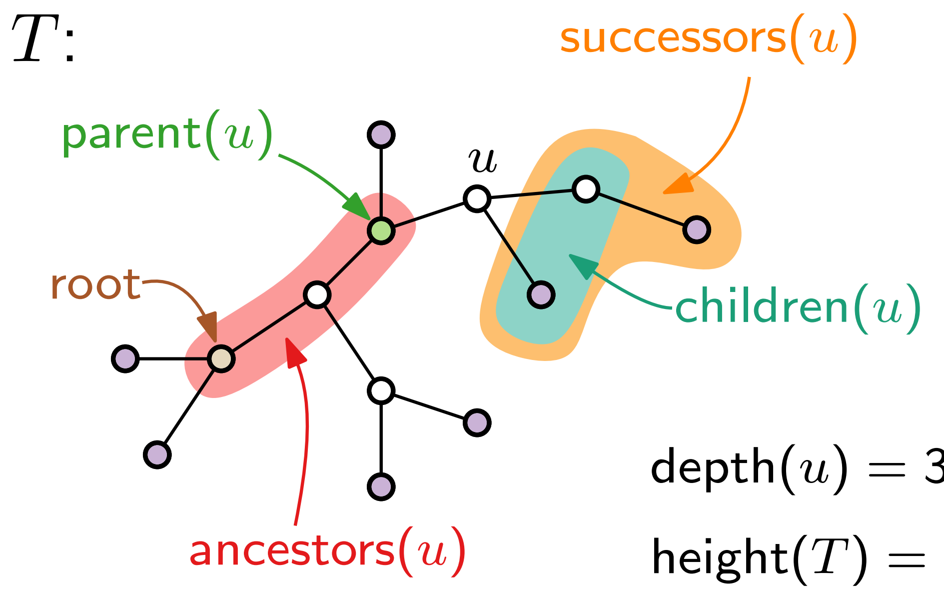
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)

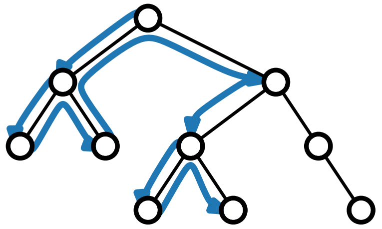


depth(u) $= 3$

height(T) $= 5$

3 types of tree traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

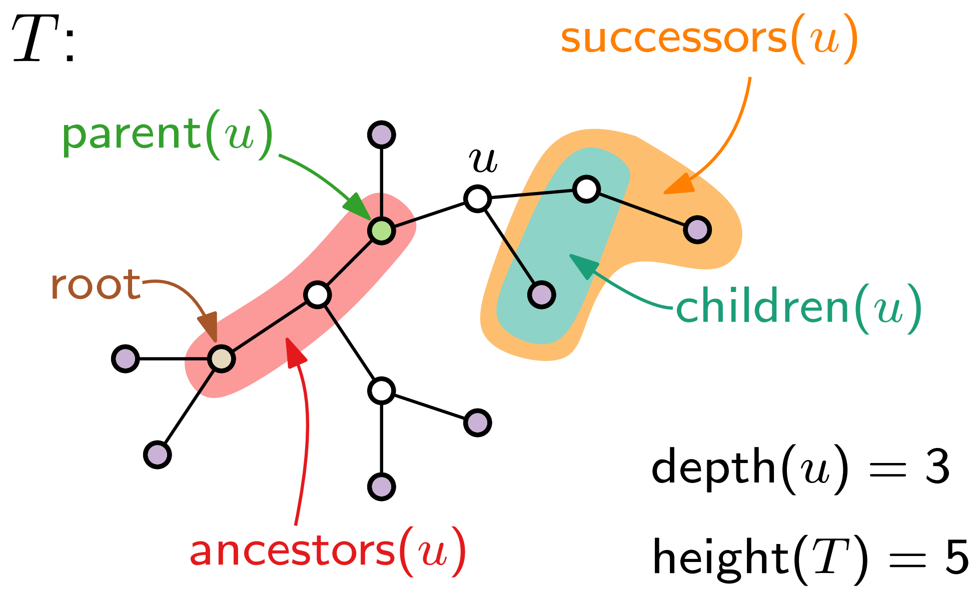
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

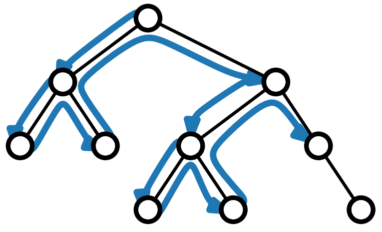
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

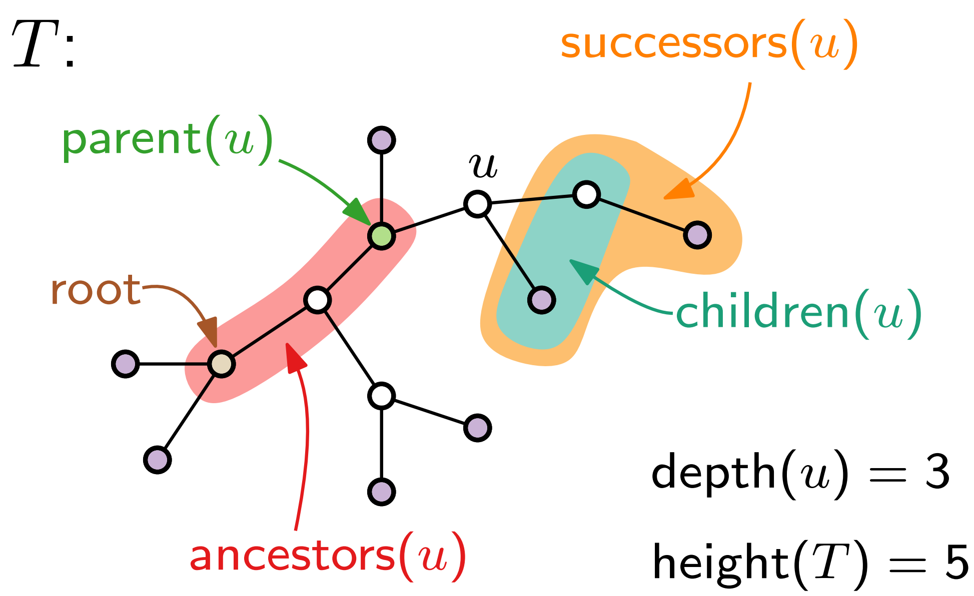
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

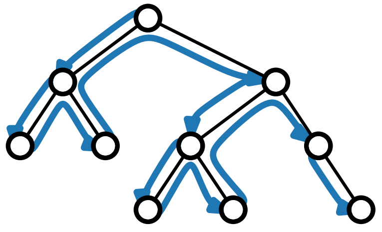
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:

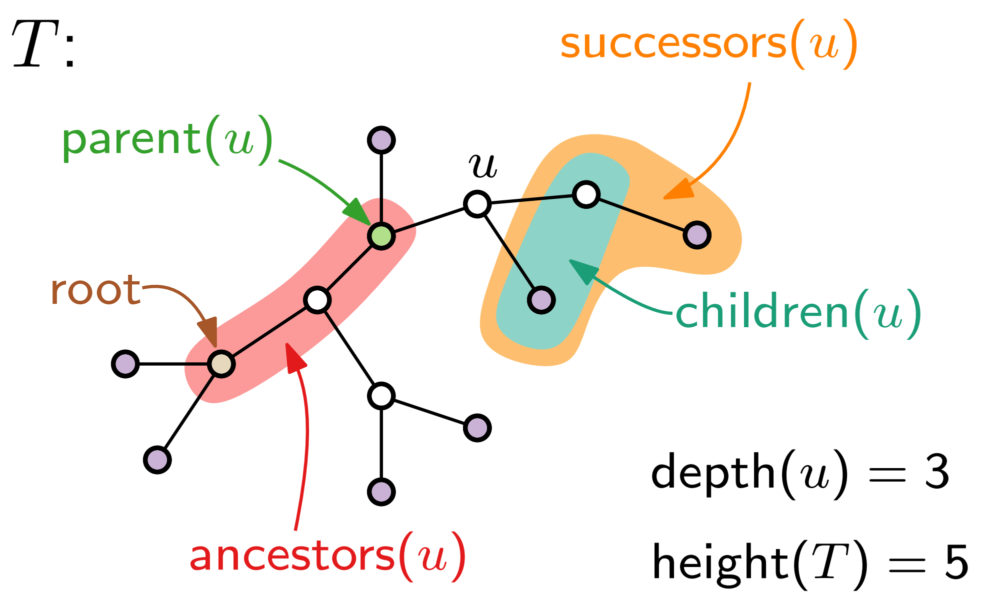
preorder



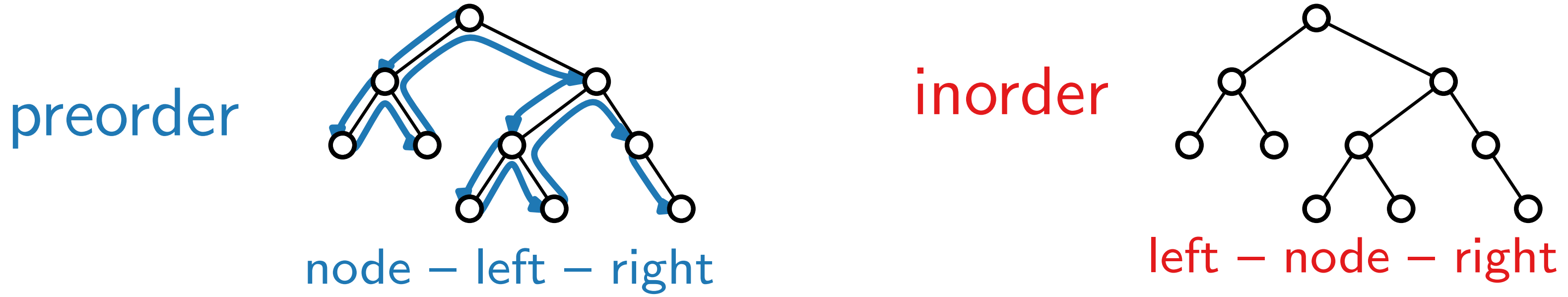
node – left – right

(Rooted) Trees

Leaf: vertex of degree 1
Rooted tree: tree with a designated **root**
Ancestor: vertex on the path to the root
Parent: neighbor on the path to the root
Successor: vertex on the path away from the root
Child: neighbor not on the path to the root
Depth: length of the path to the root
Height: maximum depth of a leaf
Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

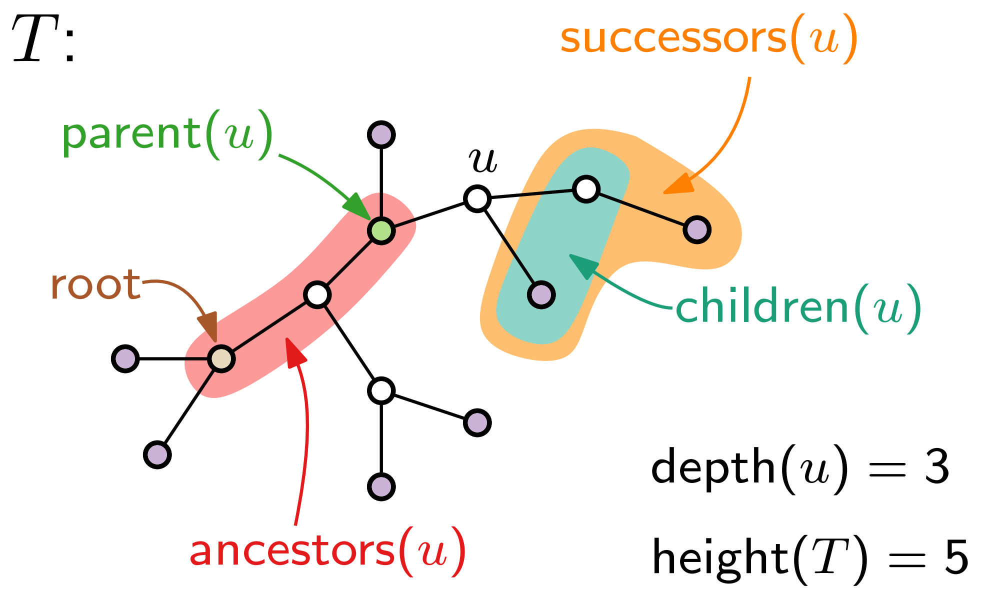
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

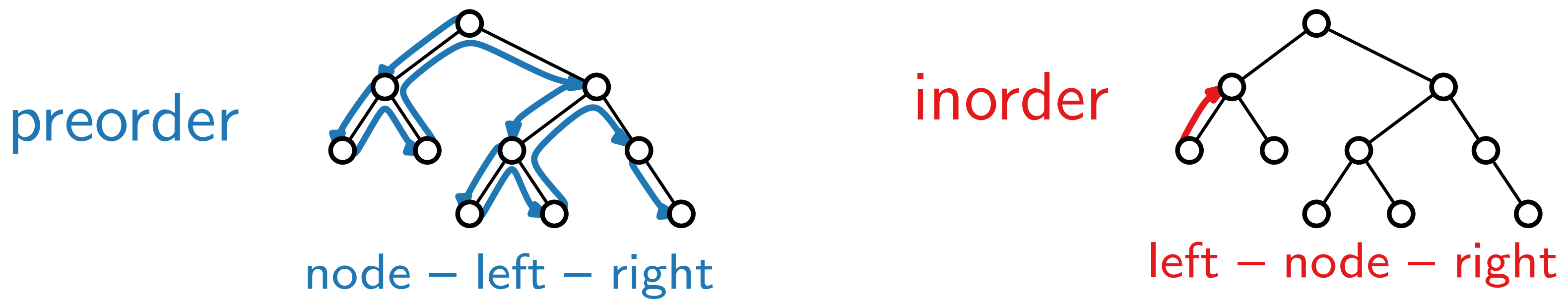
Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

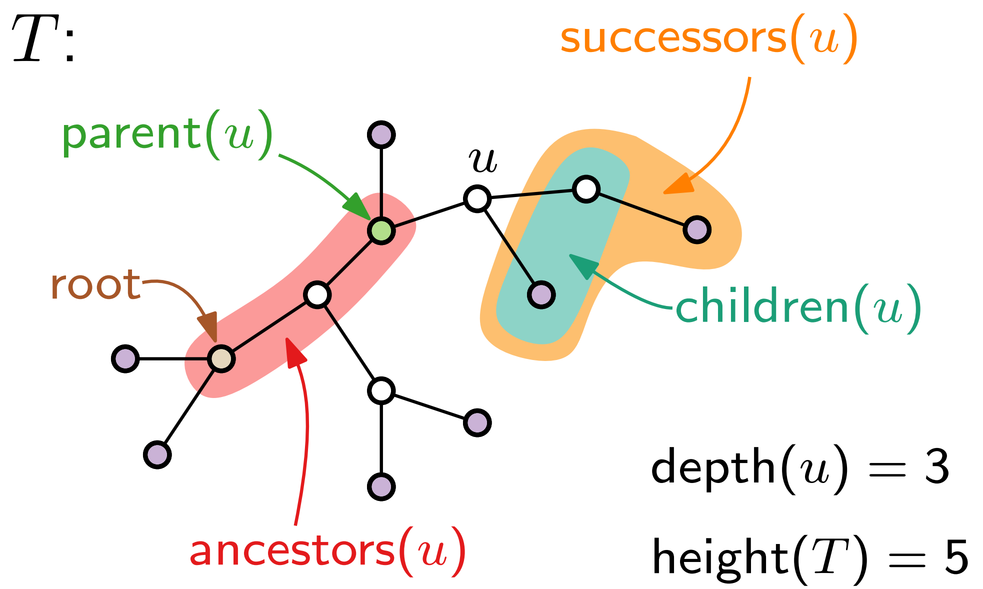
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

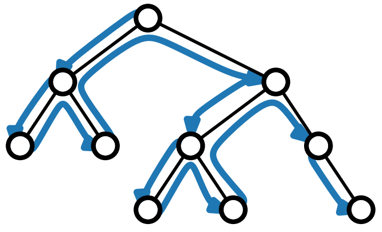
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



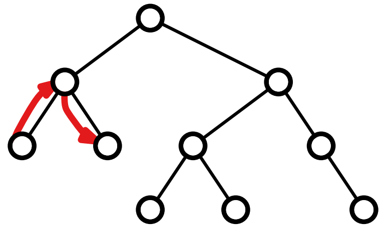
3 types of tree traversals:

preorder



node – left – right

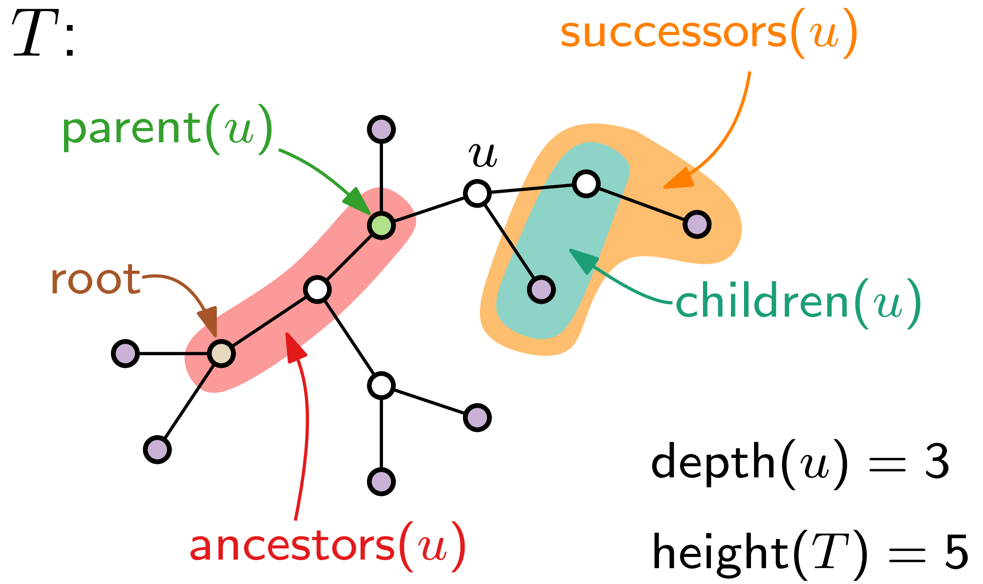
inorder



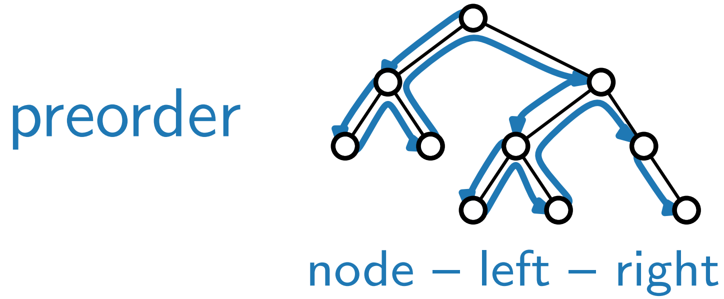
left – node – right

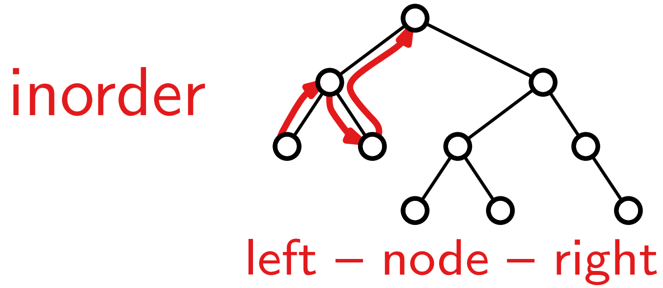
(Rooted) Trees

Leaf: vertex of degree 1
Rooted tree: tree with a designated **root**
Ancestor: vertex on the path to the root
Parent: neighbor on the path to the root
Successor: vertex on the path away from the root
Child: neighbor not on the path to the root
Depth: length of the path to the root
Height: maximum depth of a leaf
Binary Tree: at most two children per vertex (*left* and *right* child)



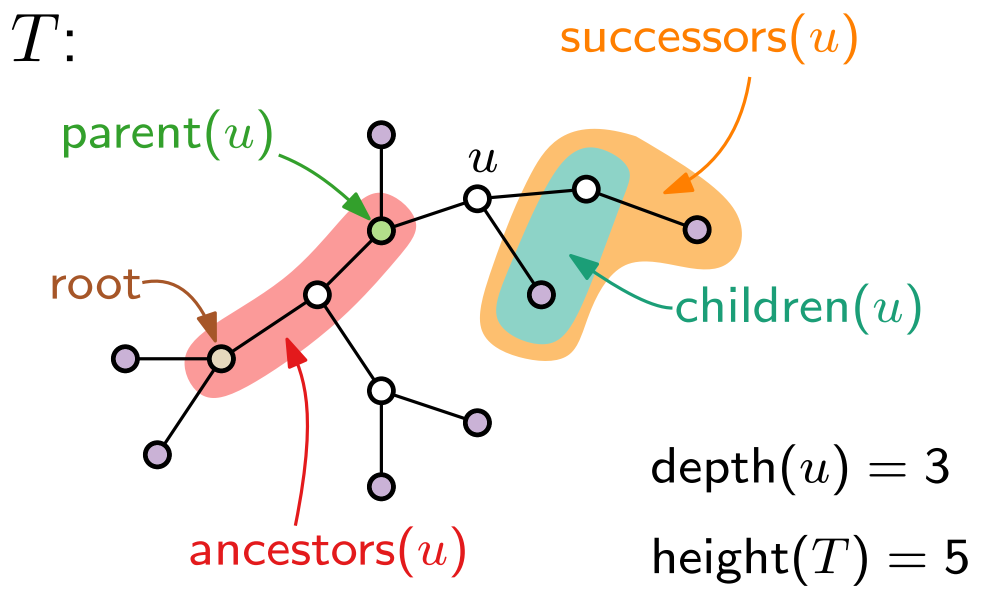
3 types of tree traversals:





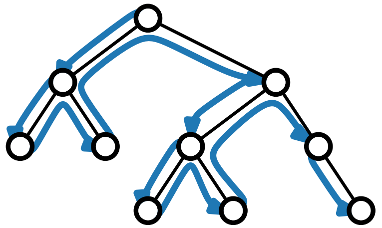
(Rooted) Trees

Leaf: vertex of degree 1
Rooted tree: tree with a designated **root**
Ancestor: vertex on the path to the root
Parent: neighbor on the path to the root
Successor: vertex on the path away from the root
Child: neighbor not on the path to the root
Depth: length of the path to the root
Height: maximum depth of a leaf
Binary Tree: at most two children per vertex (*left* and *right* child)



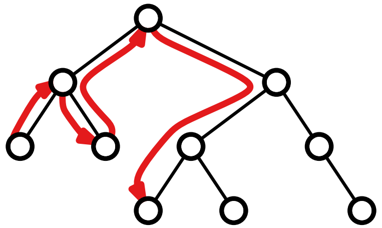
3 types of tree traversals:

preorder



node – left – right

inorder



left – node – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

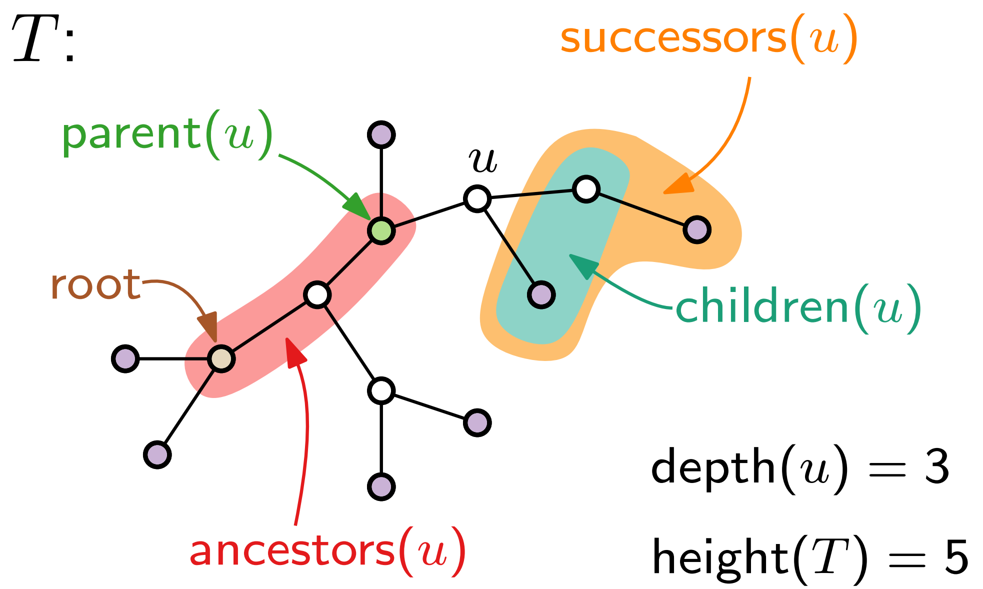
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

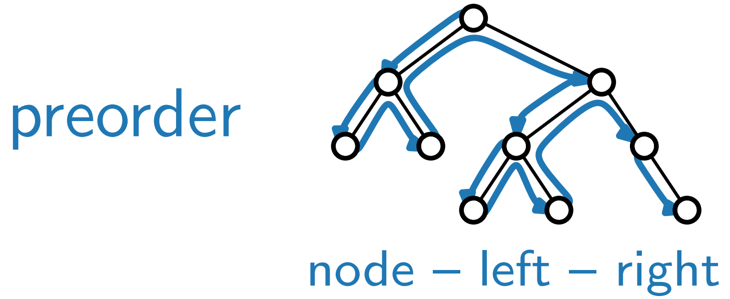
Depth: length of the path to the root

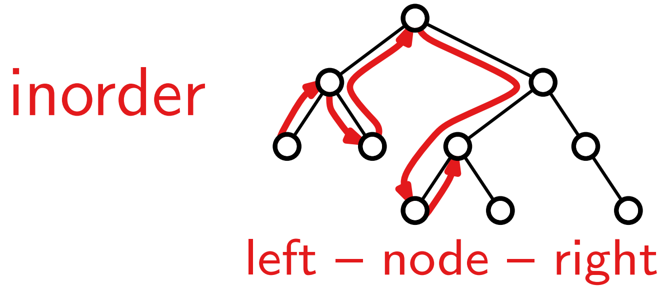
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:





(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

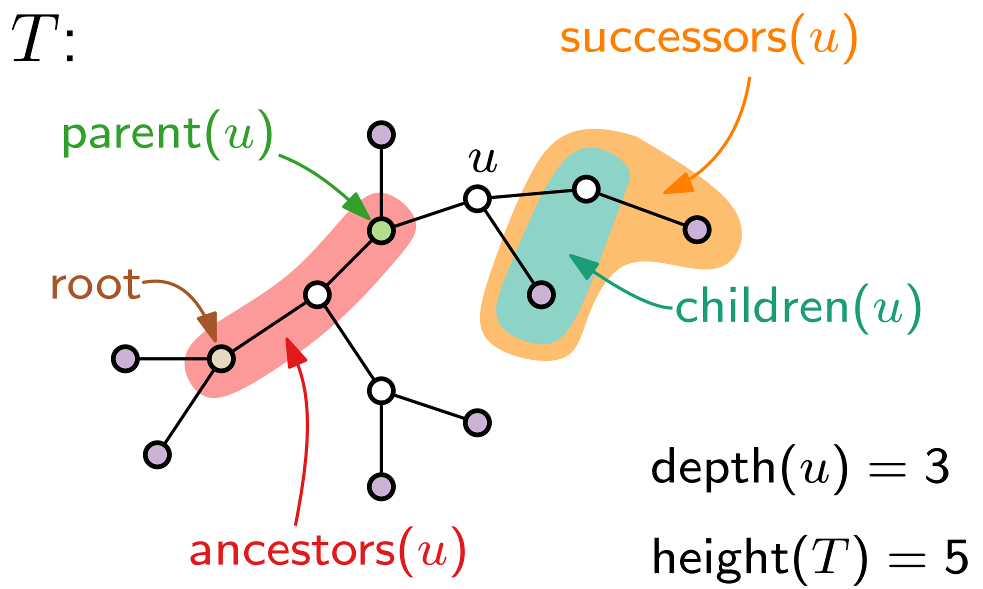
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

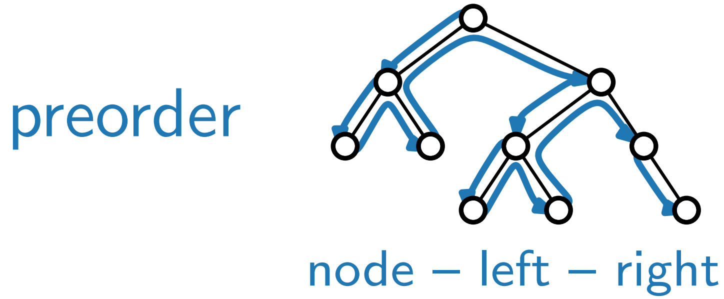
Depth: length of the path to the root

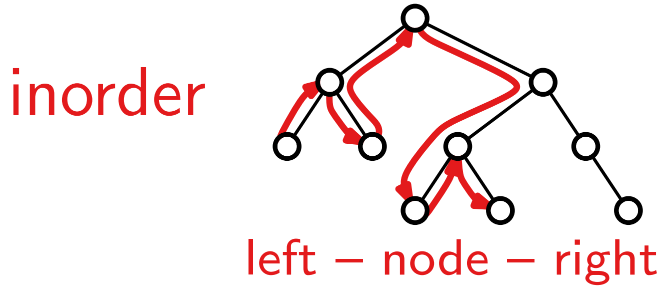
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:





(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

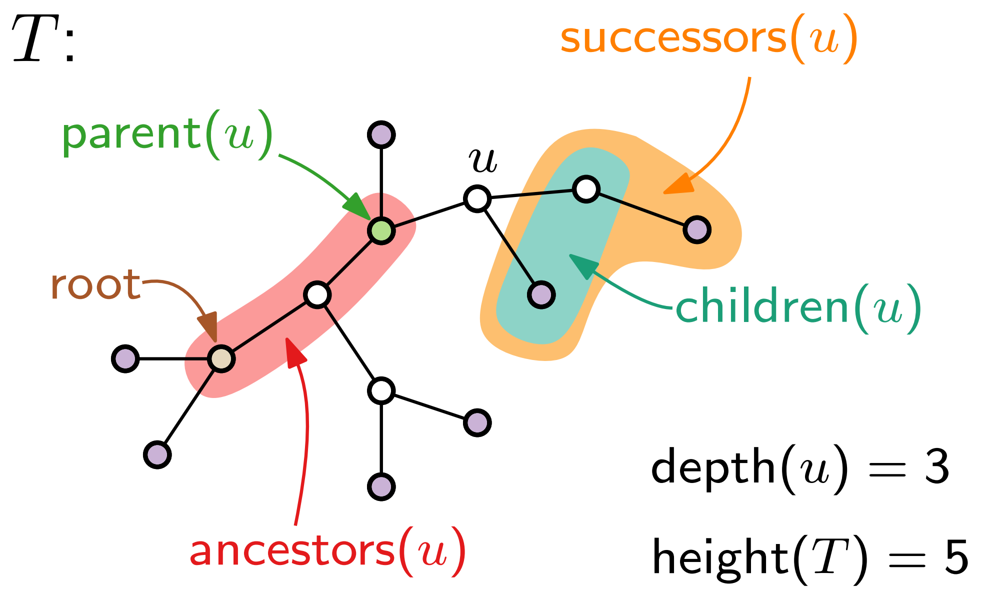
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

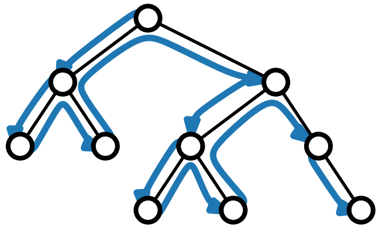
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



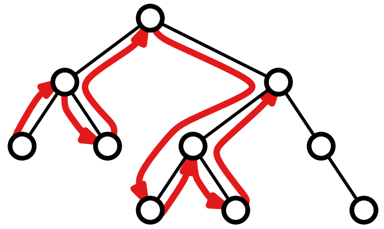
3 types of tree traversals:

preorder



node – left – right

inorder



left – node – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

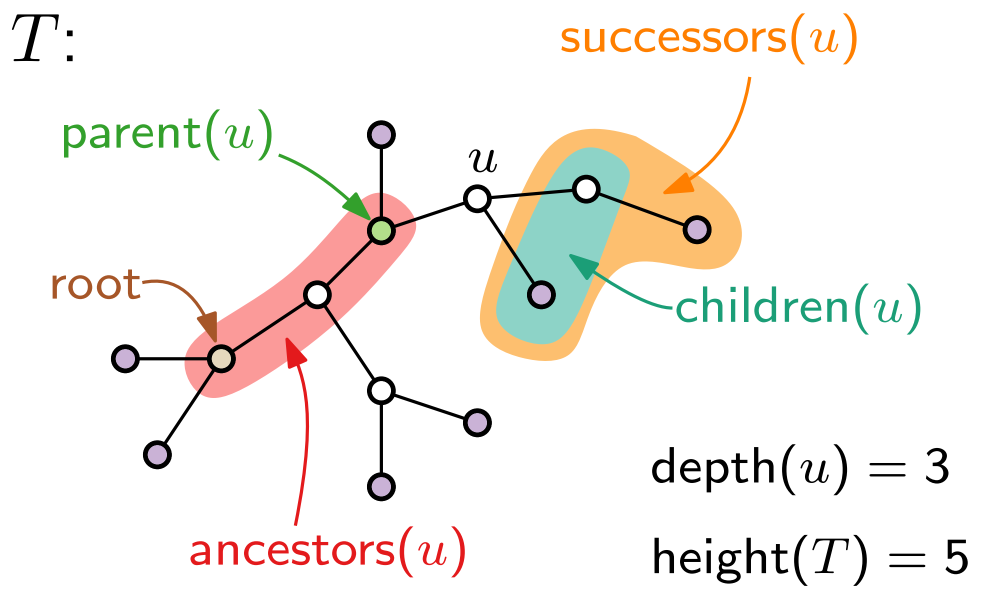
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

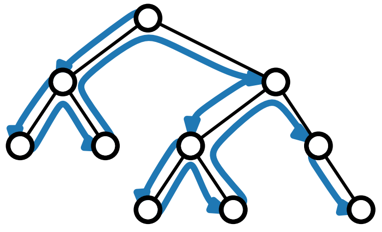
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



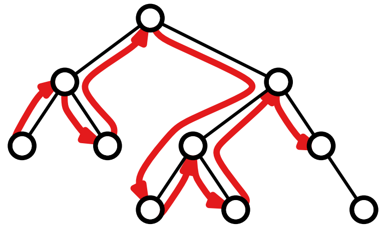
3 types of tree traversals:

preorder



node – left – right

inorder



left – node – right

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

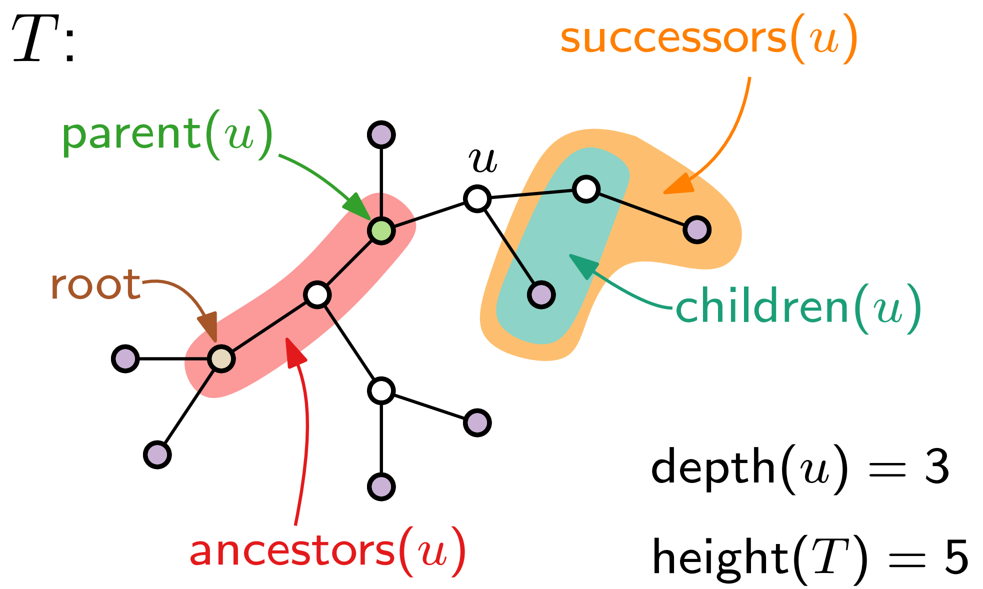
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

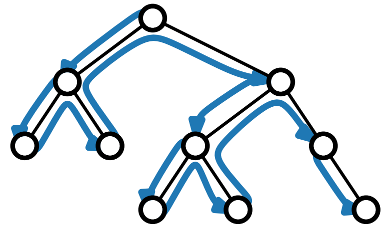
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)

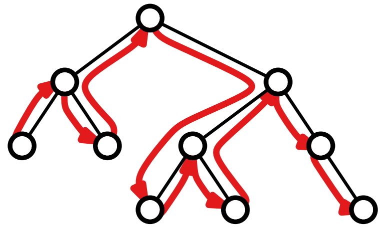


3 types of tree traversals:

preorder



inorder



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

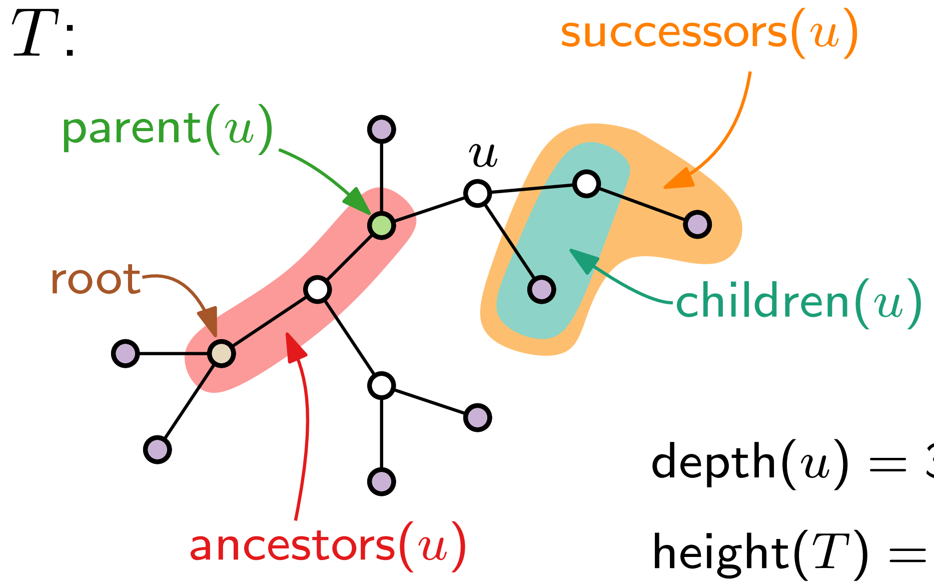
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)

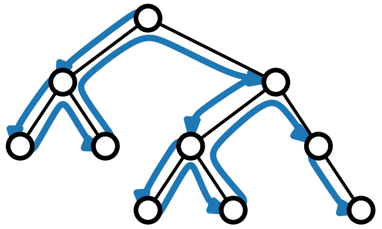


depth(u) $= 3$

height(T) $= 5$

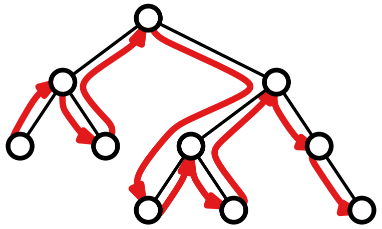
3 types of tree traversals:

preorder



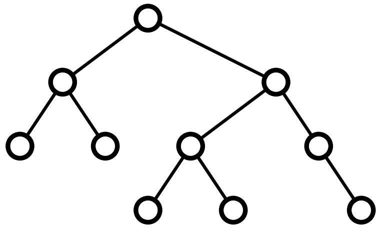
node – left – right

inorder



left – node – right

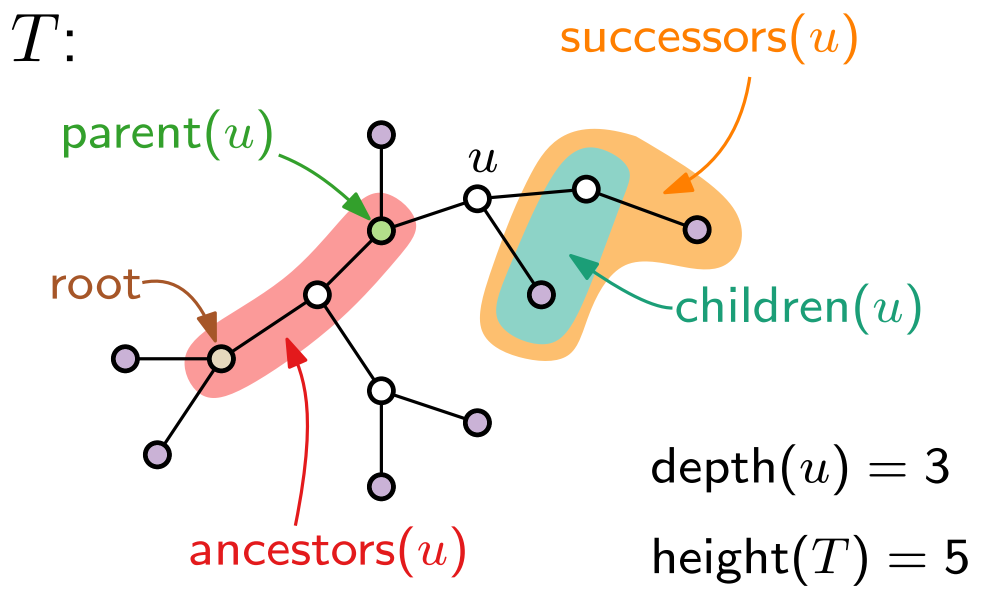
postorder



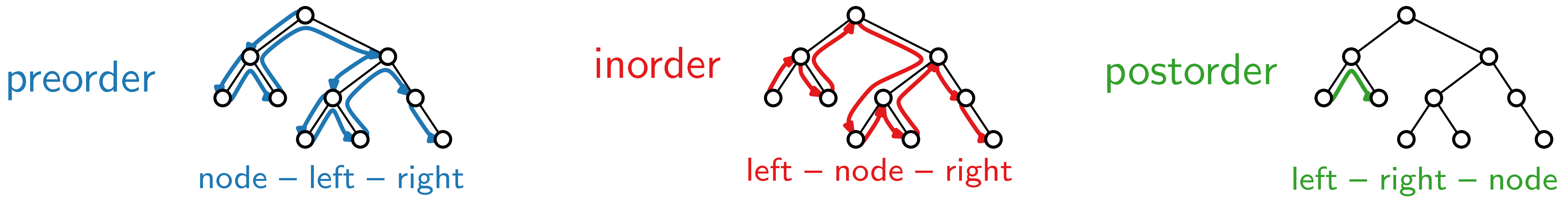
left – right – node

(Rooted) Trees

Leaf: vertex of degree 1
Rooted tree: tree with a designated **root**
Ancestor: vertex on the path to the root
Parent: neighbor on the path to the root
Successor: vertex on the path away from the root
Child: neighbor not on the path to the root
Depth: length of the path to the root
Height: maximum depth of a leaf
Binary Tree: at most two children per vertex (*left* and *right* child)

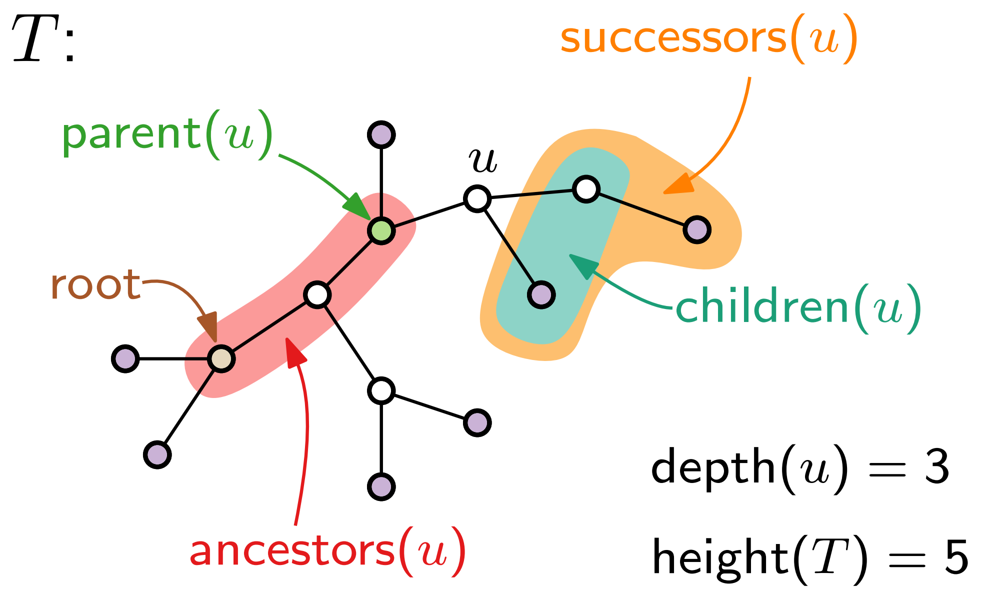


3 types of tree traversals:

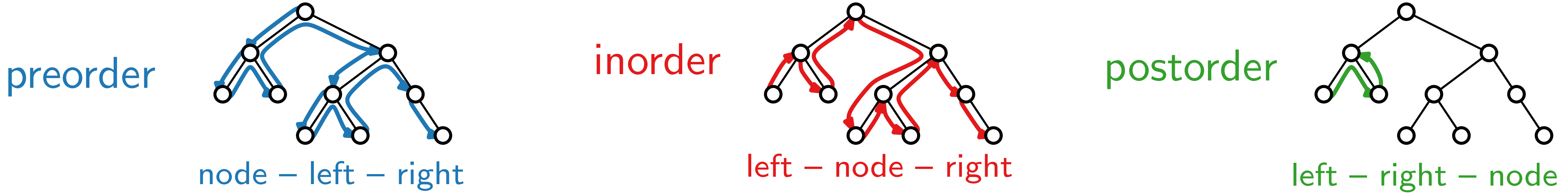


(Rooted) Trees

Leaf: vertex of degree 1
Rooted tree: tree with a designated **root**
Ancestor: vertex on the path to the root
Parent: neighbor on the path to the root
Successor: vertex on the path away from the root
Child: neighbor not on the path to the root
Depth: length of the path to the root
Height: maximum depth of a leaf
Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

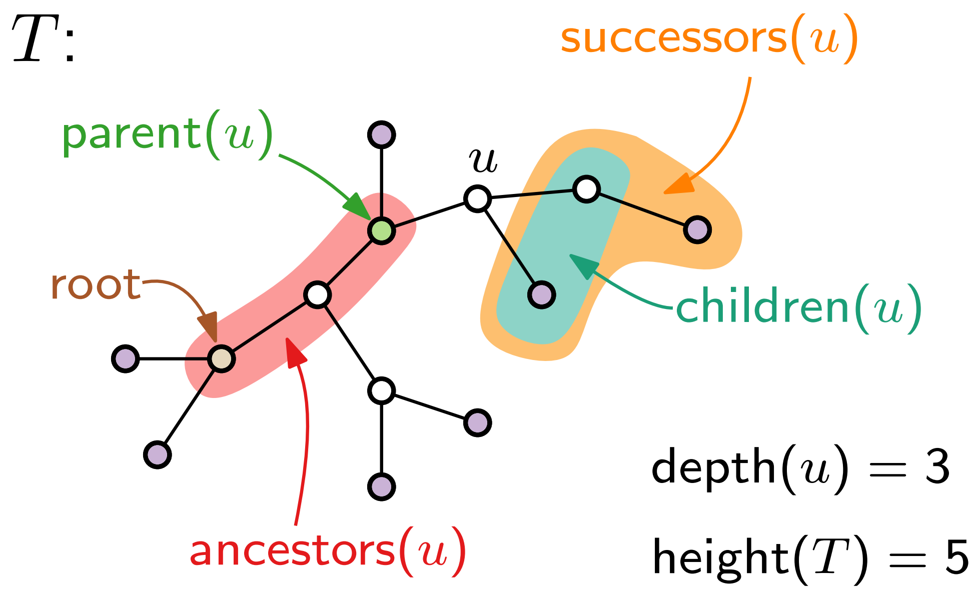
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

Height: maximum depth of a leaf

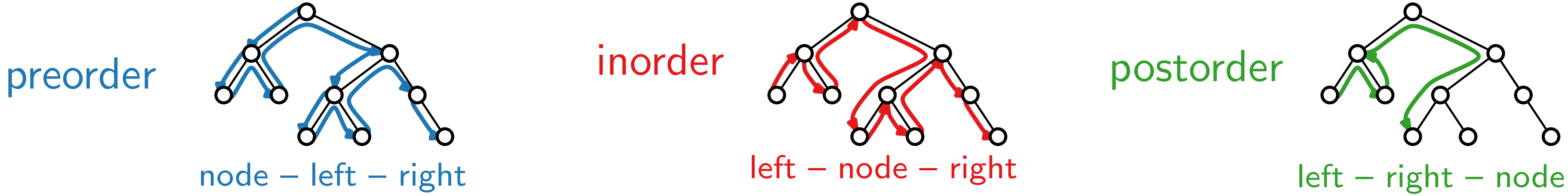
Binary Tree: at most two children per vertex (*left* and *right* child)



depth(u) $= 3$

height(T) $= 5$

3 types of tree traversals:



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

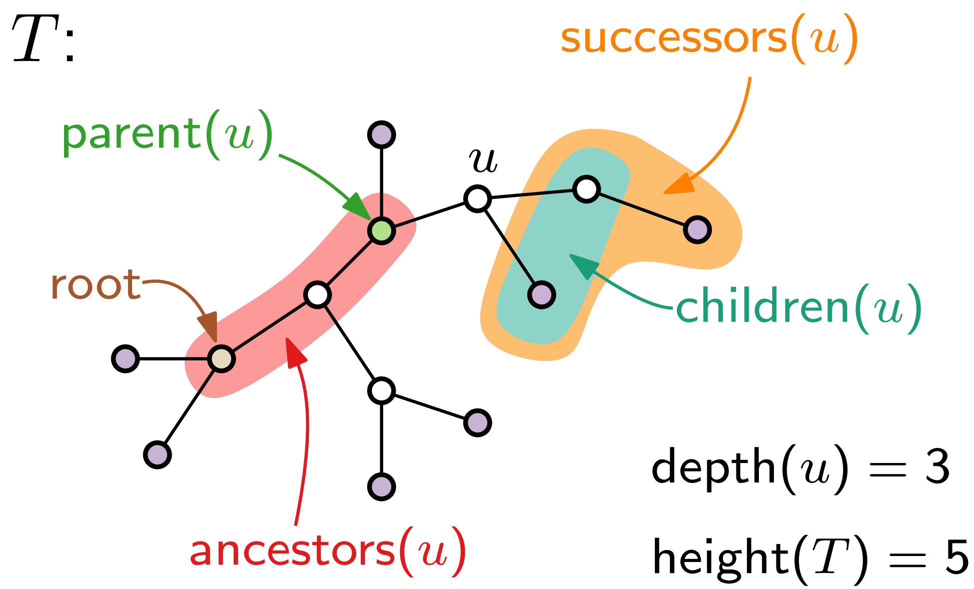
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

Depth: length of the path to the root

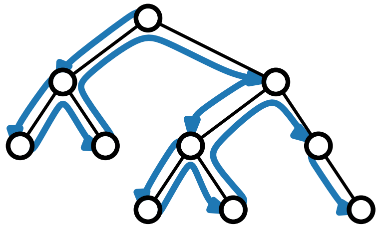
Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



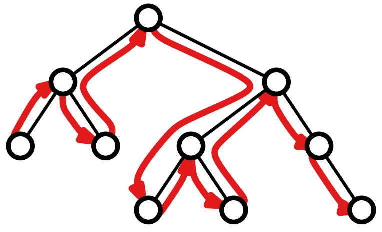
3 types of tree traversals:

preorder



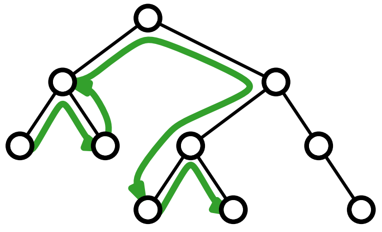
node – left – right

inorder



left – node – right

postorder



left – right – node

(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

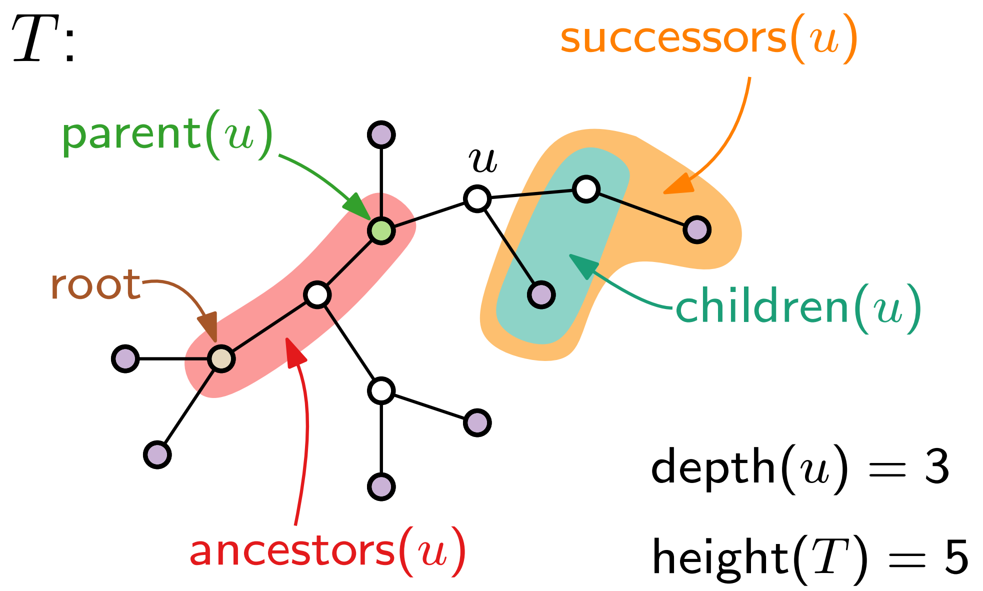
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

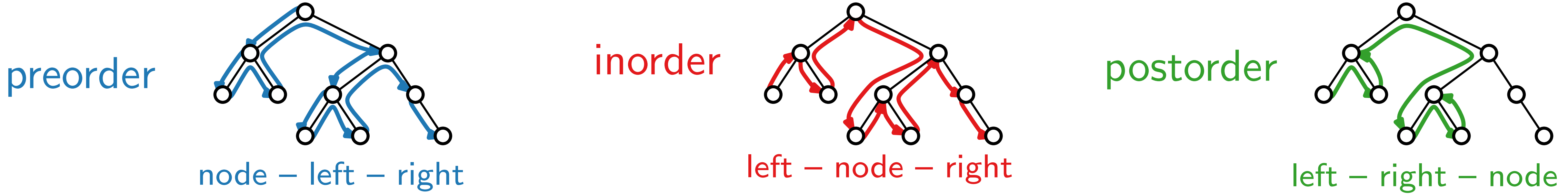
Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)

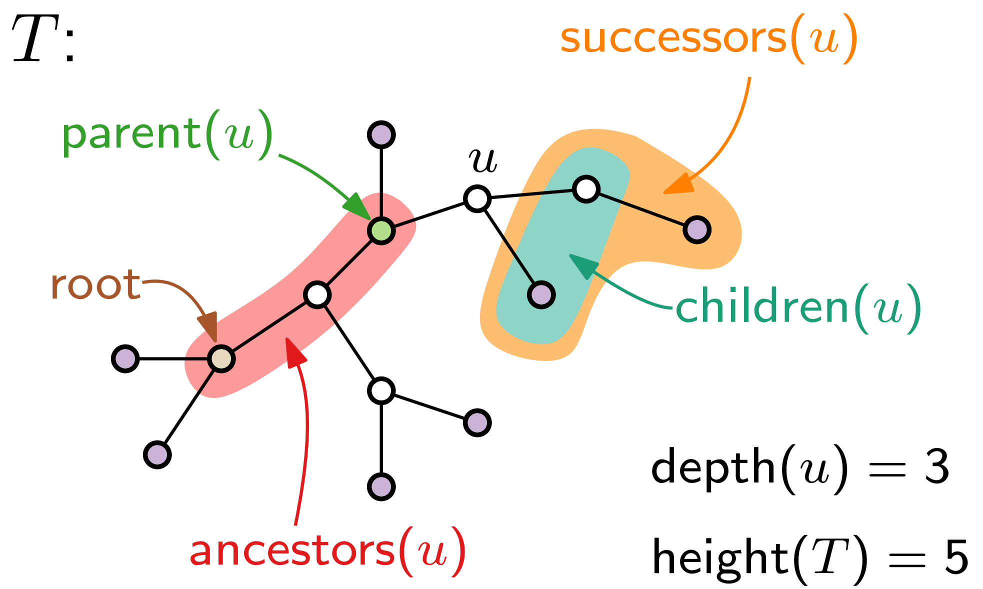


3 types of tree traversals:

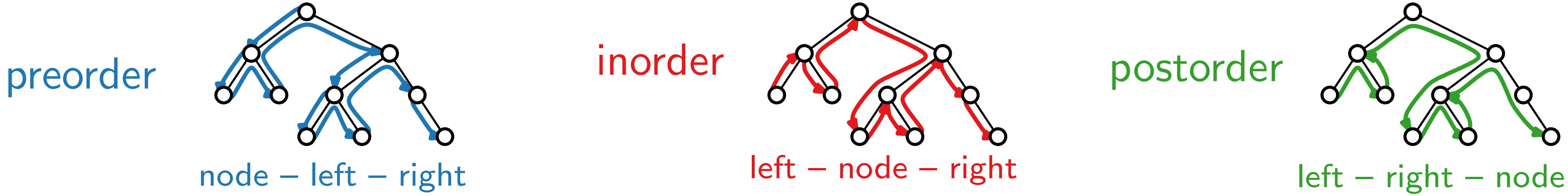


(Rooted) Trees

Leaf: vertex of degree 1
Rooted tree: tree with a designated **root**
Ancestor: vertex on the path to the root
Parent: neighbor on the path to the root
Successor: vertex on the path away from the root
Child: neighbor not on the path to the root
Depth: length of the path to the root
Height: maximum depth of a leaf
Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

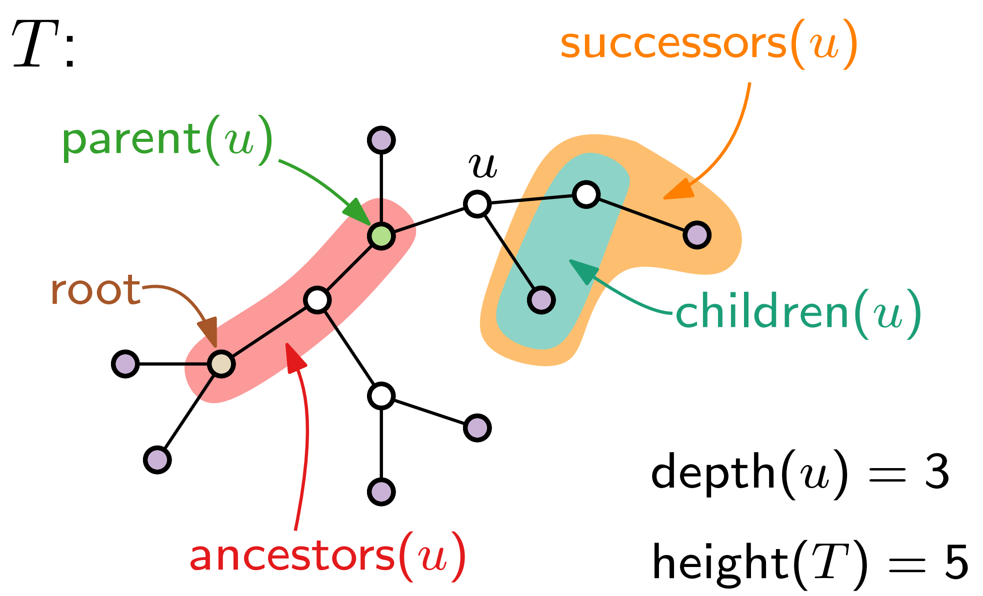
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

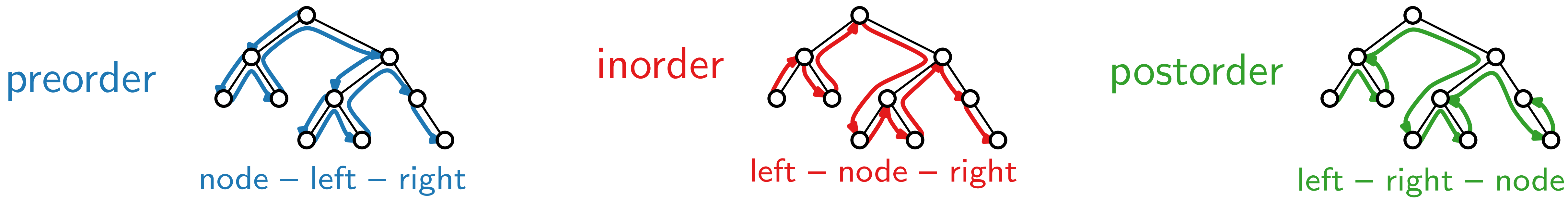
Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

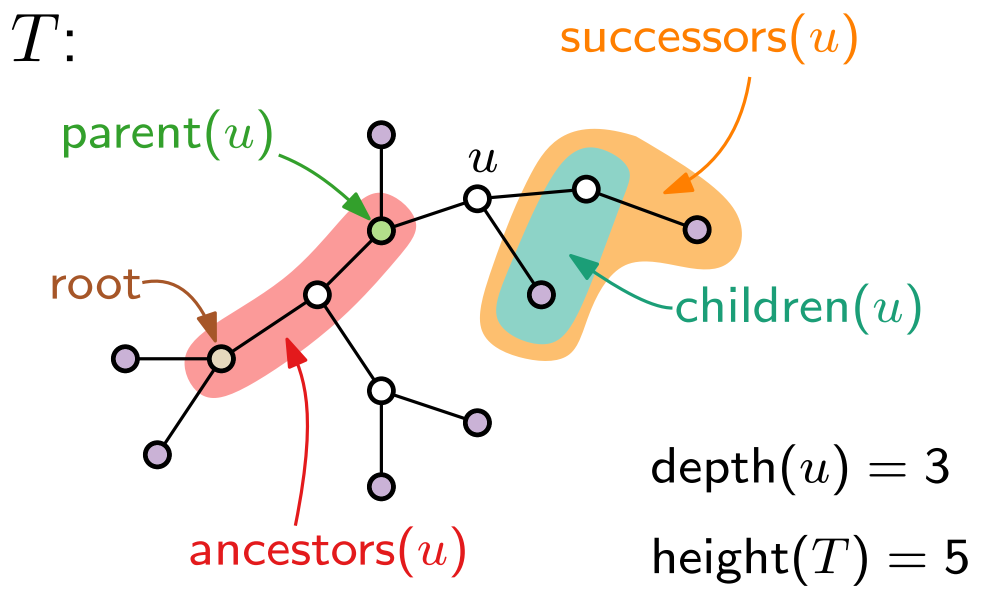
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

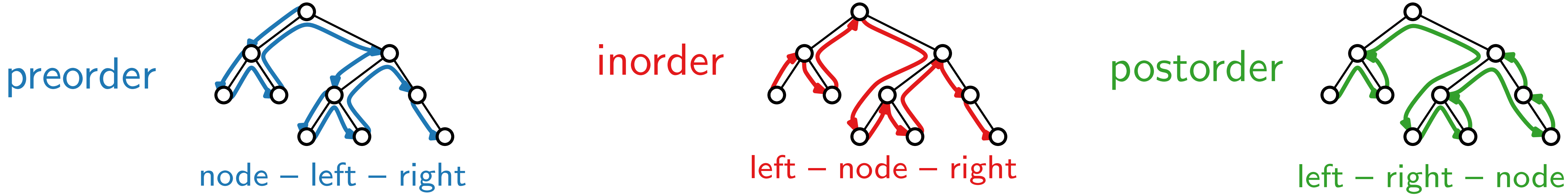
Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:



(Rooted) Trees

Leaf: vertex of degree 1

Rooted tree: tree with a designated **root**

Ancestor: vertex on the path to the root

Parent: neighbor on the path to the root

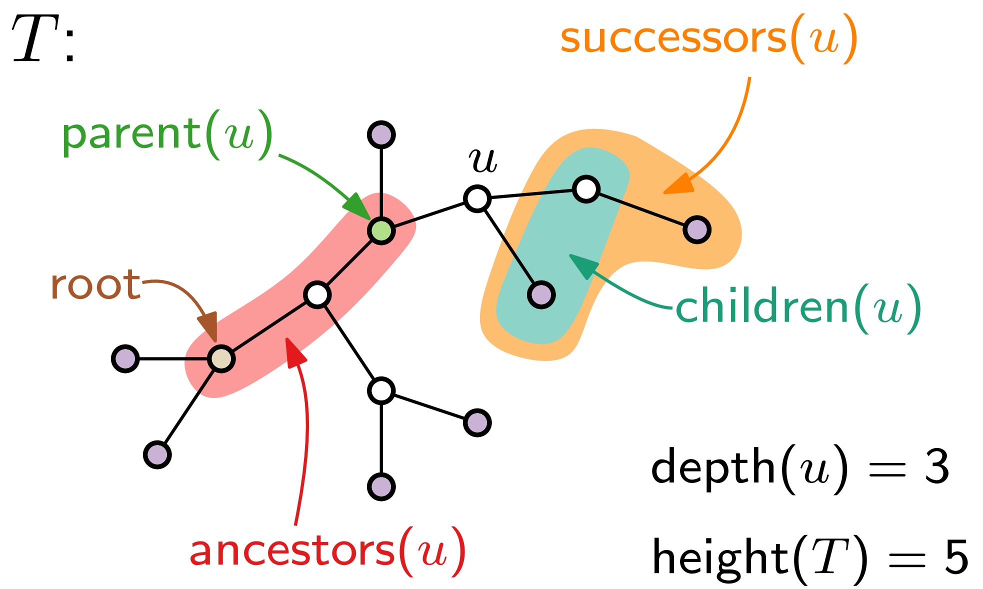
Successor: vertex on the path away from the root

Child: neighbor not on the path to the root

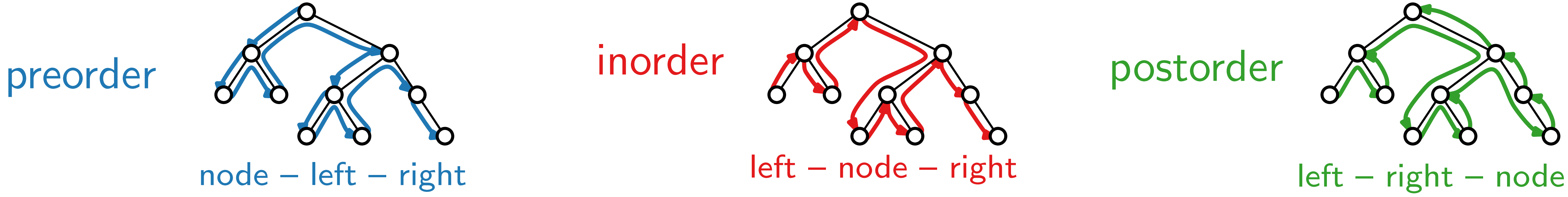
Depth: length of the path to the root

Height: maximum depth of a leaf

Binary Tree: at most two children per vertex (*left* and *right* child)



3 types of tree traversals:



First Grid Layout of Binary Trees

1. Choose y-coordinates:

First Grid Layout of Binary Trees

1. Choose y-coordinates:

2. Choose x-coordinates:

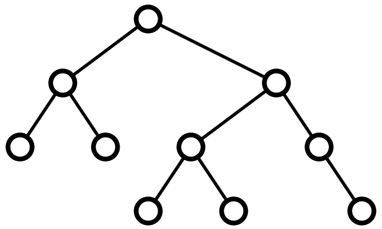
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \mathrm{depth}(u)$

2. Choose x-coordinates:

First Grid Layout of Binary Trees

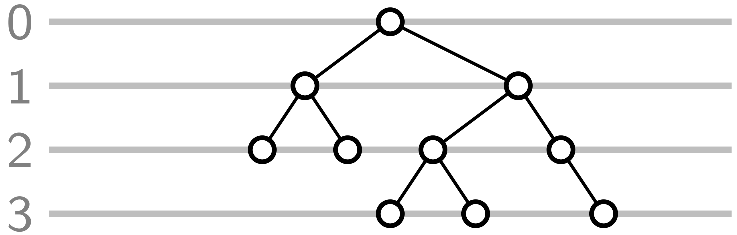
1. Choose y-coordinates: $y(u) = \text{depth}(u)$



2. Choose x-coordinates:

First Grid Layout of Binary Trees

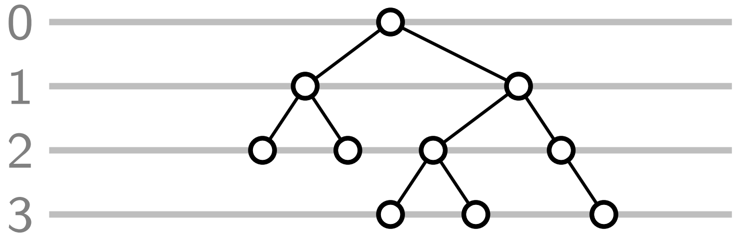
1. Choose y-coordinates: $y(u) = \mathrm{depth}(u)$



2. Choose x-coordinates:

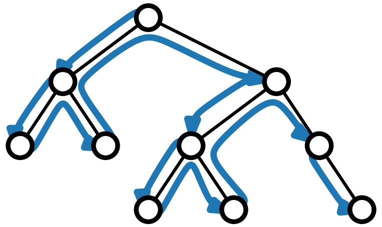
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

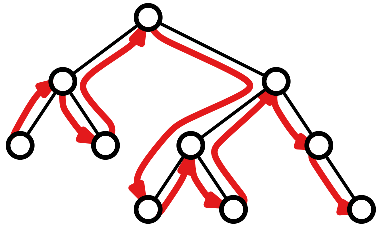


2. Choose x-coordinates:

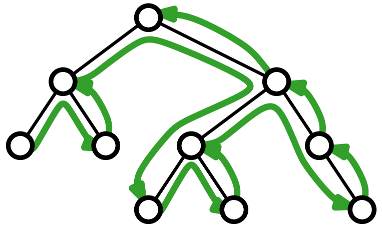
preorder



inorder

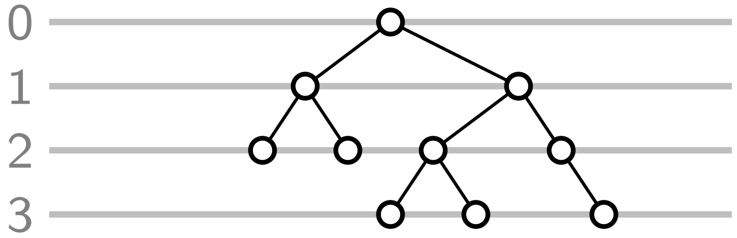


postorder

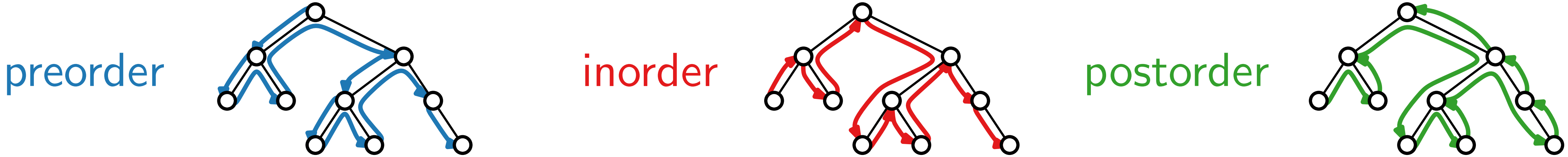


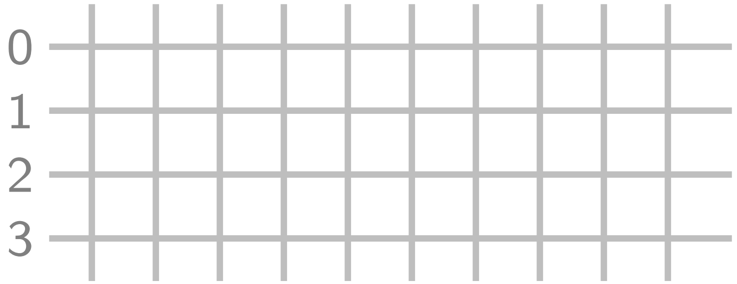
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$



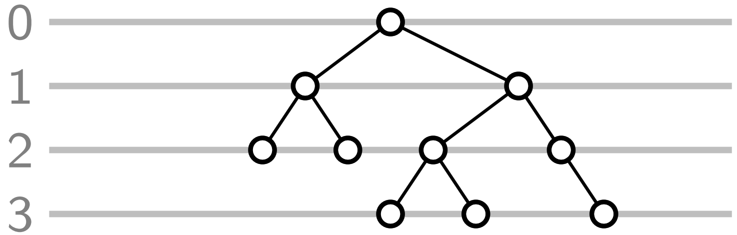
2. Choose x-coordinates:





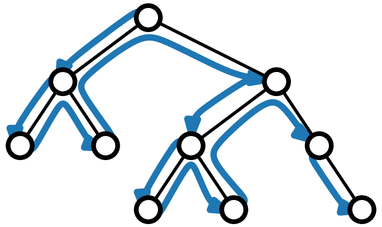
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

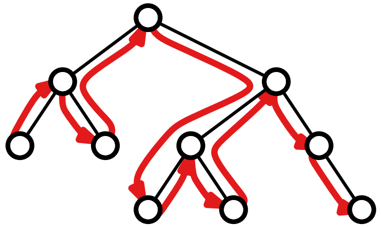


2. Choose x-coordinates:

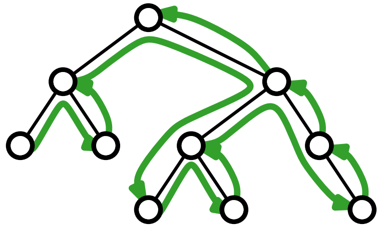
preorder

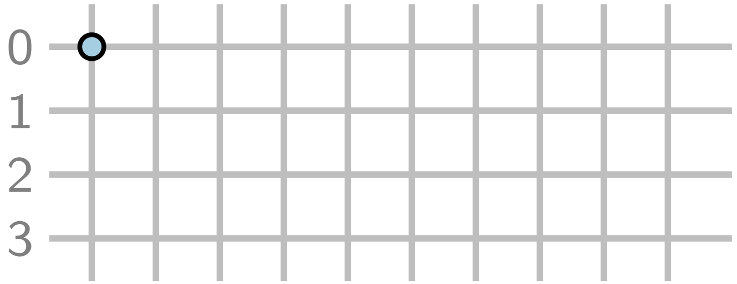


inorder



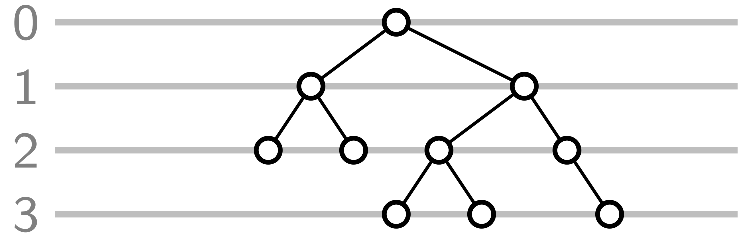
postorder



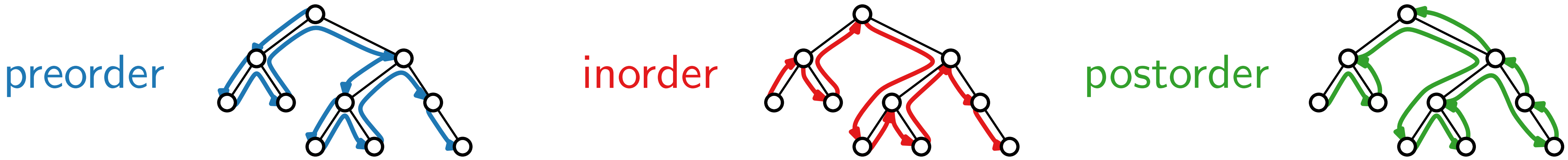


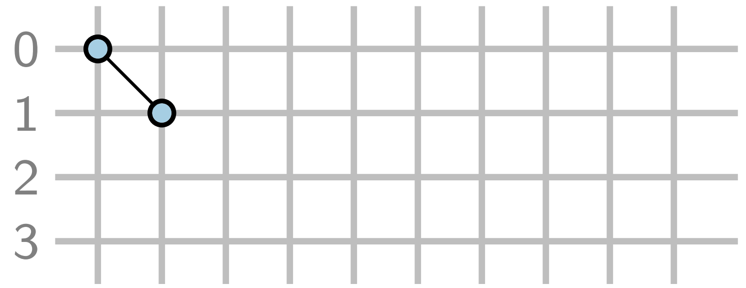
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$



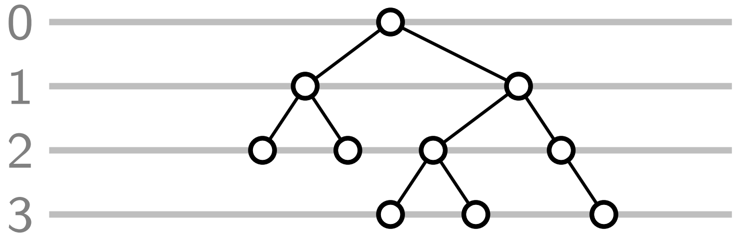
2. Choose x-coordinates:





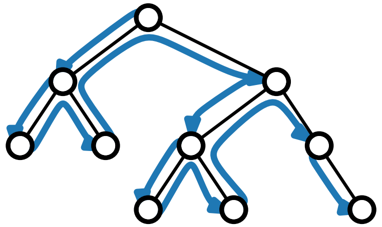
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

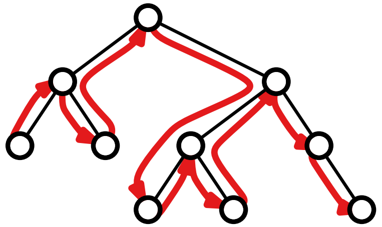


2. Choose x-coordinates:

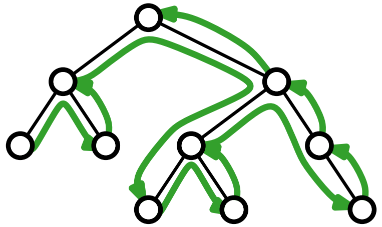
preorder

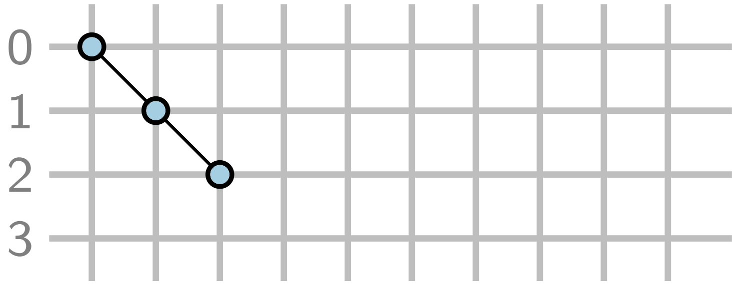


inorder



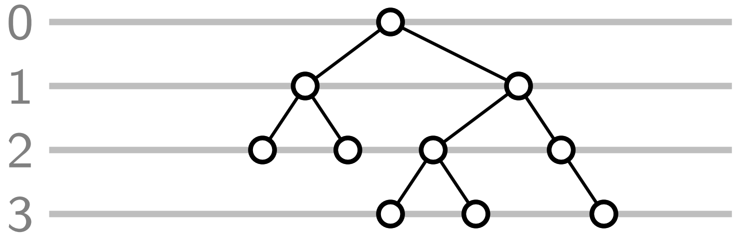
postorder





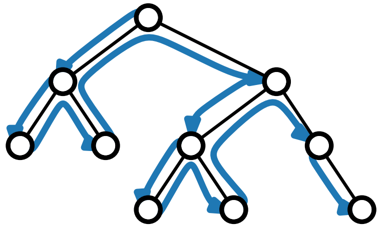
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

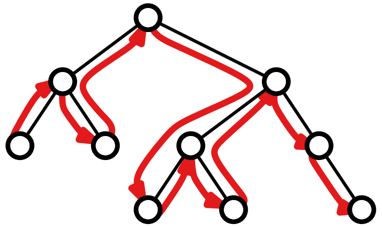


2. Choose x-coordinates:

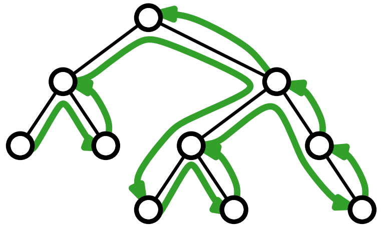
preorder

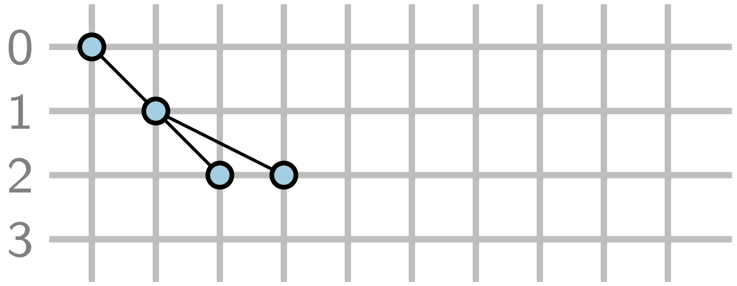


inorder



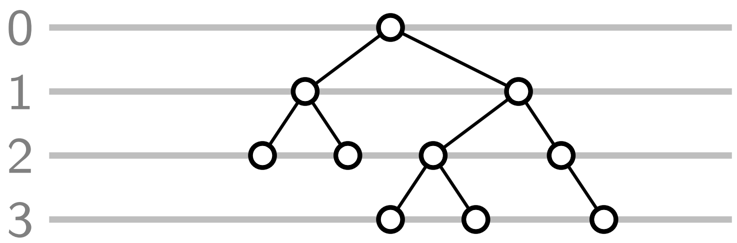
postorder





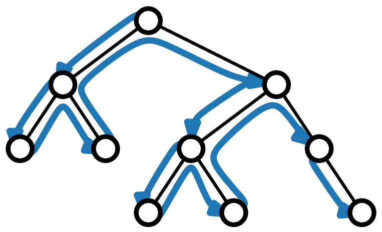
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

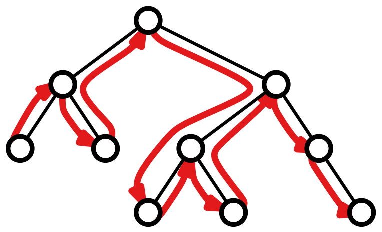


2. Choose x-coordinates:

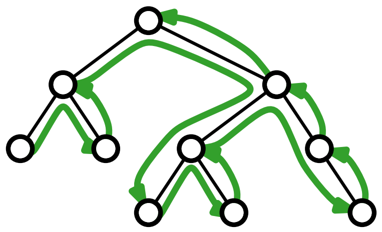
preorder

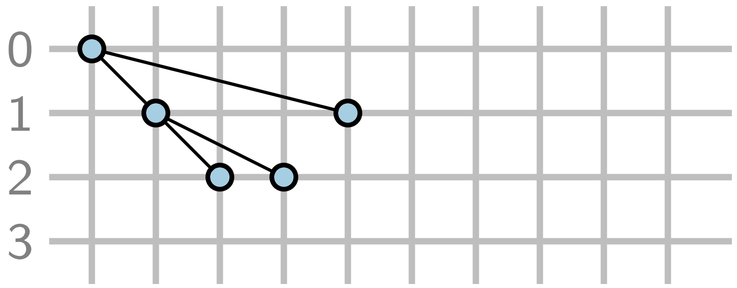


inorder



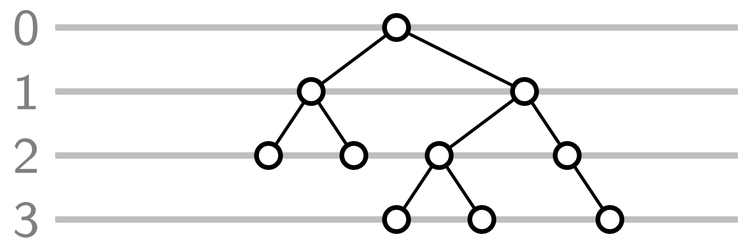
postorder



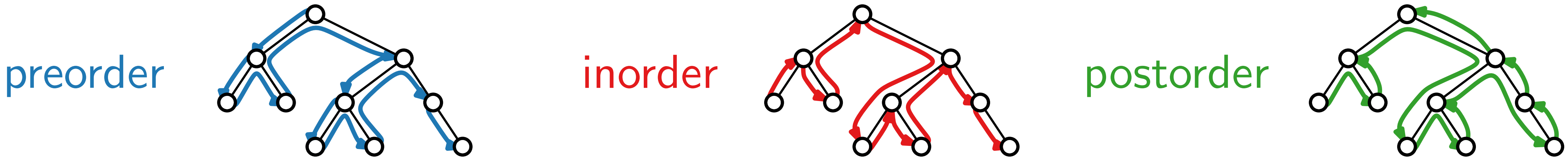


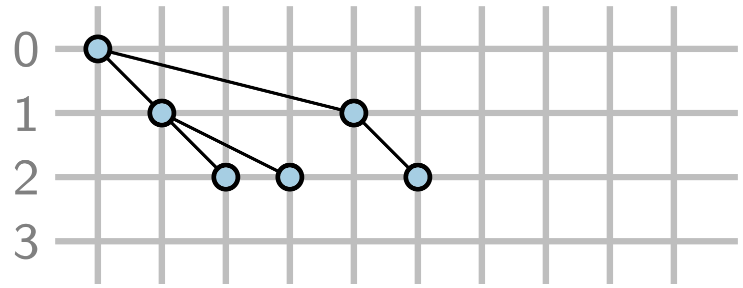
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$



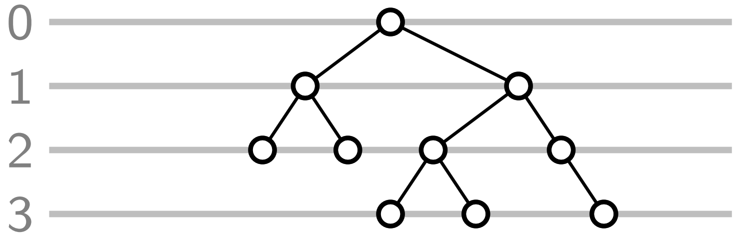
2. Choose x-coordinates:





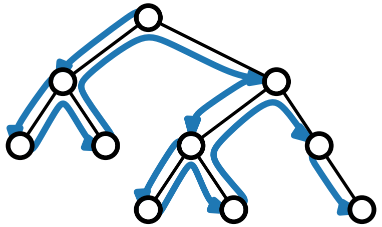
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \mathrm{depth}(u)$

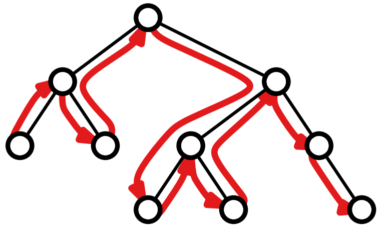


2. Choose x-coordinates:

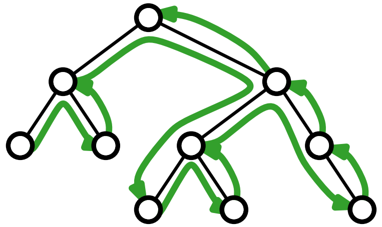
preorder

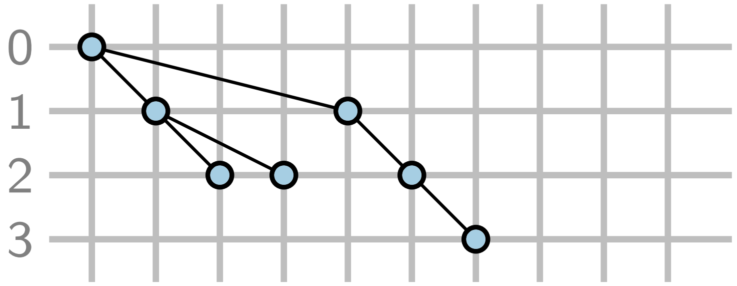


inorder



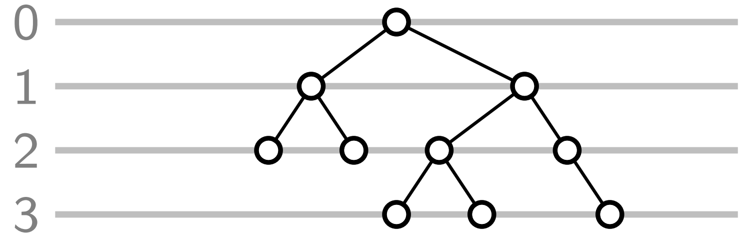
postorder





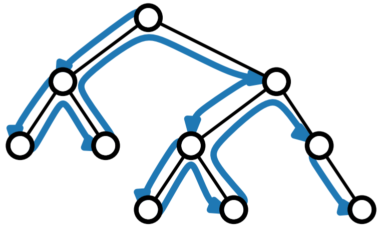
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \mathrm{depth}(u)$

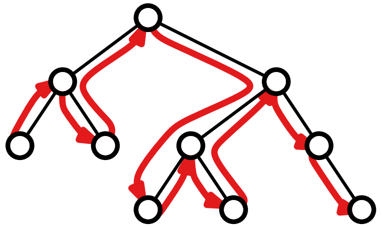


2. Choose x-coordinates:

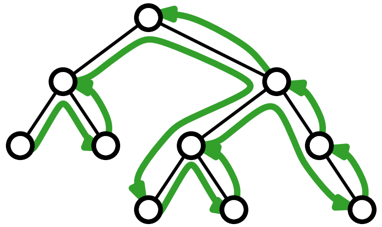
preorder

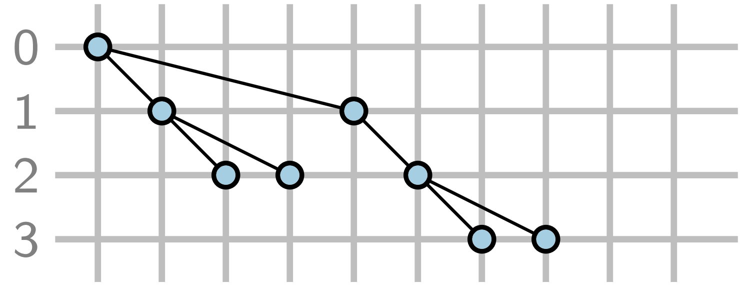


inorder



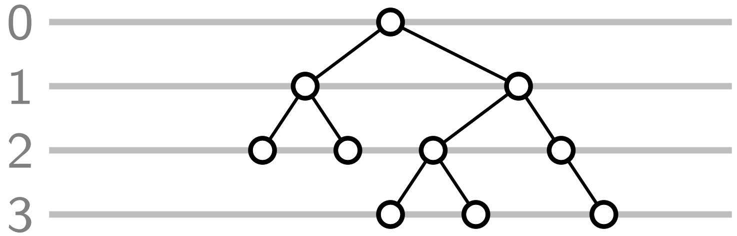
postorder





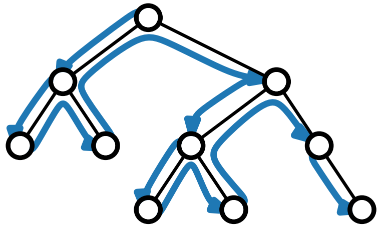
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

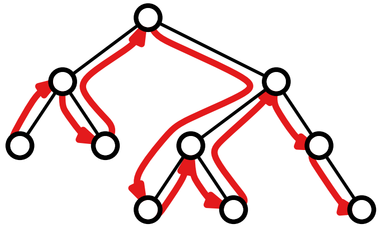


2. Choose x-coordinates:

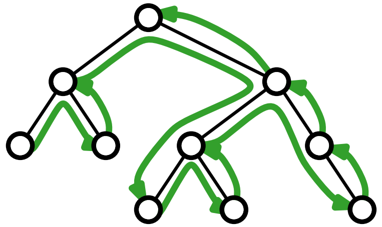
preorder

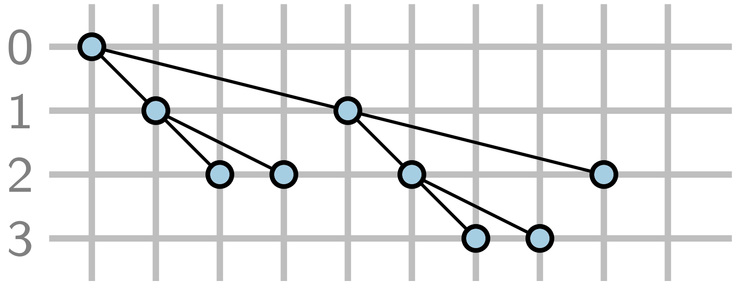


inorder



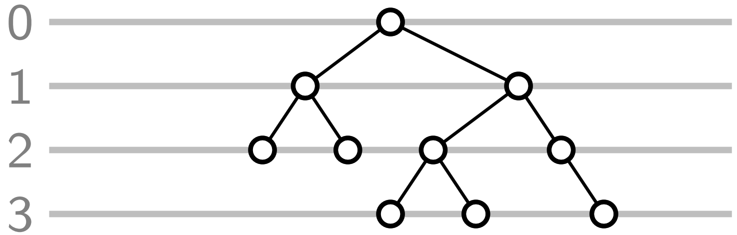
postorder





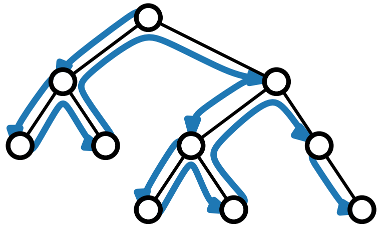
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

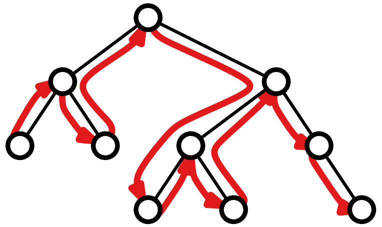


2. Choose x-coordinates:

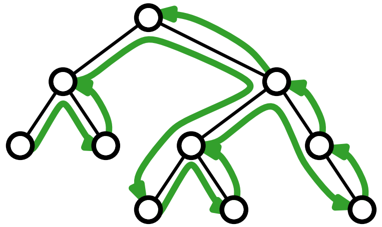
preorder

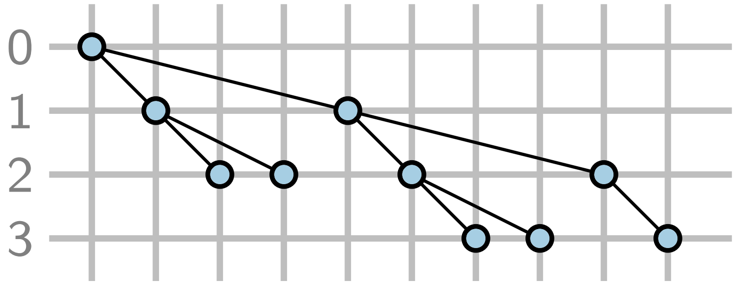


inorder



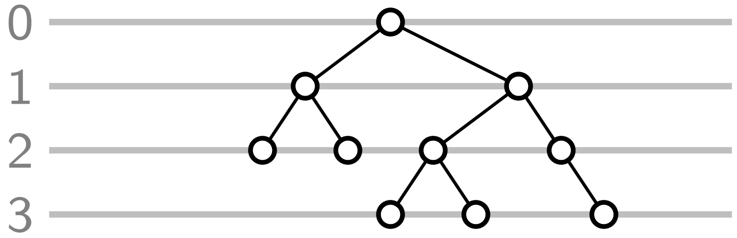
postorder





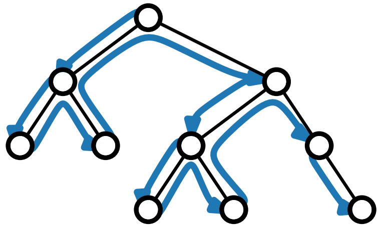
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

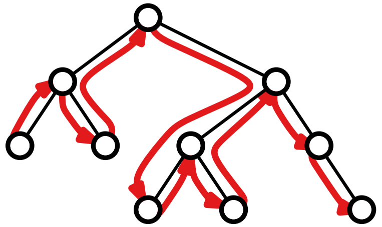


2. Choose x-coordinates:

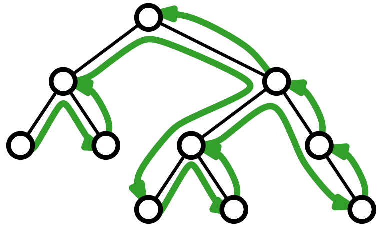
preorder

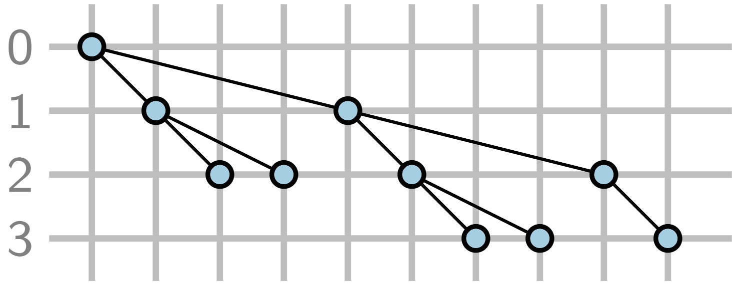


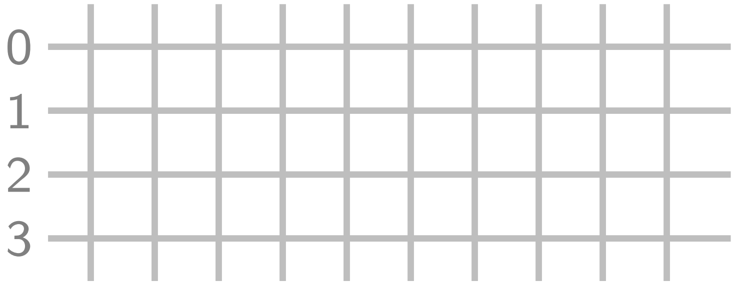
inorder



postorder

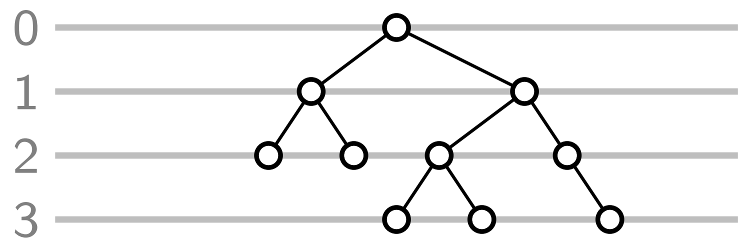




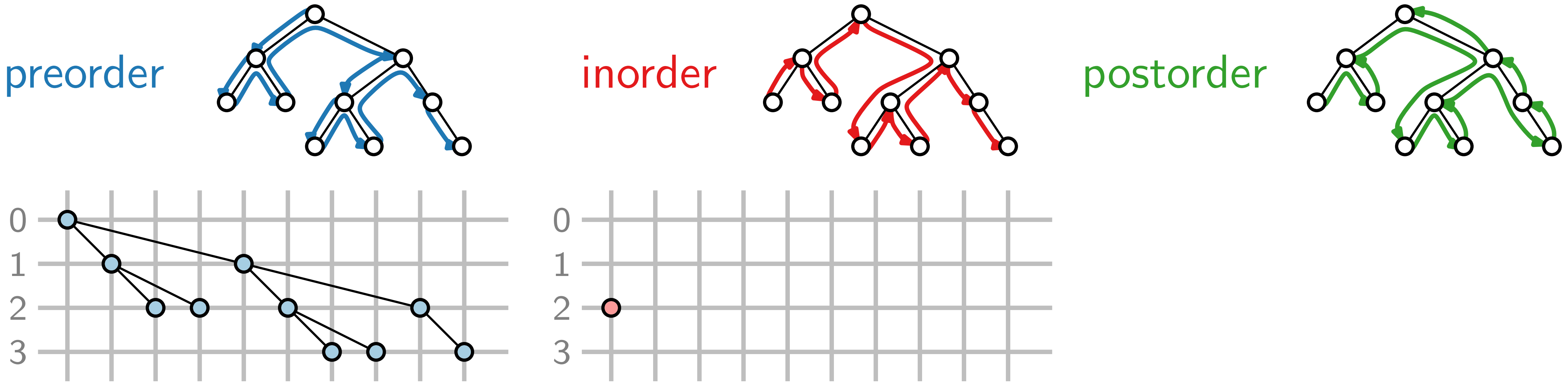


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

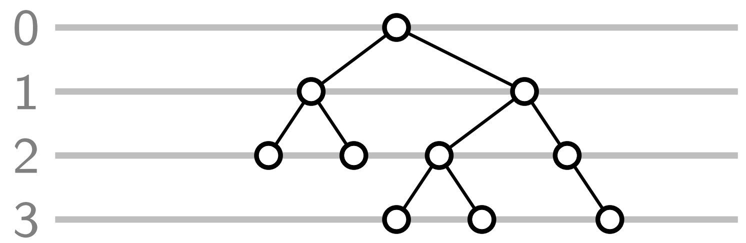


2. Choose x-coordinates:

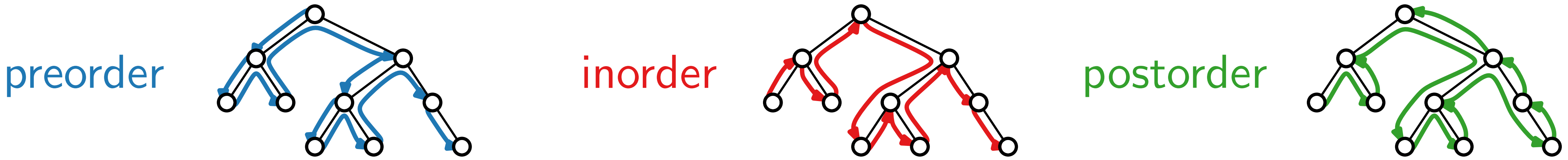


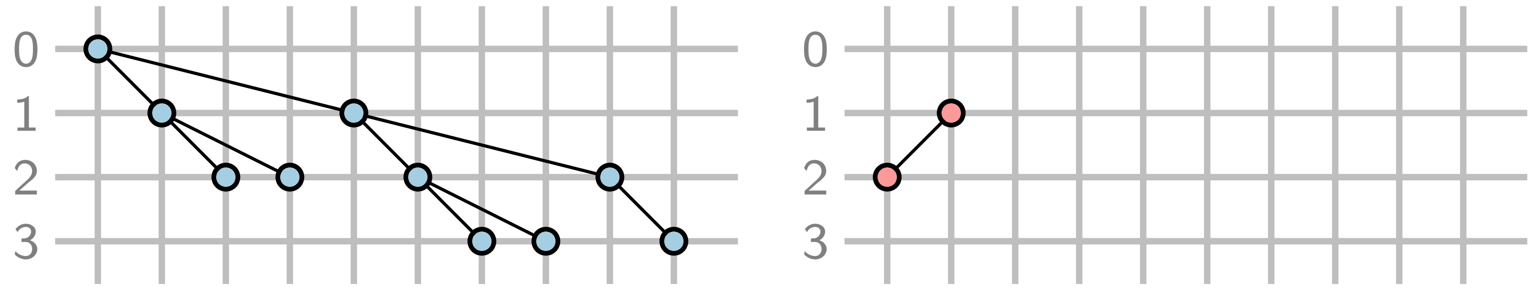
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$



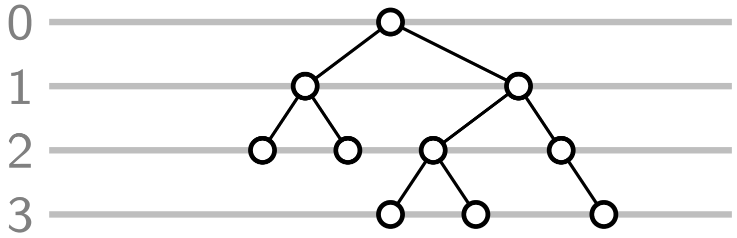
2. Choose x-coordinates:



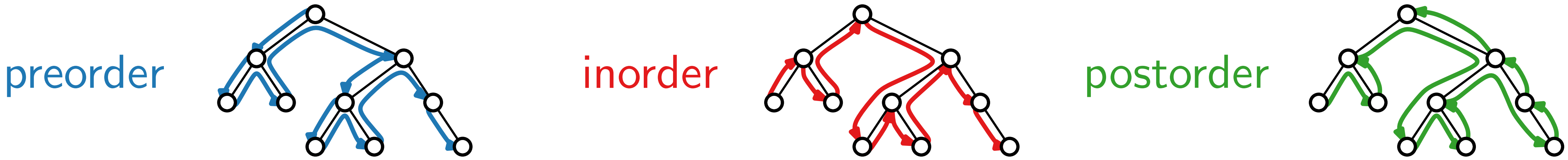


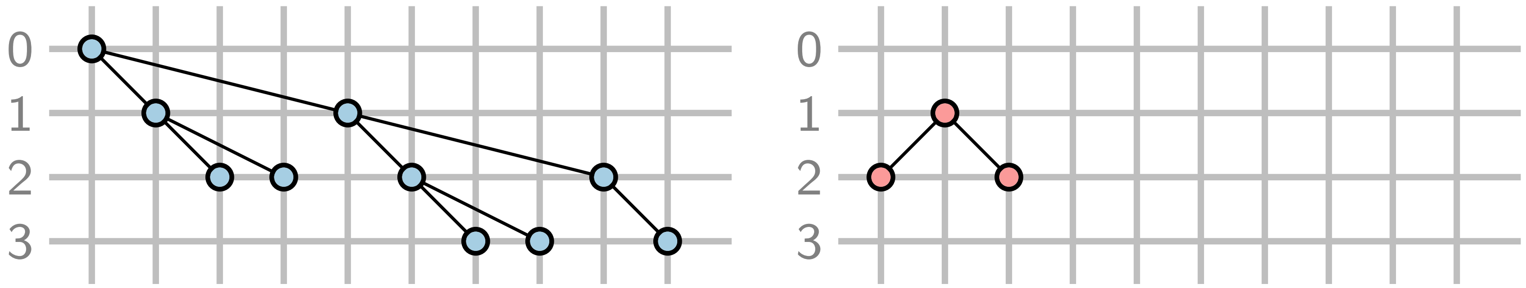
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$



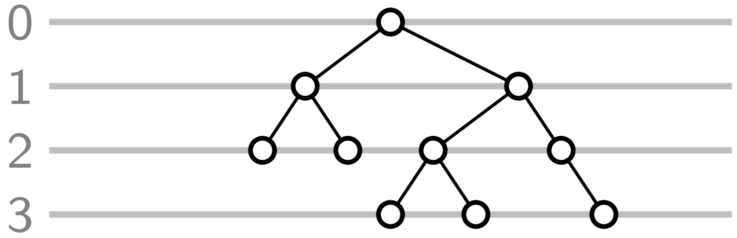
2. Choose x-coordinates:



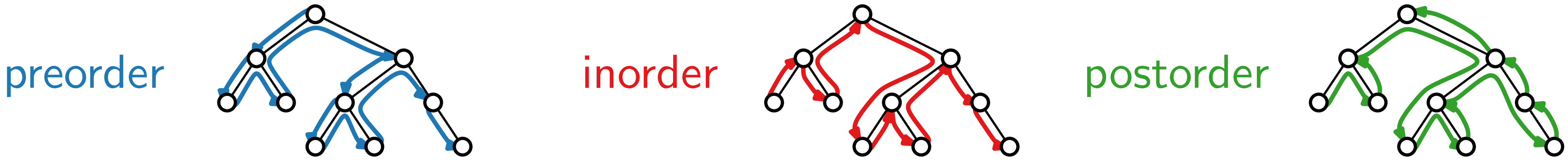


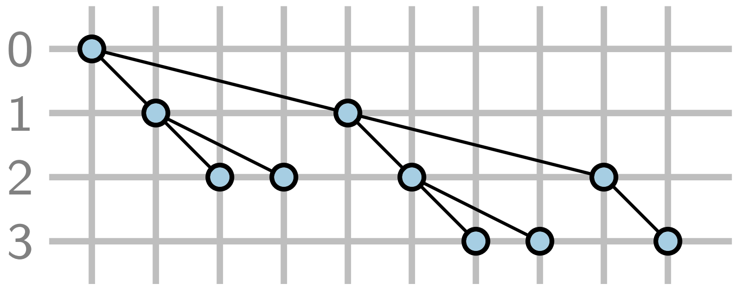
First Grid Layout of Binary Trees

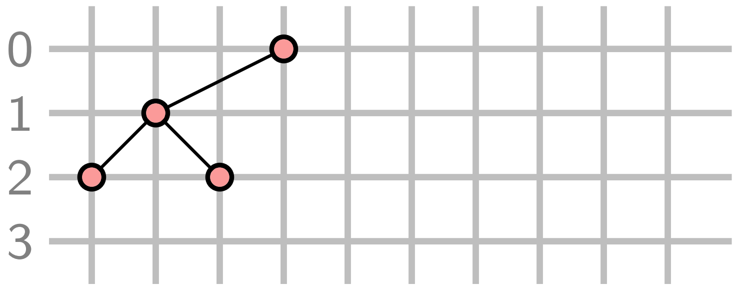
1. Choose y-coordinates: $y(u) = \text{depth}(u)$



2. Choose x-coordinates:

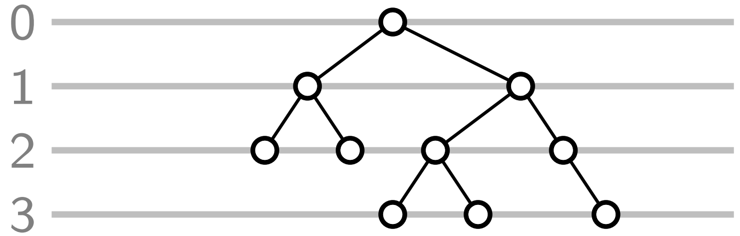




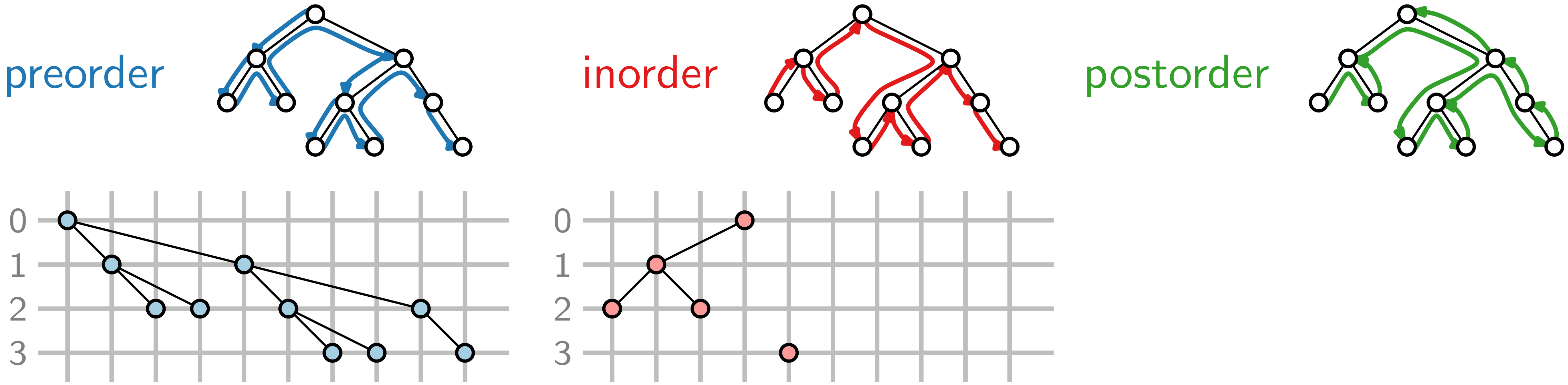


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

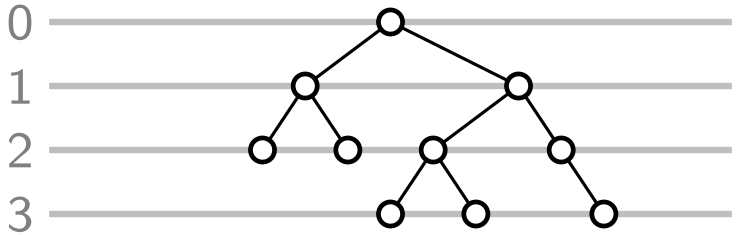


2. Choose x-coordinates:

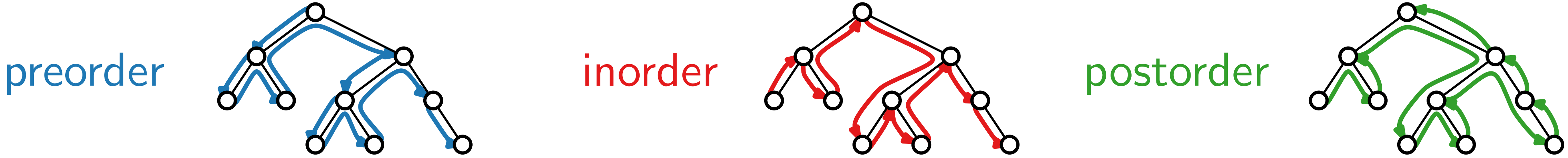


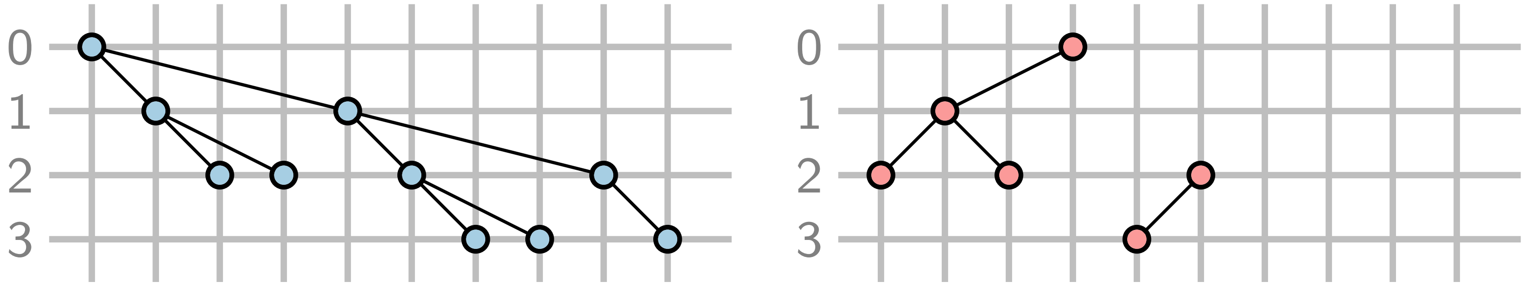
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$



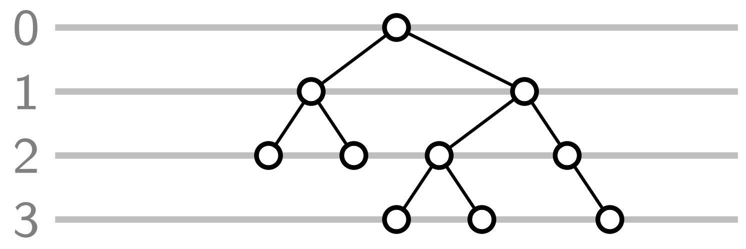
2. Choose x-coordinates:



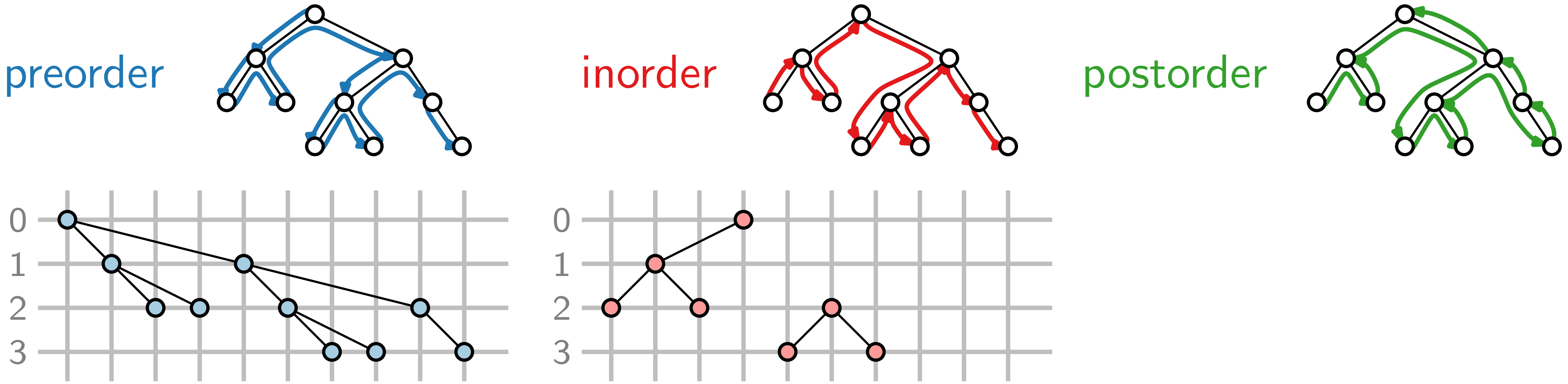


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

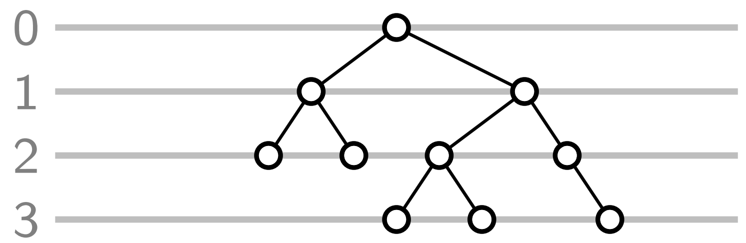


2. Choose x-coordinates:

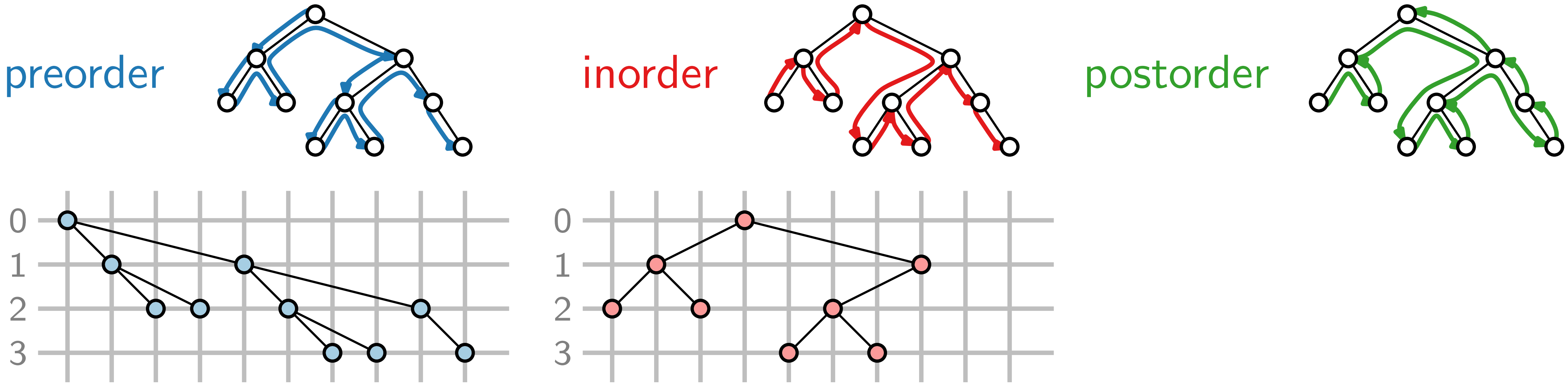


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

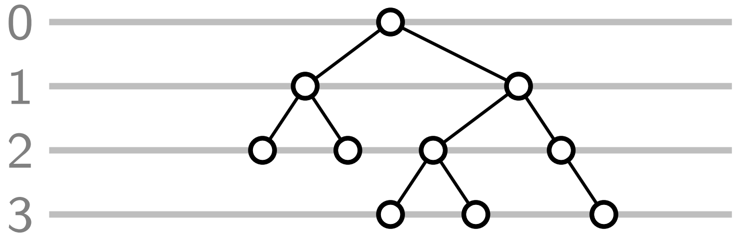


2. Choose x-coordinates:



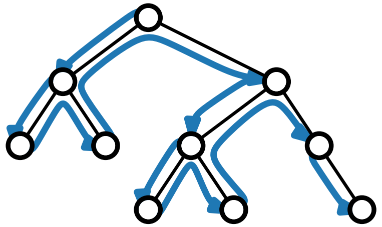
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

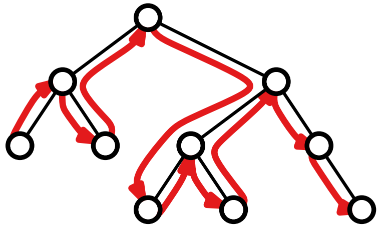


2. Choose x-coordinates:

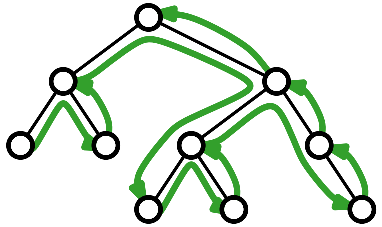
preorder

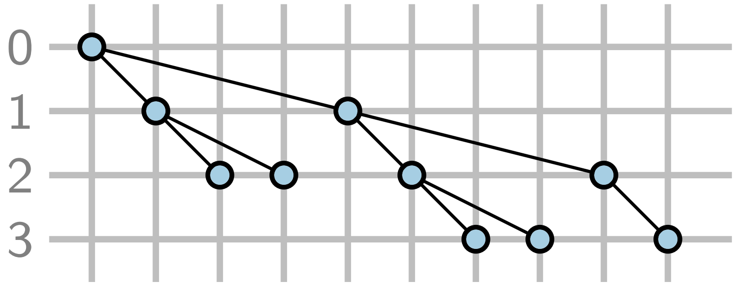


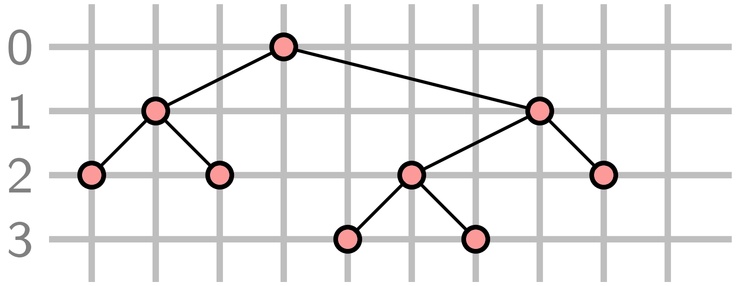
inorder



postorder

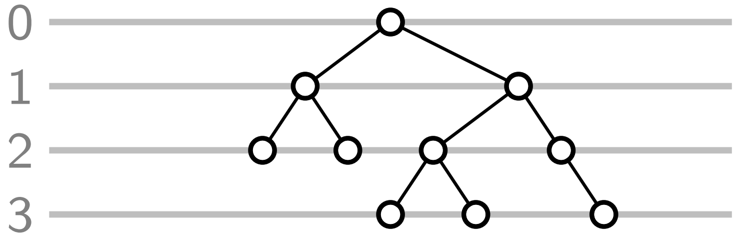






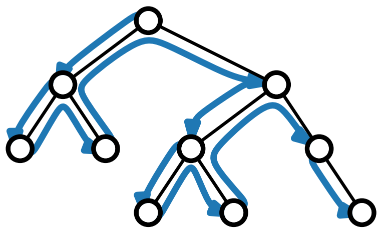
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

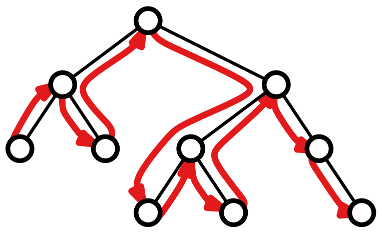


2. Choose x-coordinates:

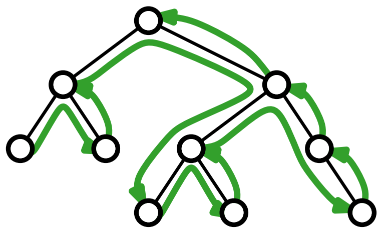
preorder

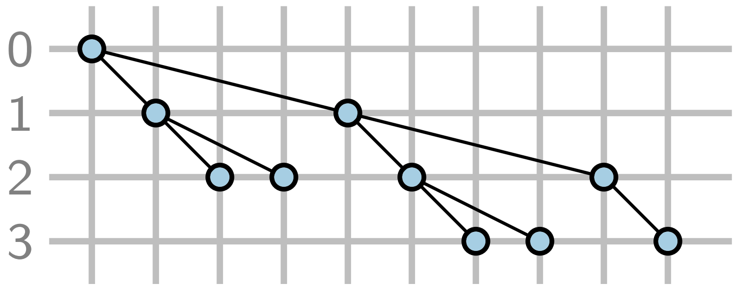


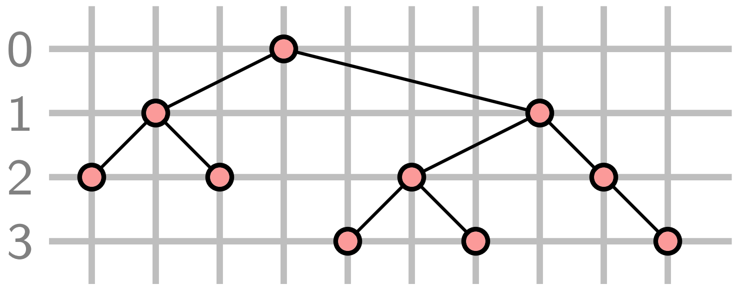
inorder



postorder

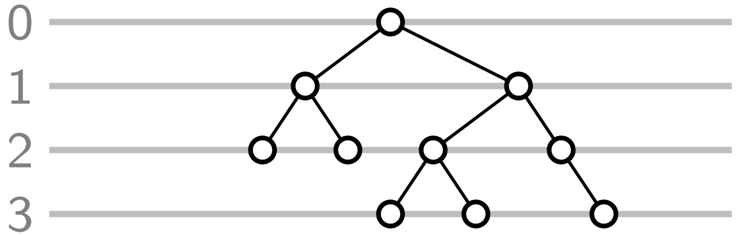






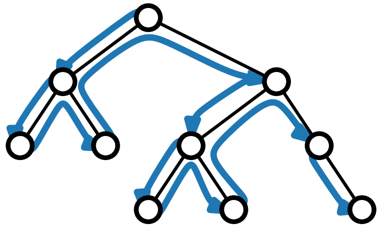
First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

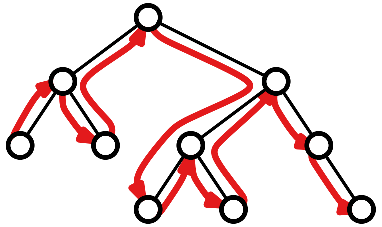


2. Choose x-coordinates:

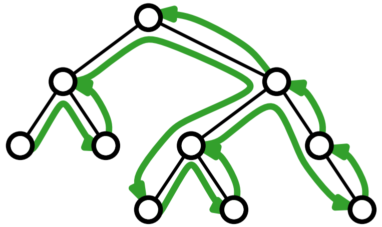
preorder

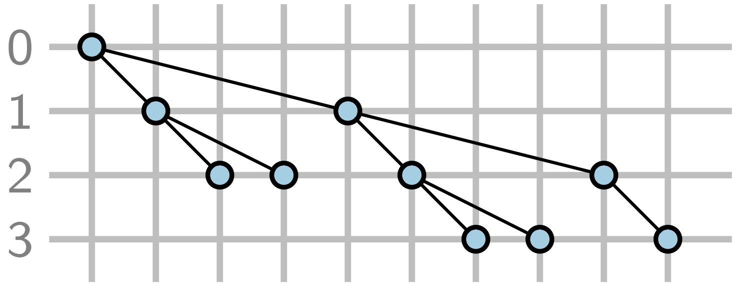


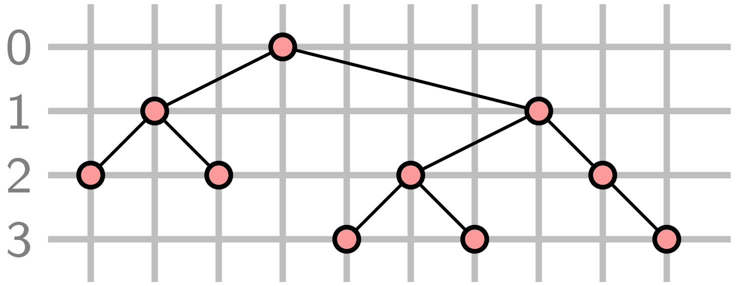
inorder

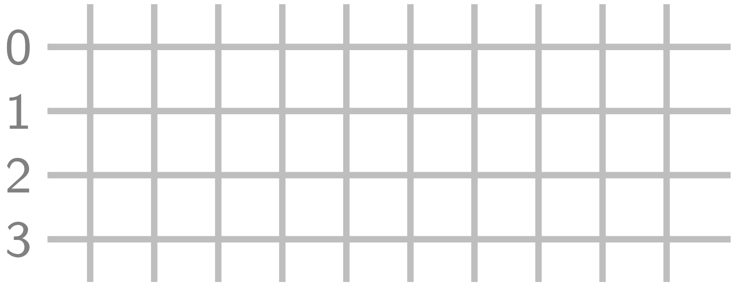


postorder



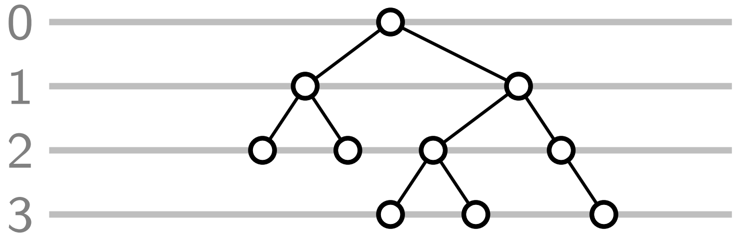






First Grid Layout of Binary Trees

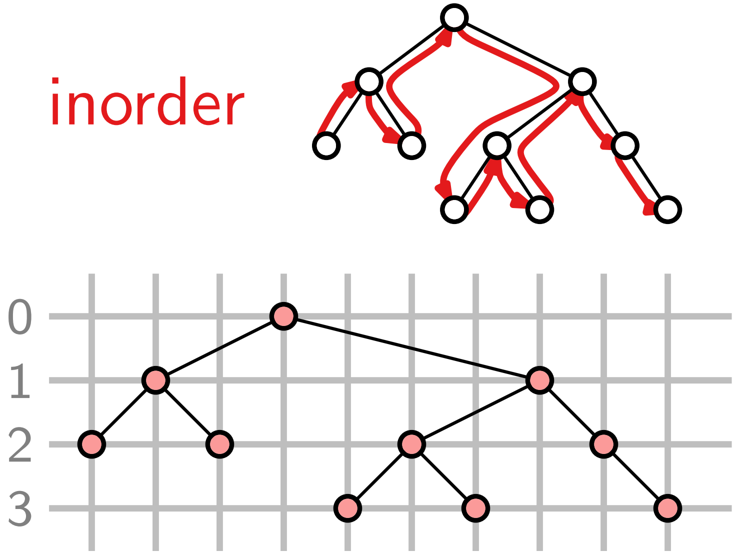
1. Choose y-coordinates: $y(u) = \mathrm{depth}(u)$



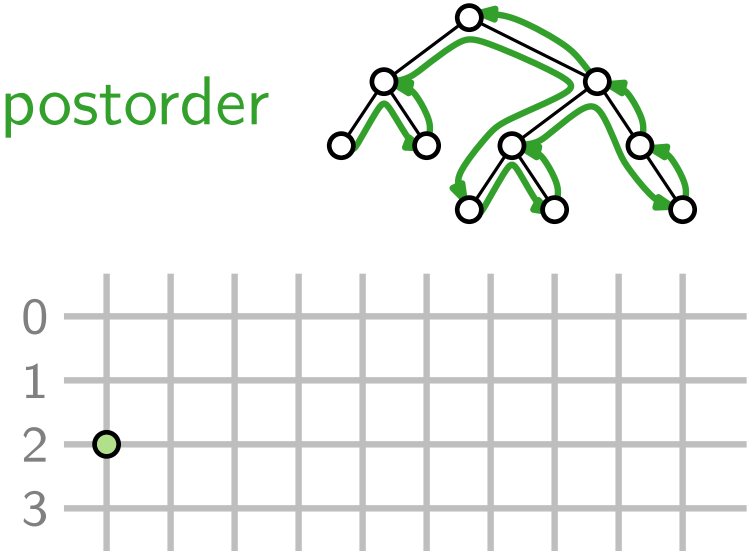
2. Choose x-coordinates:

preorder

inorder

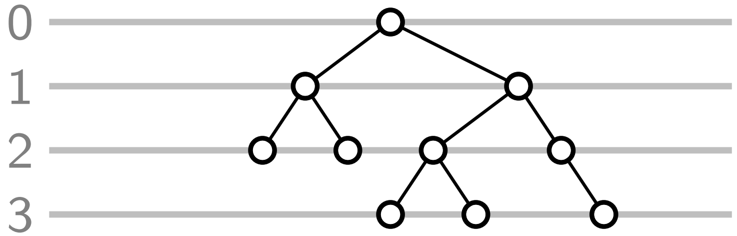


postorder

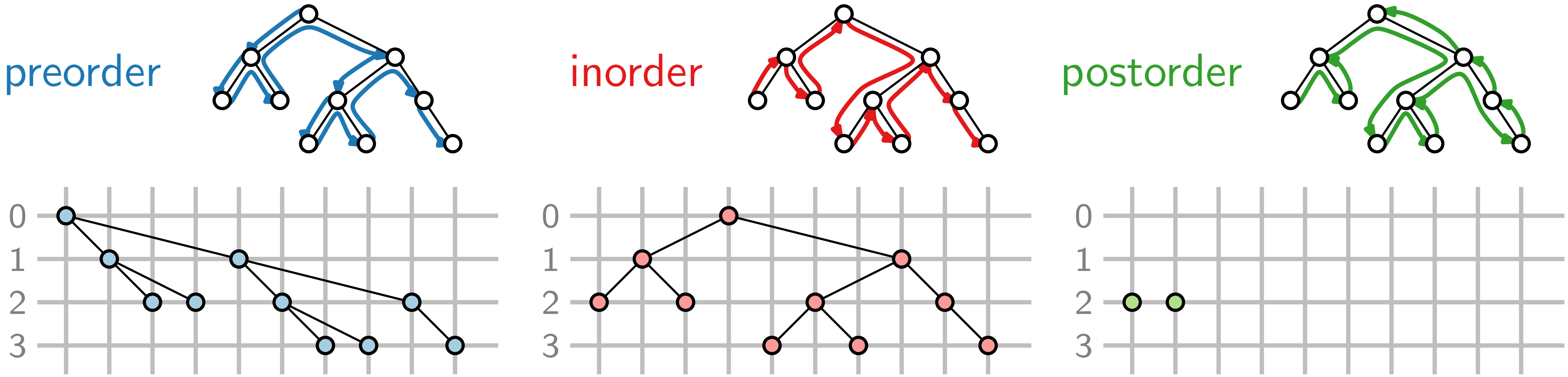


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

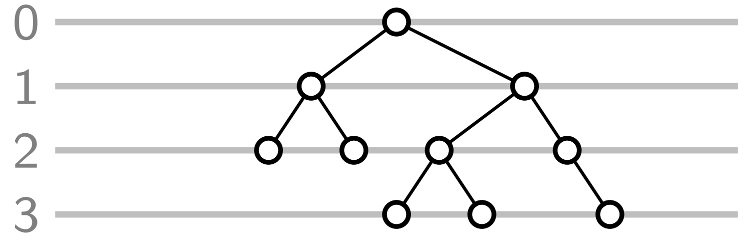


2. Choose x-coordinates:

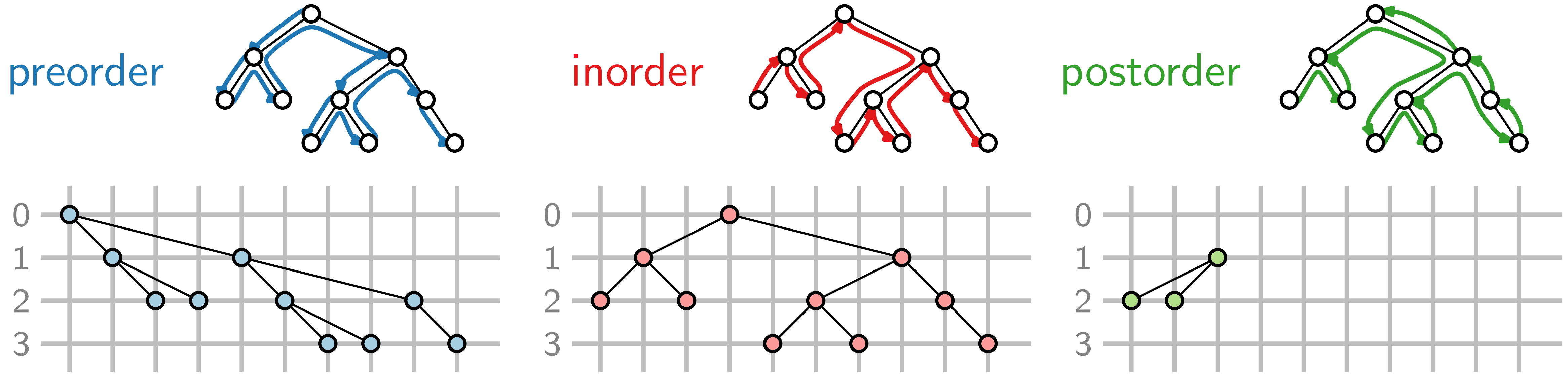


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

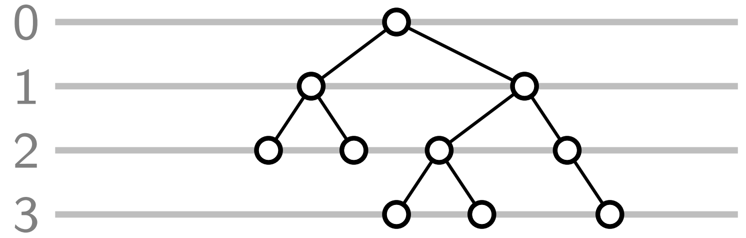


2. Choose x-coordinates:

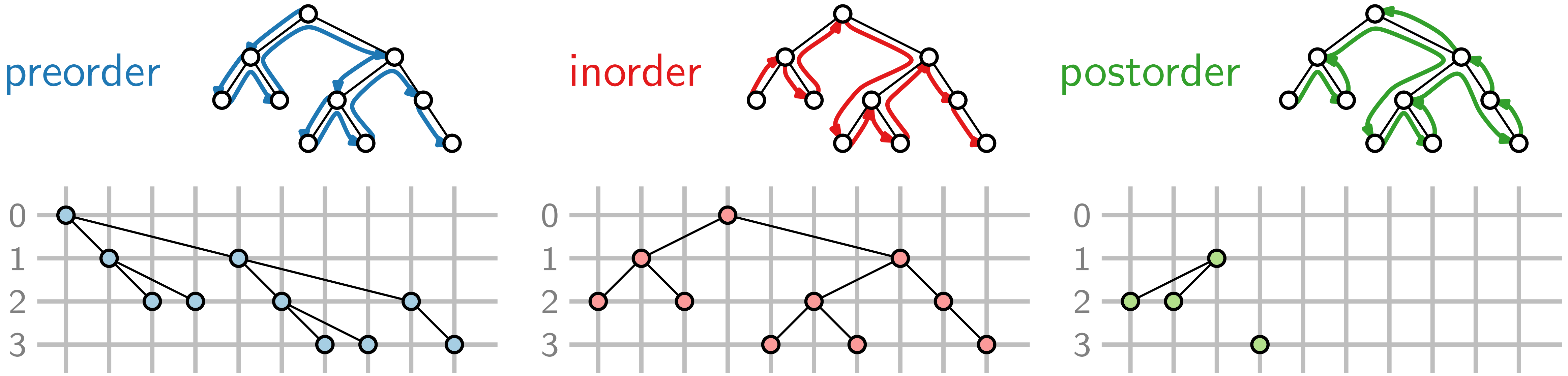


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

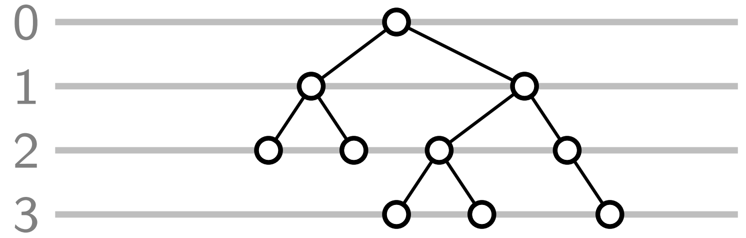


2. Choose x-coordinates:

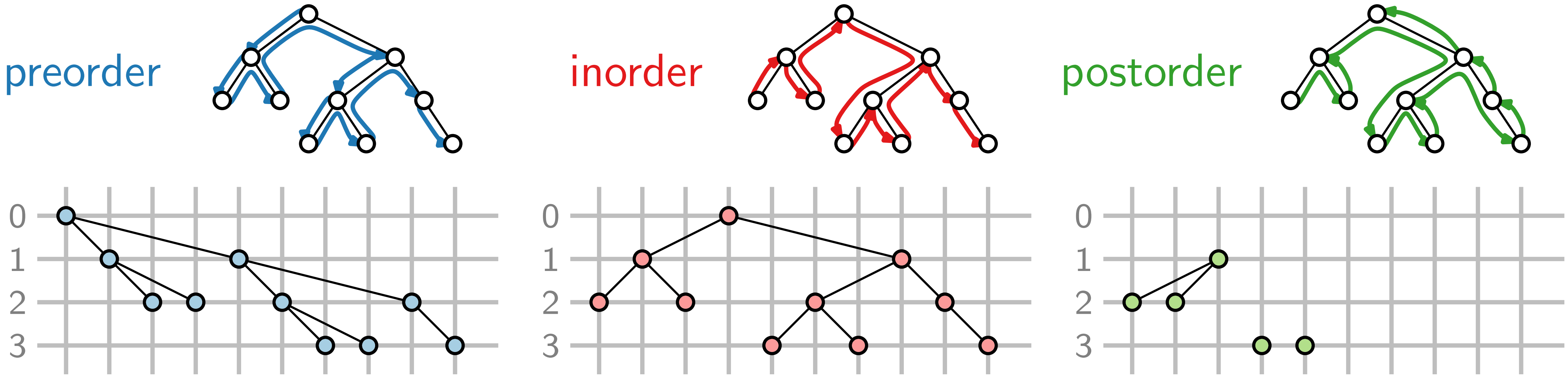


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

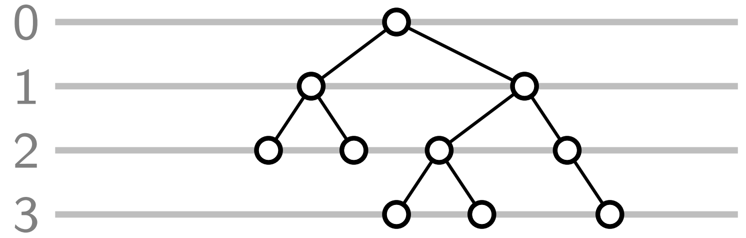


2. Choose x-coordinates:

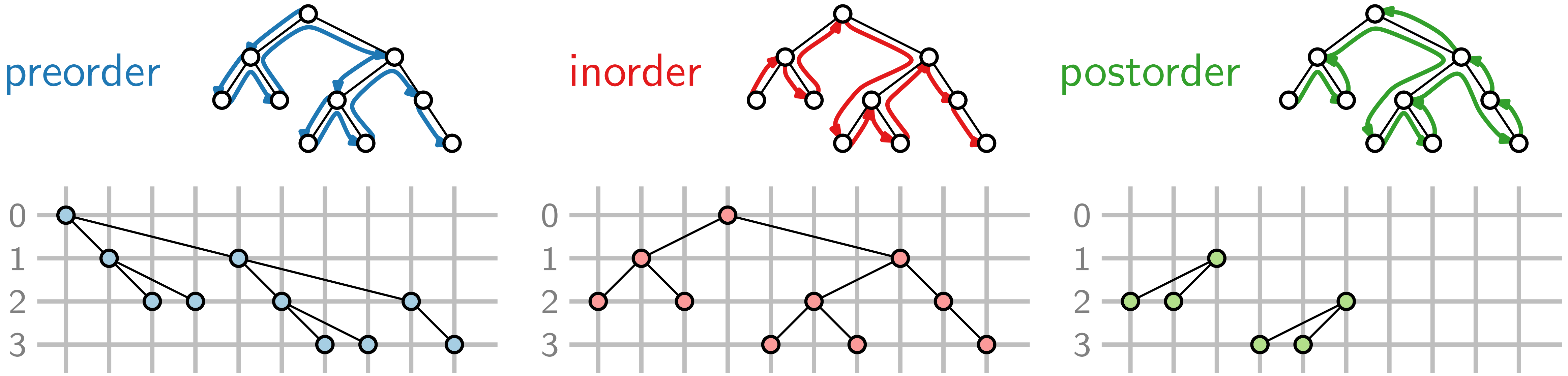


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

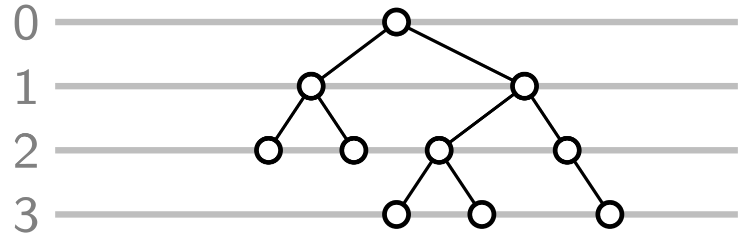


2. Choose x-coordinates:

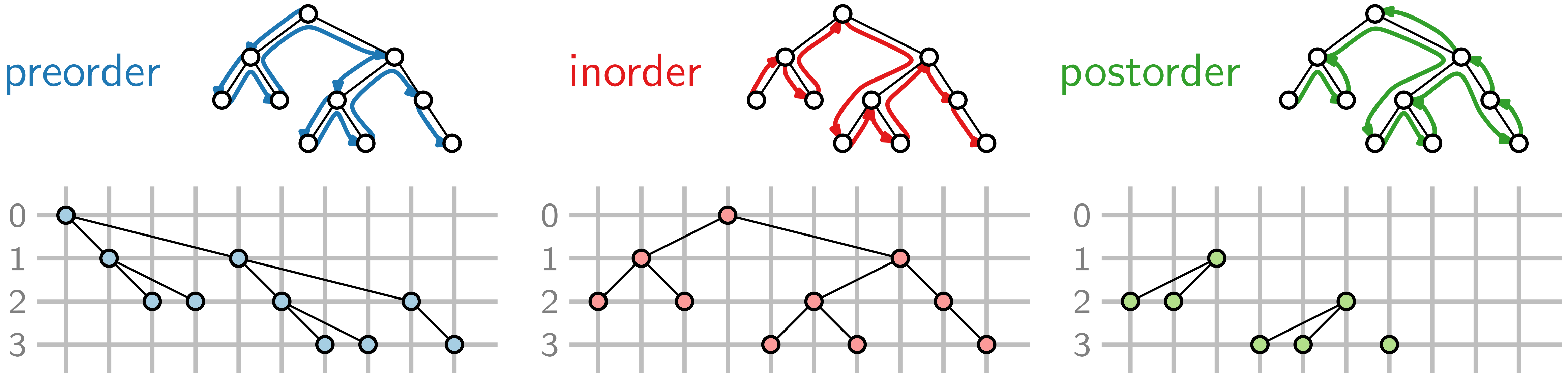


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

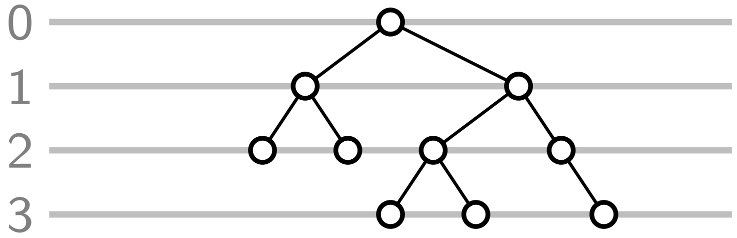


2. Choose x-coordinates:

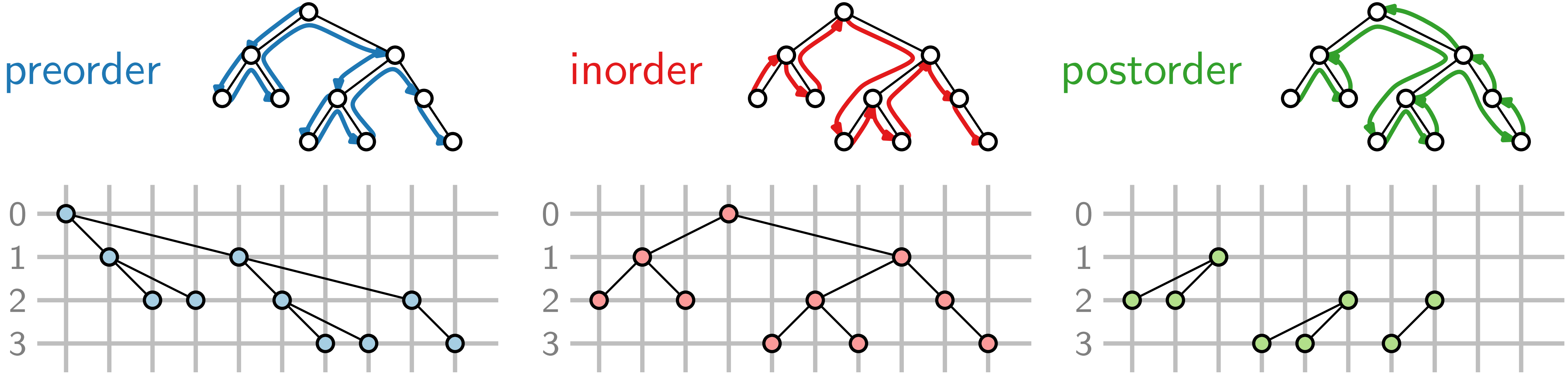


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \text{depth}(u)$

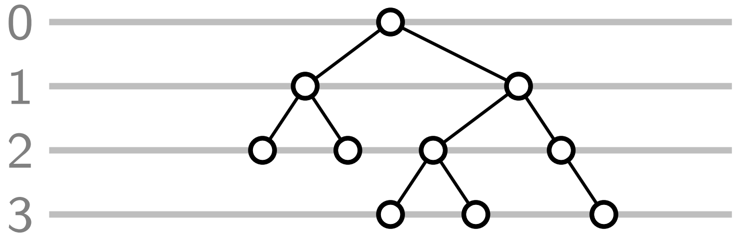


2. Choose x-coordinates:

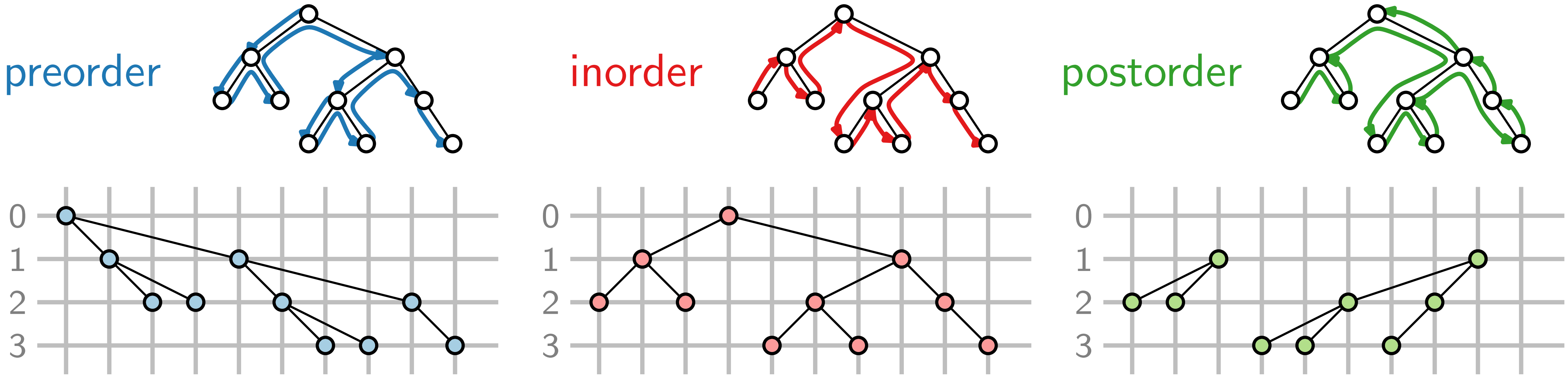


First Grid Layout of Binary Trees

1. Choose y-coordinates: $y(u) = \mathrm{depth}(u)$

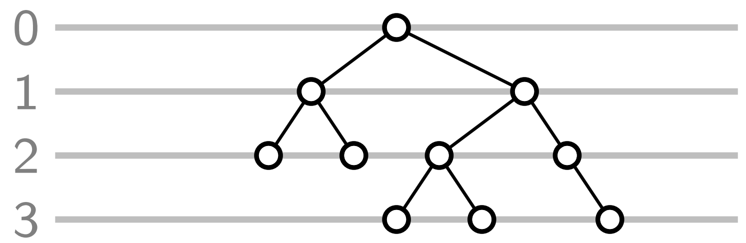


2. Choose x-coordinates:

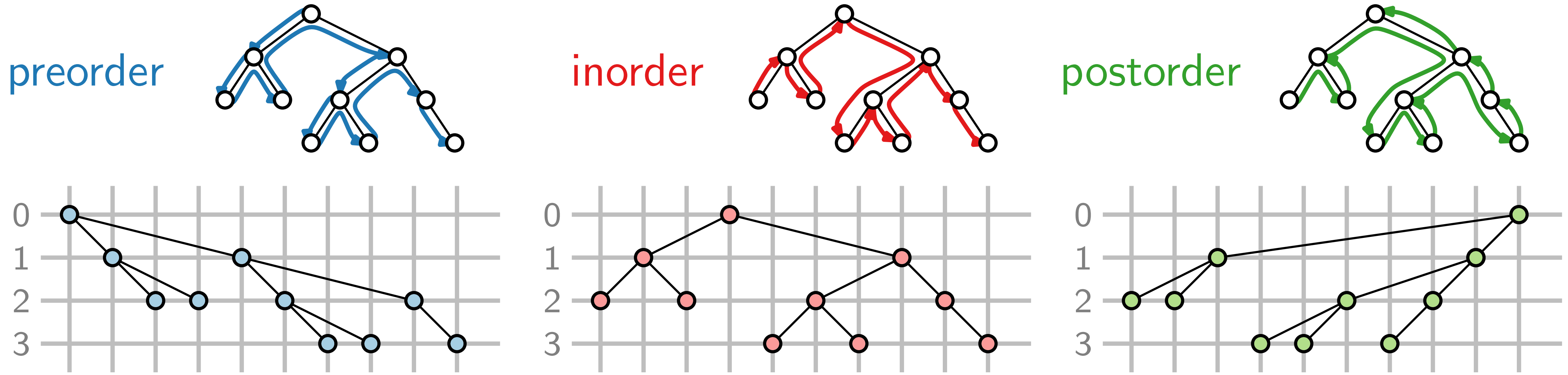


First Grid Layout of Binary Trees

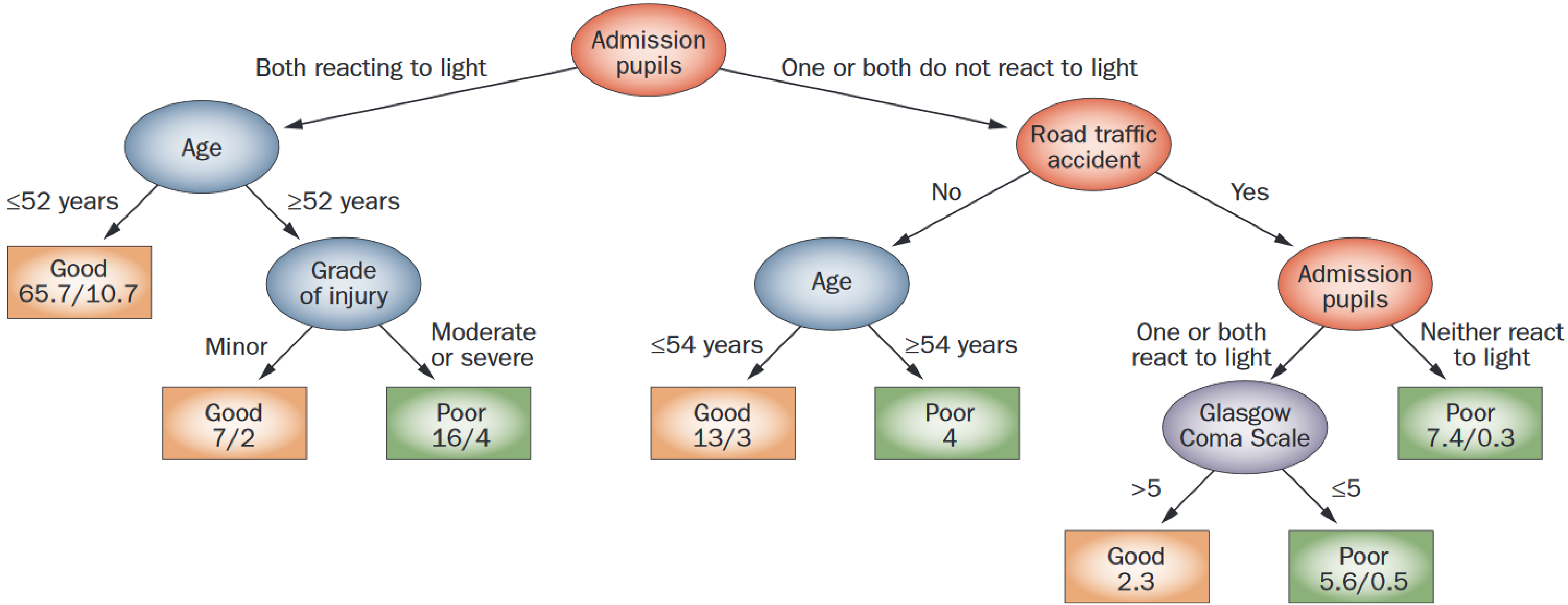
1. Choose y-coordinates: $y(u) = \text{depth}(u)$



2. Choose x-coordinates:



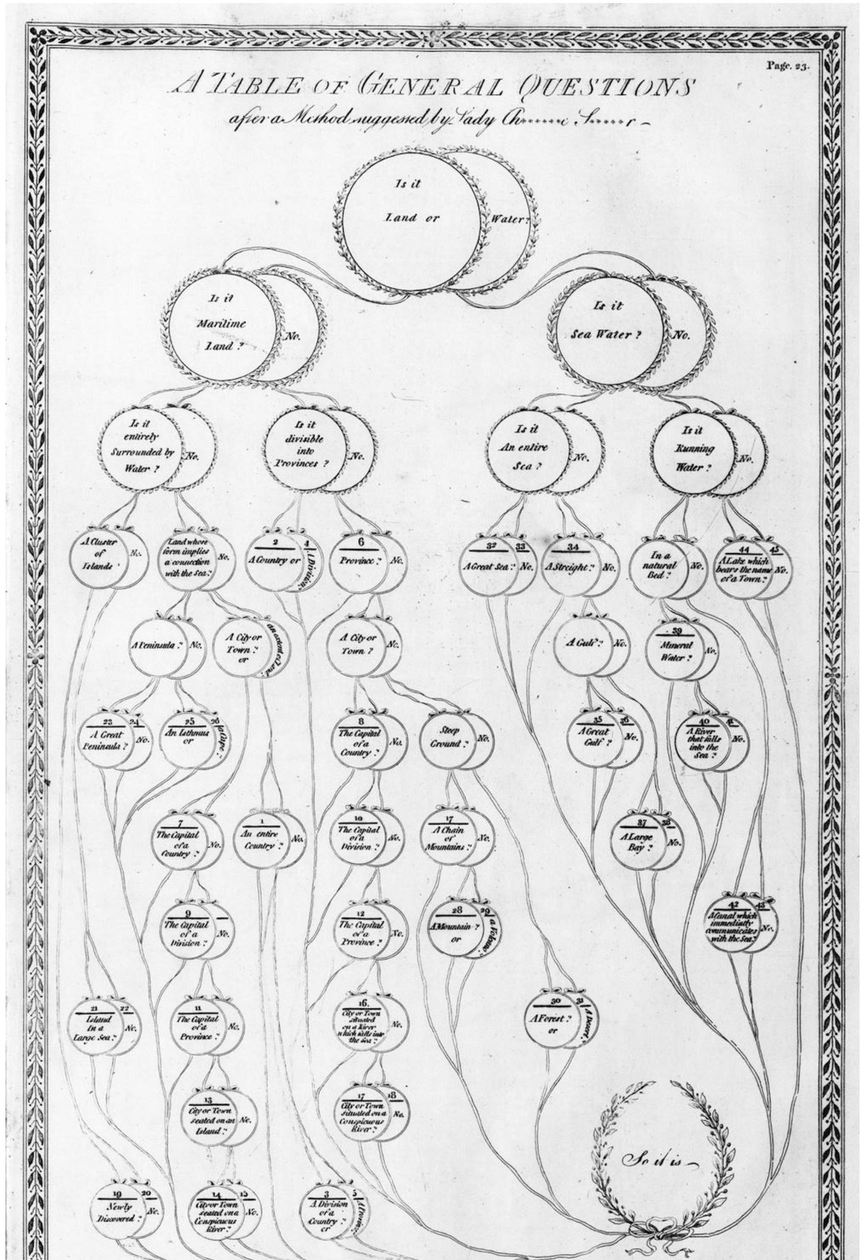
Layered Drawings – Applications



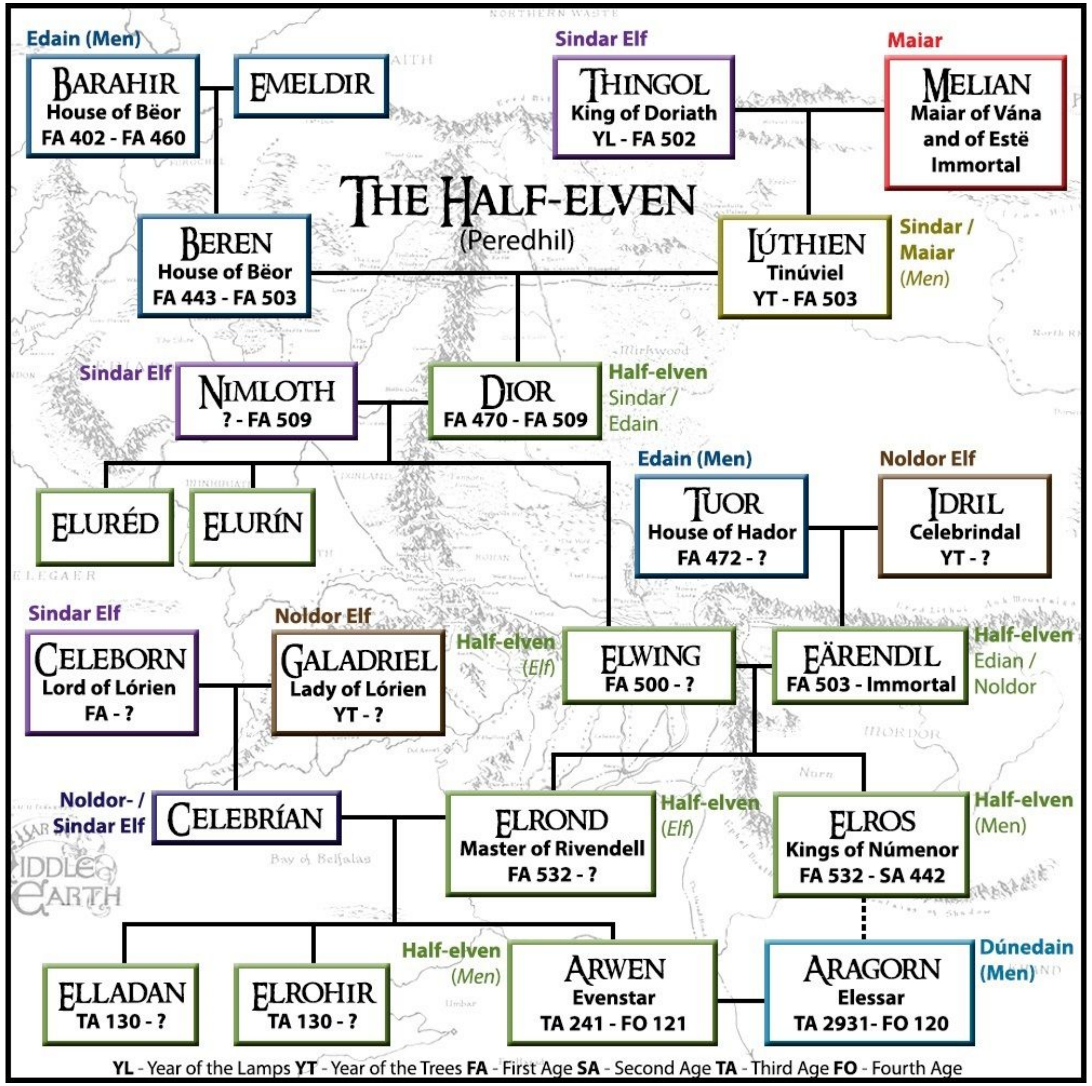
Decision tree for outcome prediction after traumatic brain injury

Source: Nature Reviews Neurology

Layered Drawings – Applications



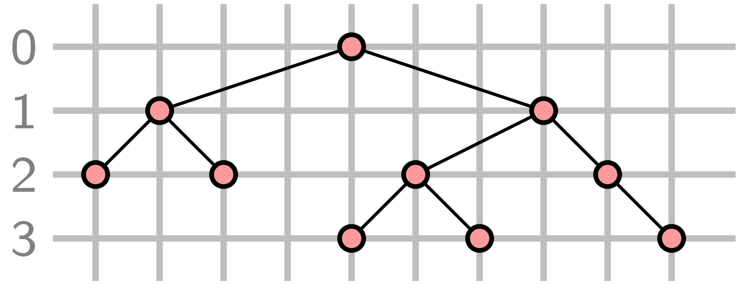
Aloïsius Gaultier 1821

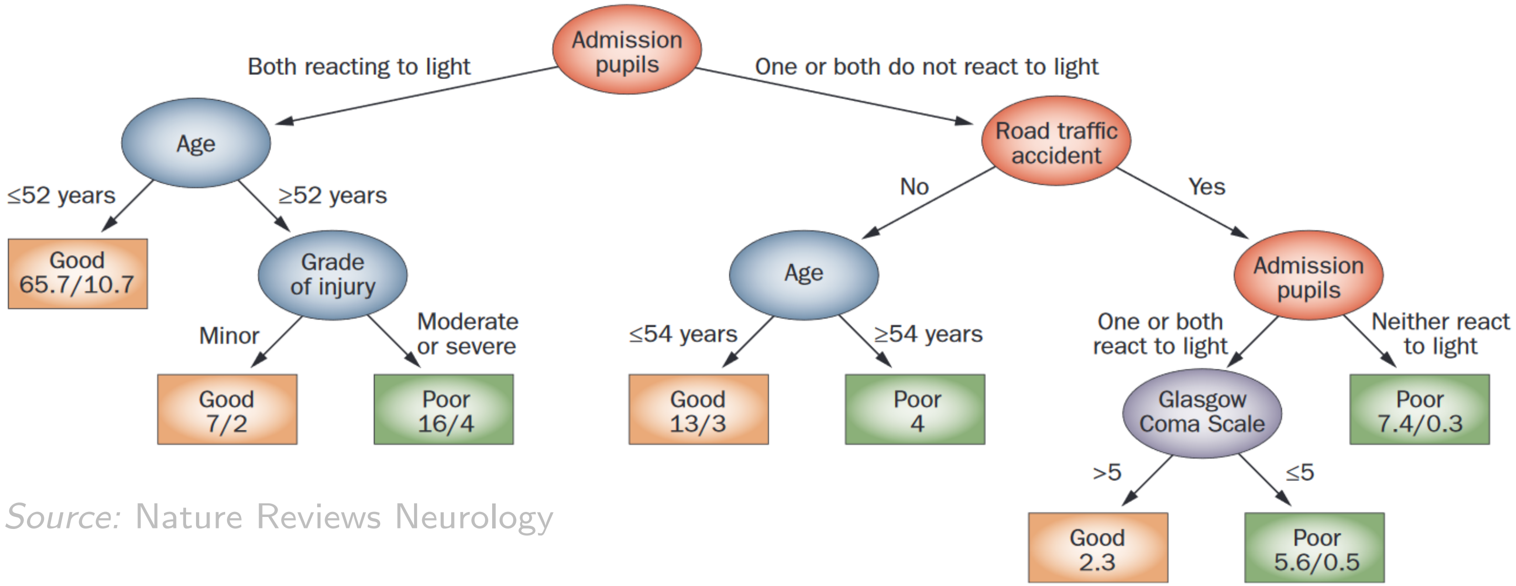


Family tree of elves and half-elves

(The Hobbit & Lord of the Rings)

Layered Drawings – Drawing Style

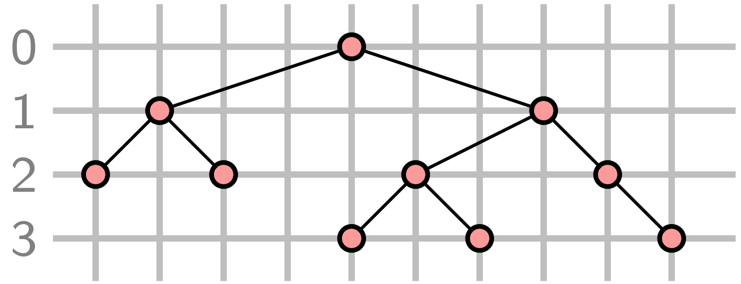


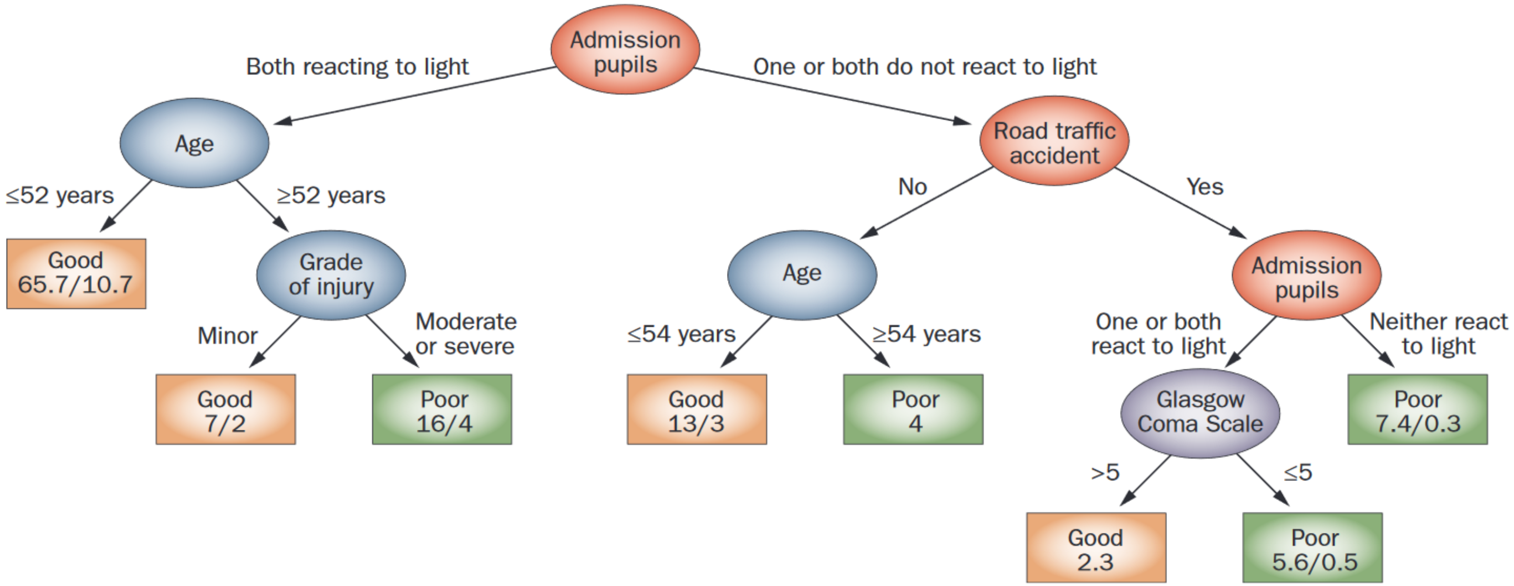


Source: Nature Reviews Neurology

- What are properties of the layout?

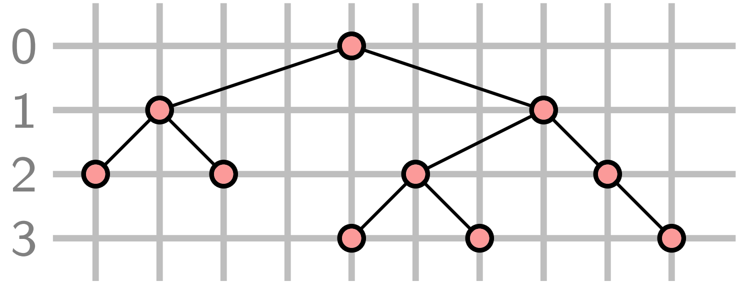
Layered Drawings – Drawing Style

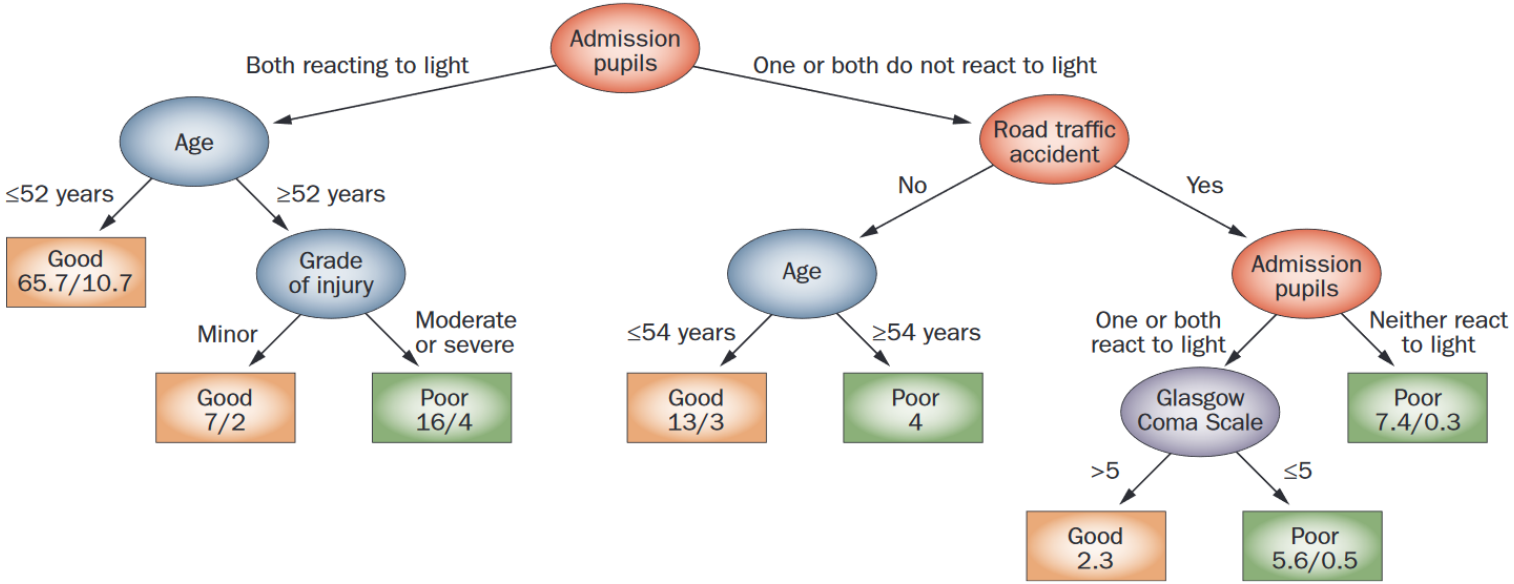




- What are properties of the layout?
- What are the drawing conventions?

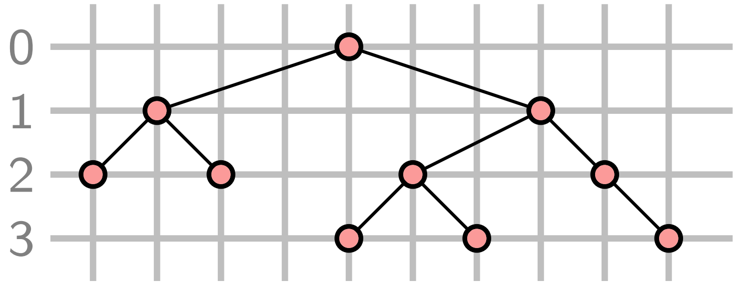
Layered Drawings – Drawing Style

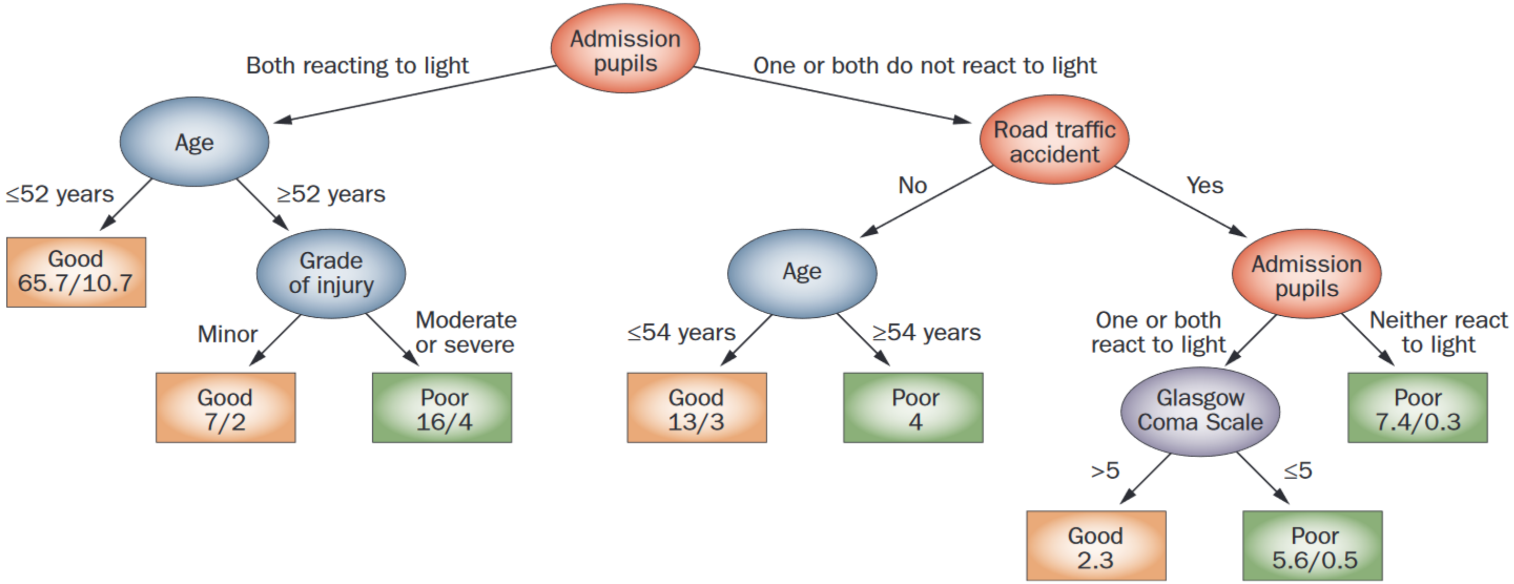




- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

Layered Drawings – Drawing Style

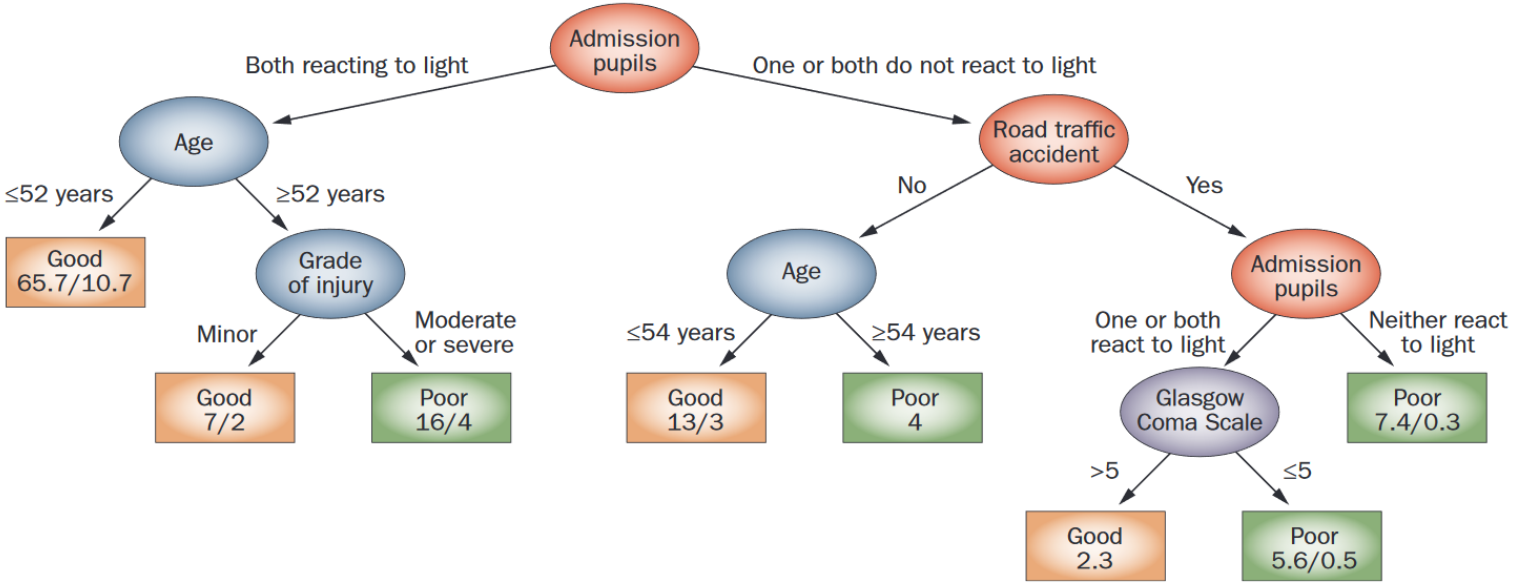


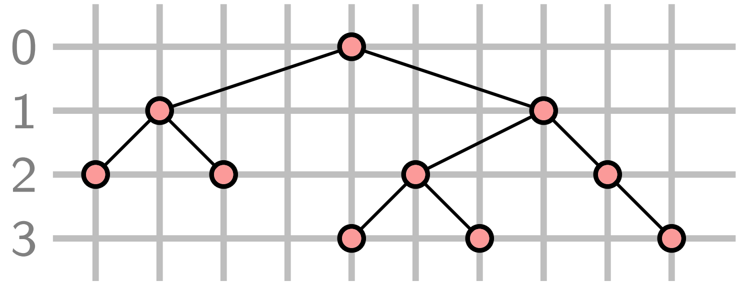


Drawing conventions

- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

Layered Drawings – Drawing Style



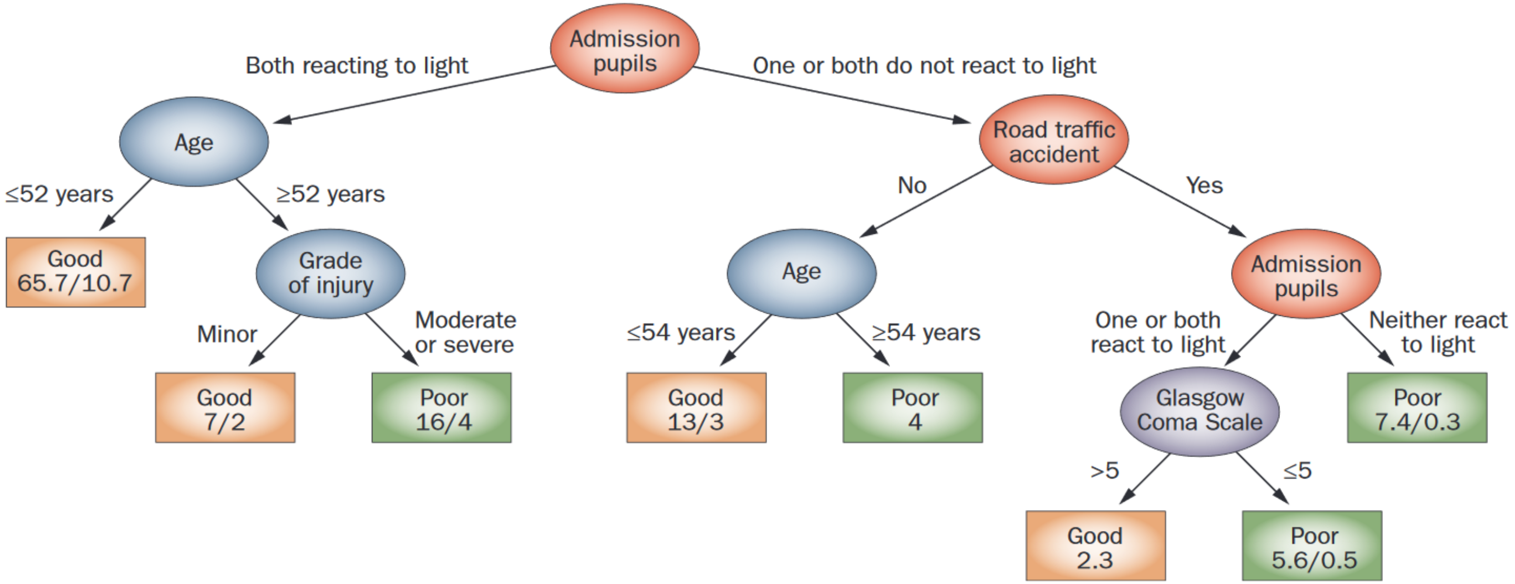


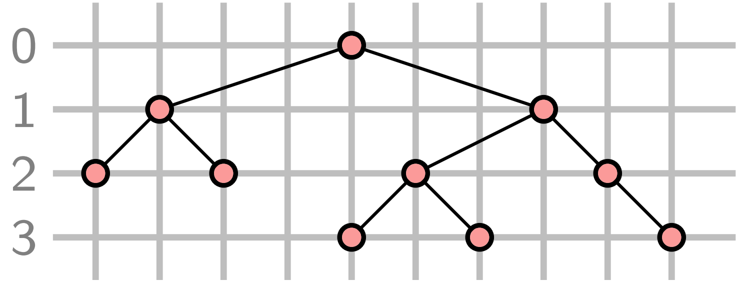
Drawing conventions

- Vertices lie on layers and have integer coordinates

- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

Layered Drawings – Drawing Style



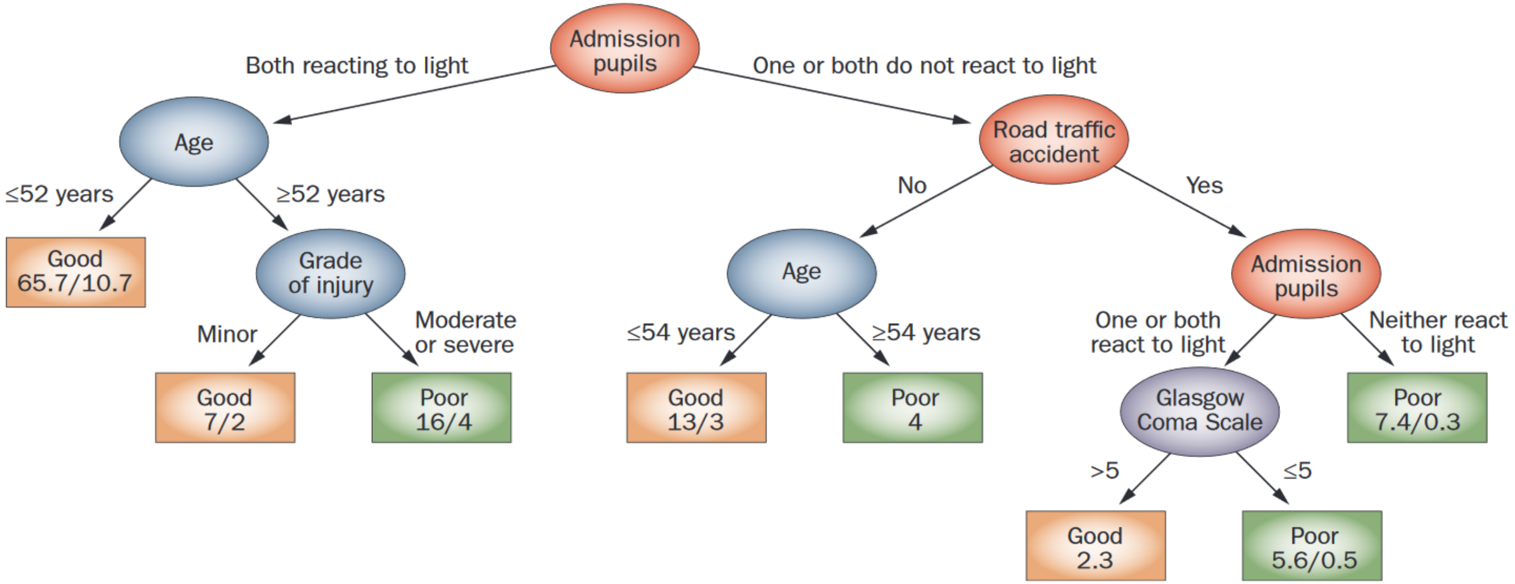


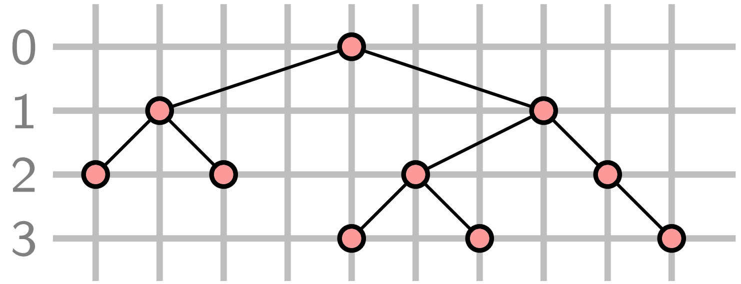
- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children (if there is more than one child)

Layered Drawings – Drawing Style



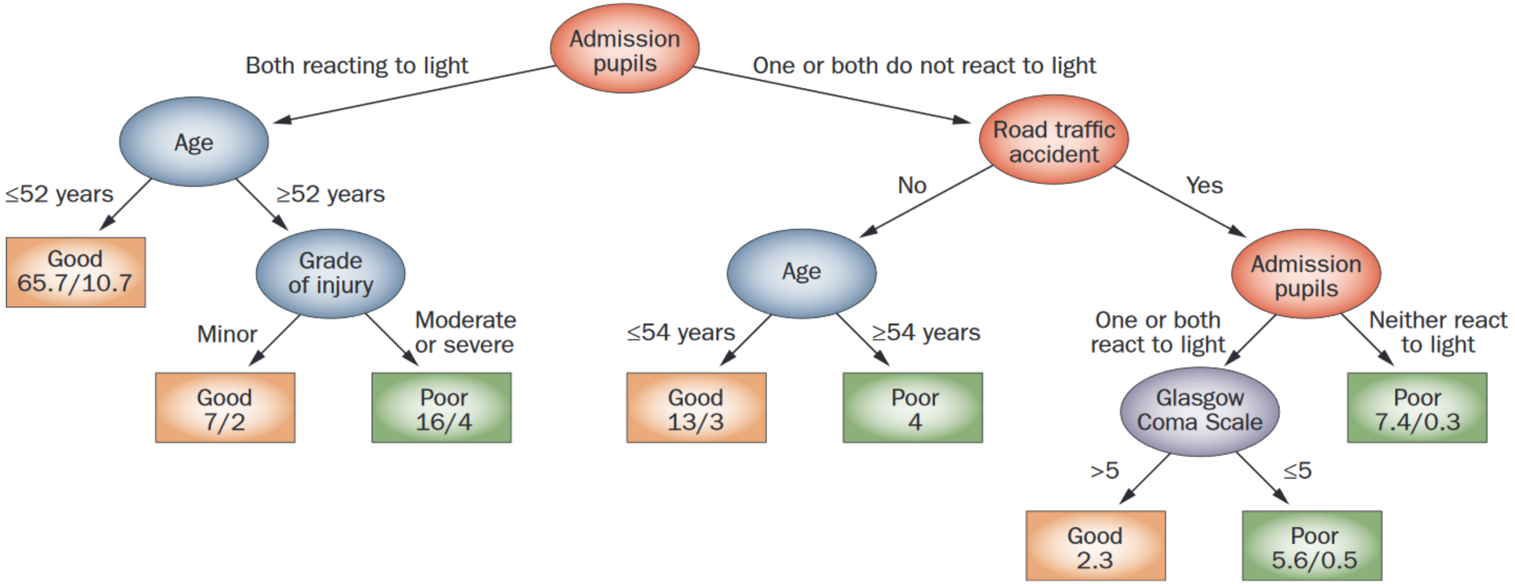


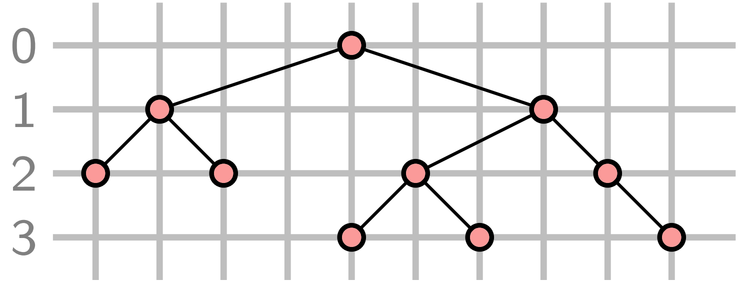
- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children (if there is more than one child)
- Edges are straight-line segments

Layered Drawings – Drawing Style



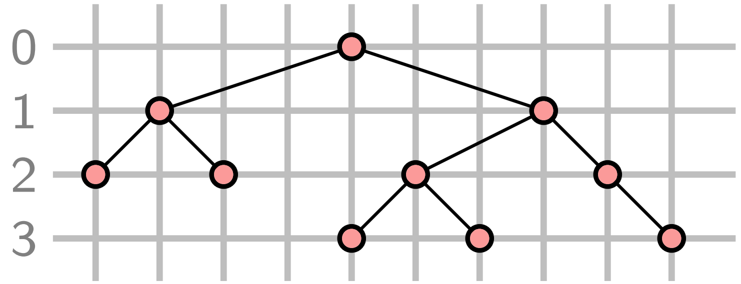


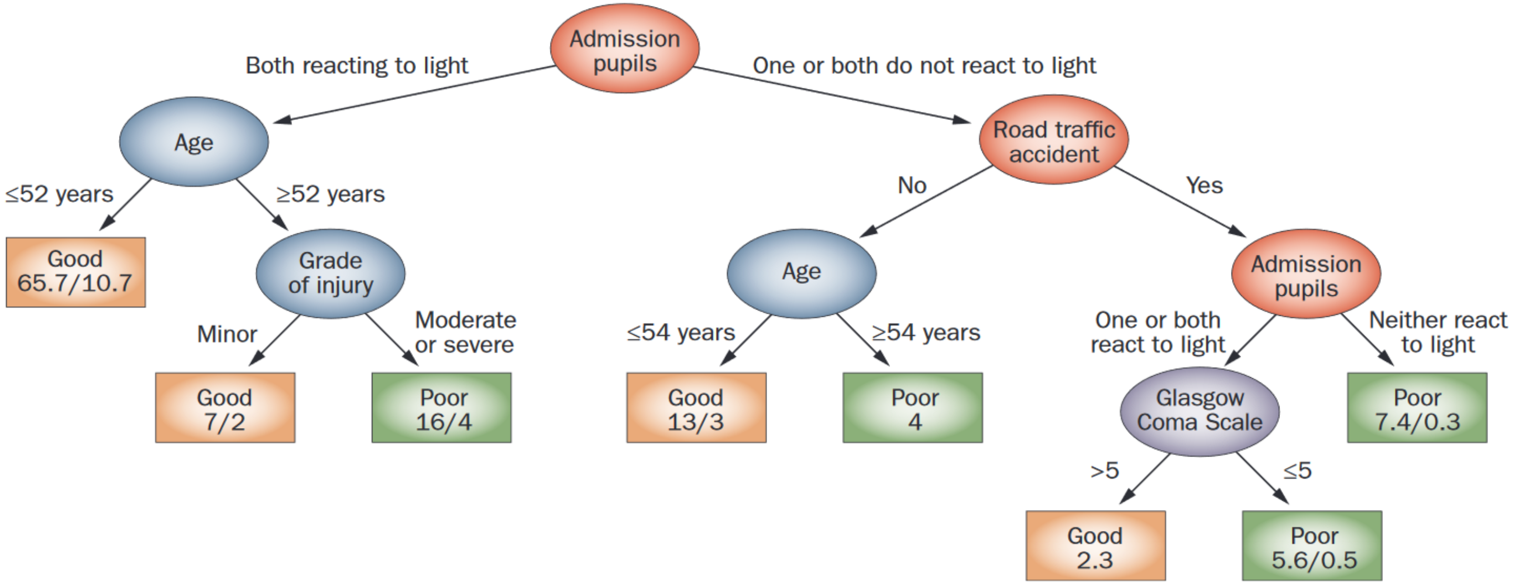
- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

Drawing conventions

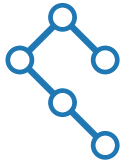
- Vertices lie on layers and have integer coordinates
- Parent centered above children (if there is more than one child)
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

Layered Drawings – Drawing Style





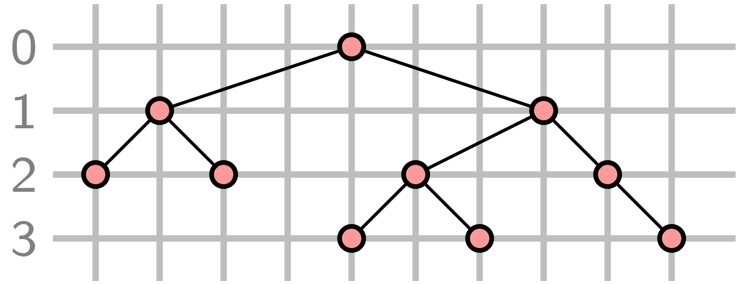
- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

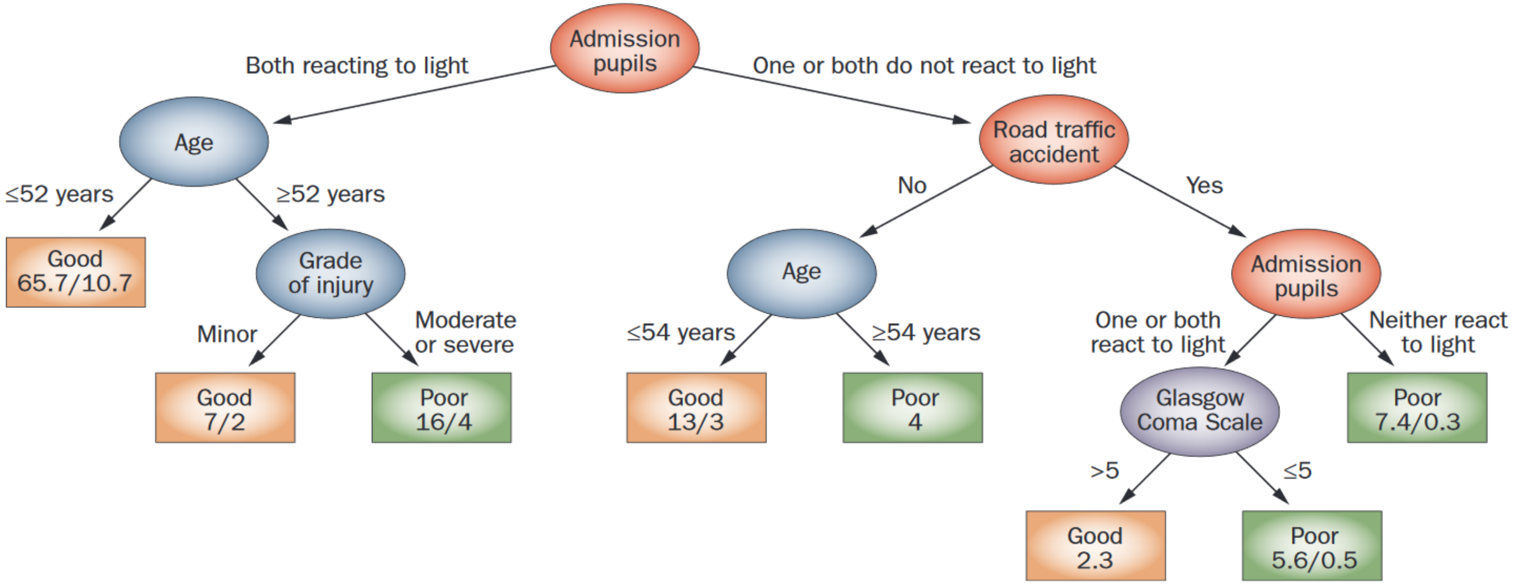


Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children (if there is more than one child)
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

Layered Drawings – Drawing Style

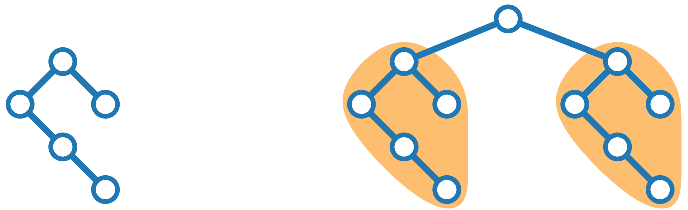




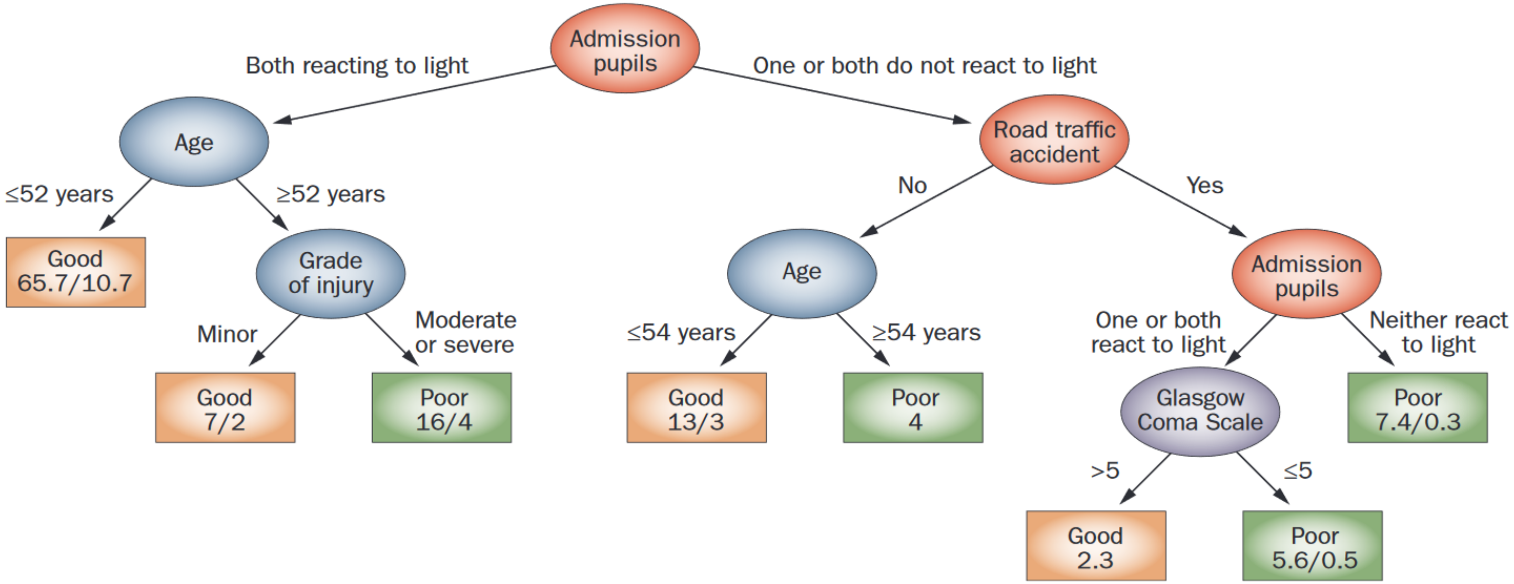
Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children (if there is more than one child)
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

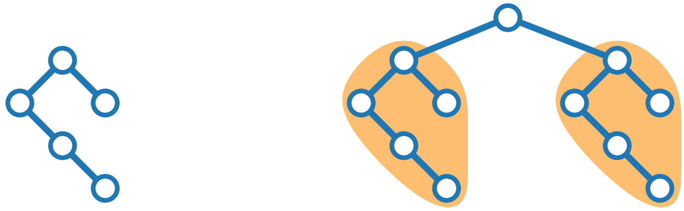
- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

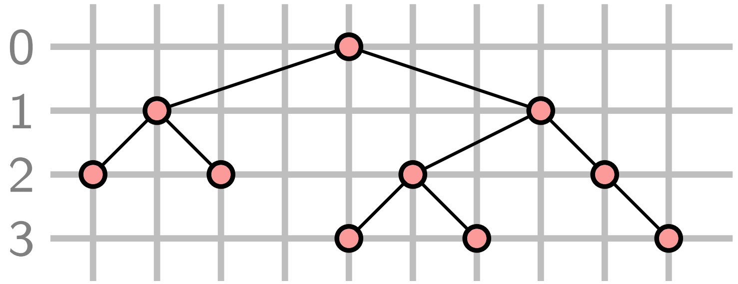


Layered Drawings – Drawing Style



- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?



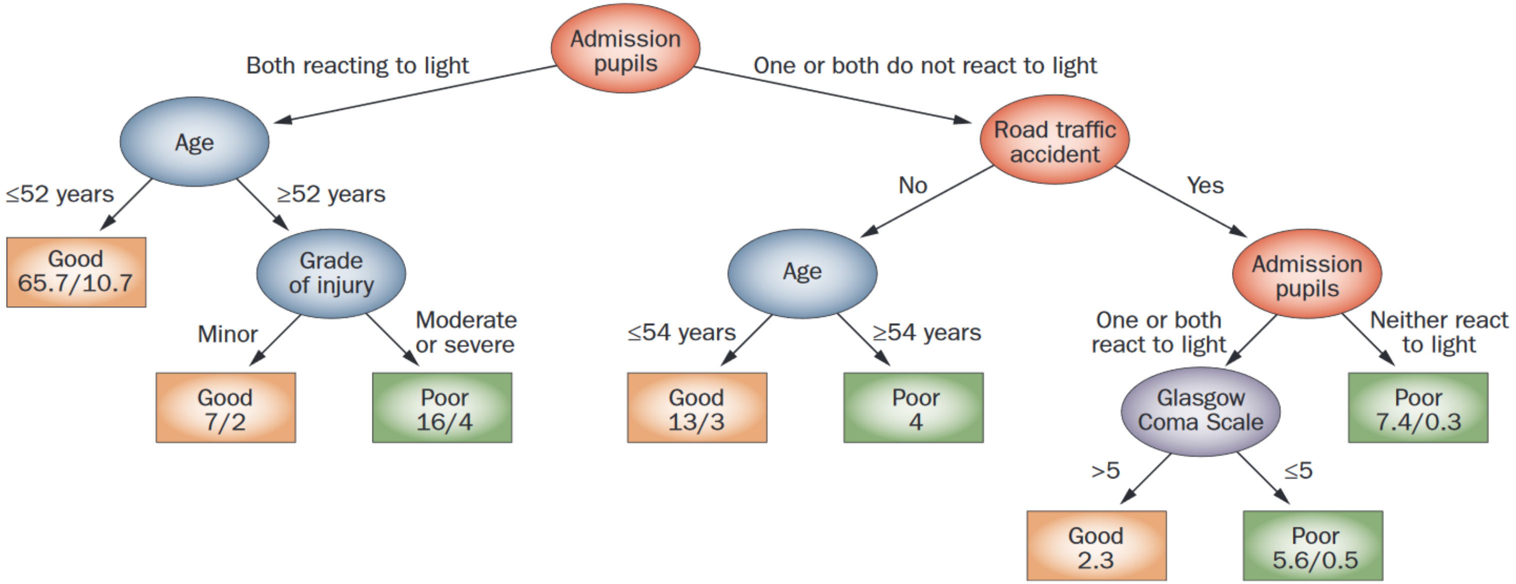


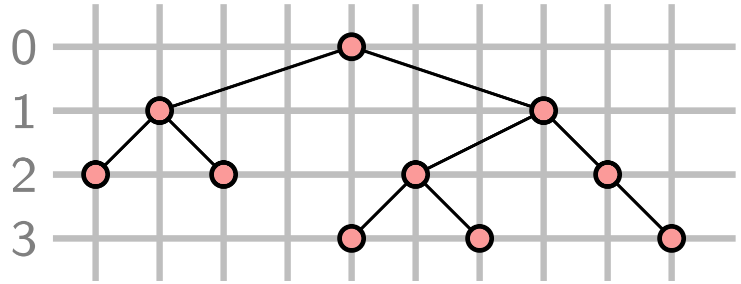
Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children (if there is more than one child)
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

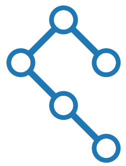
Drawing aesthetics to optimize

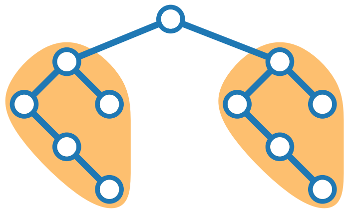
Layered Drawings – Drawing Style





- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?





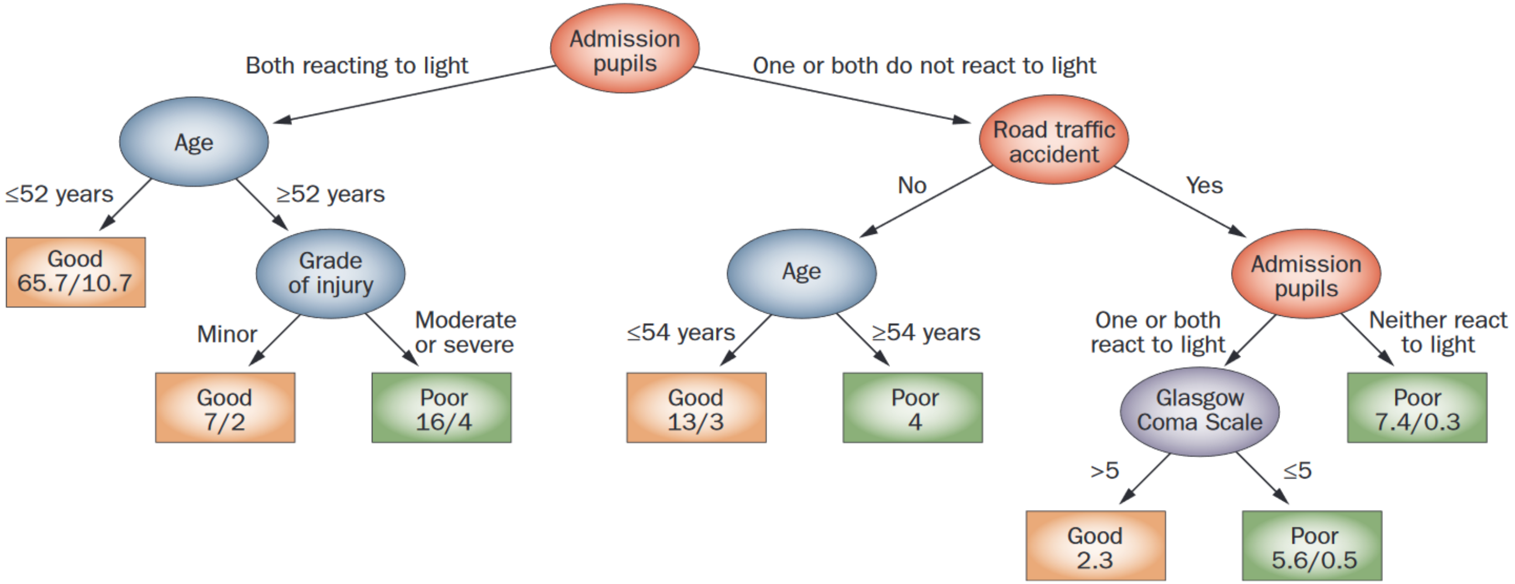
Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children (if there is more than one child)
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

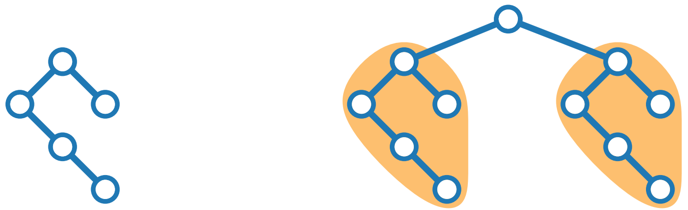
Drawing aesthetics to optimize

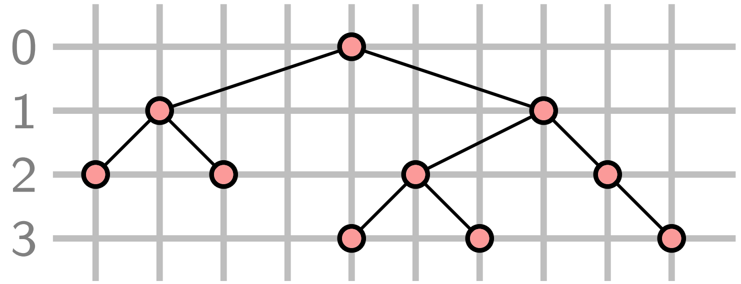
- Area

Layered Drawings – Drawing Style



- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?





Drawing conventions

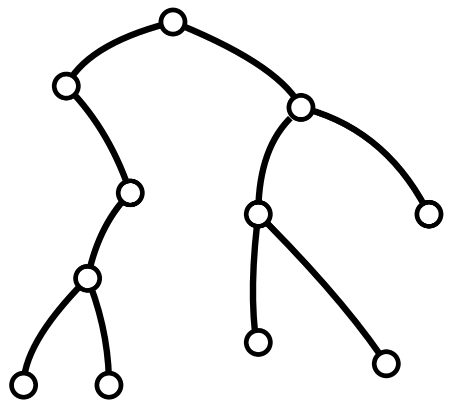
- Vertices lie on layers and have integer coordinates
- Parent centered above children (if there is more than one child)
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

Drawing aesthetics to optimize

- Area
- Symmetries

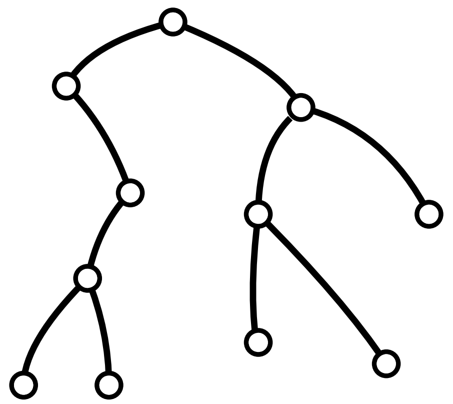
Layered Drawings – Algorithm

Input: A binary tree T
Output: A layered drawing of T



Layered Drawings – Algorithm

Input: A binary tree T
Output: A layered drawing of T



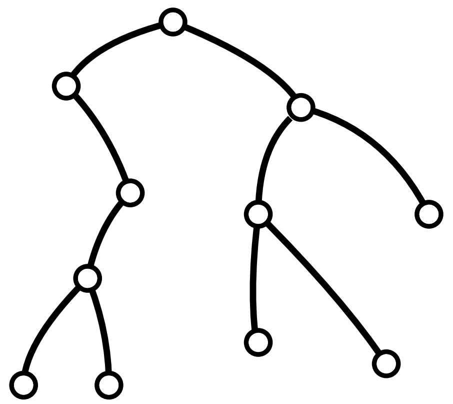
Base case:

Divide:

Conquer:

Layered Drawings – Algorithm

Input: A binary tree T
Output: A layered drawing of T



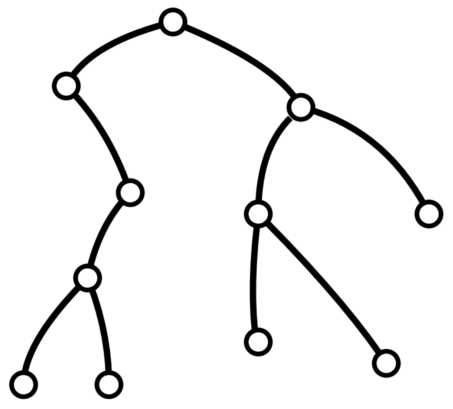
Base case: A single vertex

Divide:

Conquer:

Layered Drawings – Algorithm

Input: A binary tree T
Output: A layered drawing of T



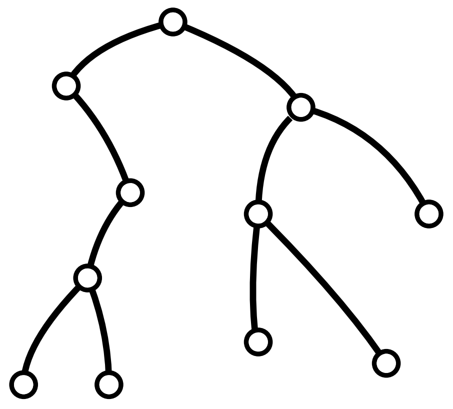
Base case: A single vertex 

Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:

Layered Drawings – Algorithm

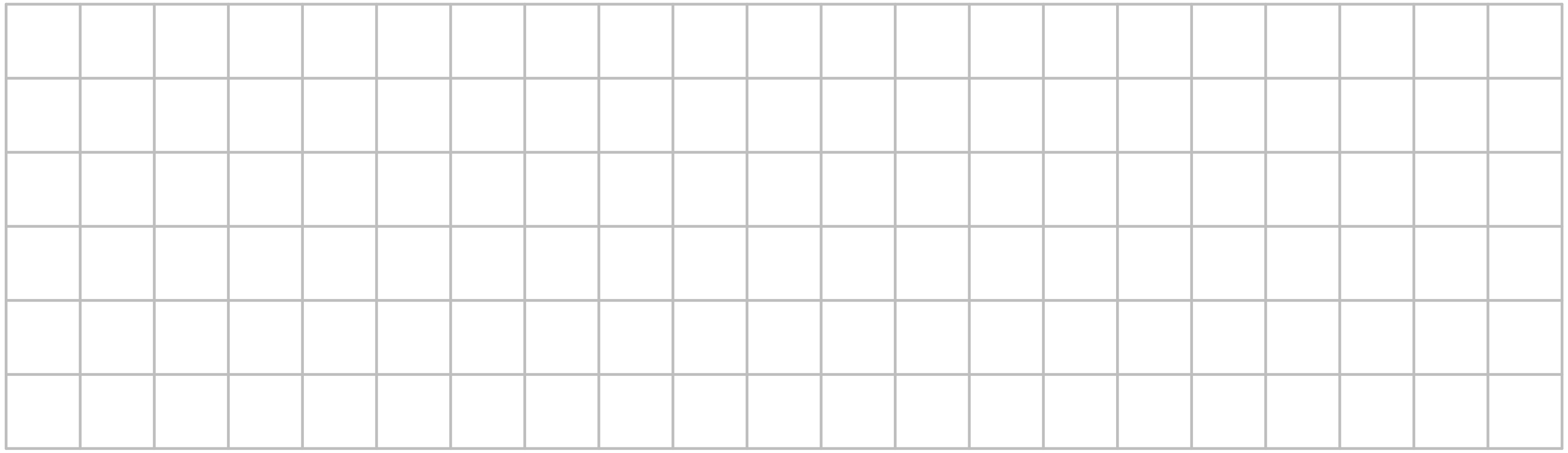
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

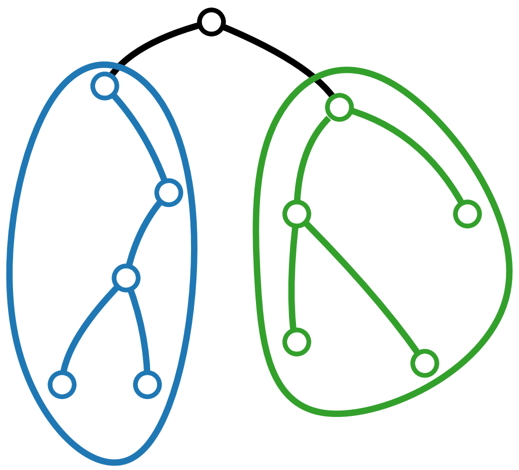
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

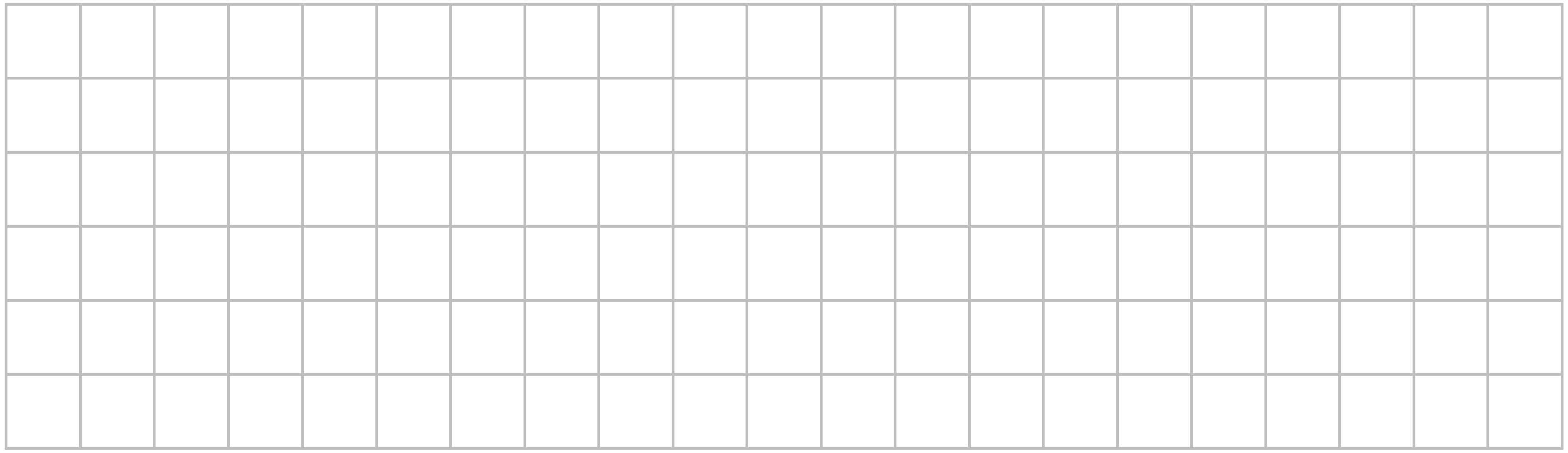
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex 

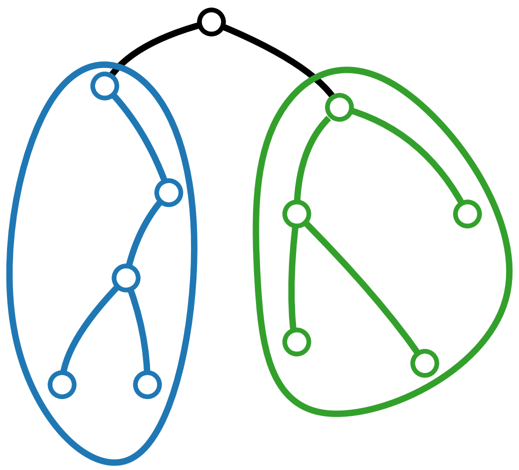
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

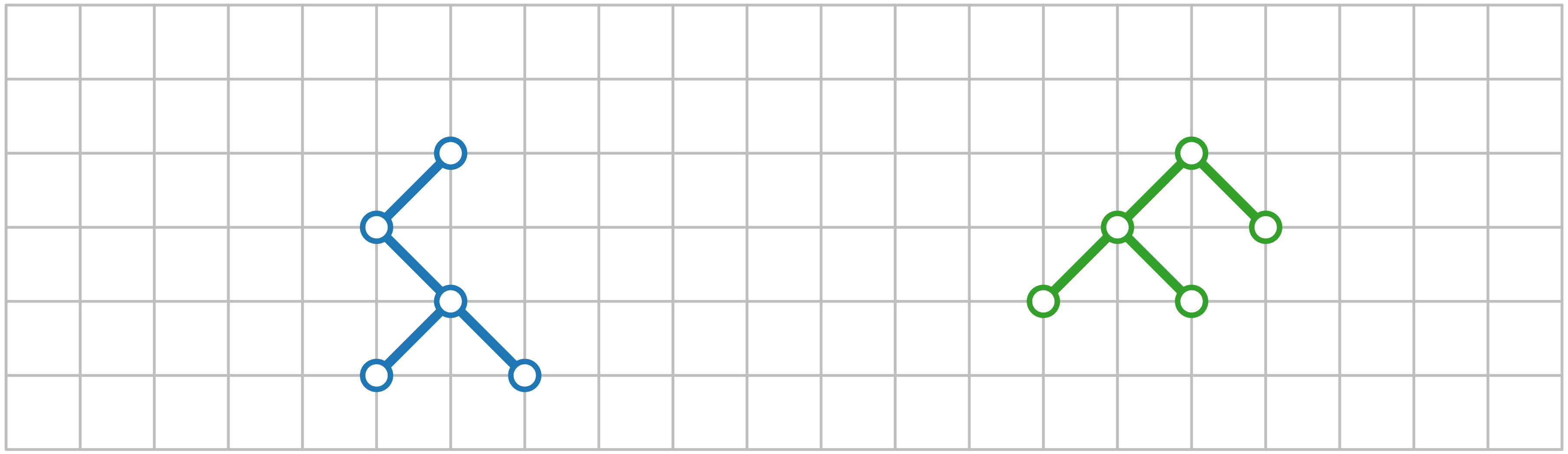
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

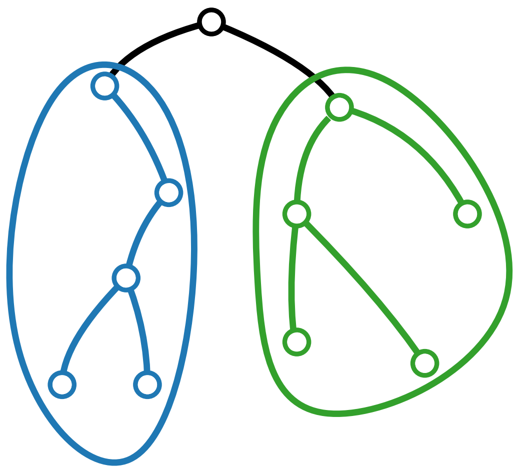
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

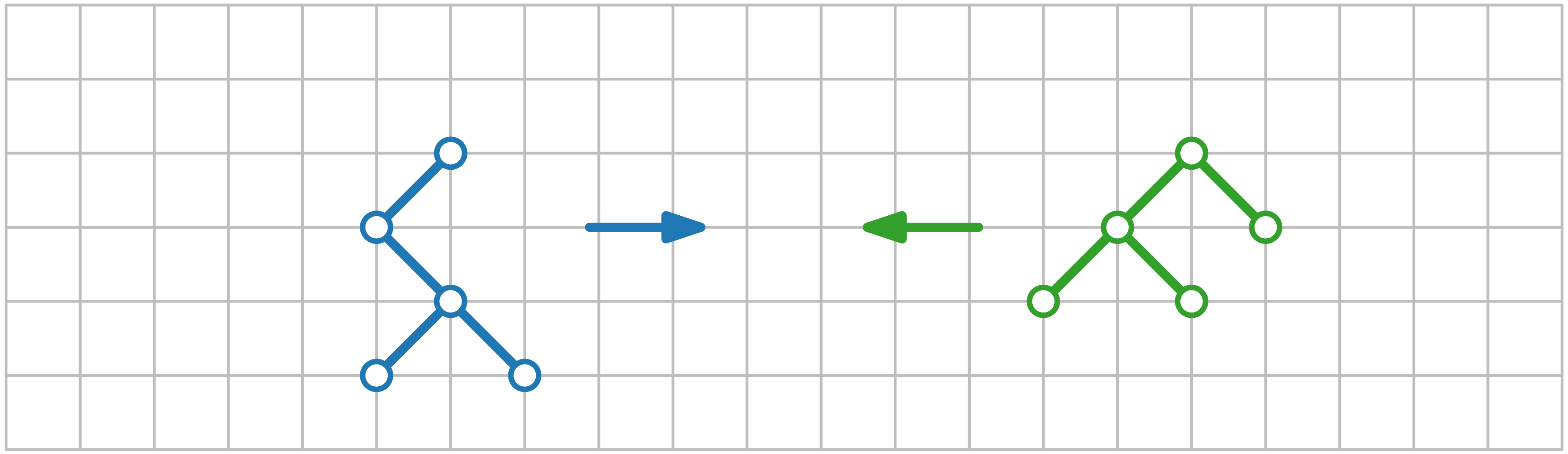
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

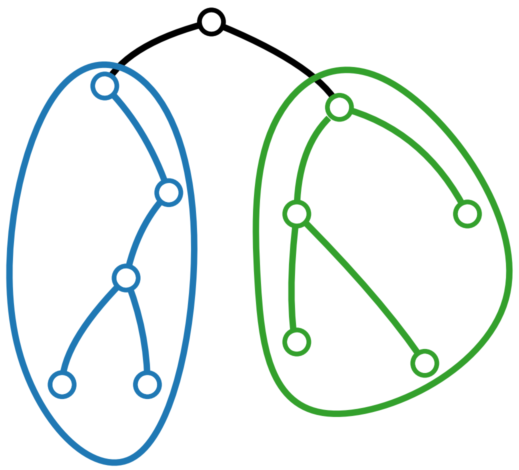
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

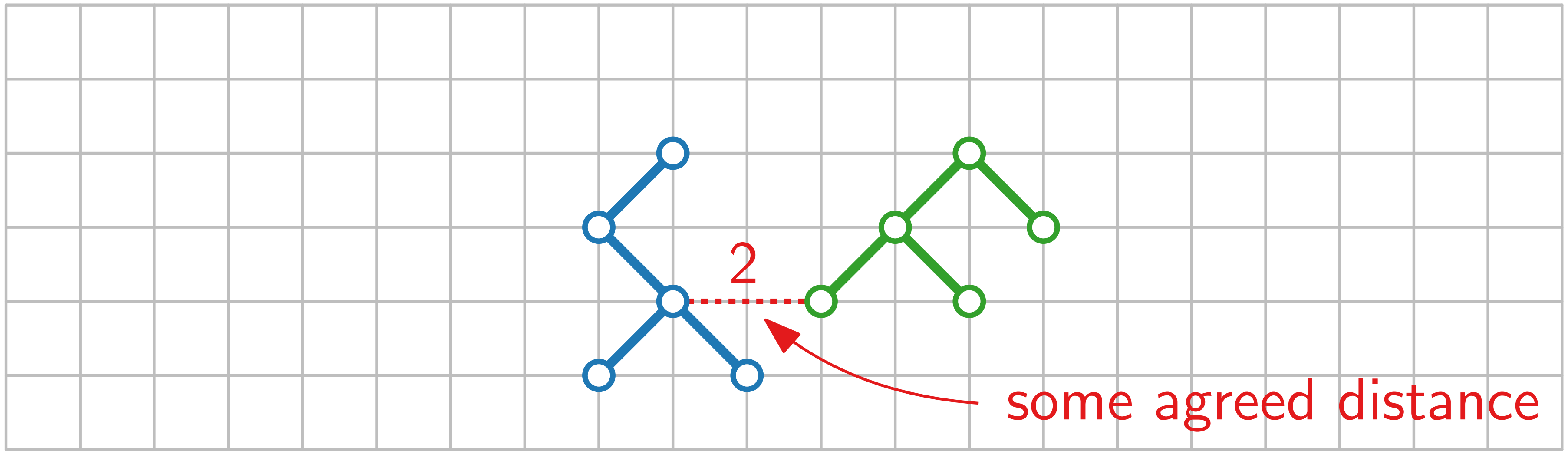
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex ○

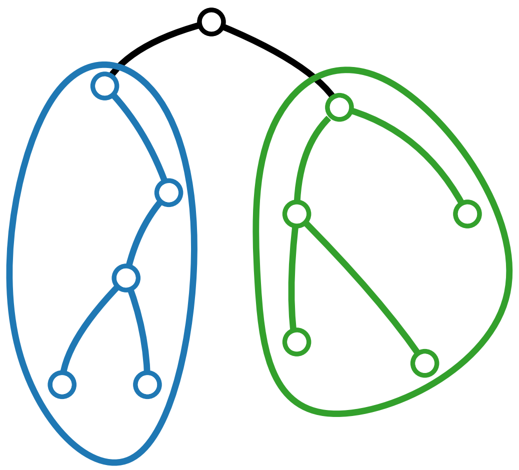
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

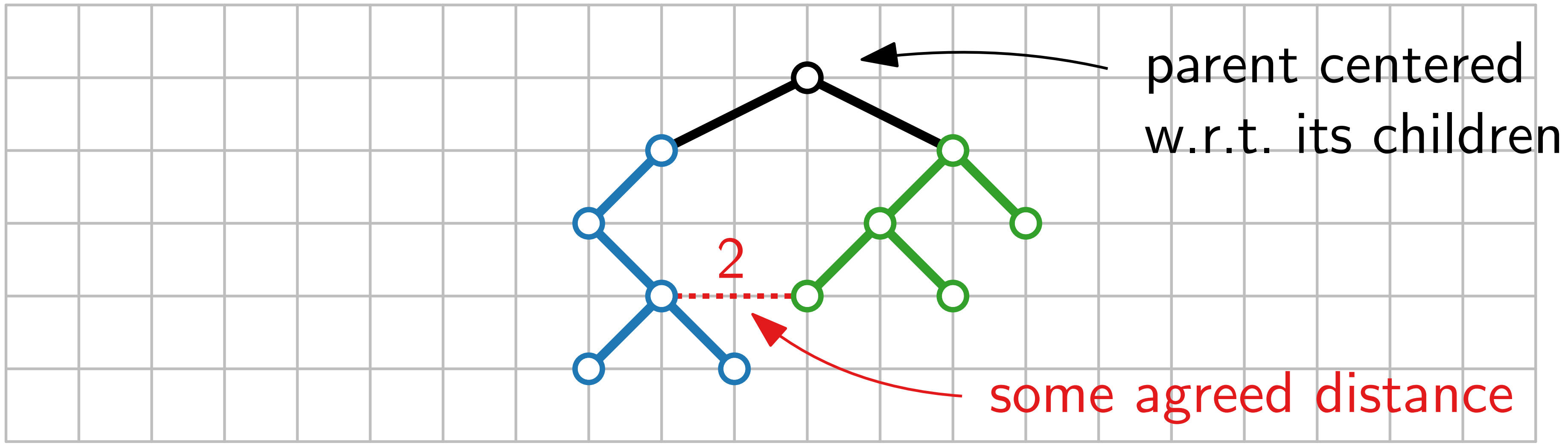
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

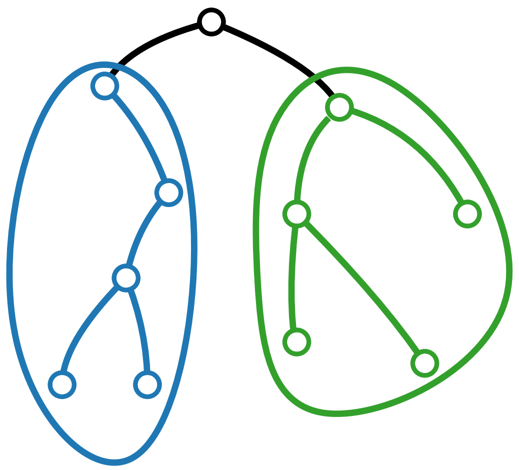
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

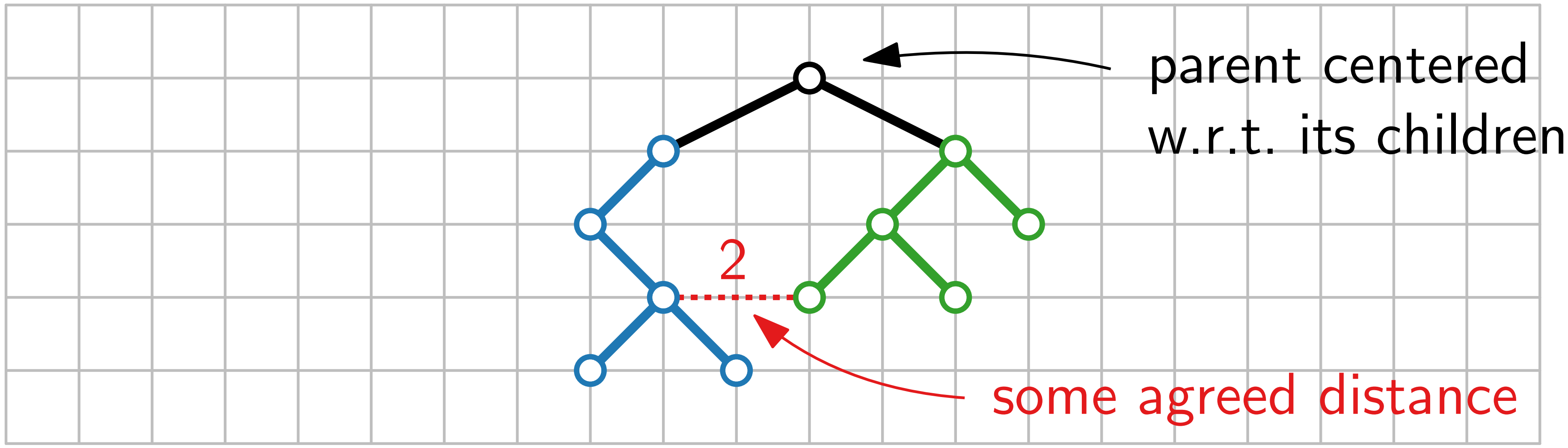
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex ○

Divide: Recursively apply the algorithm to draw the left and right subtrees

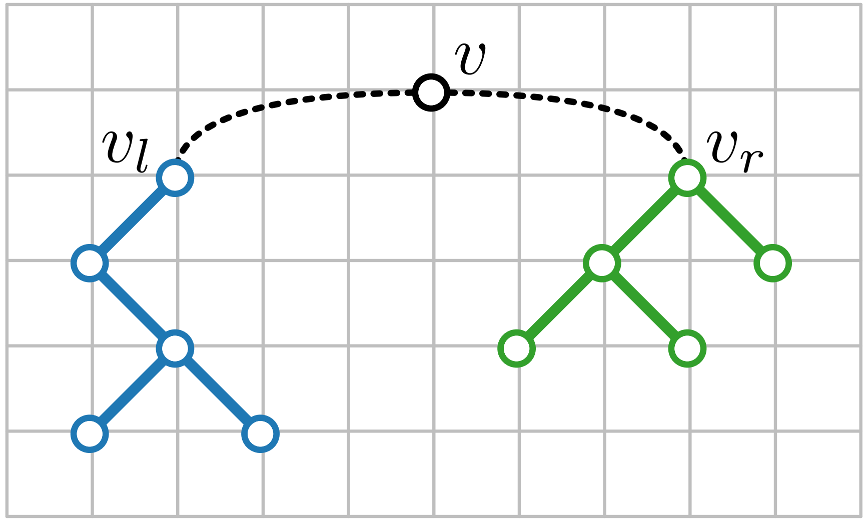
Conquer:



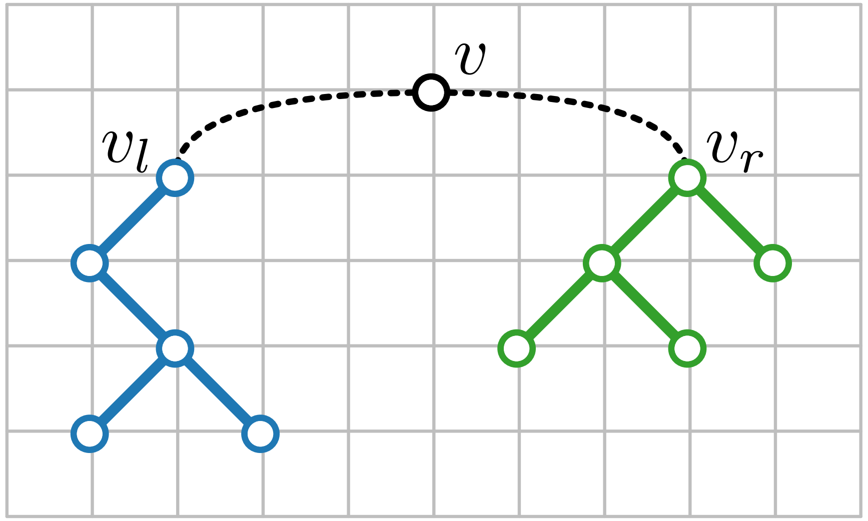
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.



Layered Drawings – Algorithm Details



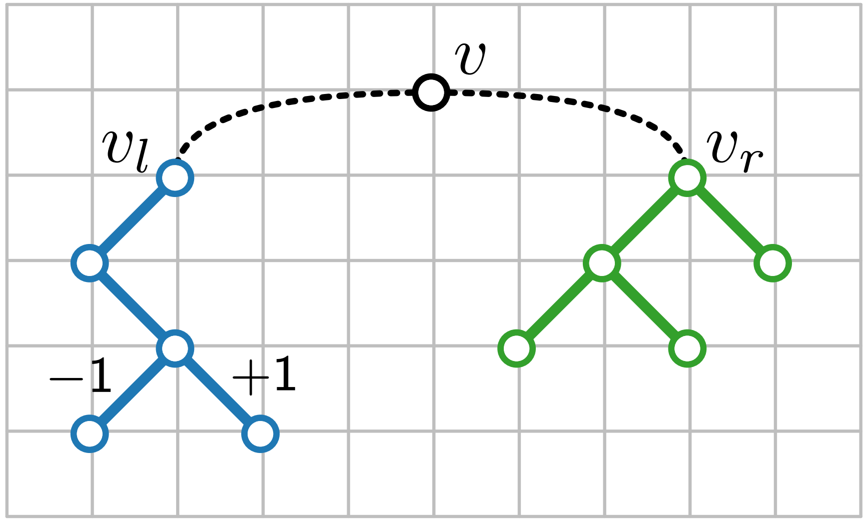
Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.

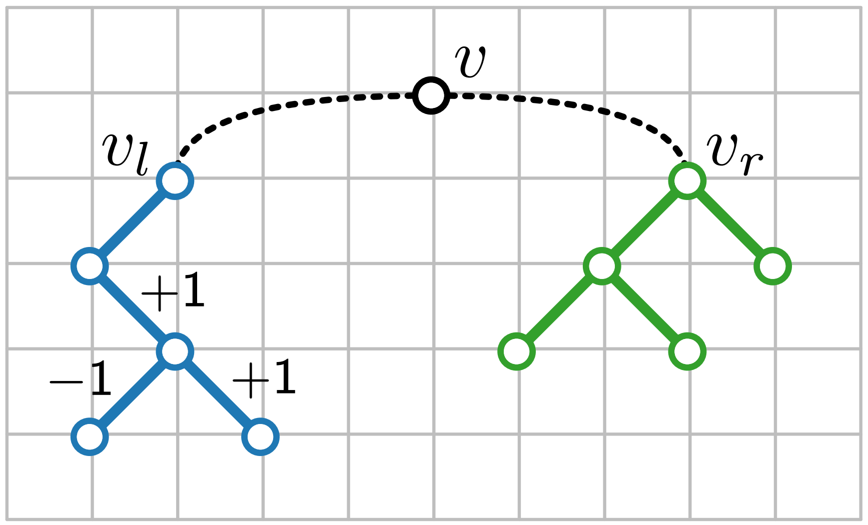
Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.



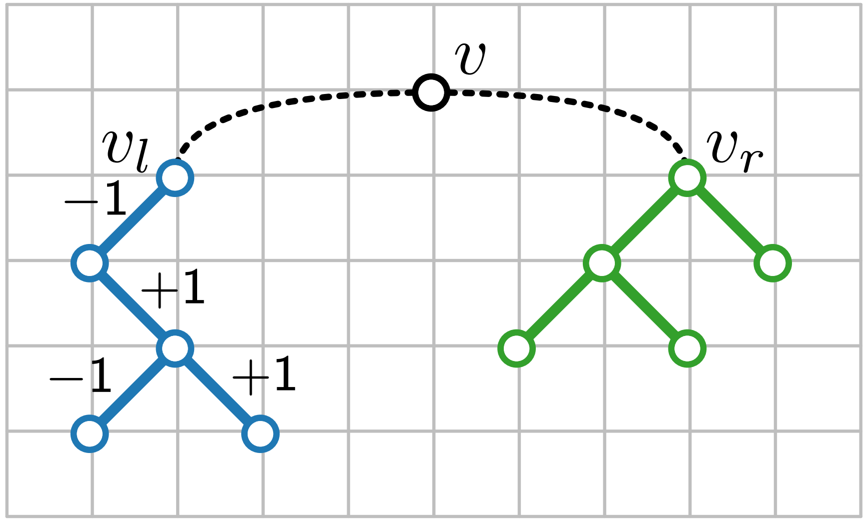
Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.



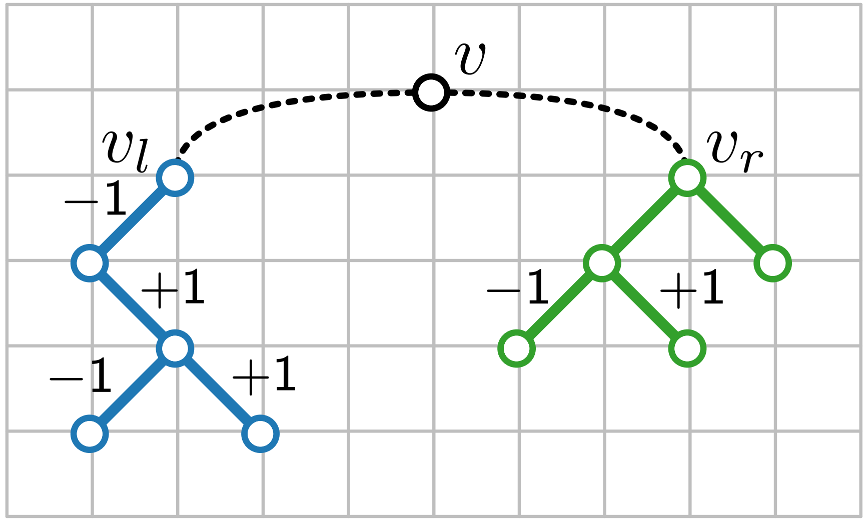
Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

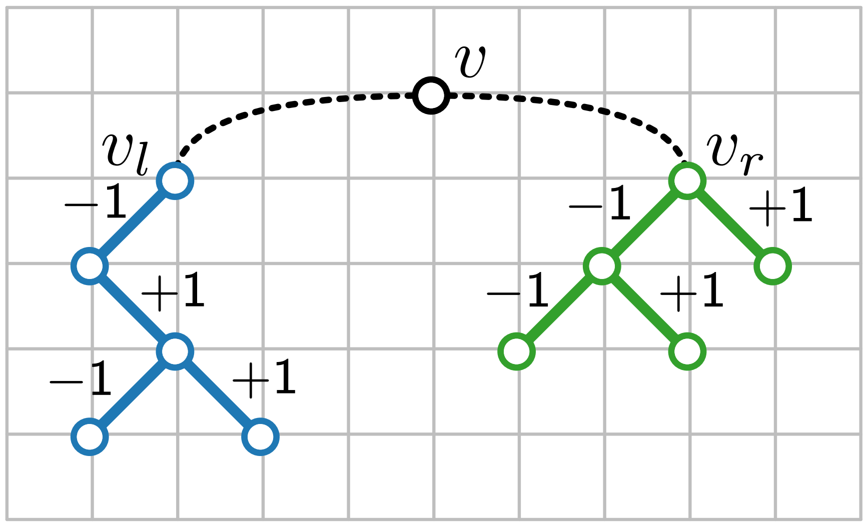
- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.



Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.

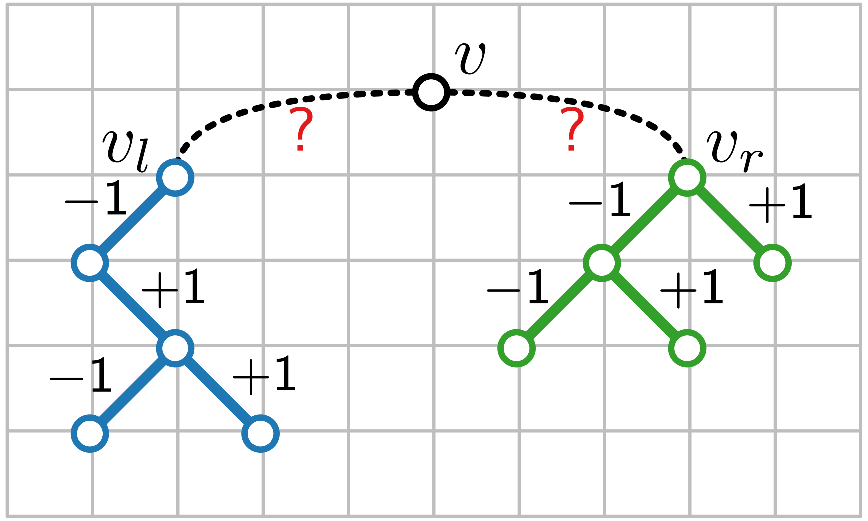
Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

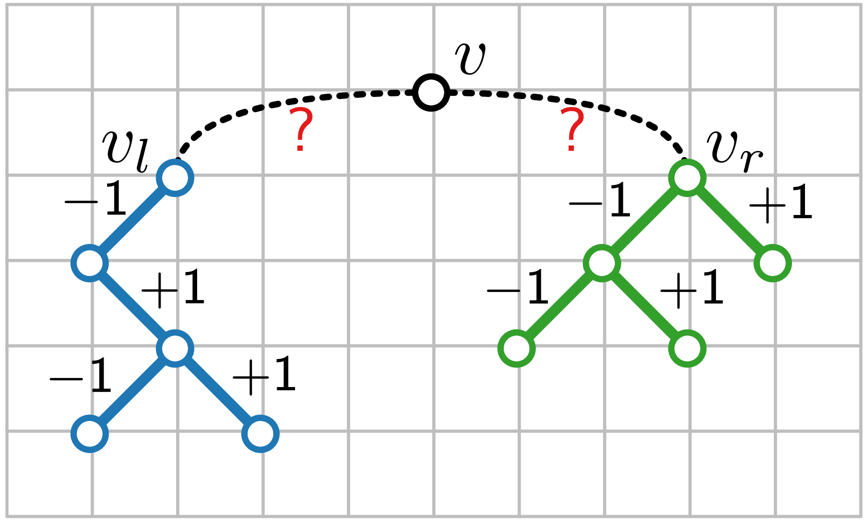
- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.



Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

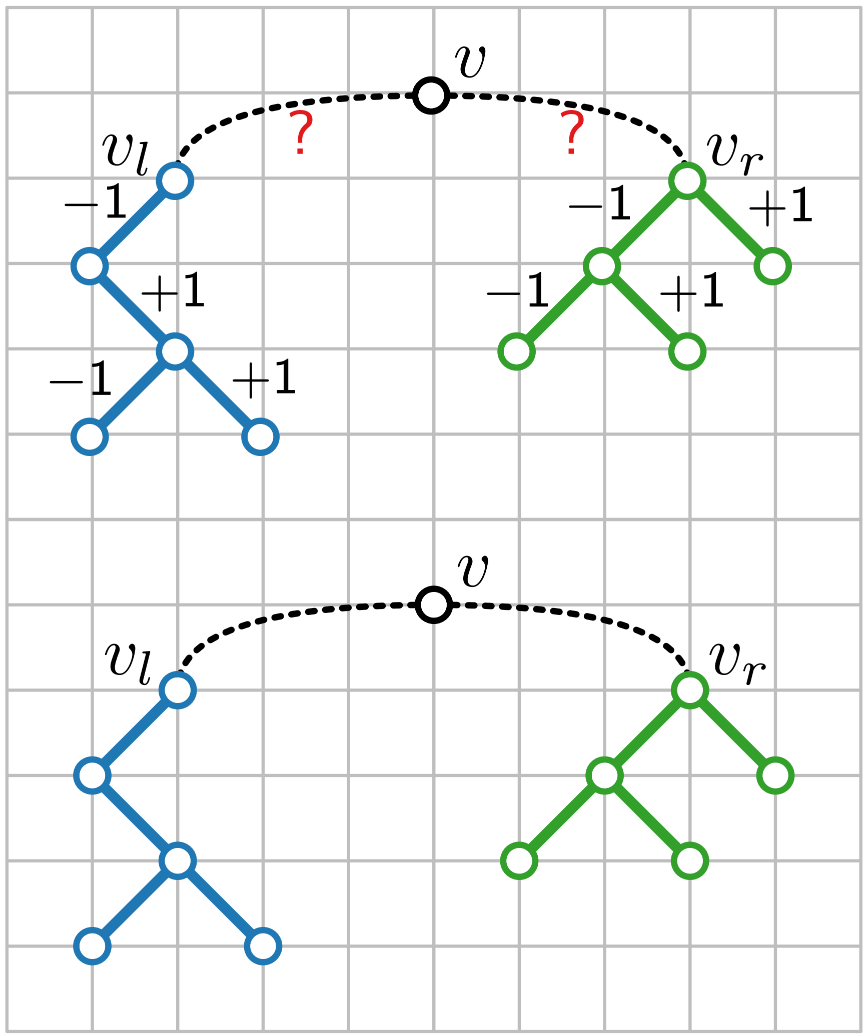
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.

Phase 2 – preorder traversal:

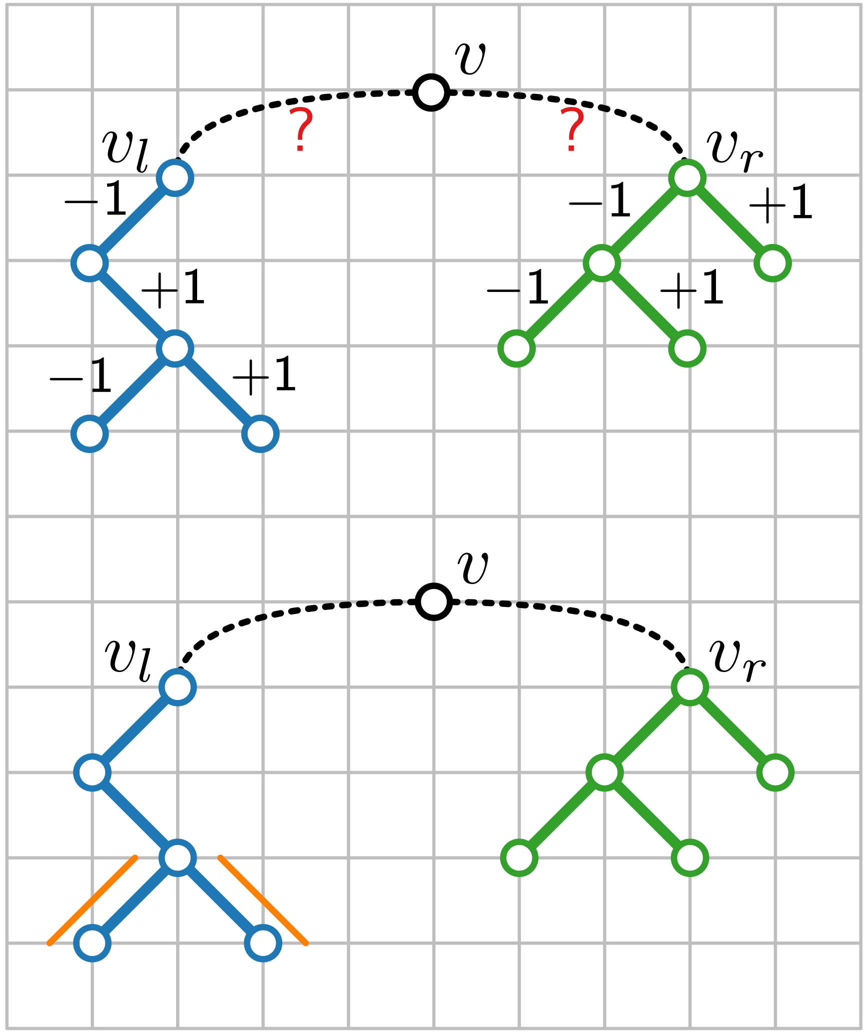
- Compute x- and y-coordinates



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.



Phase 2 – preorder traversal:

- Compute x- and y-coordinates

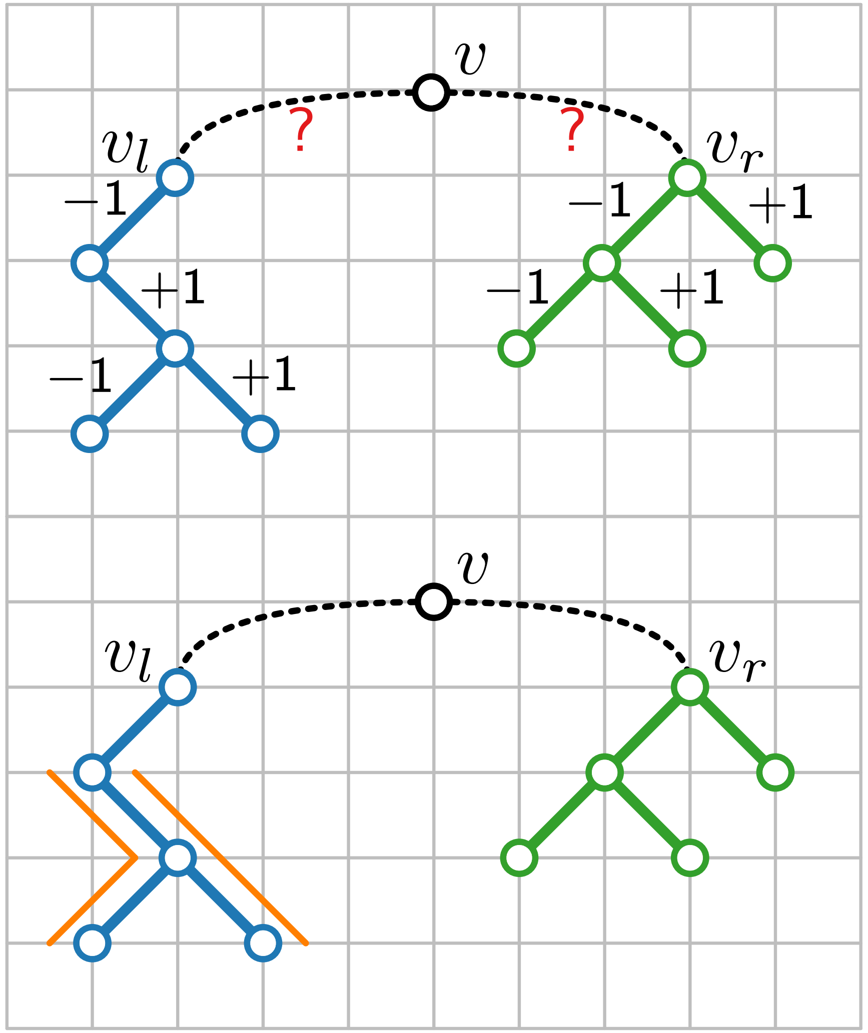
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



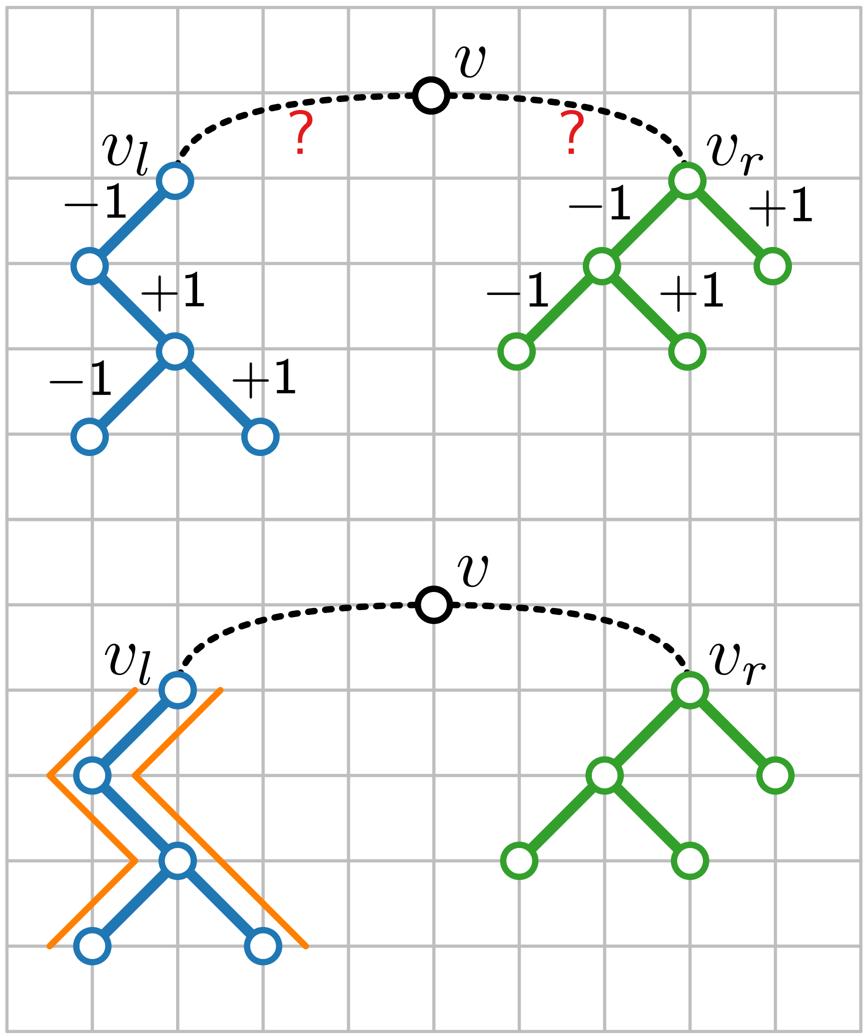
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



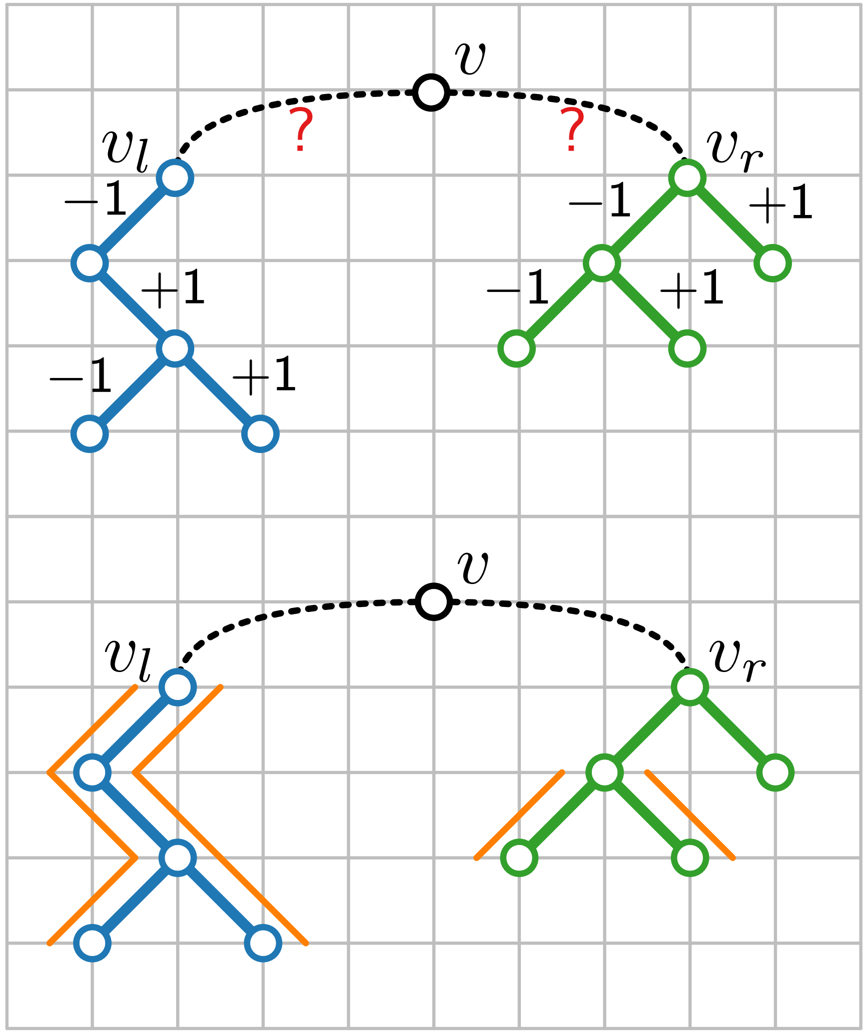
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

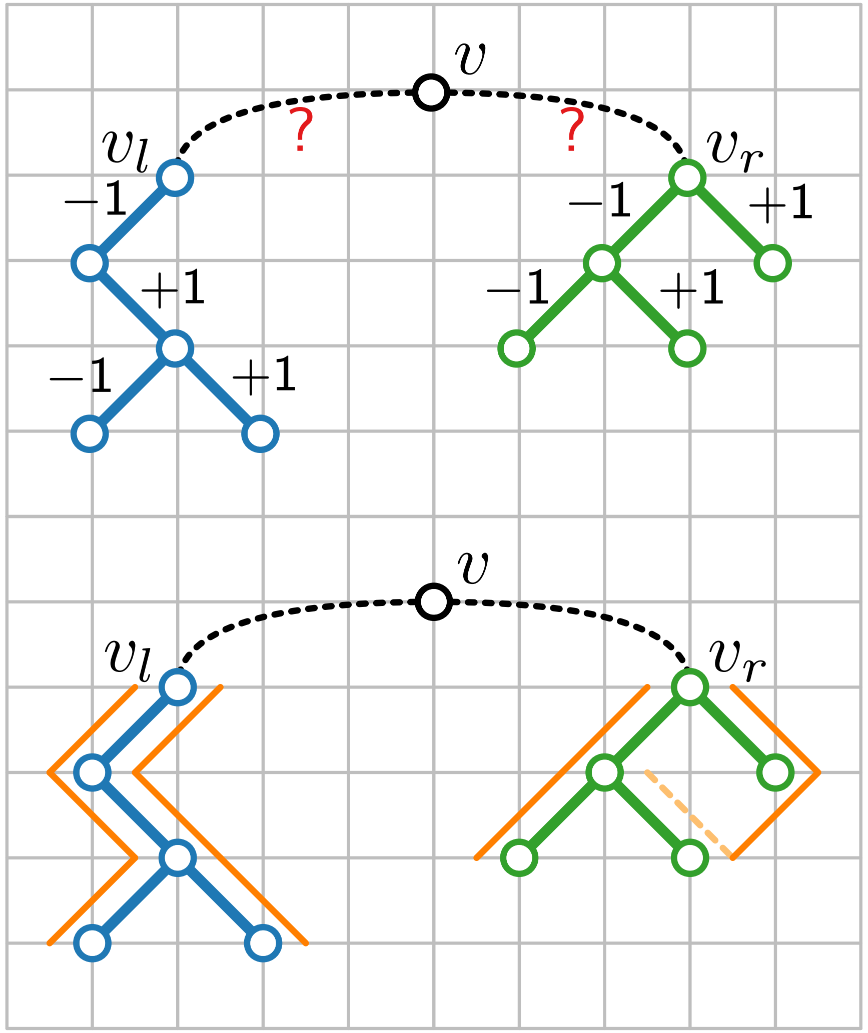


Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.

- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.



Phase 2 – preorder traversal:

- Compute x- and y-coordinates

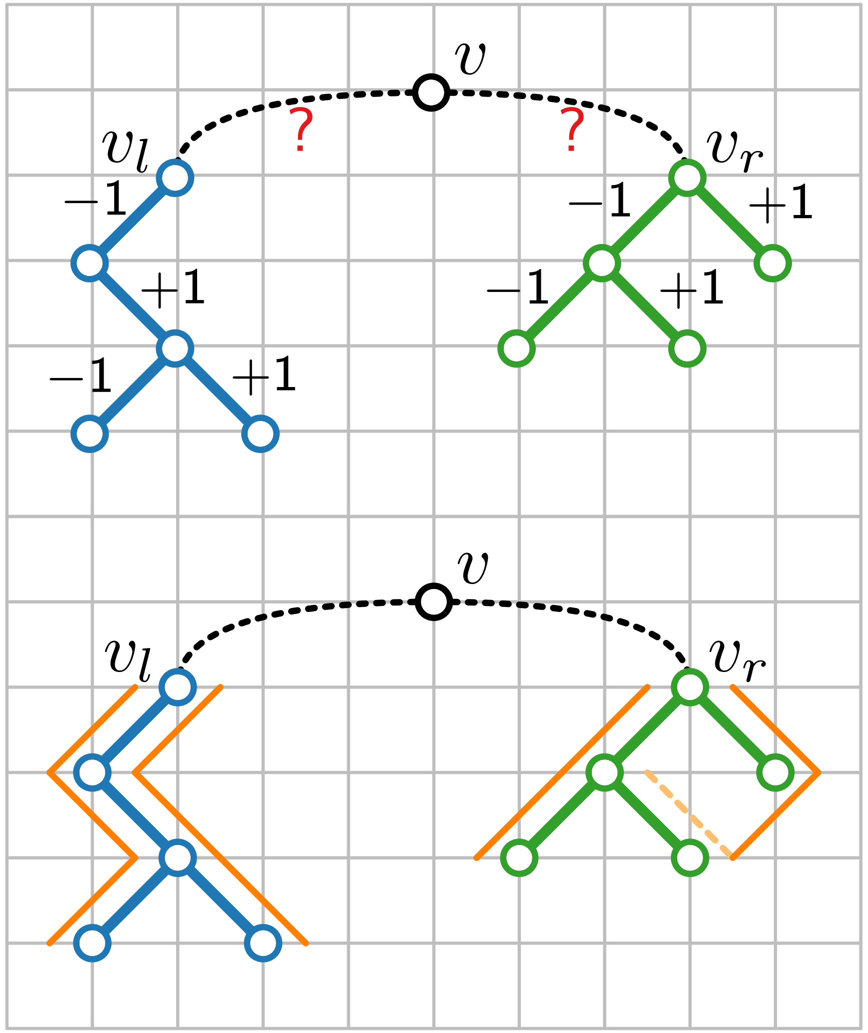
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



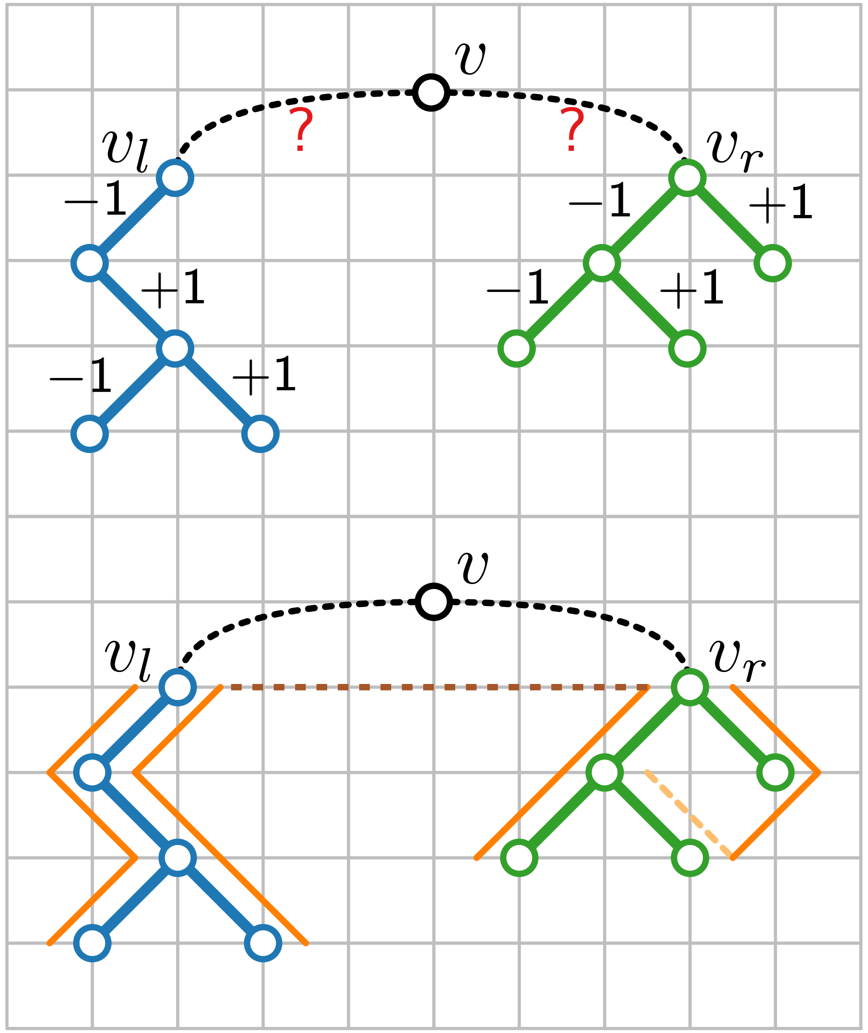
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



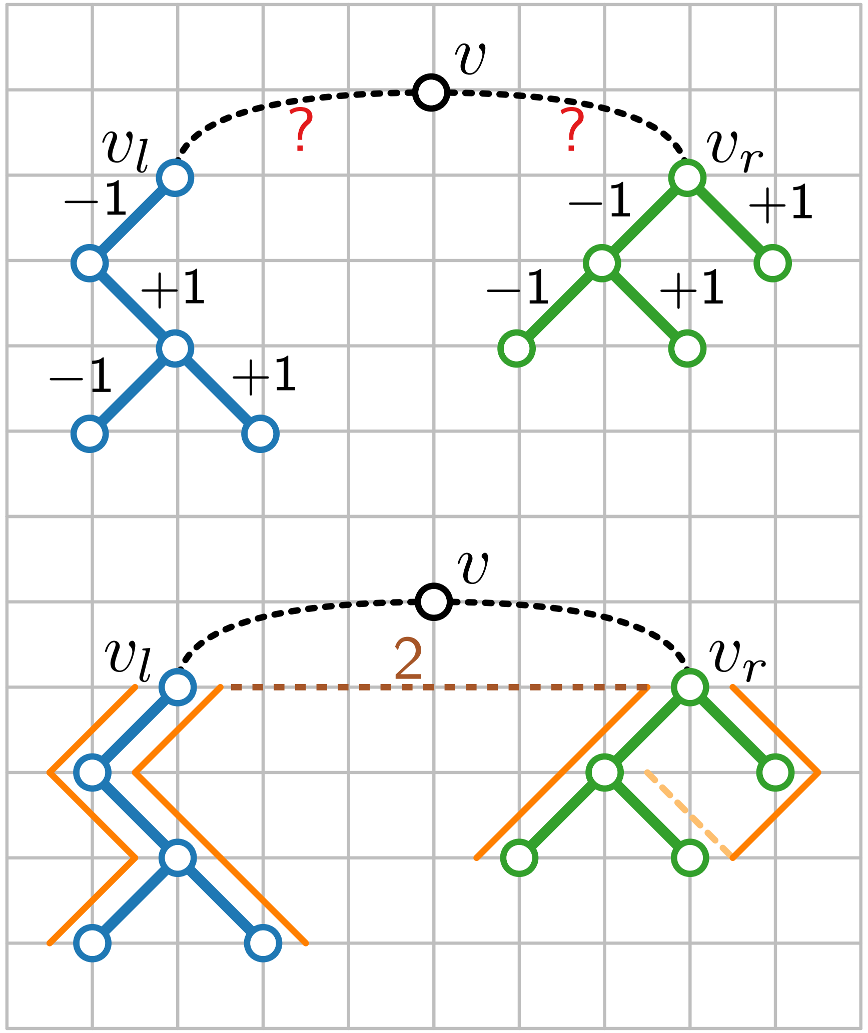
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find $d_v =$ min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



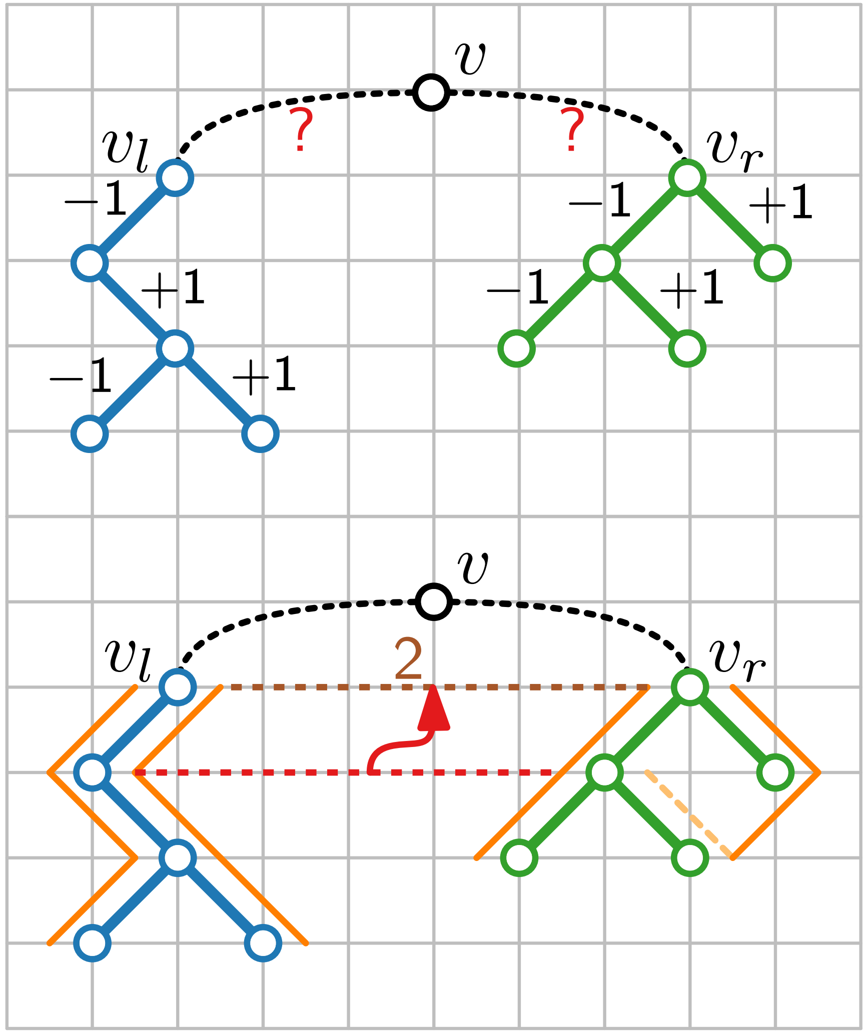
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



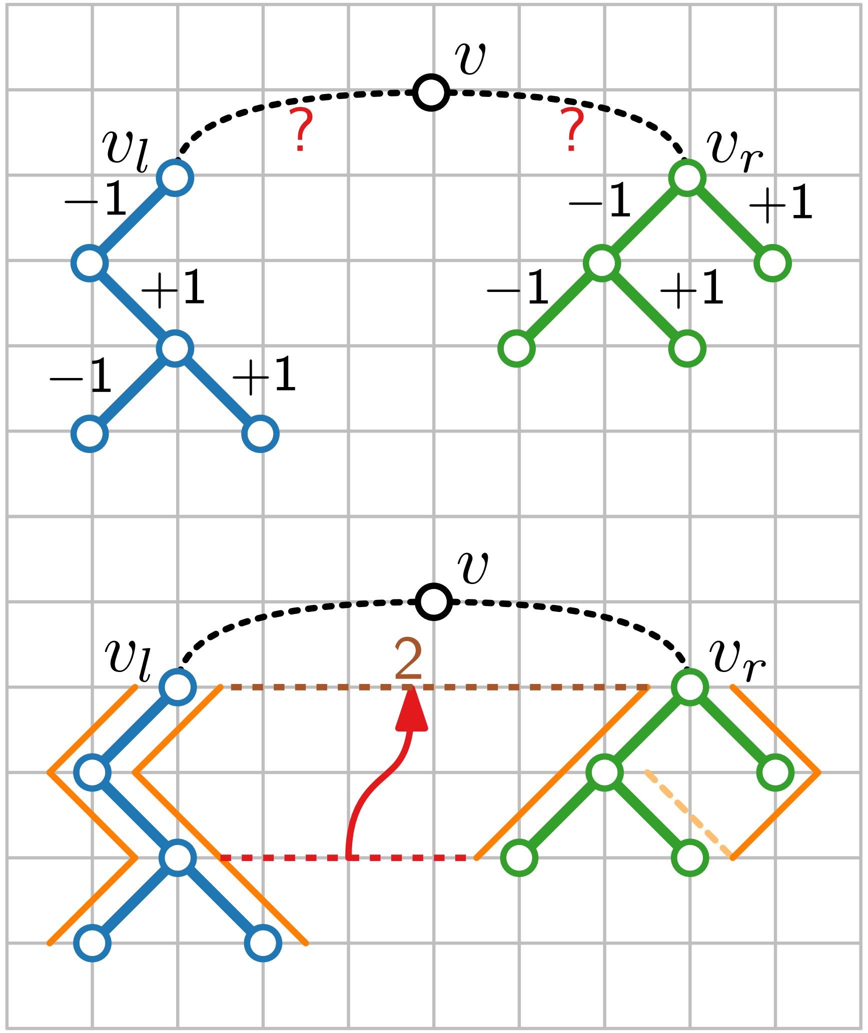
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



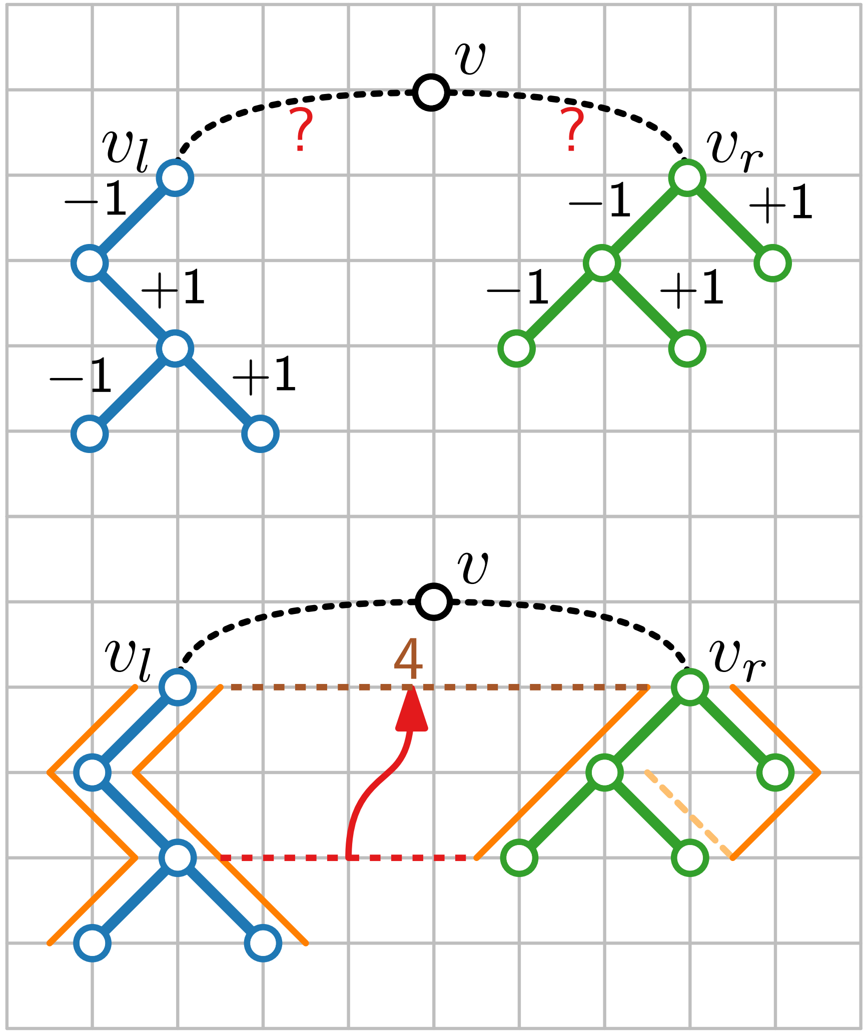
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



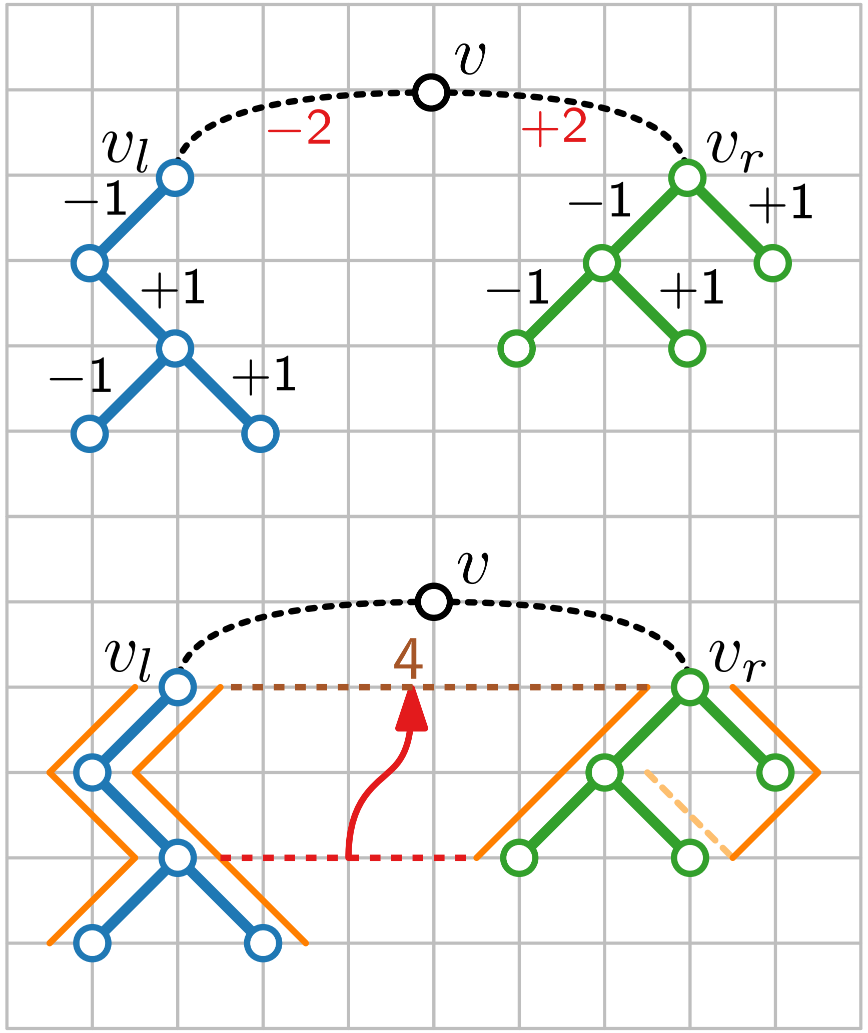
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



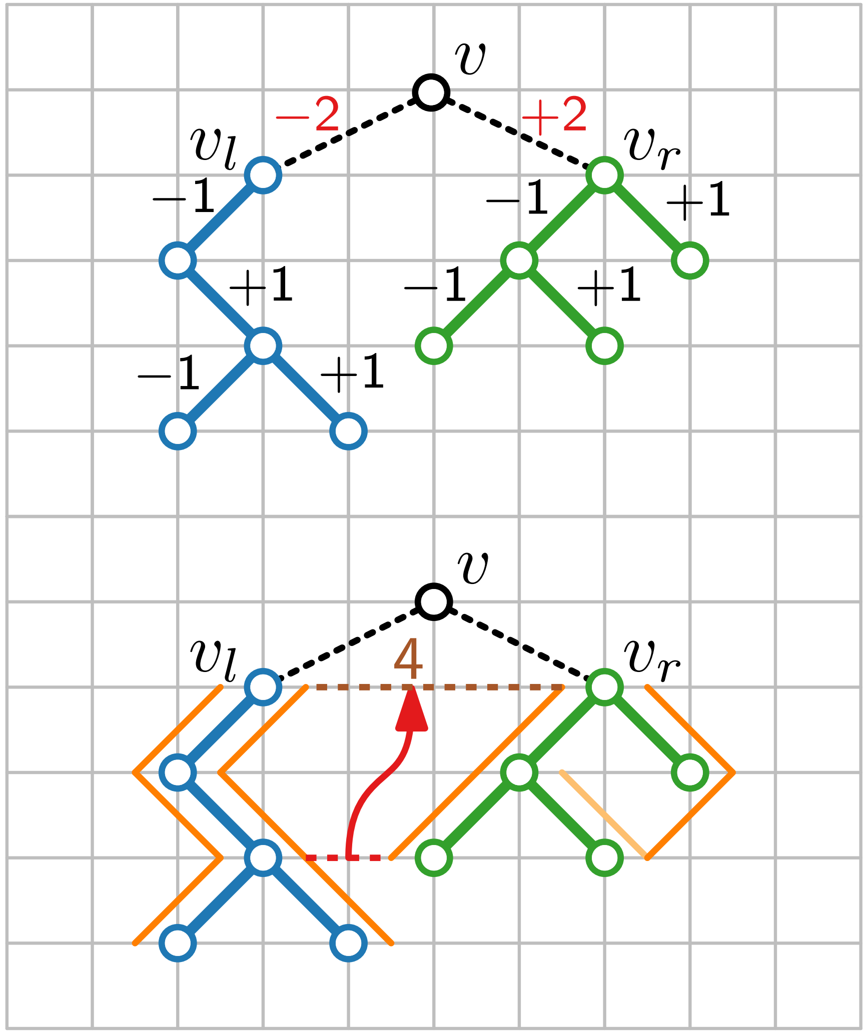
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



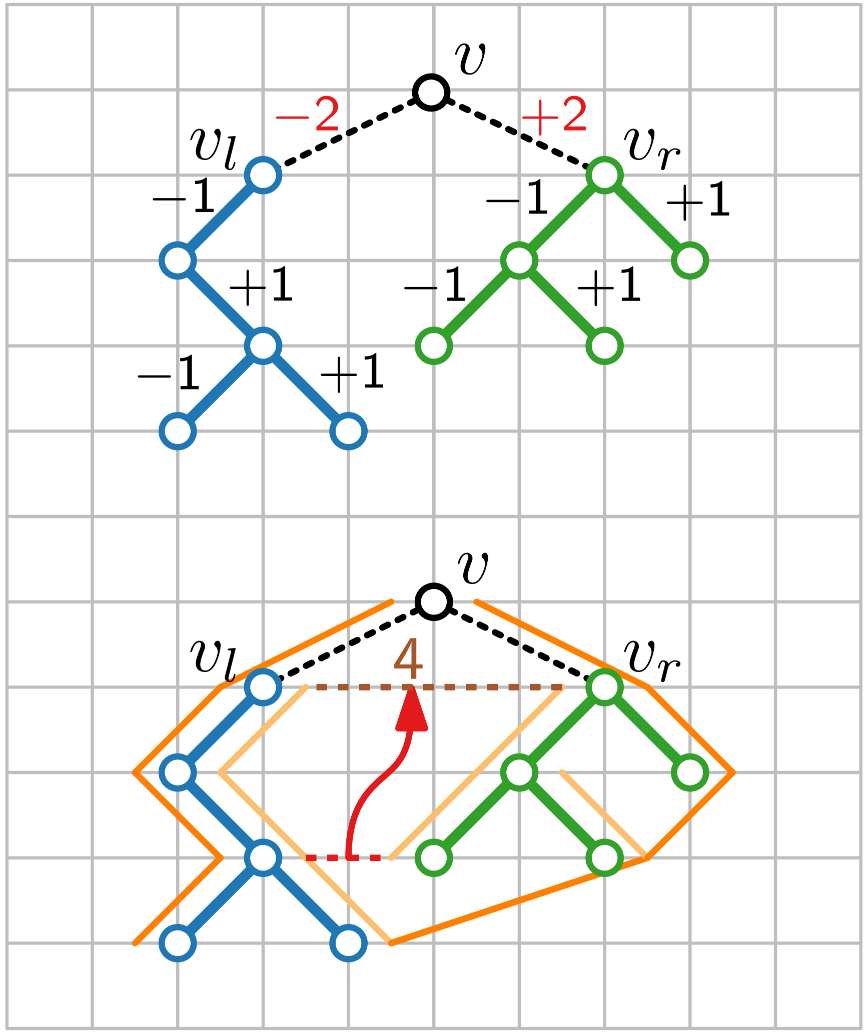
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of sub-tree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

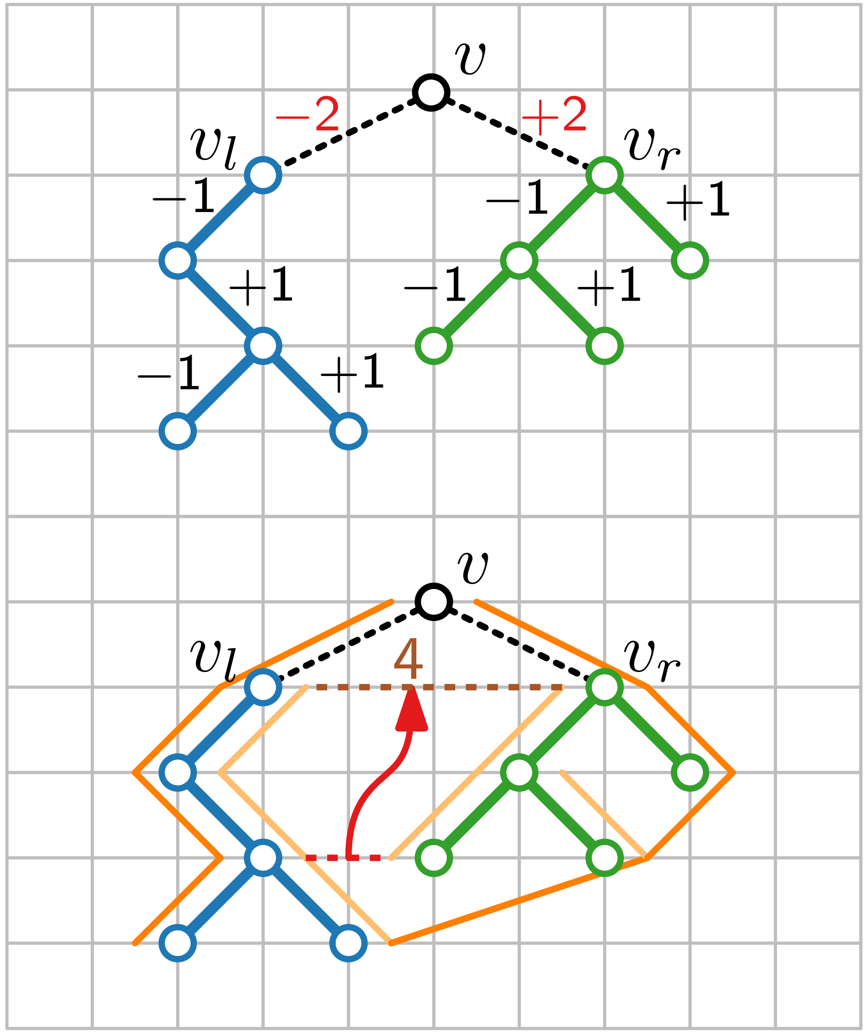
- Compute x- and y-coordinates



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.



Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

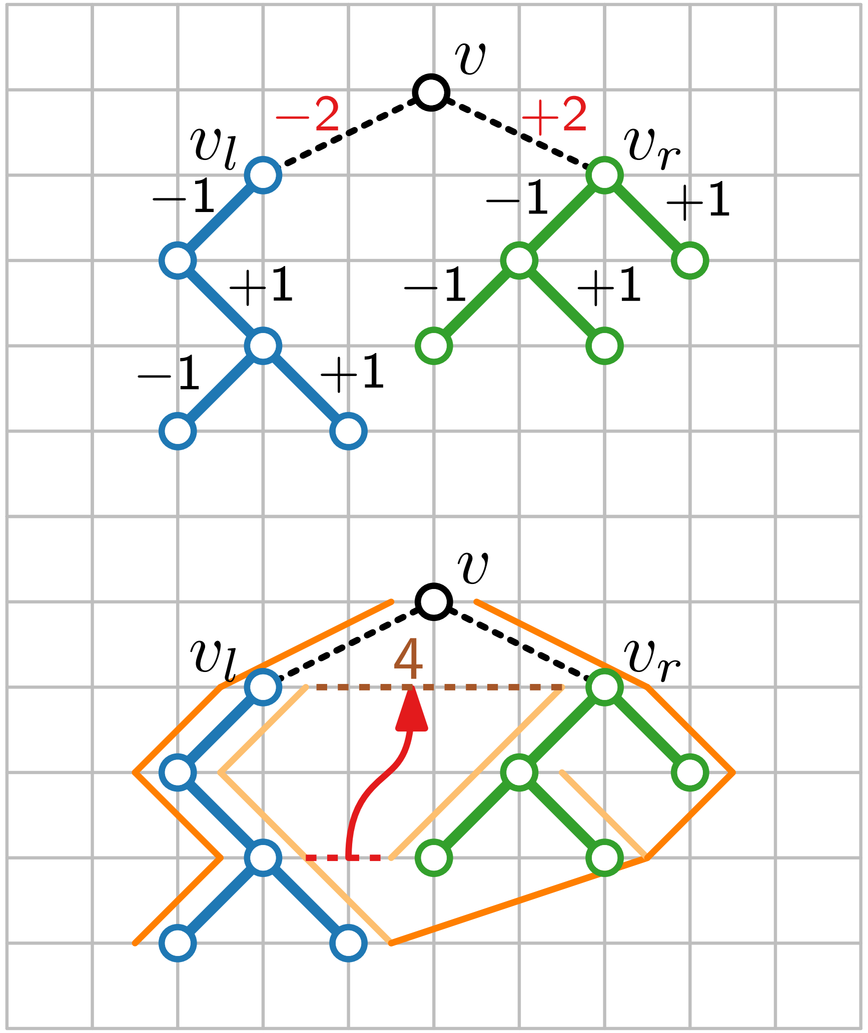
- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of sub-tree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

- How often do we take a step along a contour?



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

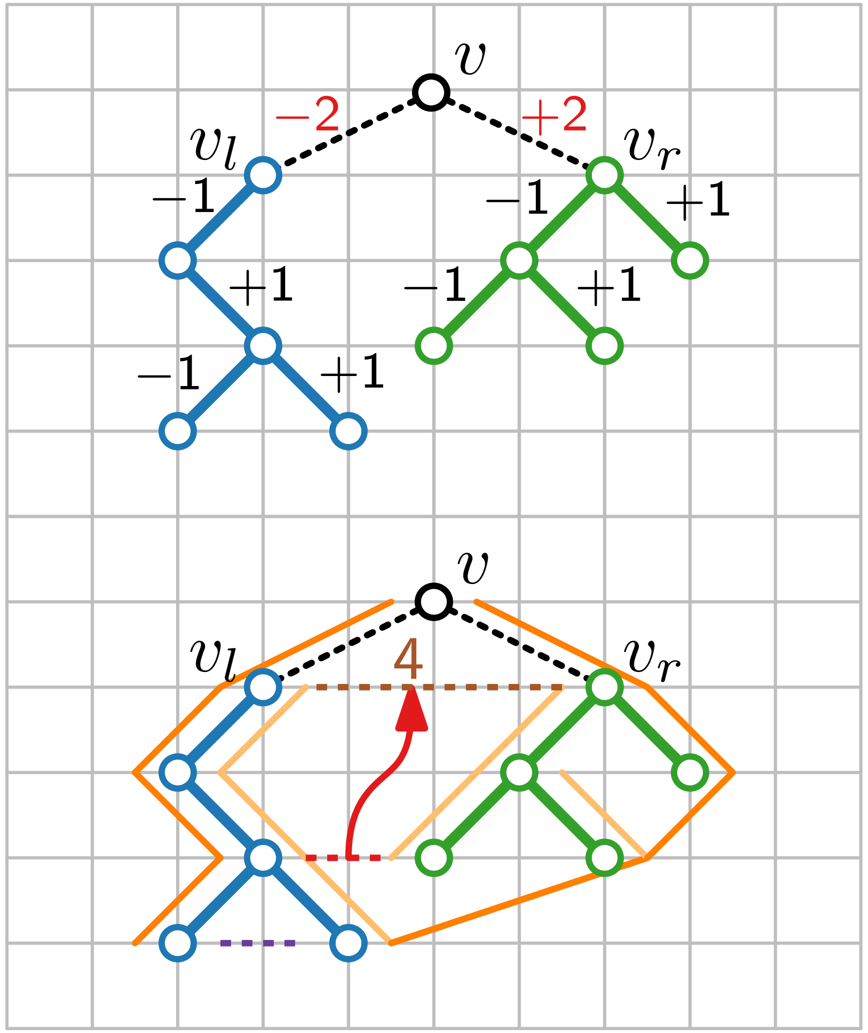
- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

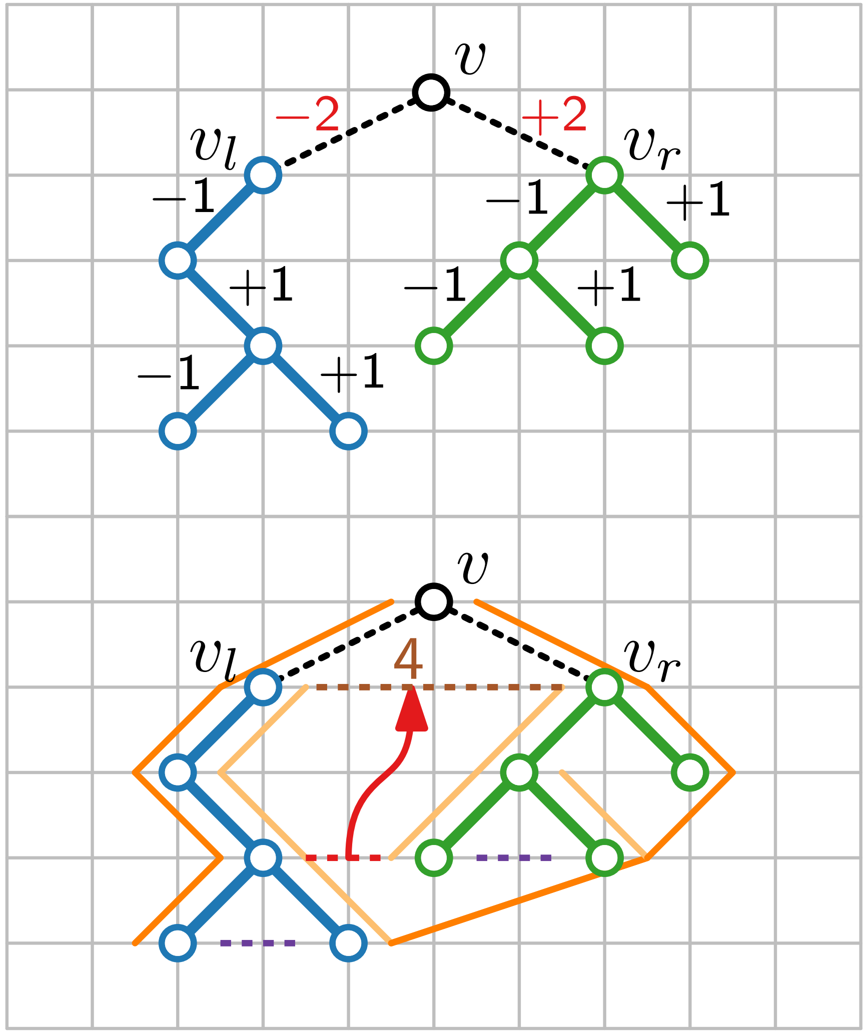
- Compute x- and y-coordinates

Runtime?

- How often do we take a step along a contour?



Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

- How often do we take a step along a contour?

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

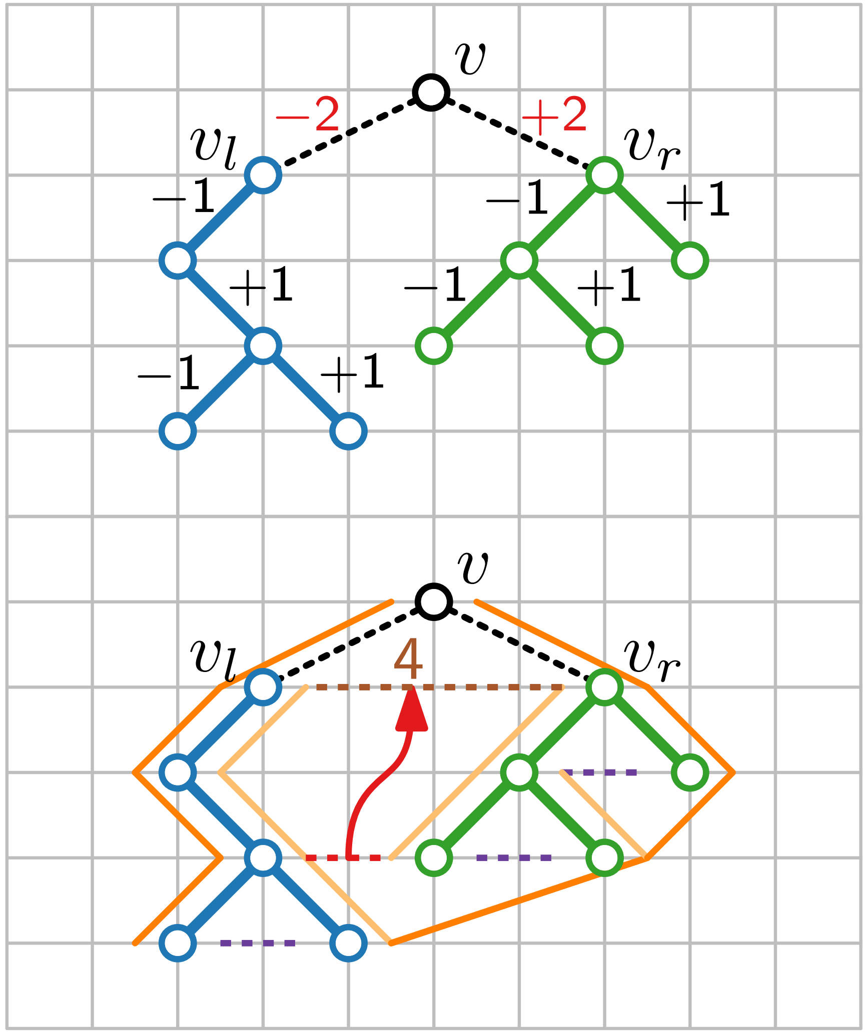
- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

- How often do we take a step along a contour?



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

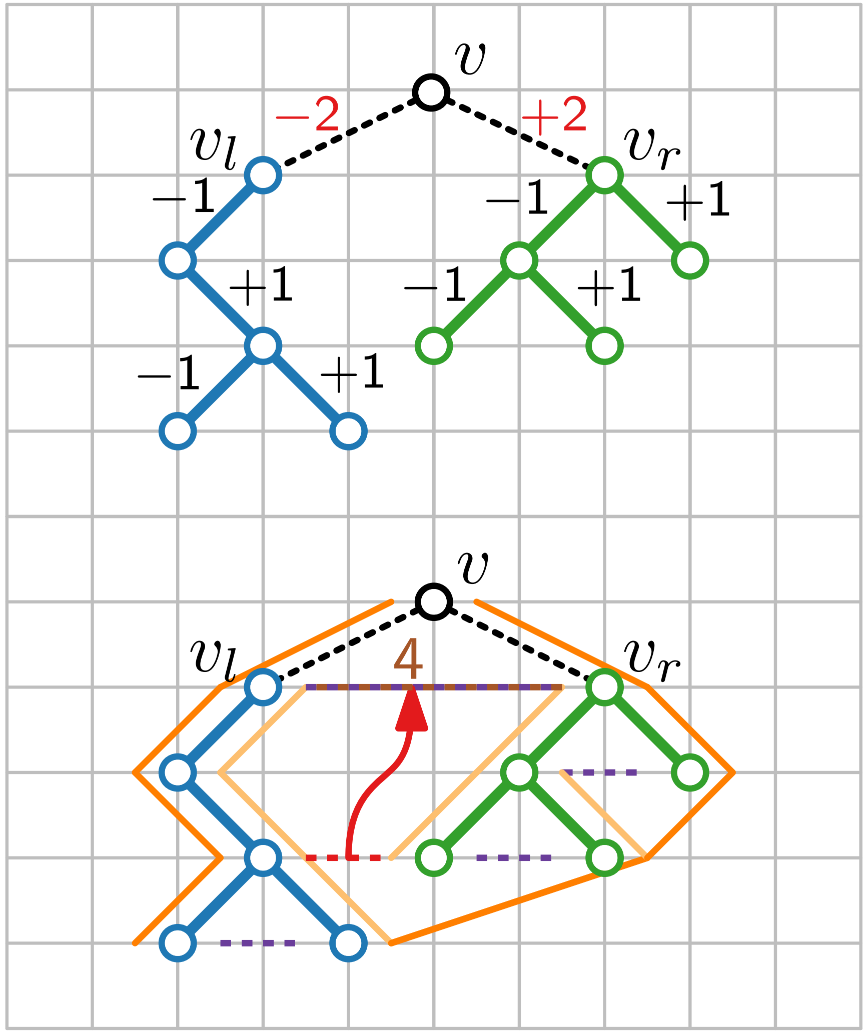
- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of sub-tree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

- How often do we take a step along a contour?



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

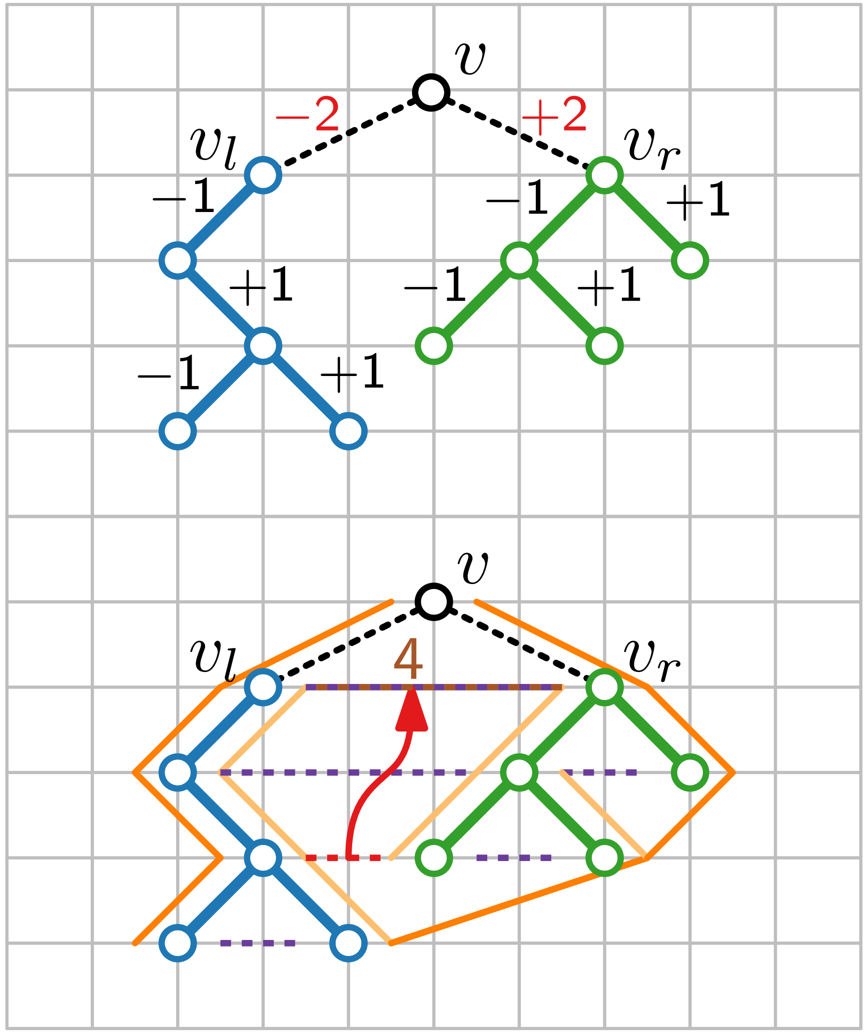
- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

- How often do we take a step along a contour?



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

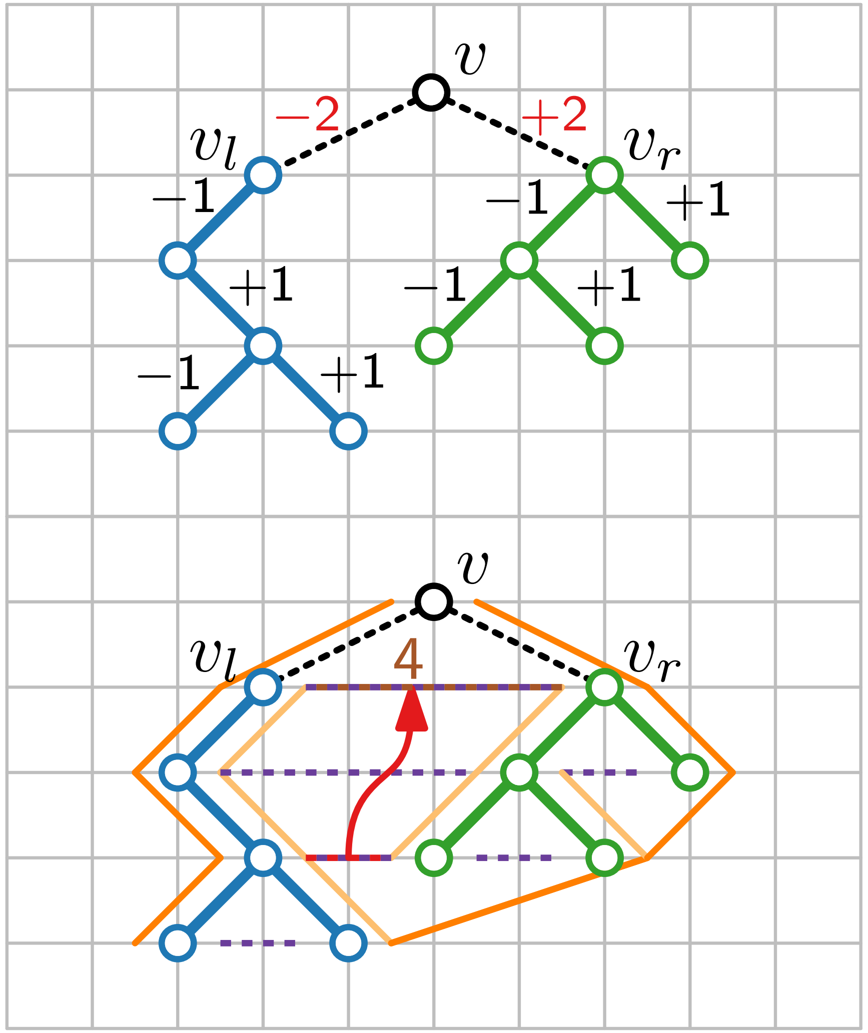
- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

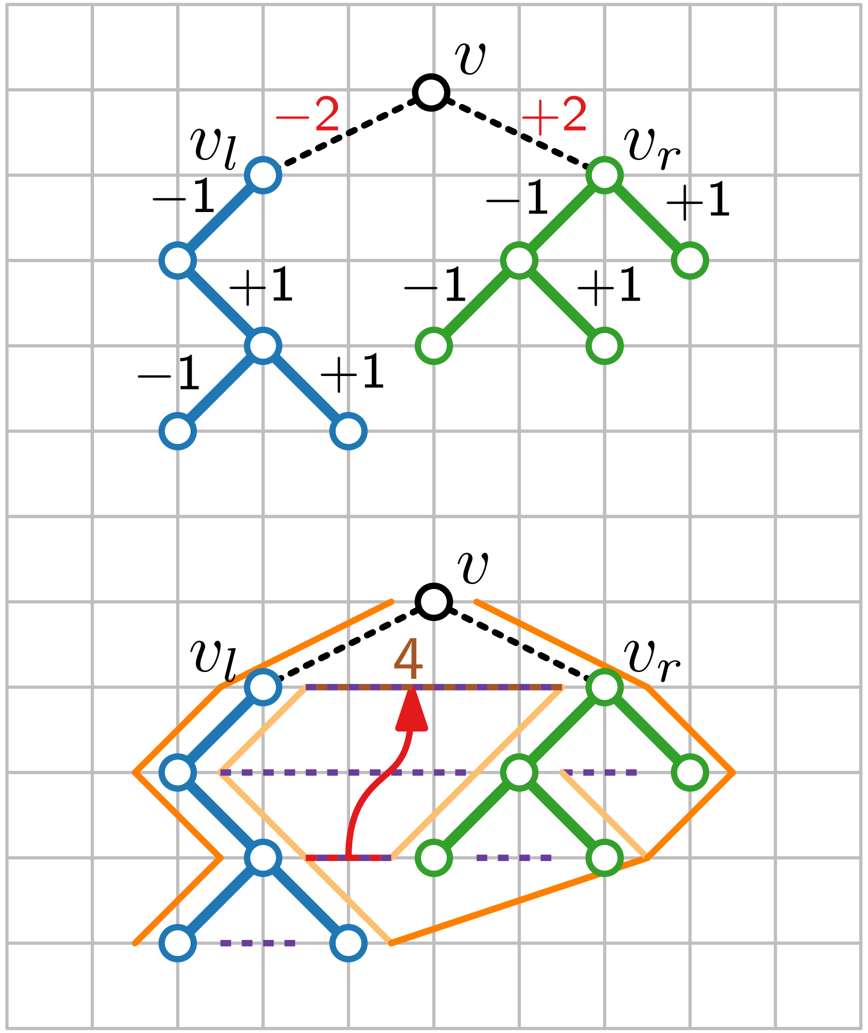
- Compute x- and y-coordinates

Runtime?

- How often do we take a step along a contour?



Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex v, compute horizontal displacement of left child v_l and right child v_r.
- x-offset$(v_l) = -\lceil d_v/2 \rceil$, x-offset$(v_r) = \lceil d_v/2 \rceil$
- At every vertex v store left and right contour of subtree $T(v)$.
- A contour is a linked list of vertex coordinates/offsets.
- Find d_v = min. horiz. distance between v_l and v_r.

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

- How often do we take a step along a contour? *in total $\mathcal{O}(n)$ times!* where $n = \#$ vertices

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\mathrm{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$

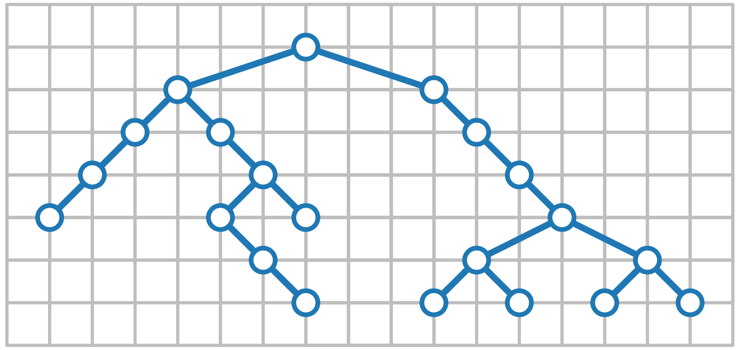
Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

Layered Drawings – Result

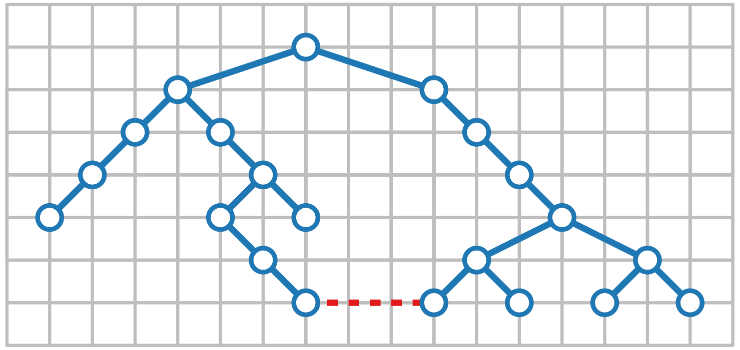


Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

Layered Drawings – Result

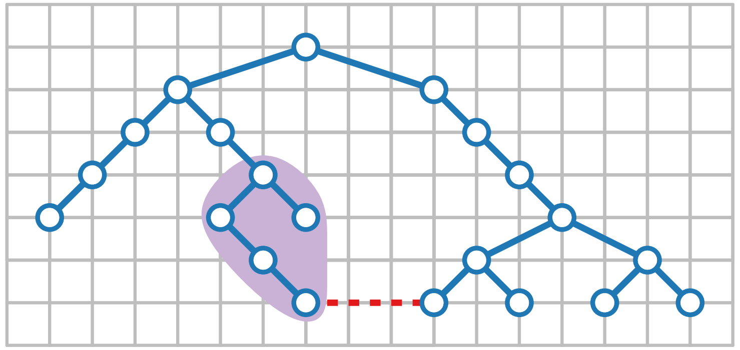


Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-$depth(v)
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

Layered Drawings – Result



Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

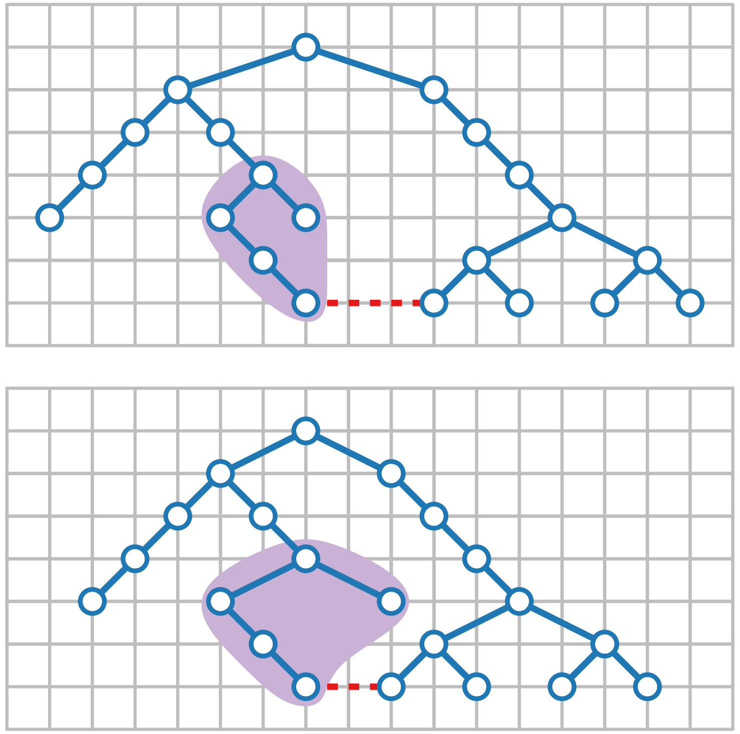
- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-$depth(v)
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-$depth(v)
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

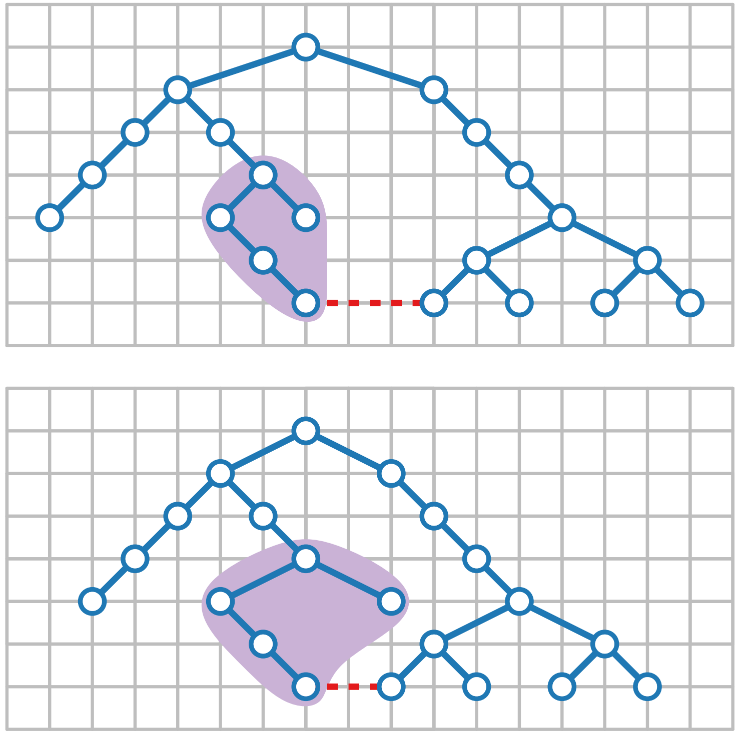


Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

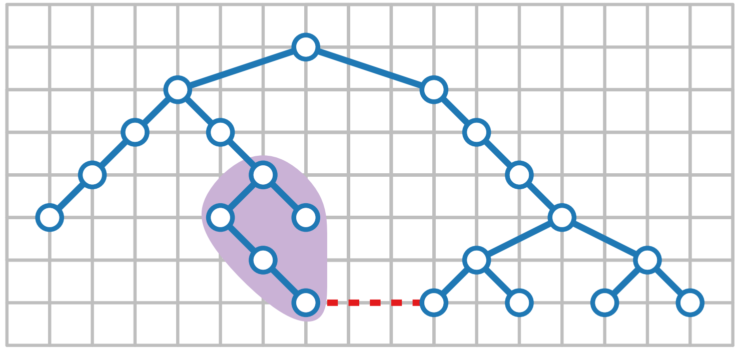
- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-$depth(v)
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard

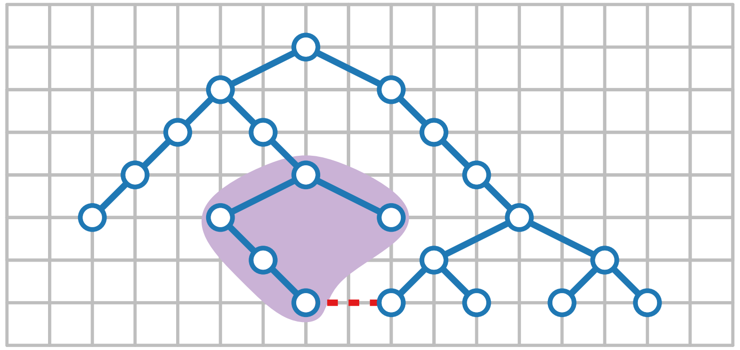


Layered Drawings – Result

Theorem. [Reingold & Tilford '81]
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation

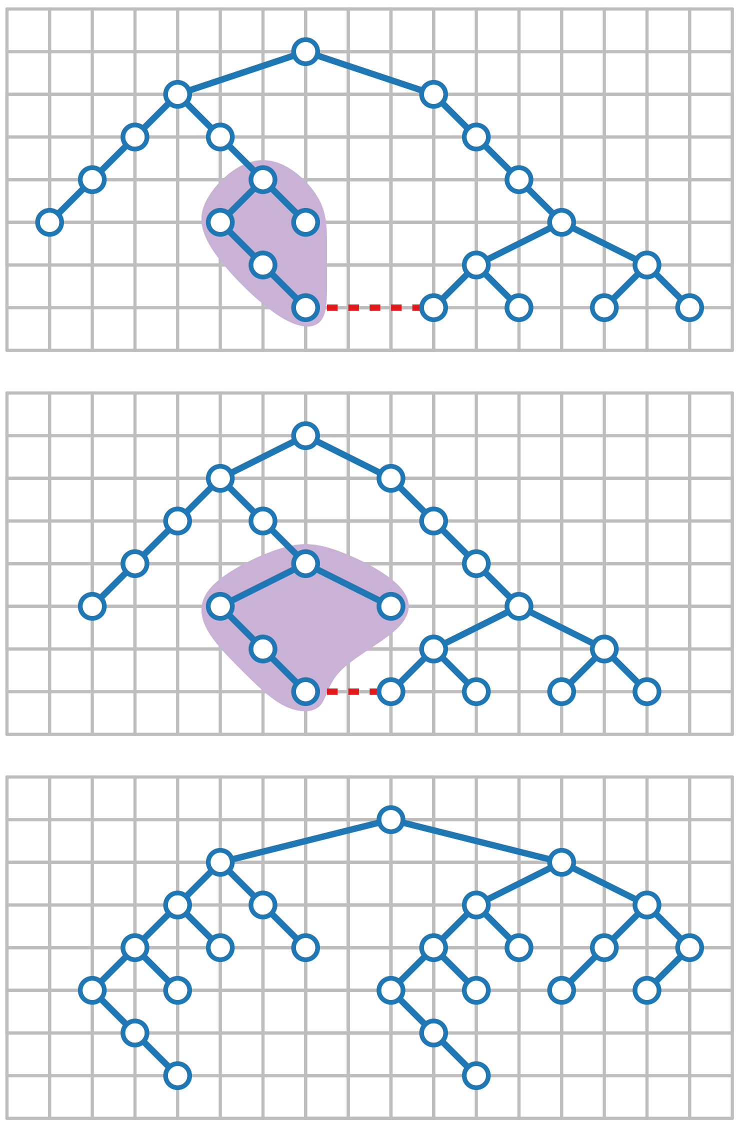




Layered Drawings – Result

Theorem. [Reingold & Tilford '81]
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

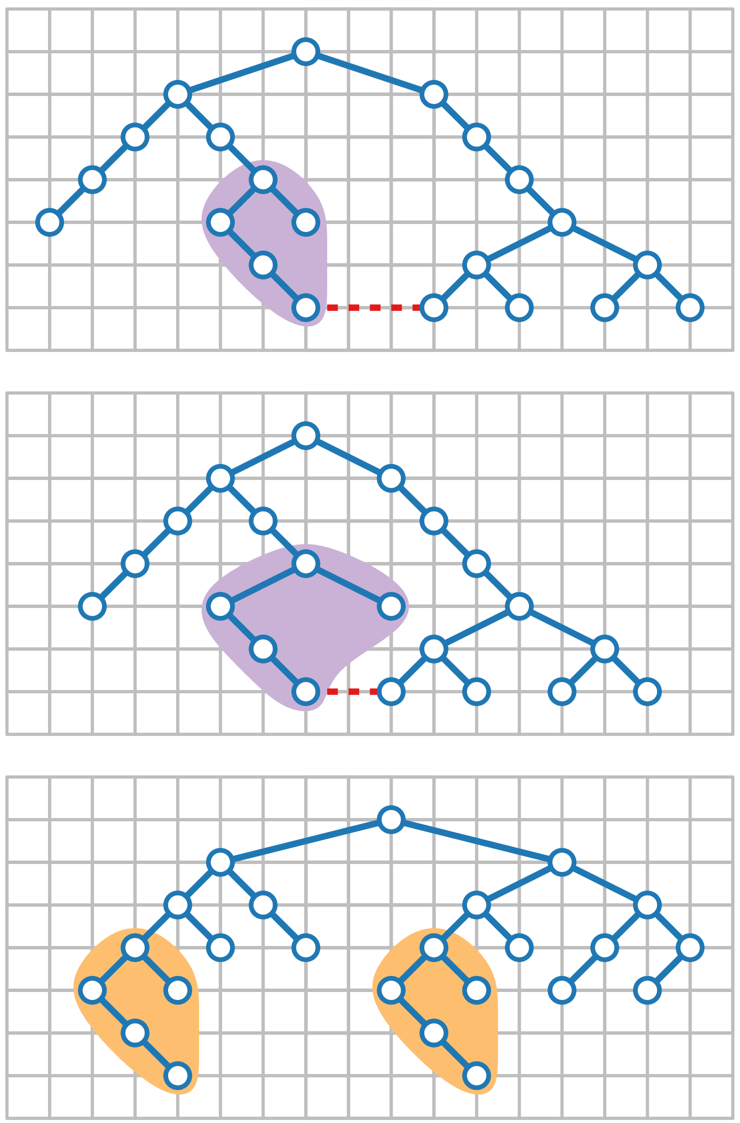
- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation



Layered Drawings – Result

Theorem. [Reingold & Tilford '81]
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

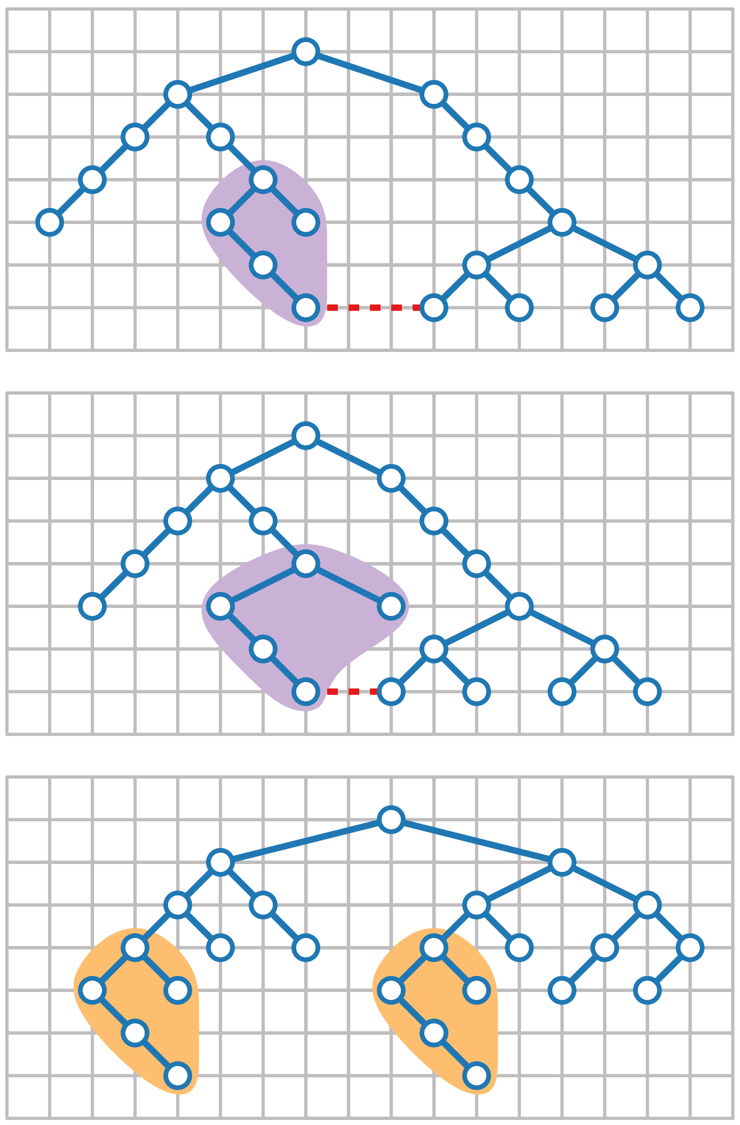
- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation



Layered Drawings – Result

Theorem. [Reingold & Tilford '81]
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection

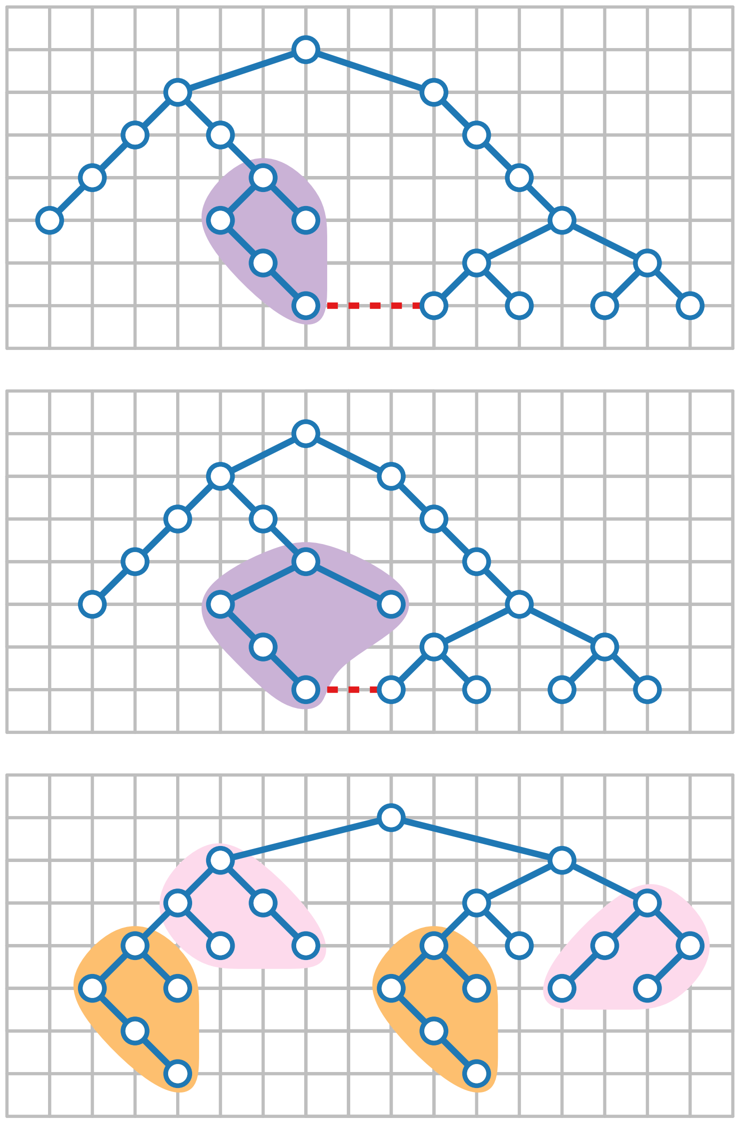


Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection

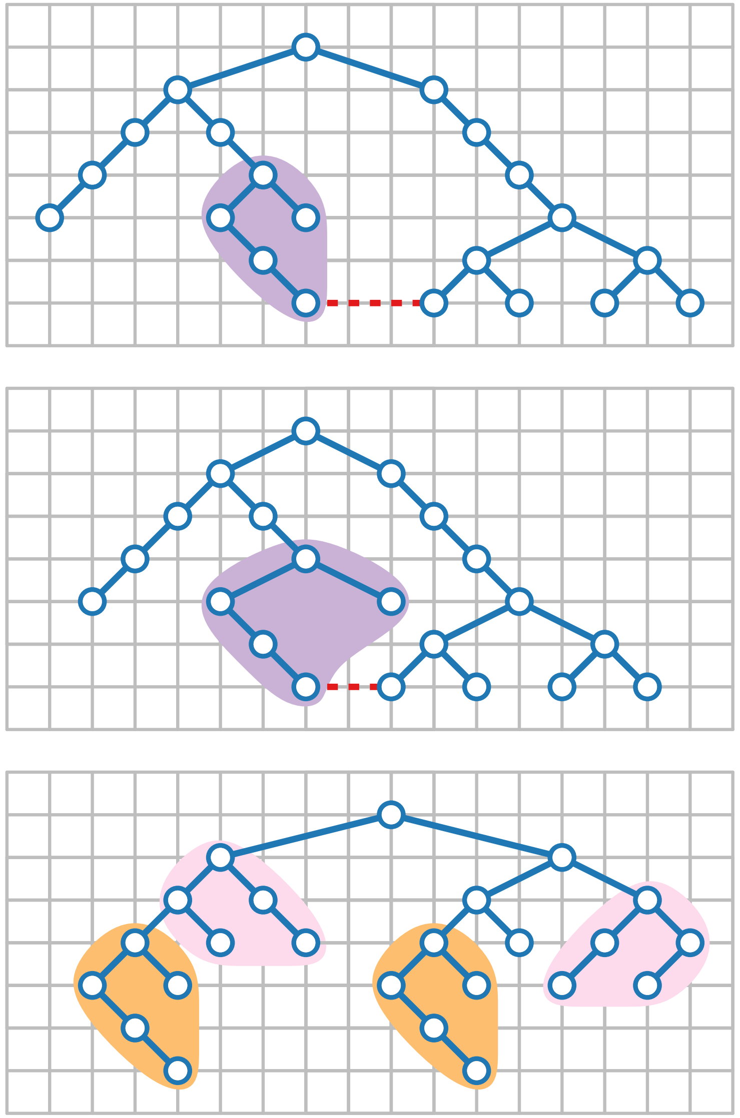


Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection

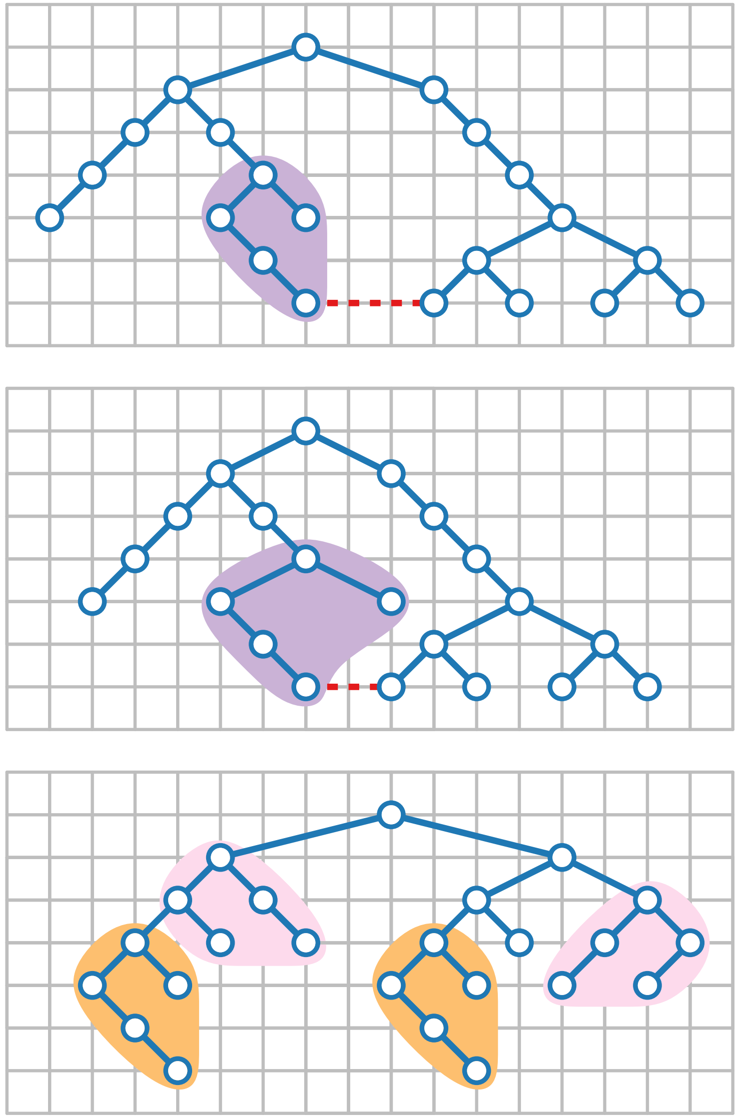


Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection



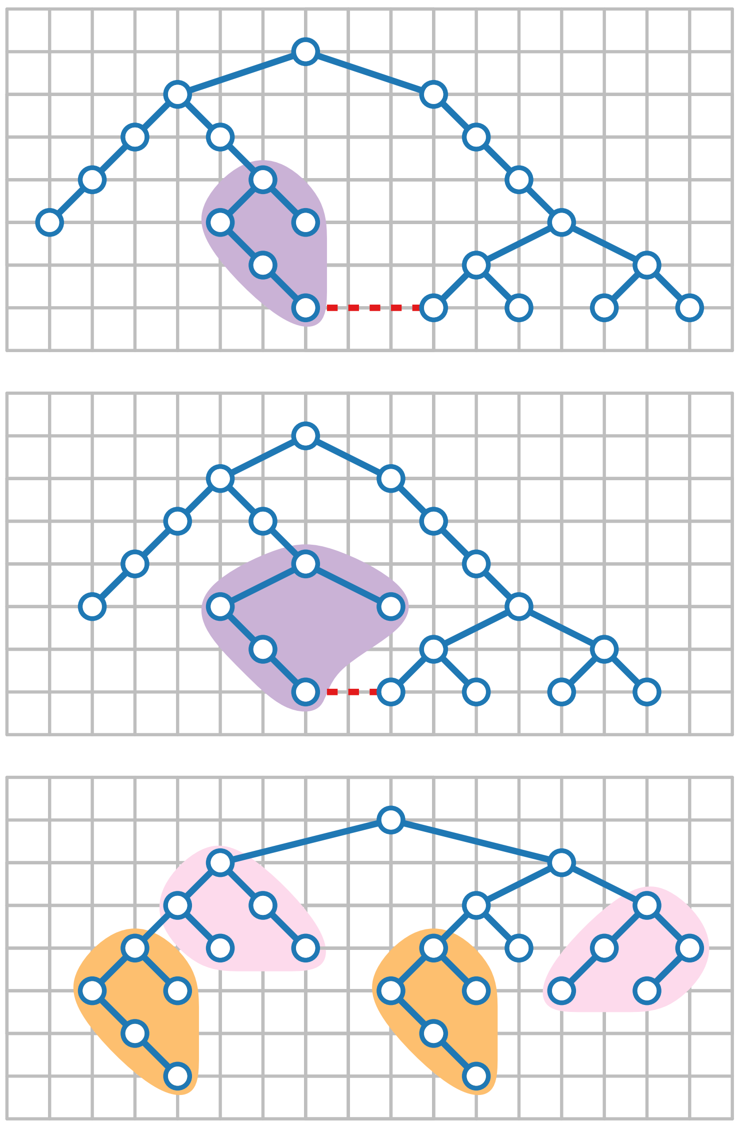
extension to non-binary rooted trees

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection



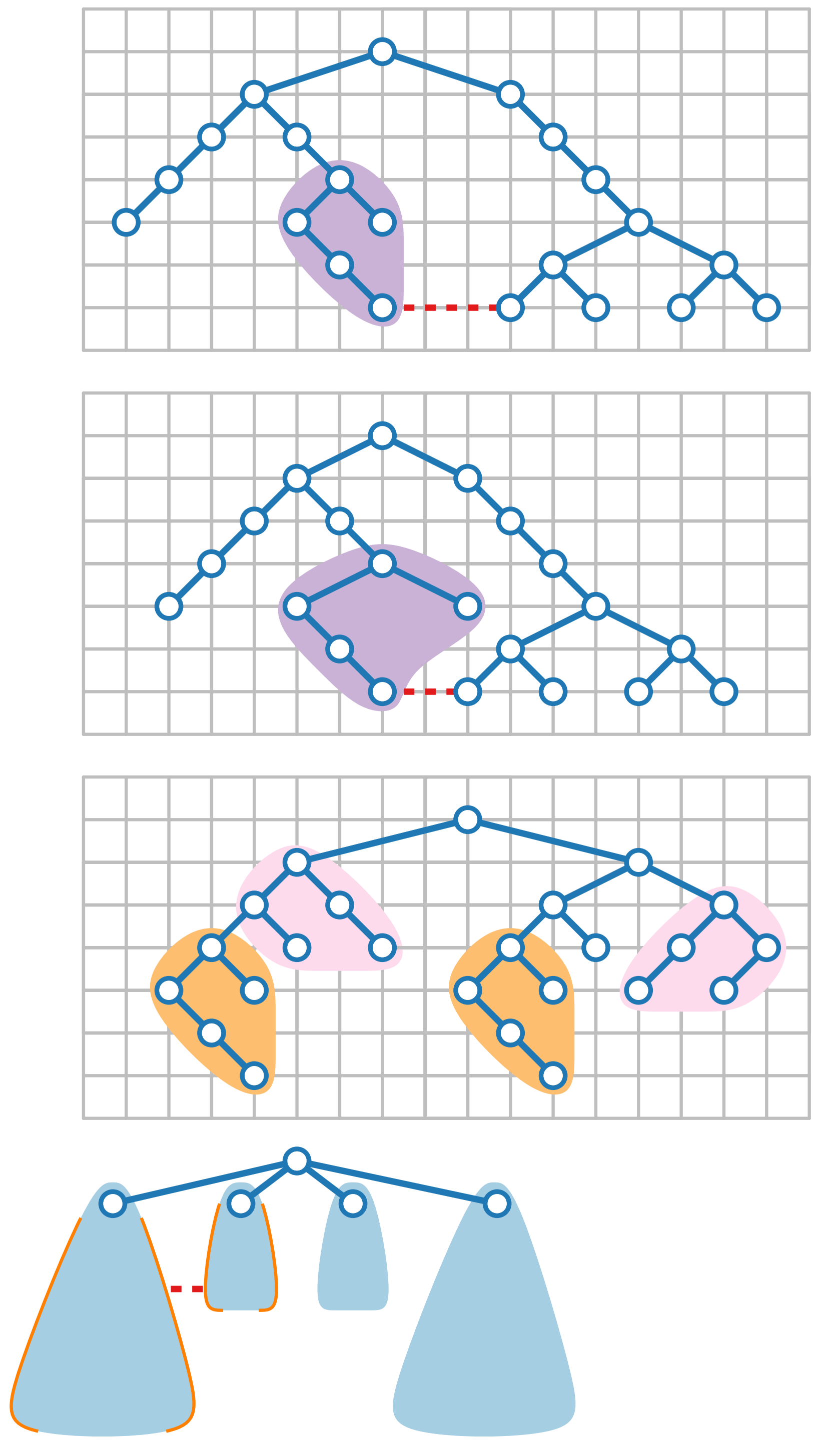
extension to non-binary rooted trees

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-$depth(v)
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection



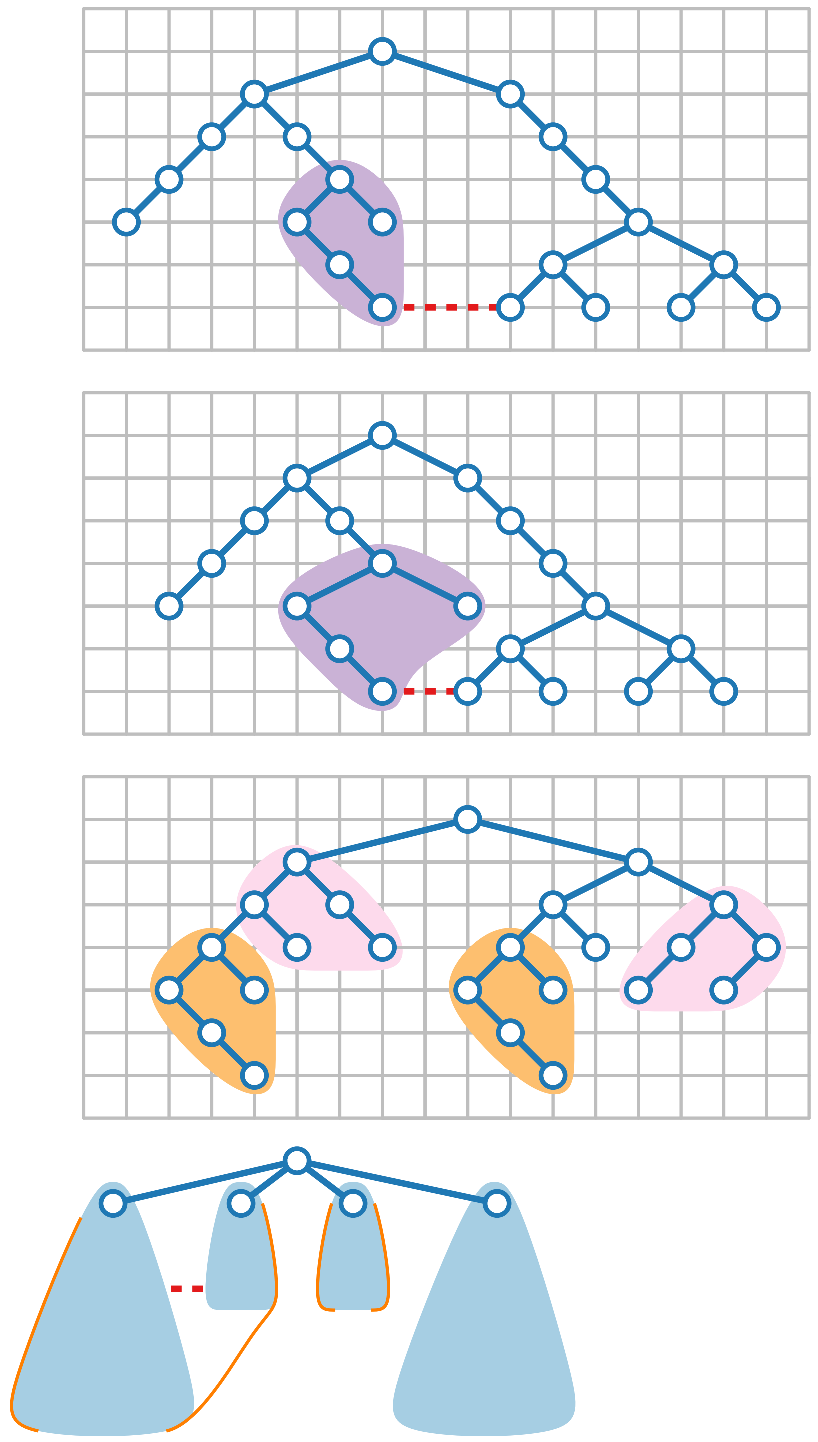
extension to non-binary rooted trees

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection



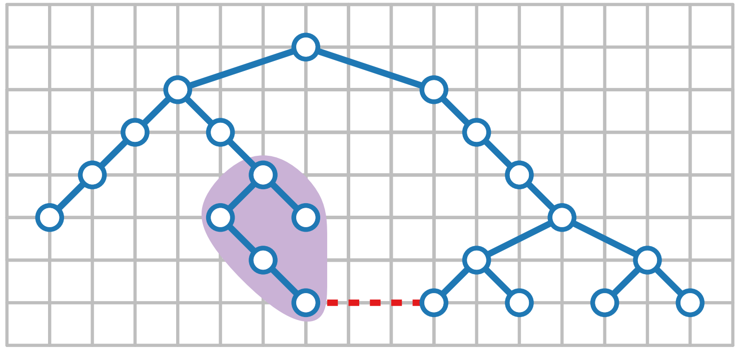
extension to non-binary rooted trees

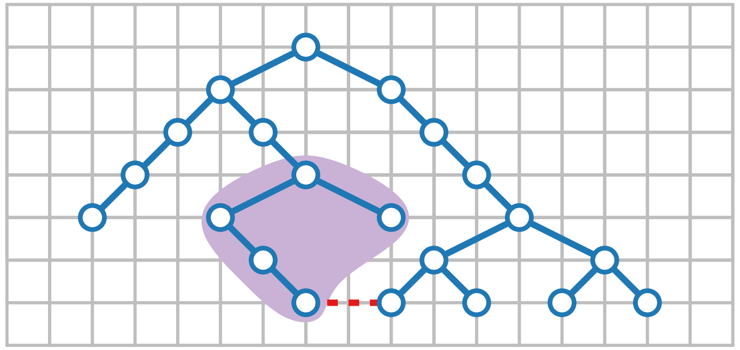
Layered Drawings – Result

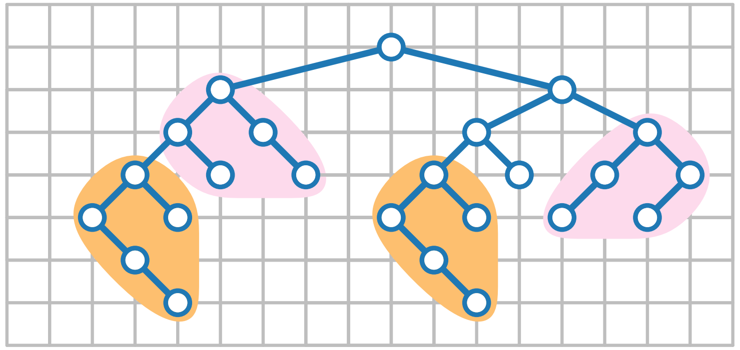
Theorem. [Reingold & Tilford '81]

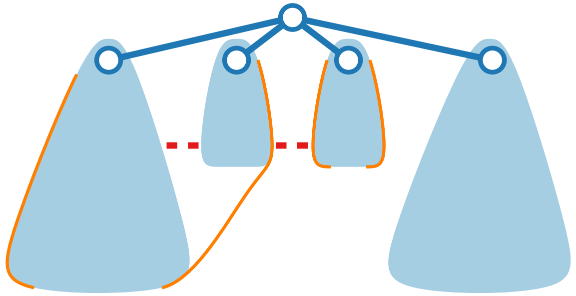
Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection









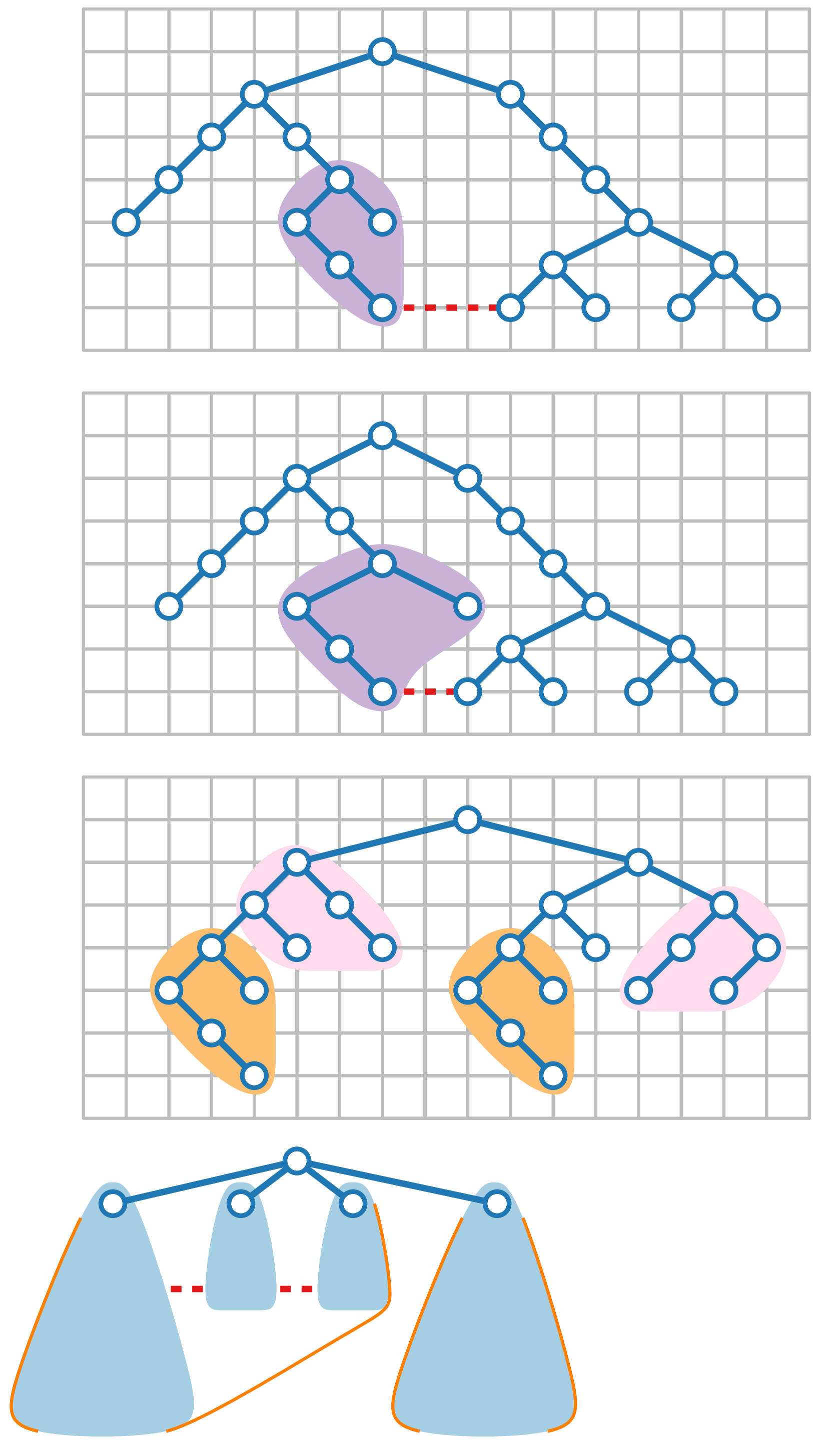
extension to non-binary rooted trees

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection



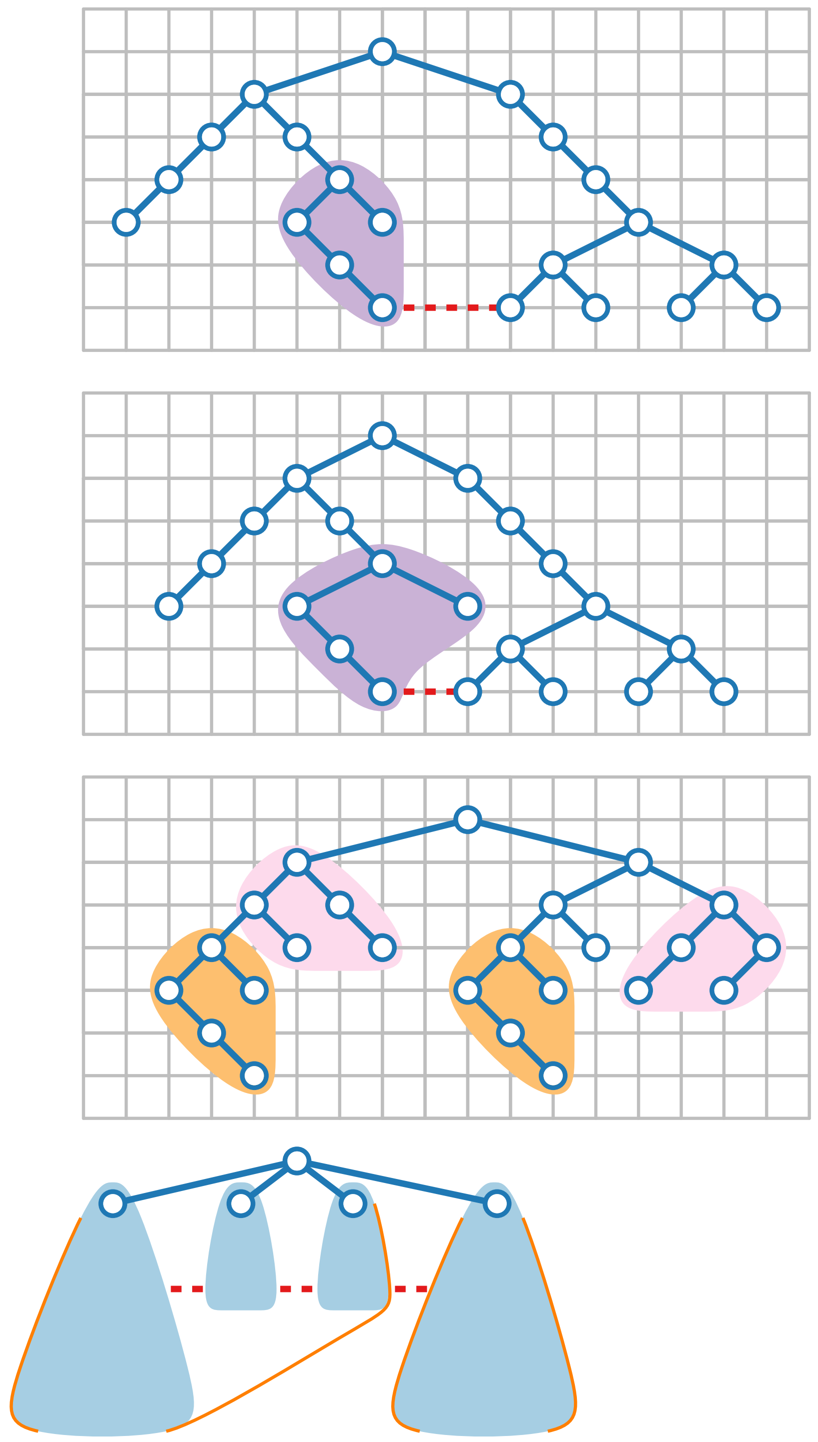
extension to non-binary rooted trees

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection



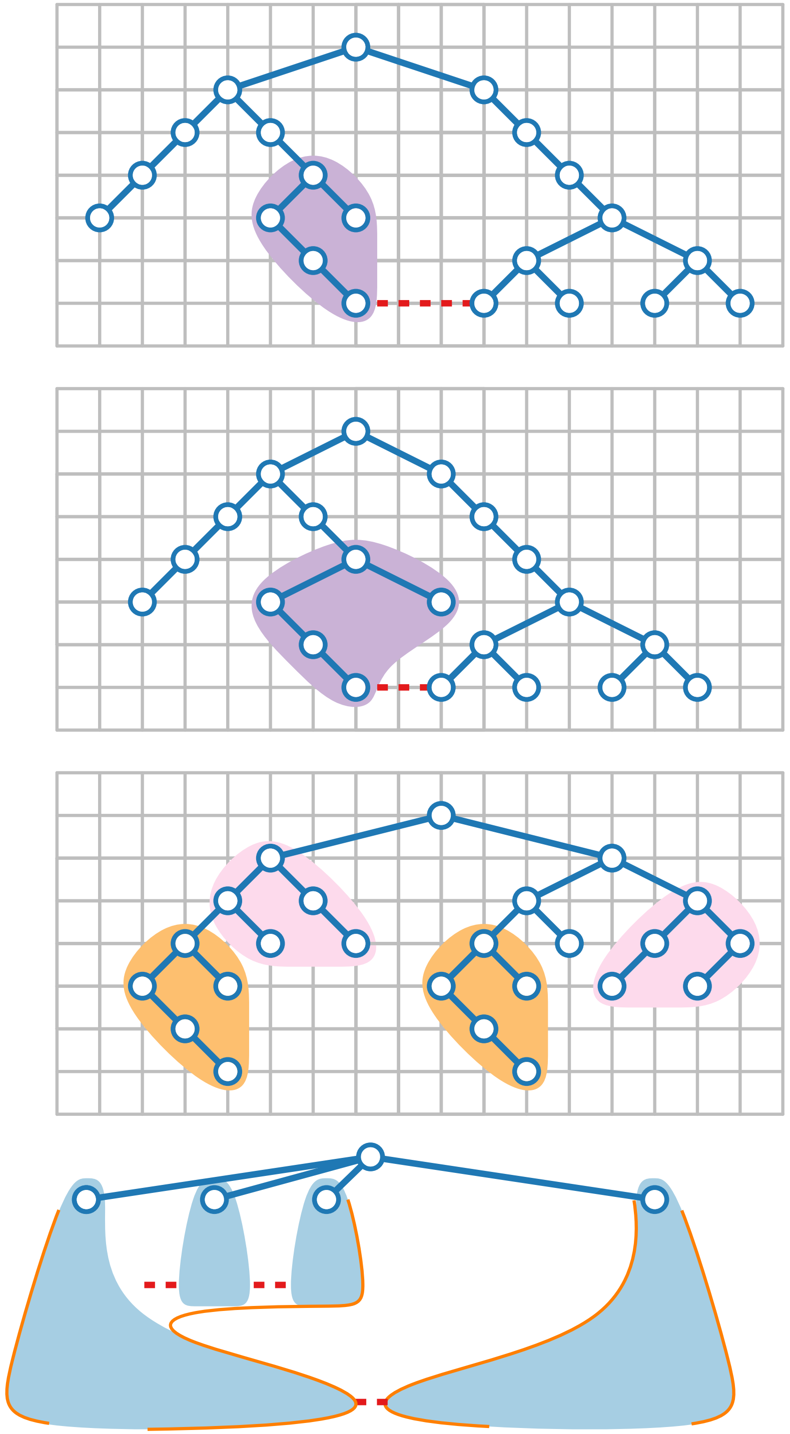
extension to non-binary rooted trees

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection



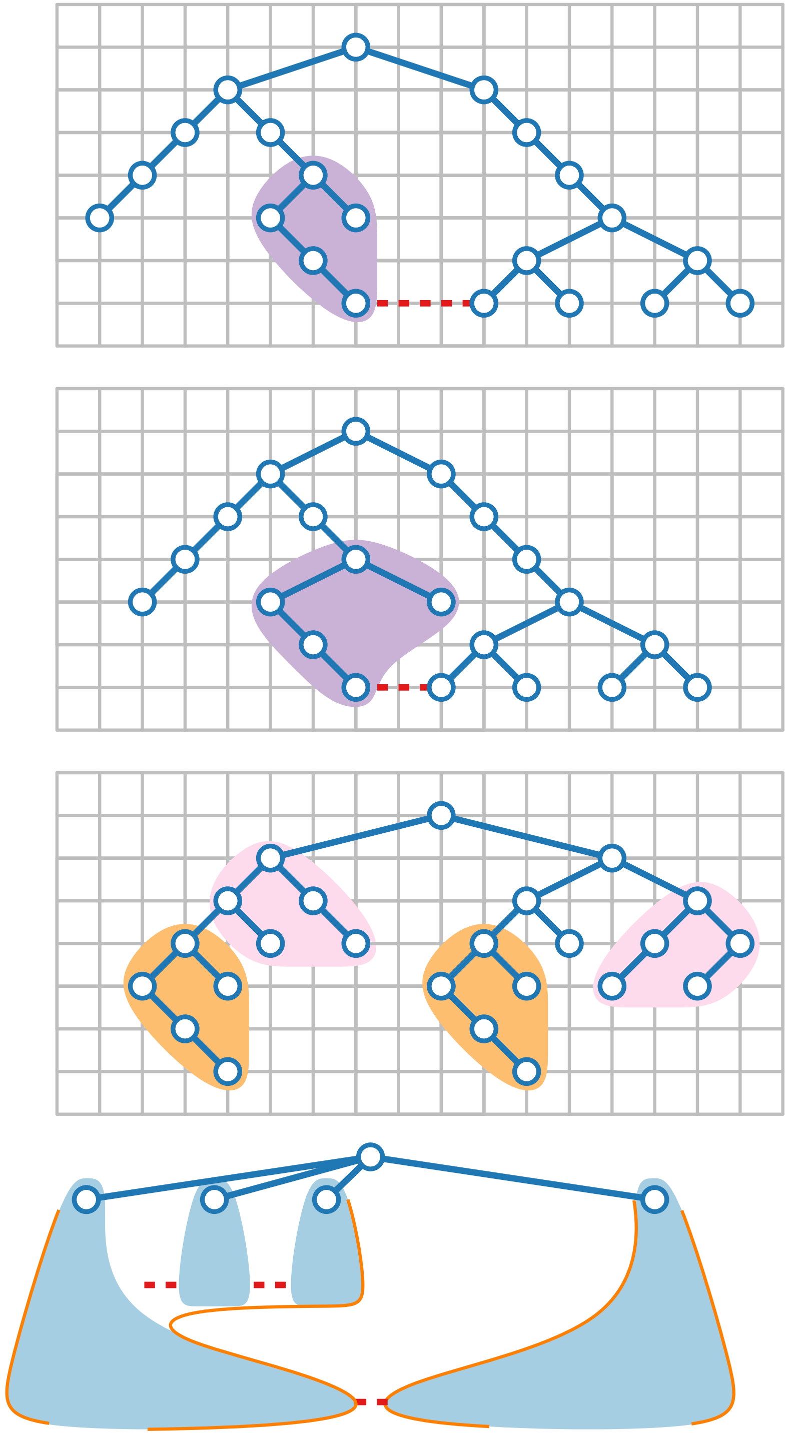
extension to non-binary rooted trees

Layered Drawings – Result

Theorem. [Reingold & Tilford '81]

Let T be a ~~binary~~ rooted tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time such that:

- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and vertical distances are at least 1
- Each vertex with > 1 child is centered w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! ← NP-hard
- Simply isomorphic subtrees have congruent drawings, up to translation
- ~~Axially isomorphic subtrees have congruent drawings, up to translation and reflection~~



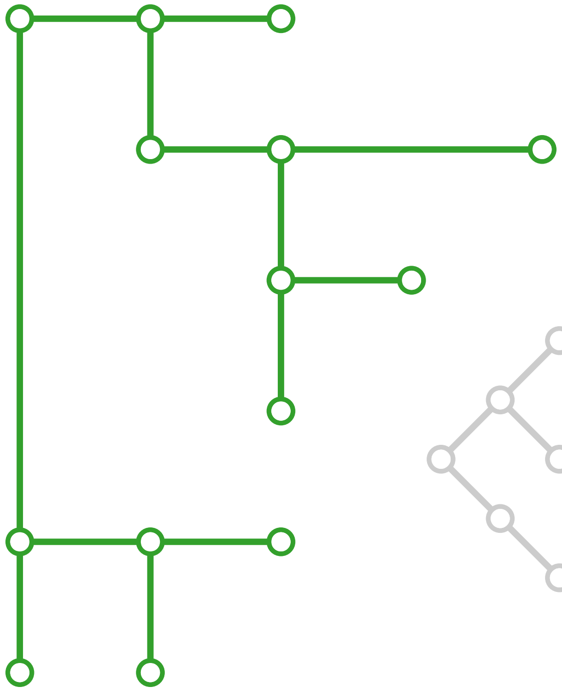
extension to non-binary rooted trees

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Visualization of Graphs

Lecture 1b:
Drawing Trees

Part II:
HV-Drawings



HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP

HV-Drawings – Drawing Style

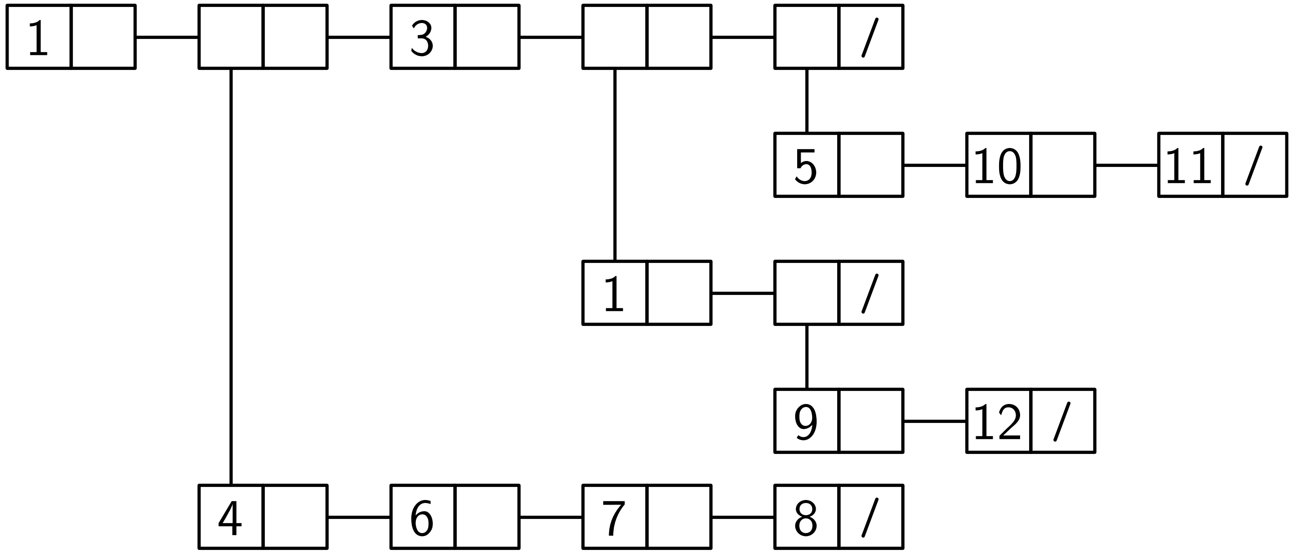
Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values

HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values

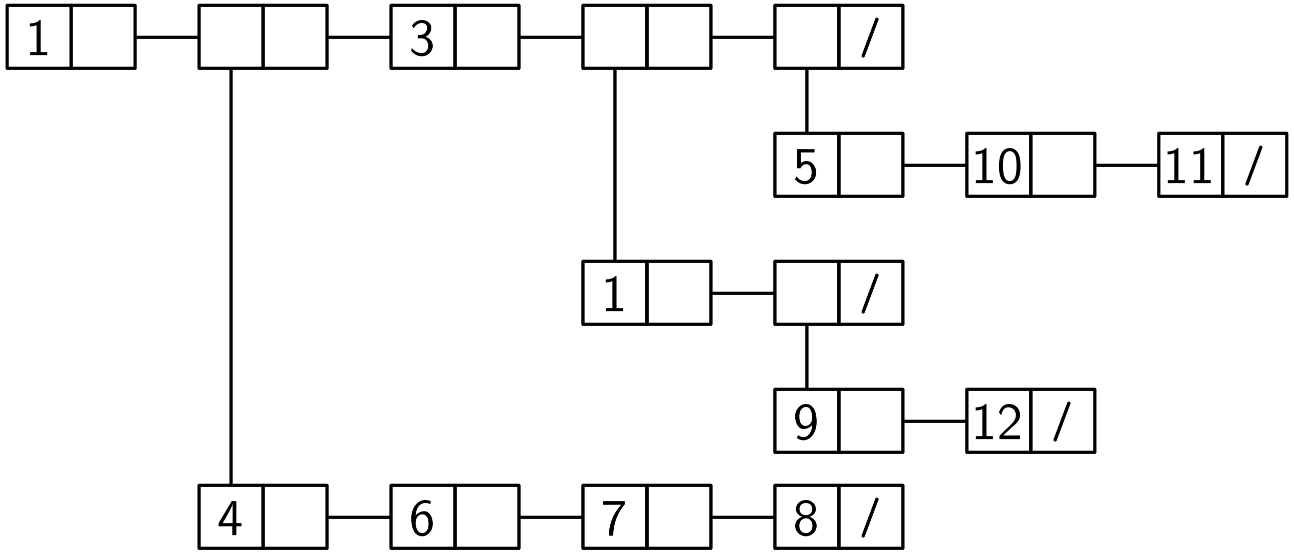


Source: after `gajon.org/trees-linked-lists-common-lisp/`

HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

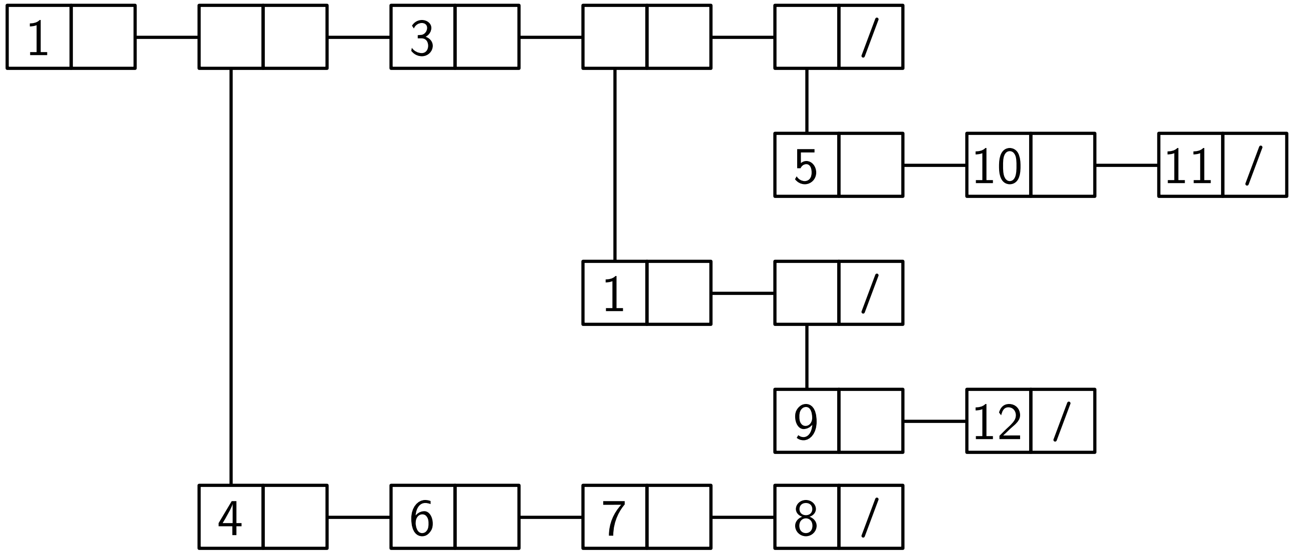
Drawing conventions

Drawing aesthetics to optimize

HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

Drawing conventions

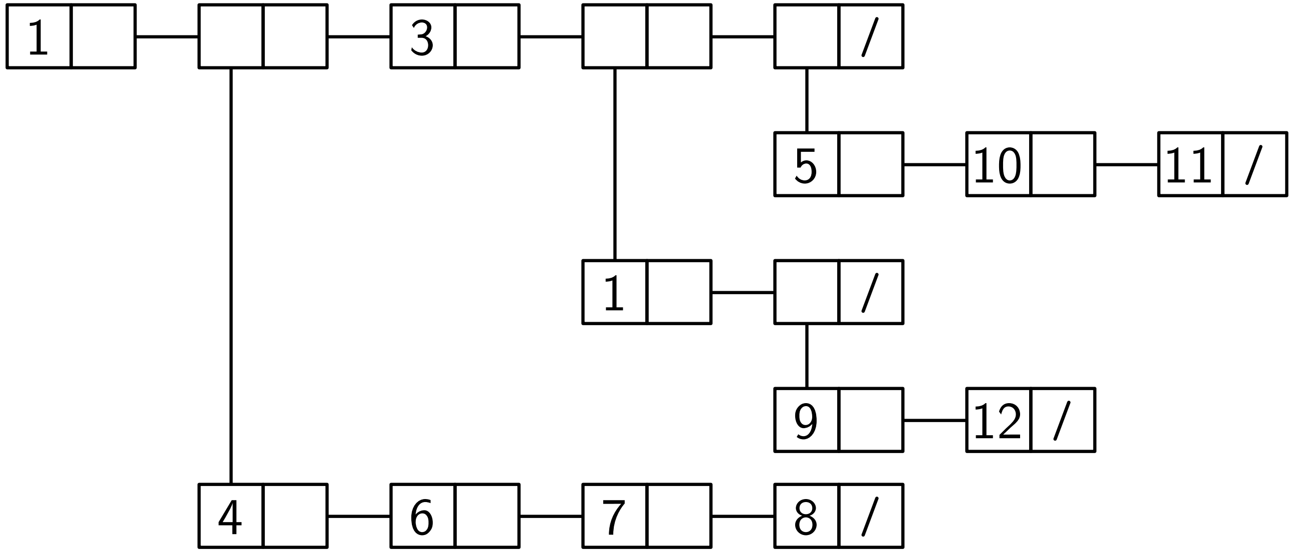
- Children are vertically or horizontally aligned with their parent

Drawing aesthetics to optimize

HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

Drawing conventions

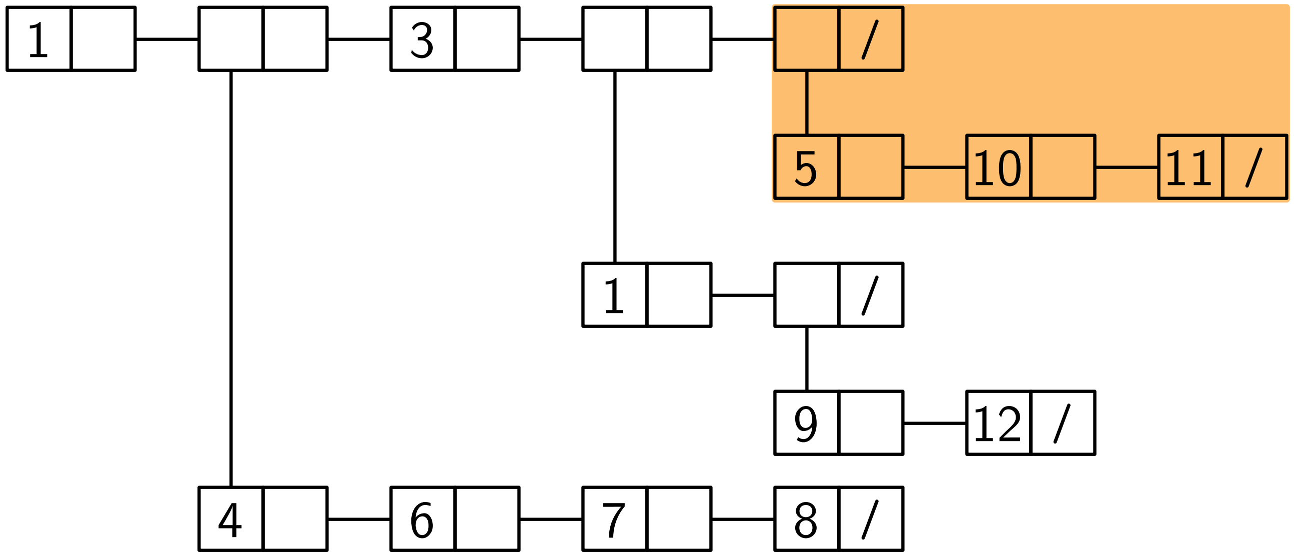
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics to optimize

HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

Drawing conventions

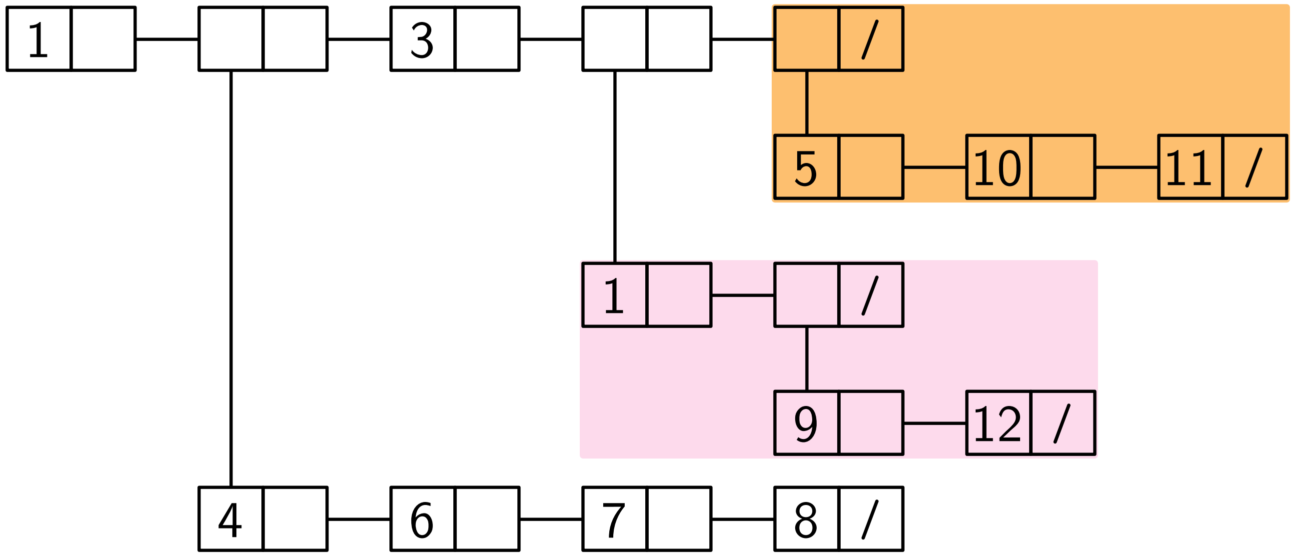
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics to optimize

HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

Drawing conventions

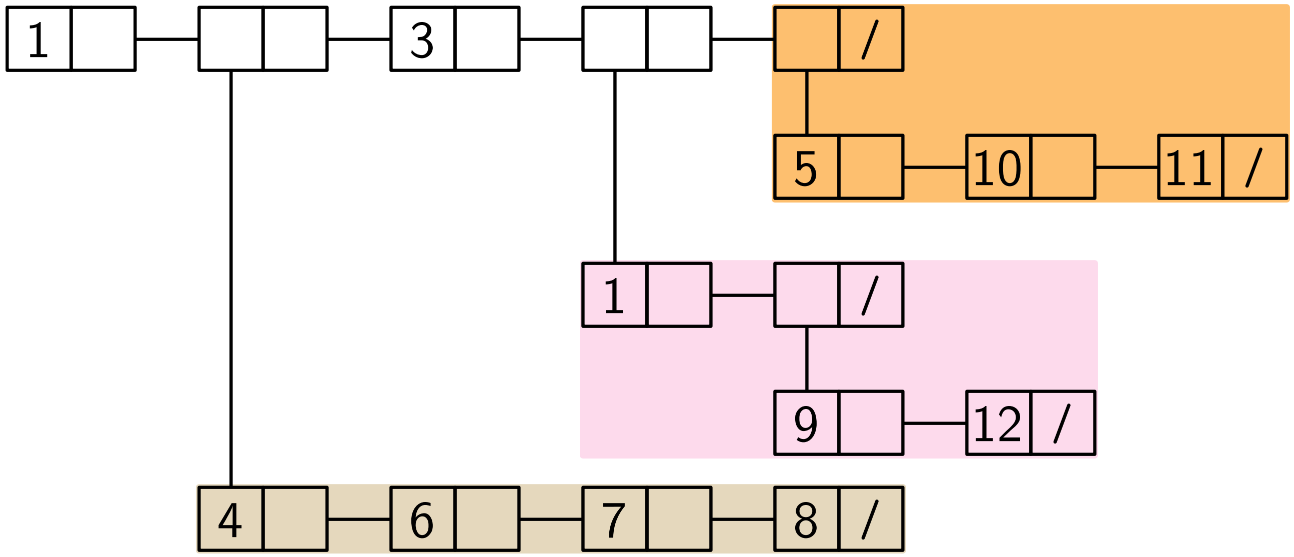
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics to optimize

HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

Drawing conventions

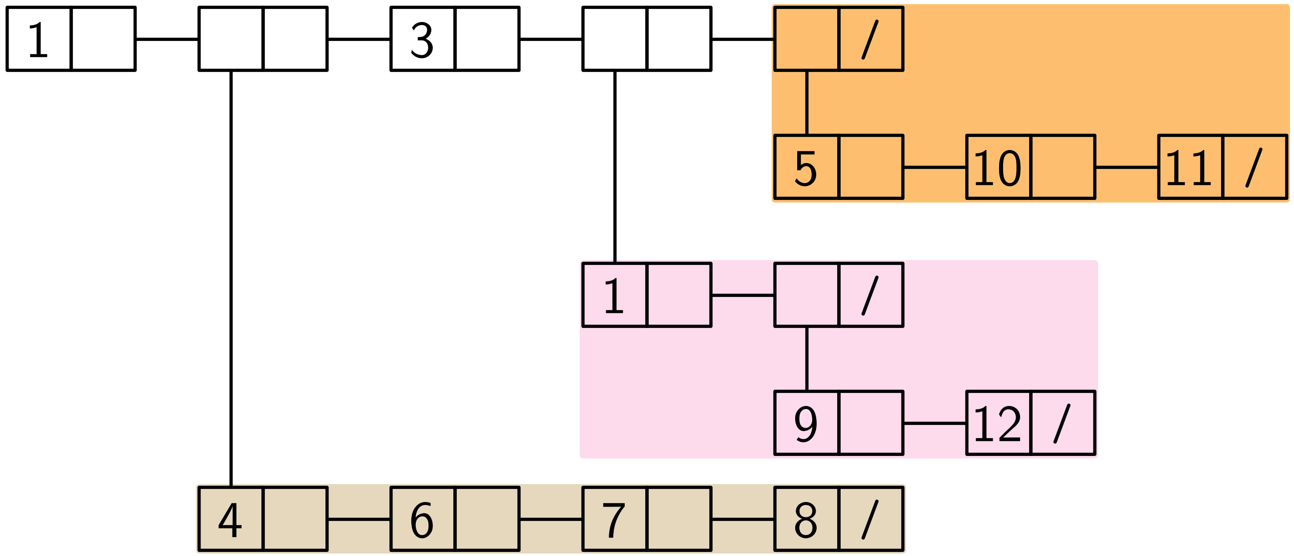
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics to optimize

HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

Drawing conventions

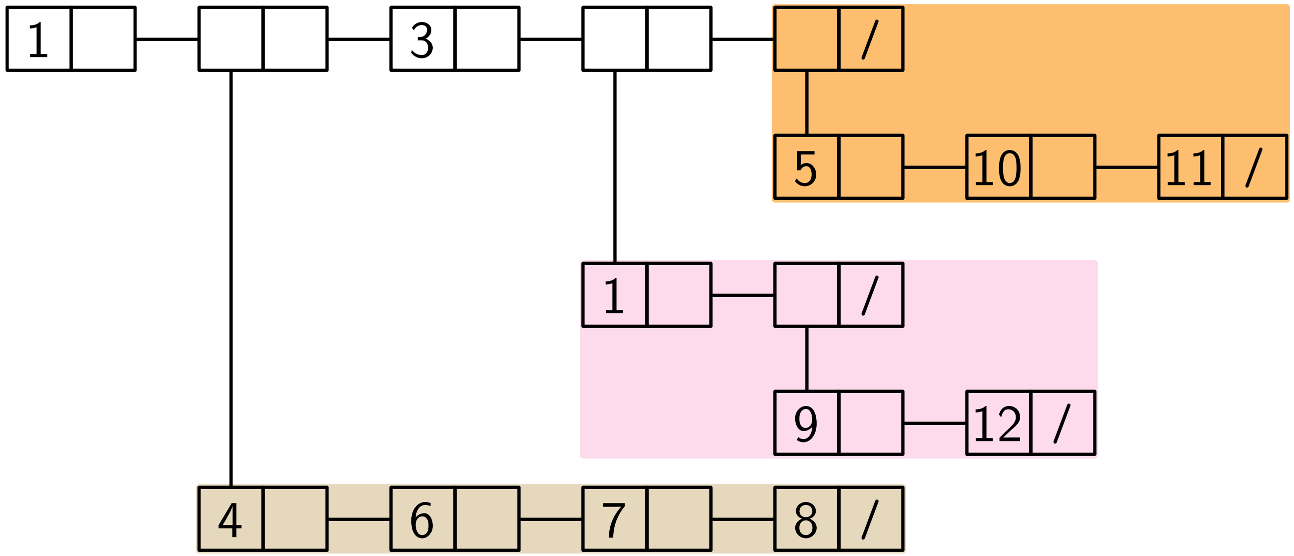
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint
- Edges are strictly down- or rightwards

Drawing aesthetics to optimize

HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP
- *Cons* (constructs) are memory objects that hold two values or pointers to values



Source: after `gajon.org/trees-linked-lists-common-lisp/`

Drawing conventions

- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint
- Edges are strictly down- or rightwards

Drawing aesthetics to optimize

- Height, width, area

HV-Drawings – Algorithm

Input: A binary tree T
Output: An HV-drawing of T

HV-Drawings – Algorithm

Input: A binary tree T
Output: An HV-drawing of T

Base case: 

HV-Drawings – Algorithm

Input: A binary tree T
Output: An HV-drawing of T

Base case: 

Divide: Recursively apply the algorithm to draw the left and right subtrees

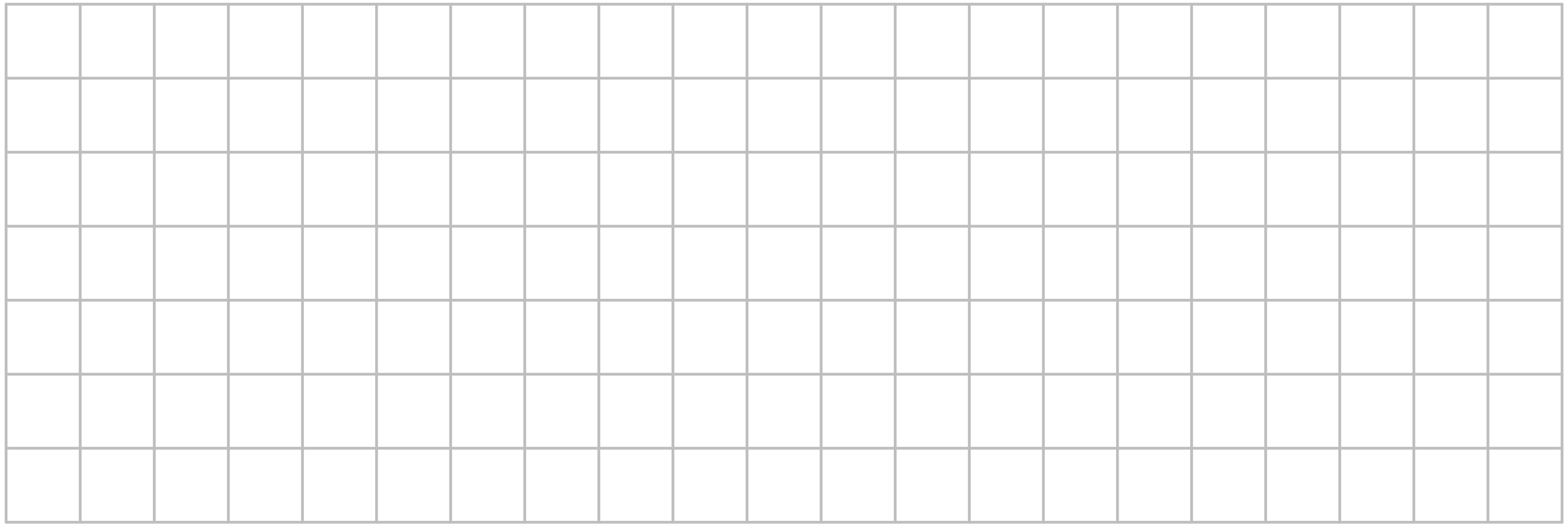
HV-Drawings – Algorithm

Input: A binary tree T
Output: An HV-drawing of T

Base case: 

Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



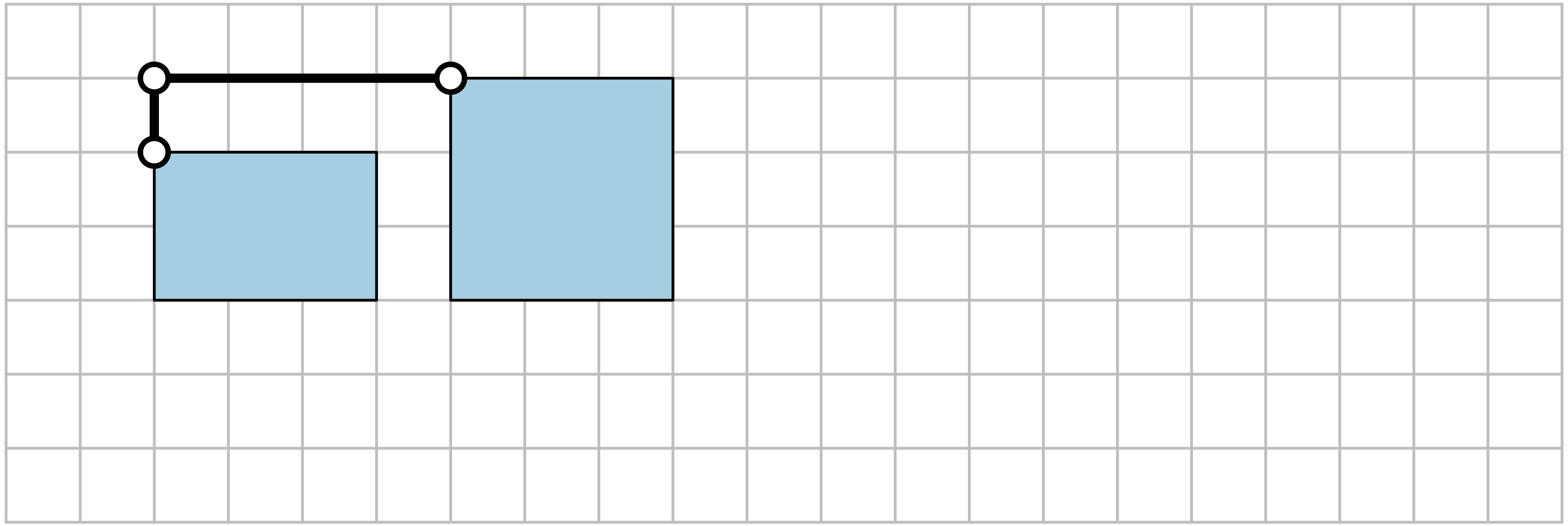
HV-Drawings – Algorithm

Input: A binary tree T
Output: An HV-drawing of T

Base case: 

Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer: horizontal combination



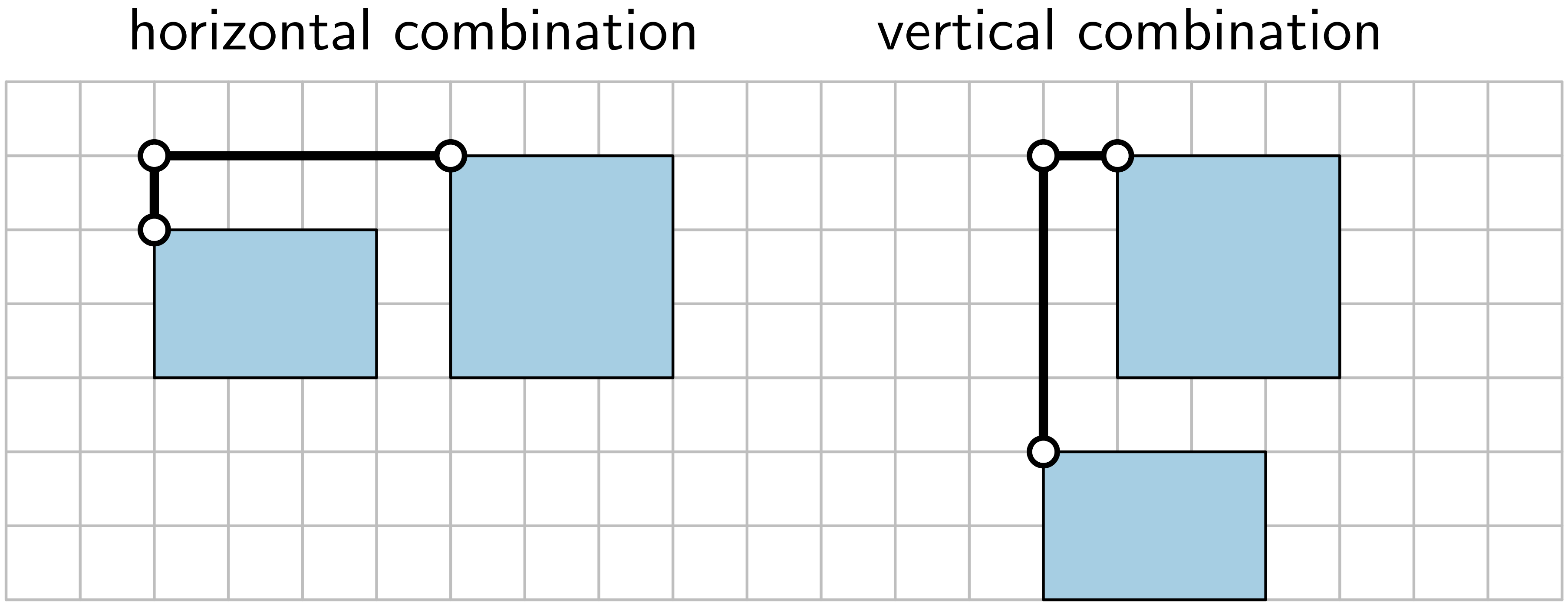
HV-Drawings – Algorithm

Input: A binary tree T
Output: An HV-drawing of T

Base case: 

Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*

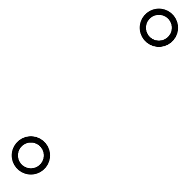
  Size of subtree := number of vertices

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*

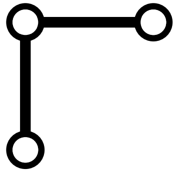
  Size of subtree $:=$ number of vertices



HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*
  Size of subtree := number of vertices

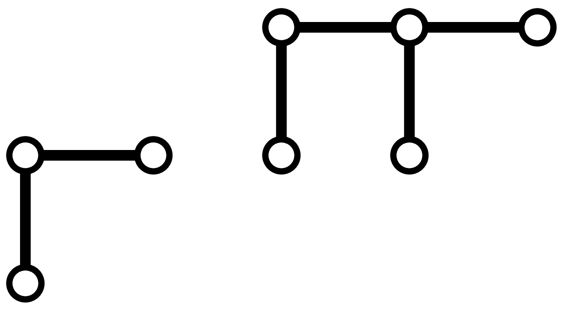


HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right ← *This can change the embedding!*

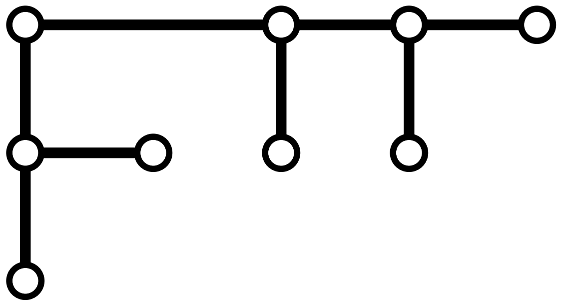
  Size of subtree := number of vertices



HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*
  Size of subtree := number of vertices

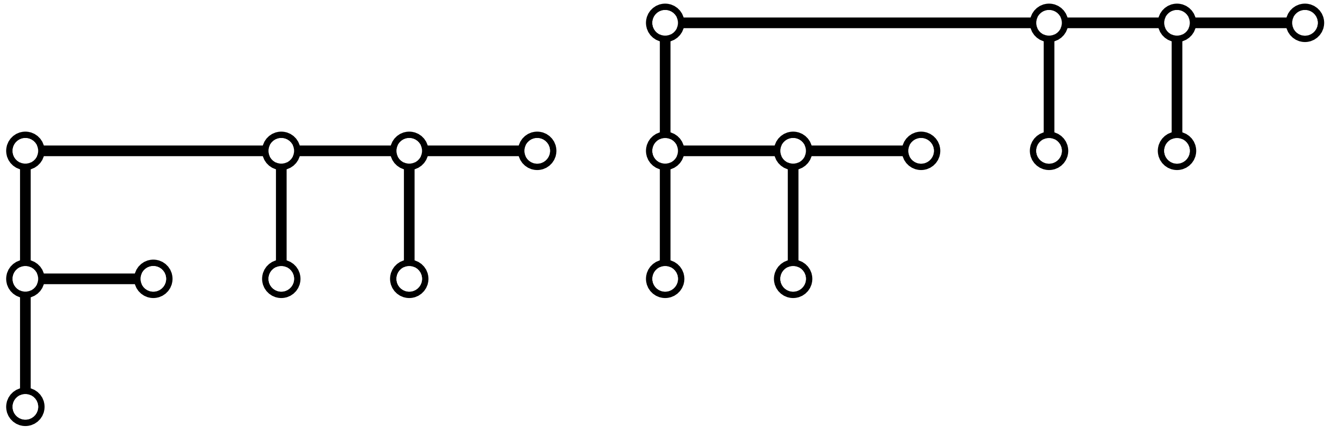


HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*

  Size of subtree := number of vertices

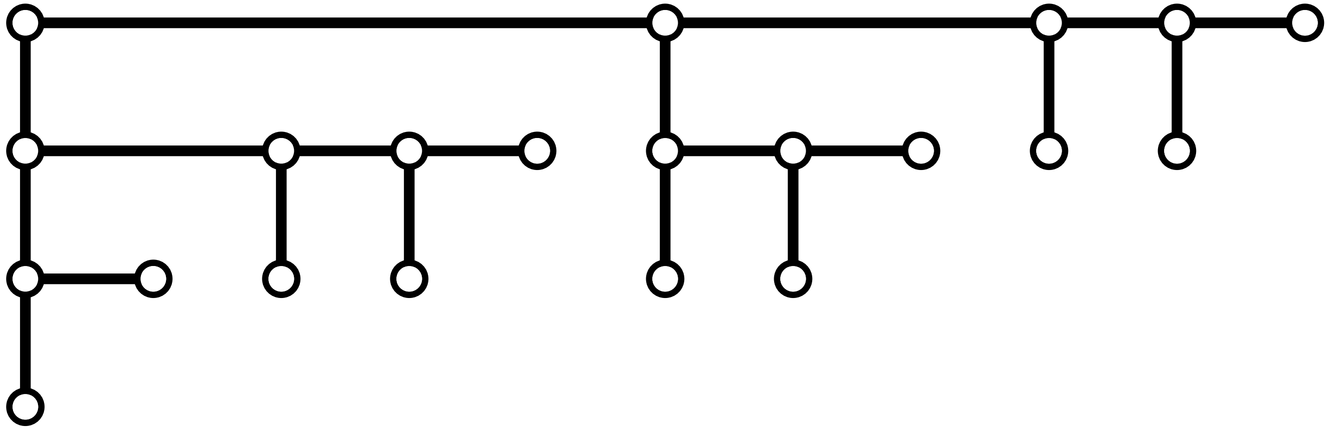


HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*

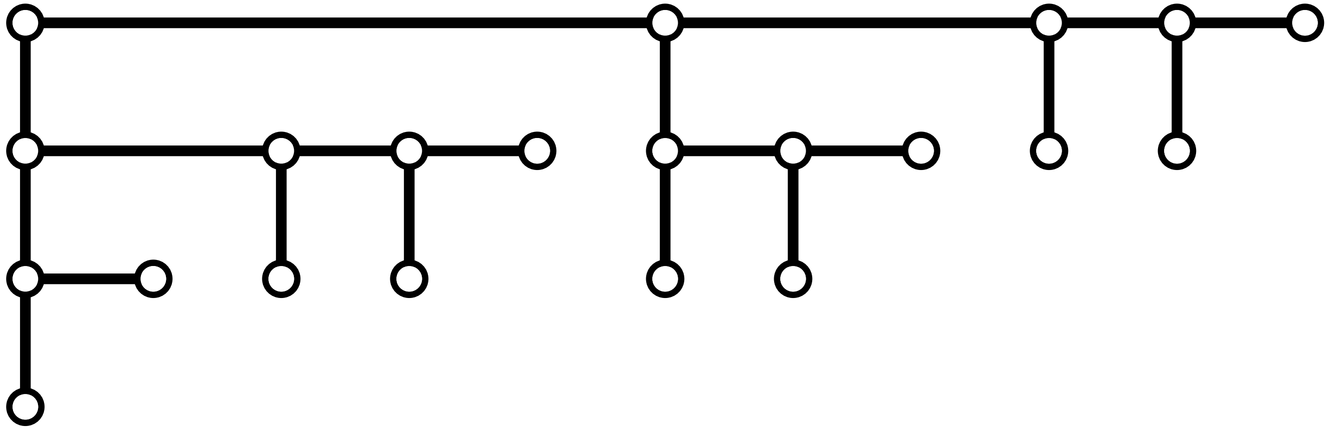
  Size of subtree := number of vertices



HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*
  Size of subtree := number of vertices



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

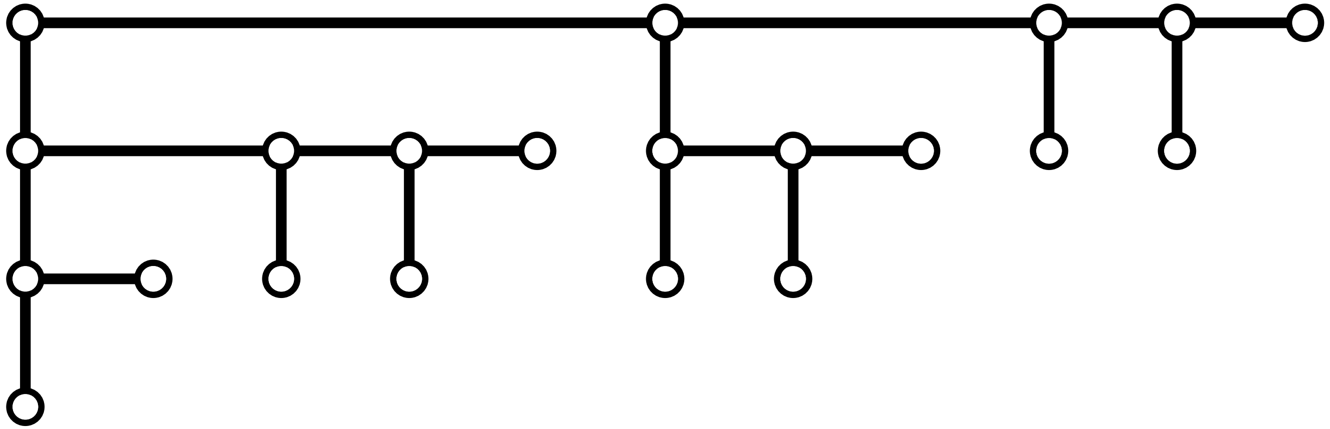
- width at most and
- height at most .

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*

  Size of subtree $:=$ number of vertices



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

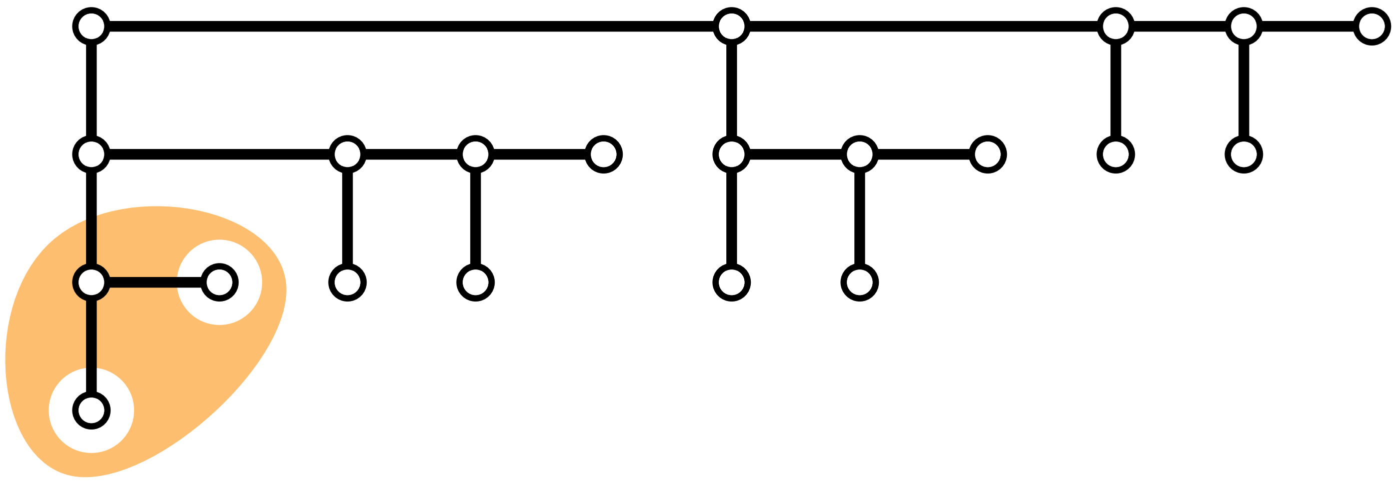
- width at most $n - 1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right ← *This can change the embedding!*

  Size of subtree := number of vertices



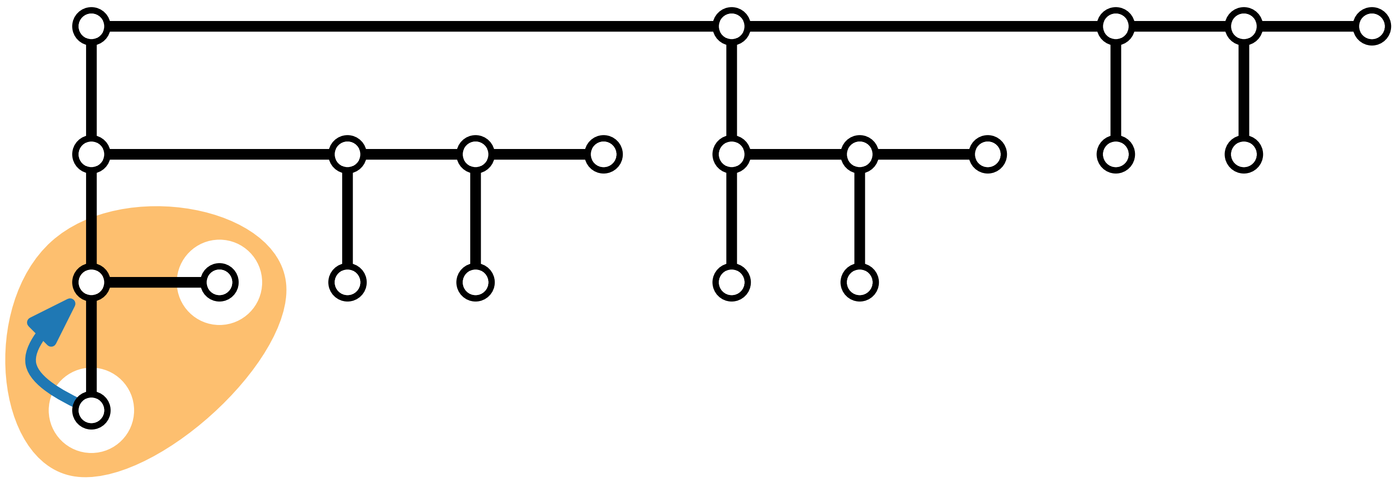
Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n-1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right ← *This can change the embedding!*
  Size of subtree := number of vertices



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

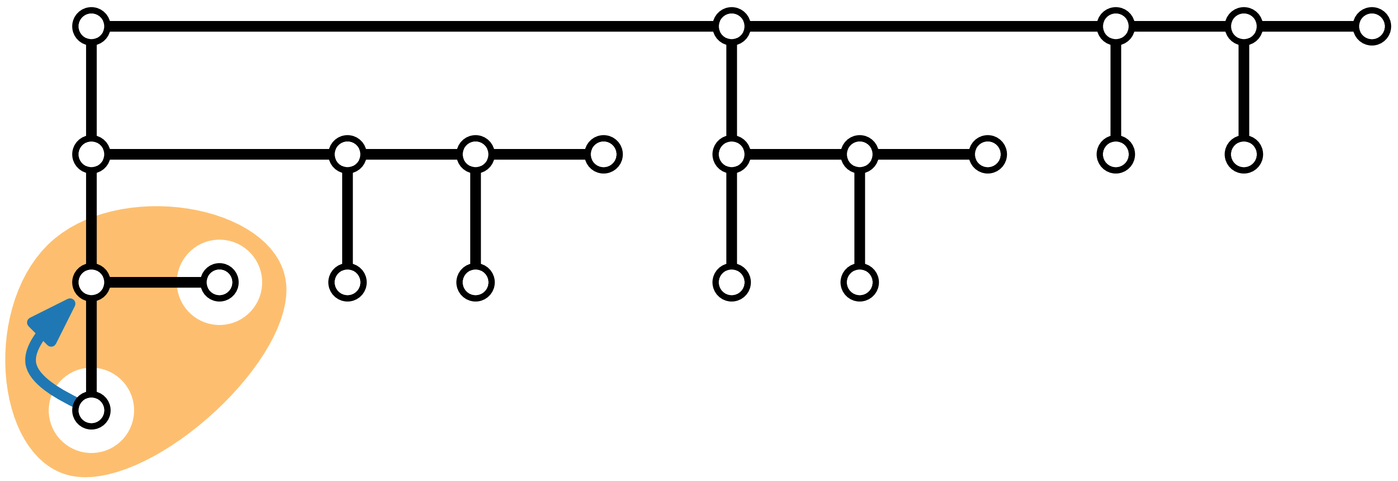
- width at most $n-1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*
  Size of subtree := number of vertices

at least $\cdot 2$



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n - 1$ and
- height at most

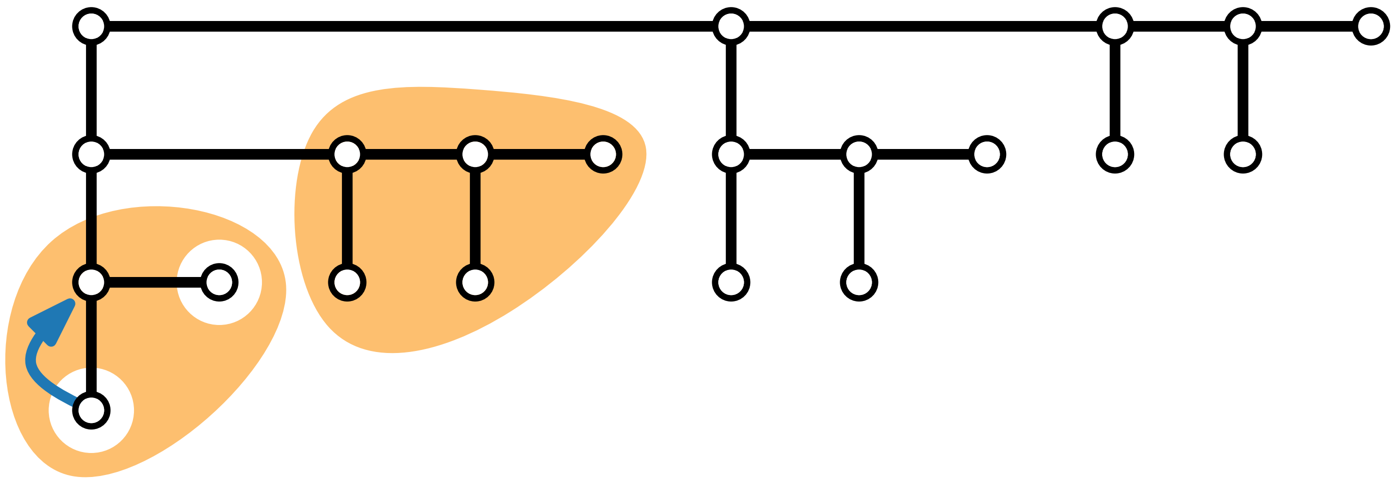
HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*

  Size of subtree := number of vertices

at least $\cdot 2$



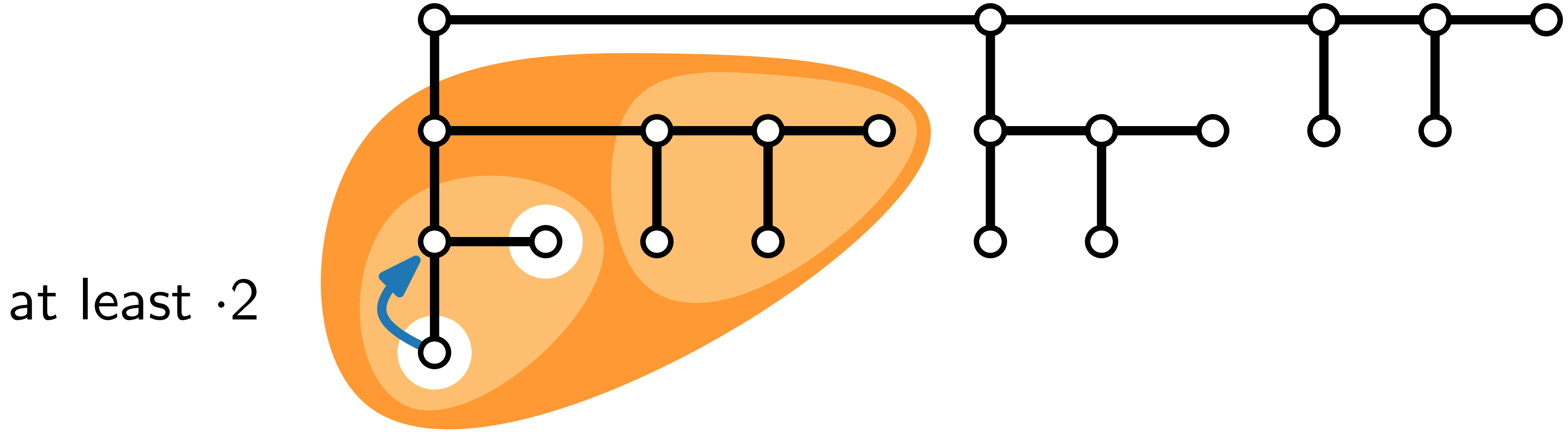
Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n-1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right ← *This can change the embedding!*
  Size of subtree := number of vertices



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n - 1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

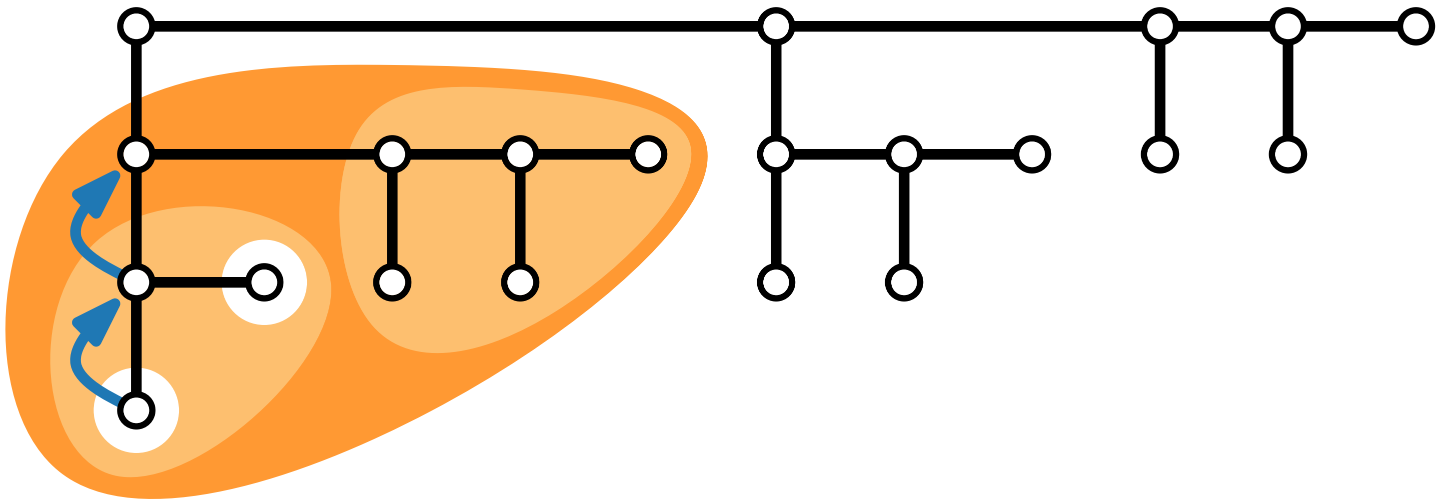
Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right ← *This can change the embedding!*

Size of subtree := number of vertices

at least ·2

at least ·2



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n - 1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

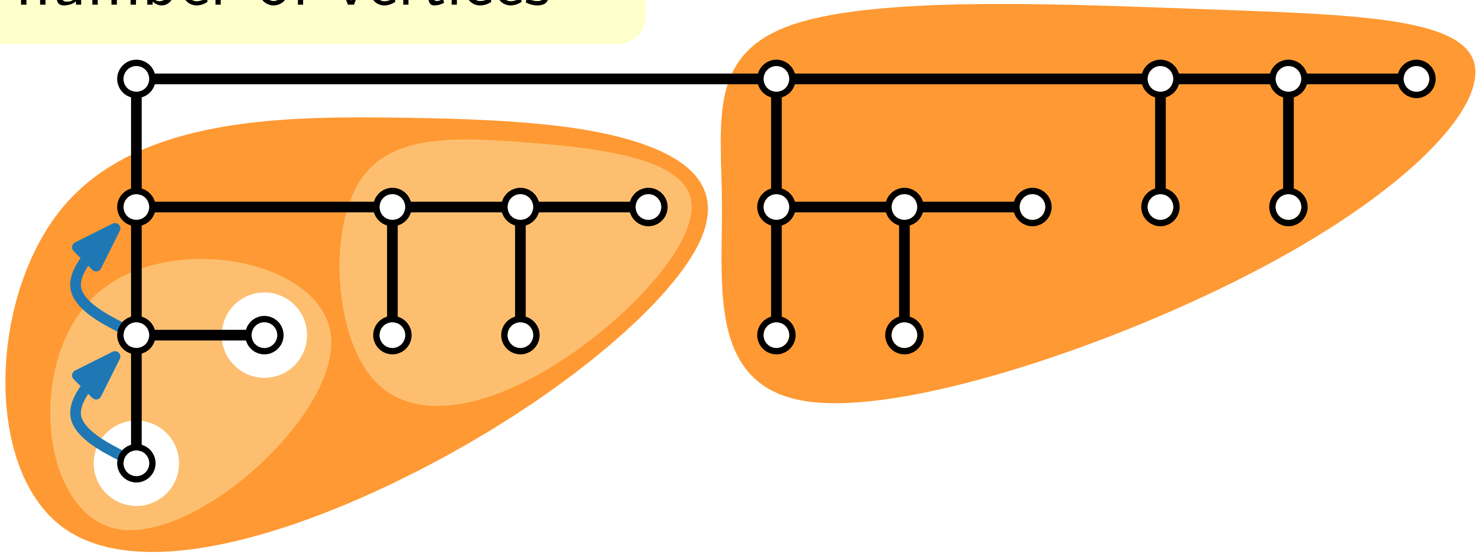
Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right ← *This can change the embedding!*

  Size of subtree := number of vertices

at least ·2

at least ·2



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- width at most $n-1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

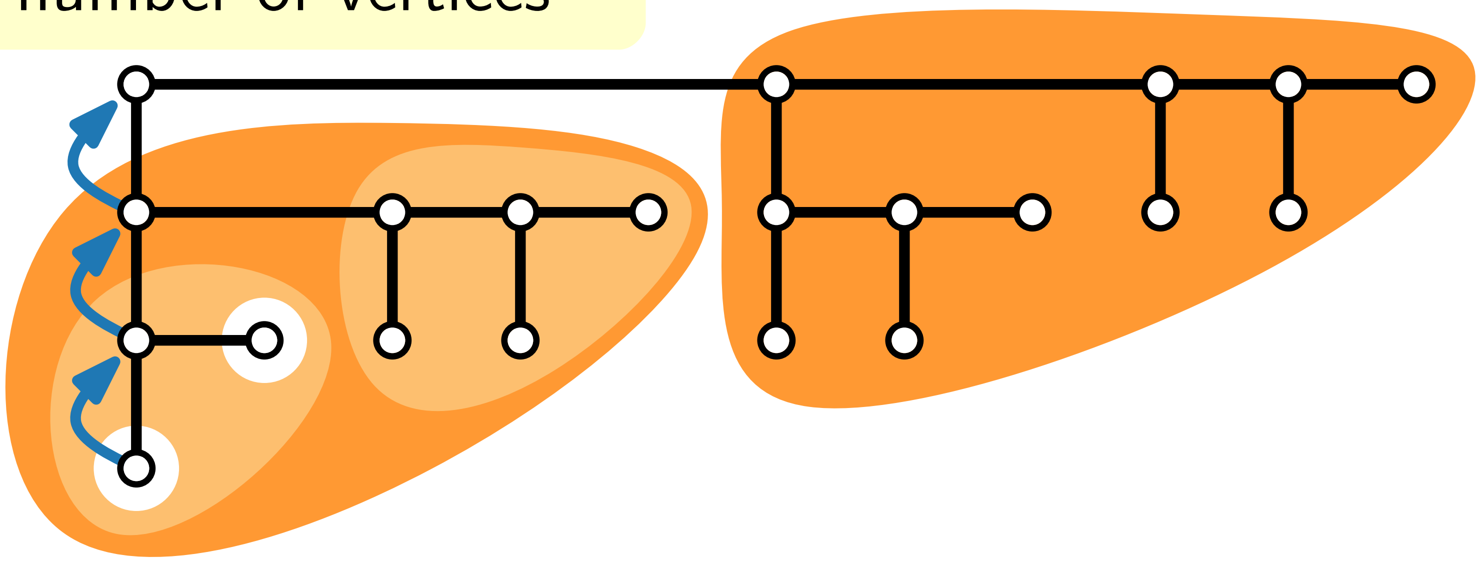
- Always apply horizontal combination
- Place the larger subtree to the right ← *This can change the embedding!*

  Size of subtree := number of vertices

at least ·2

at least ·2

at least ·2



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

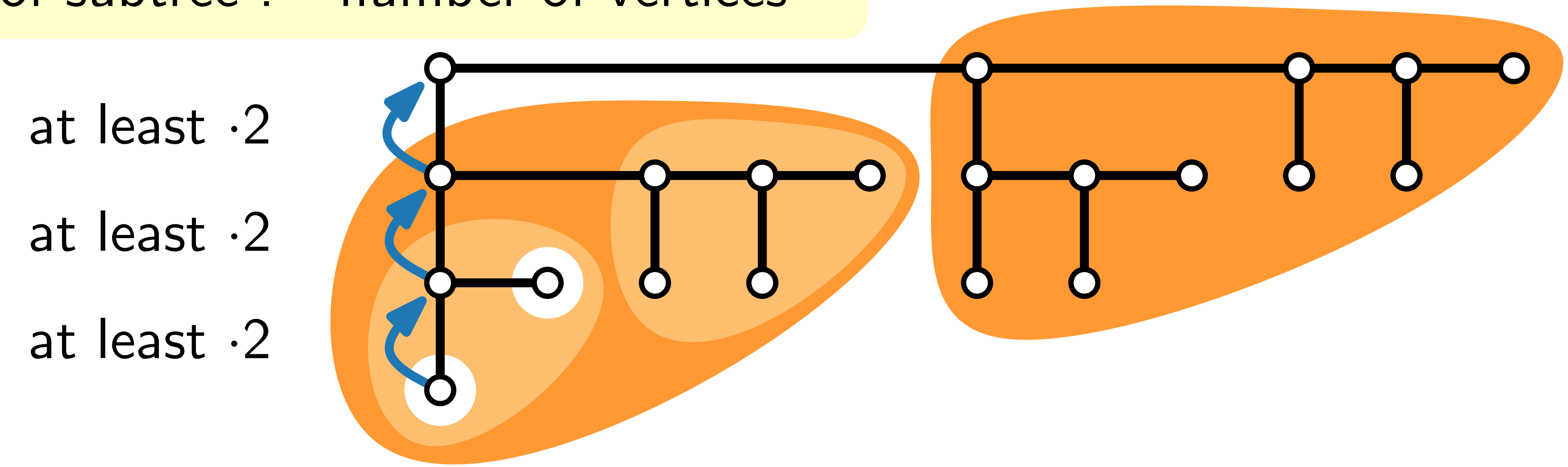
- width at most $n-1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*

  Size of subtree := number of vertices



Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

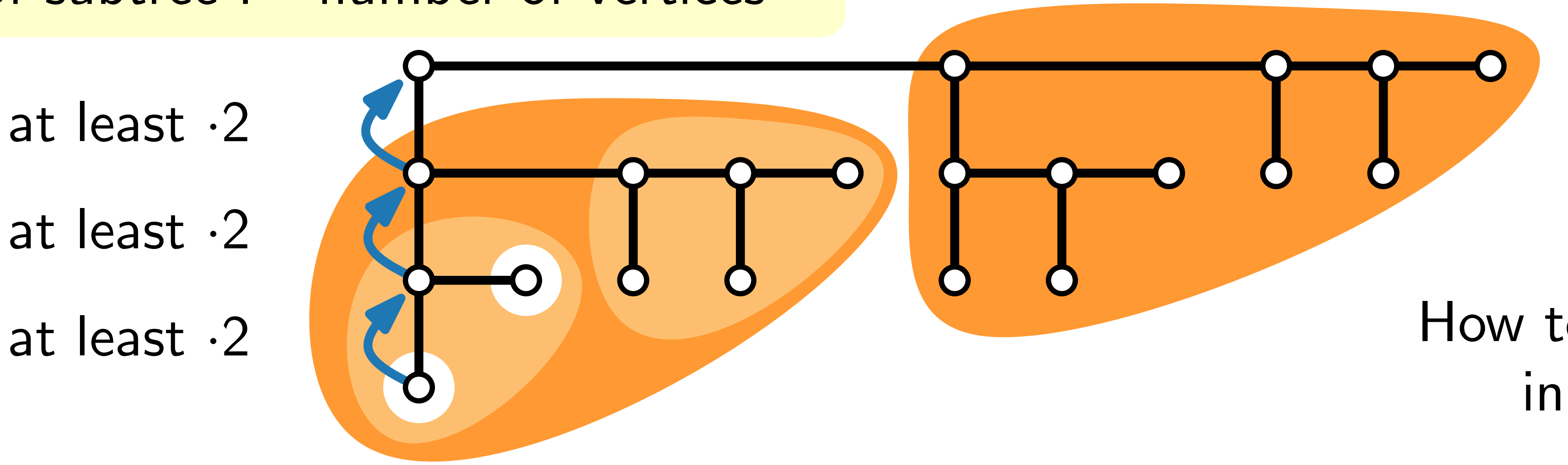
- width at most $n - 1$ and
- height at most $\log_2 n$.

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

- Always apply horizontal combination
- Place the larger subtree to the right ← *This can change the embedding!*

  Size of subtree := number of vertices



How to implement this in linear time?

Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has
- width at most $n - 1$ and
- height at most $\log_2 n$.

HV-Drawings – Result

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

HV-Drawings – Result

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing
  (planar, orthogonal, strictly right-/downward)

HV-Drawings – Result

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing (planar, orthogonal, strictly right-/downward)
- Width is at most $n - 1$

HV-Drawings – Result

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing
  (planar, orthogonal, strictly right-/downward)
- Width is at most $n - 1$
- Height is at most $\log_2 n$

HV-Drawings – Result

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing (planar, orthogonal, strictly right-/downward)
- Width is at most $n - 1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$

HV-Drawings – Result

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing
  (planar, orthogonal, strictly right-/downward)
- Width is at most $n-1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ← worst-case optimal [exercise]

HV-Drawings – Result

Theorem.
Let T be a binary tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing
  (planar, orthogonal, strictly right-/downward)
- Width is at most $n - 1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ⟵ worst-case optimal [exercise]
- Simply and axially isomorphic subtrees have congruent drawings up to translation

HV-Drawings – Result

Theorem.
Let T be a ~~binary~~ rooted tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing (planar, orthogonal, strictly right-/downward)
- Width is at most $n - 1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ← worst-case optimal [exercise]
- Simply and axially isomorphic subtrees have congruent drawings up to translation

HV-Drawings – Result

Theorem.
rooted
Let T be a ~~binary~~ tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing
  (planar, orthogonal, strictly right-/downward)
- Width is at most $n-1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ← worst-case optimal [exercise]
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree

HV-Drawings – Result

Theorem.
Let T be a ~~binary~~ rooted tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing (planar, orthogonal, strictly right-/downward)
- Width is at most $n - 1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ← worst-case optimal [exercise]
- Simply and axially isomorphic subtrees have congruent drawings up to translation

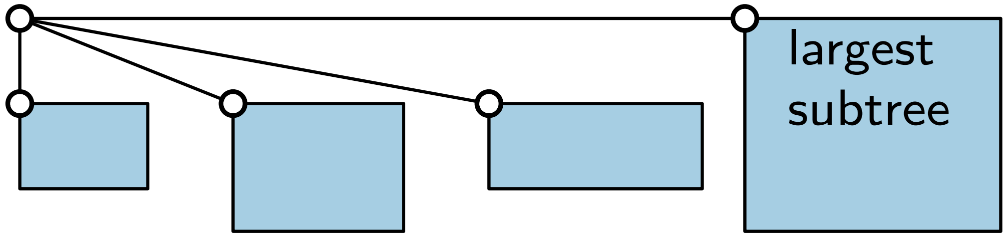
General rooted tree

HV-Drawings – Result

Theorem.
Let T be a ~~binary~~ rooted tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing (planar, orthogonal, strictly right-/downward)
- Width is at most $n-1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ← worst-case optimal [exercise]
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree

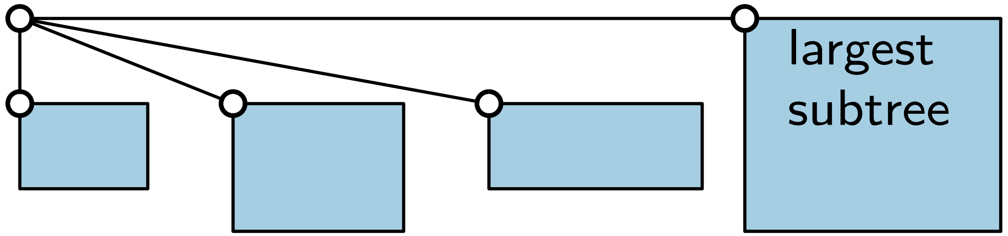


HV-Drawings – Result

Theorem.
Let T be a ~~binary~~ rooted tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing (planar, ~~orthogonal,~~ strictly right-/downward)
- Width is at most $n - 1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ← worst-case optimal [exercise]
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree

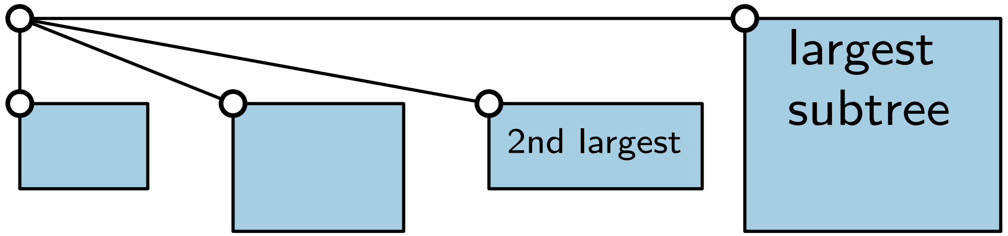


HV-Drawings – Result

Theorem.
Let T be a ~~binary~~ rooted tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing (planar, ~~orthogonal,~~ strictly right-/downward)
- Width is at most $n - 1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ← worst-case optimal [exercise]
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree

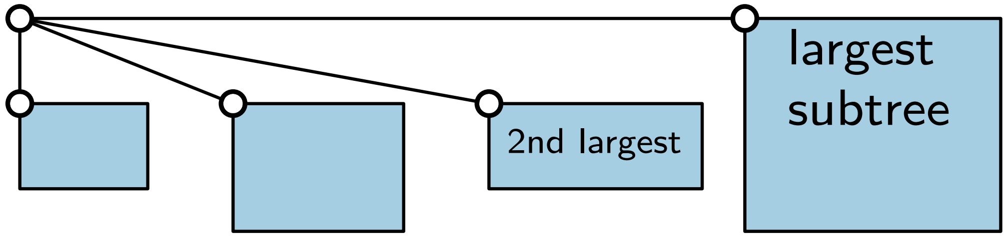


HV-Drawings – Result

Theorem.
Let T be a ~~binary~~ rooted tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing
  (planar, ~~orthogonal,~~ strictly right-/downward)
- Width is at most $n - 1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ⟵ worst-case optimal [exercise]
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree



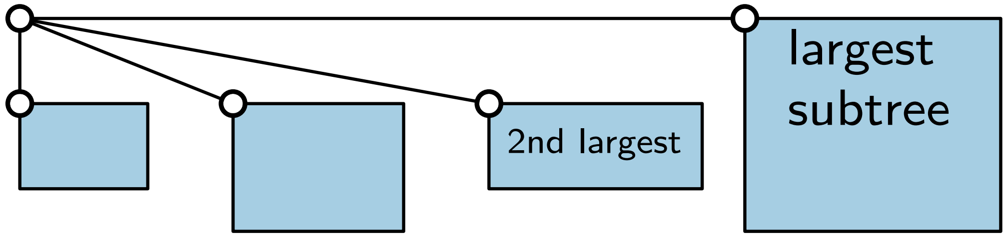
Optimal area?

HV-Drawings – Result

Theorem.
Let T be a ~~binary~~ rooted tree with n vertices. The right-heavy algorithm constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is an HV-drawing (planar, ~~orthogonal,~~ strictly right-/downward)
- Width is at most $n - 1$
- Height is at most $\log_2 n$
- Area is in $\mathcal{O}(n \log n)$ ⟵ worst-case optimal [exercise]
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree



Optimal area?

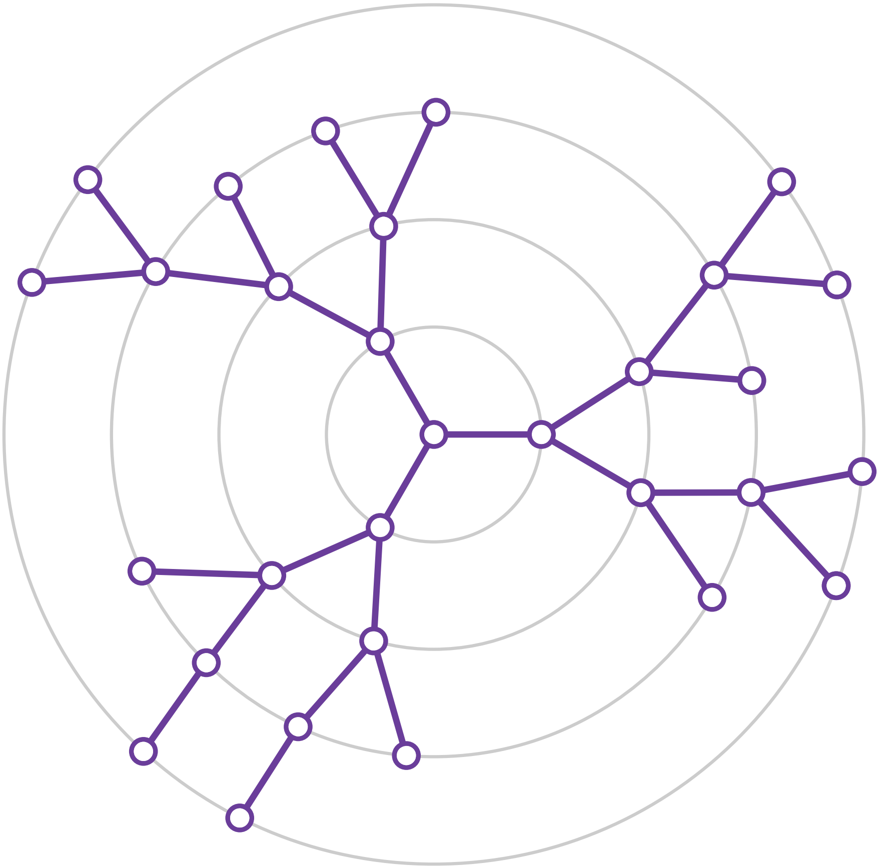
Not with divide & conquer approach, but can be computed with Dynamic Programming.

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

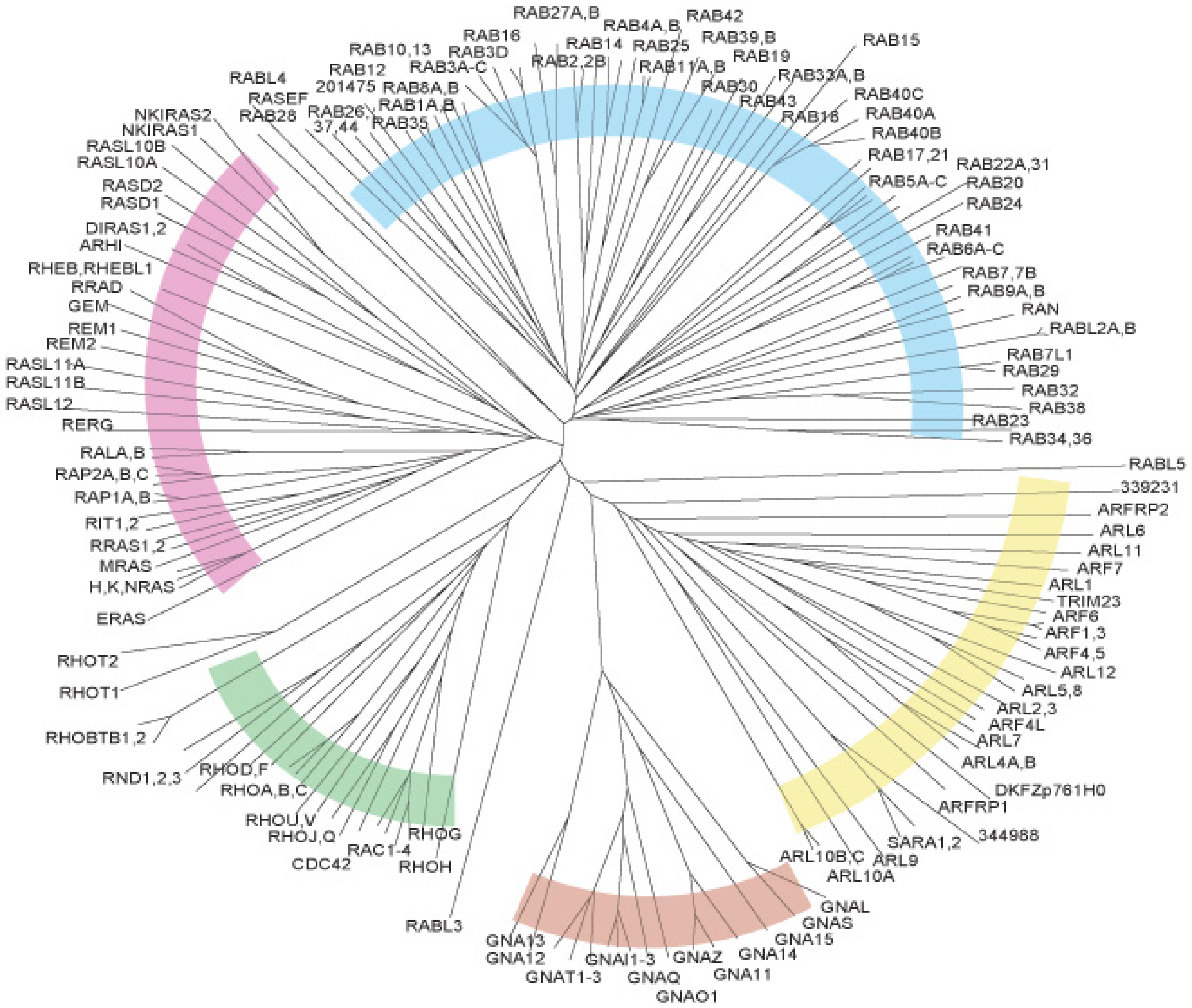
Visualization of Graphs

Lecture 1b:
Drawing Trees

Part III:
Radial Layouts

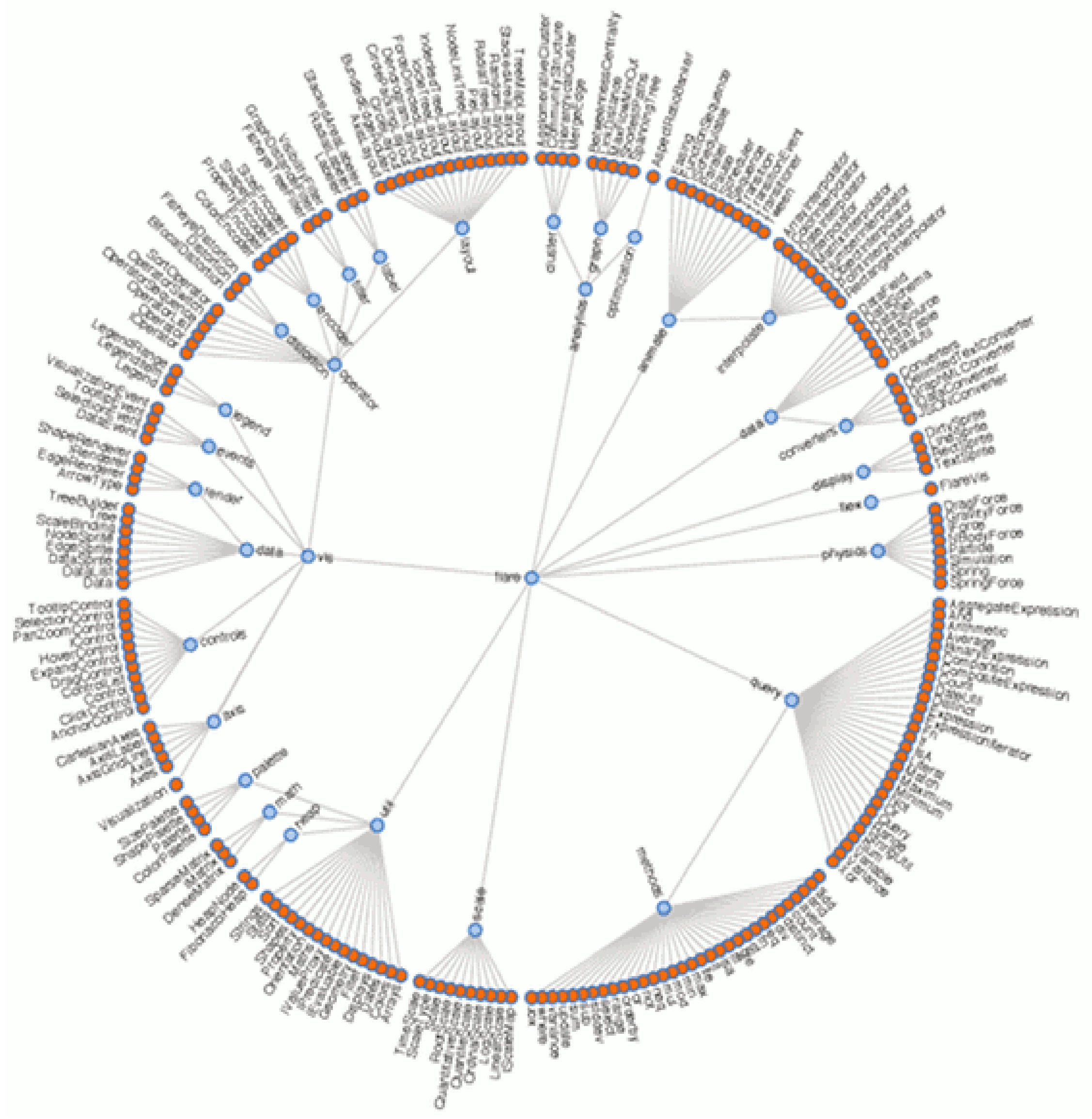


Radial Layouts – Applications

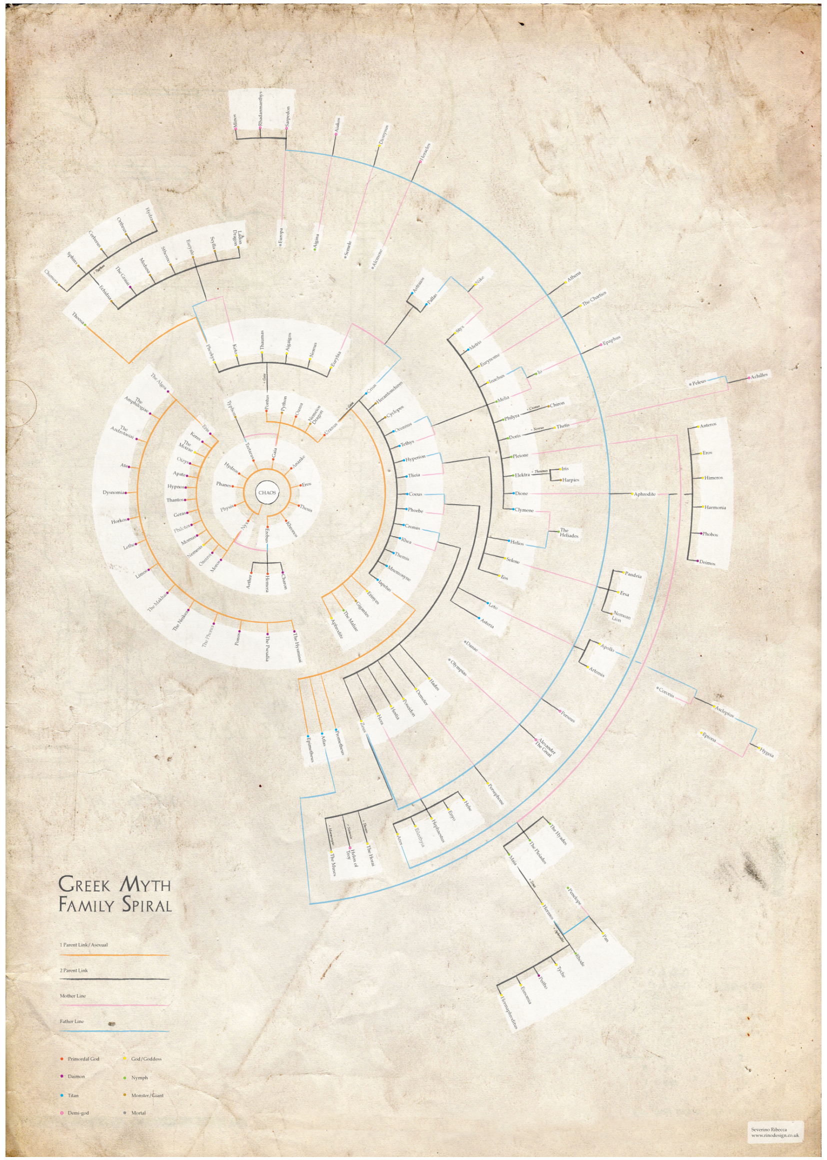


Phylogenetic tree
by Colicelli, ScienceSignaling, 2004

Radial Layouts – Applications

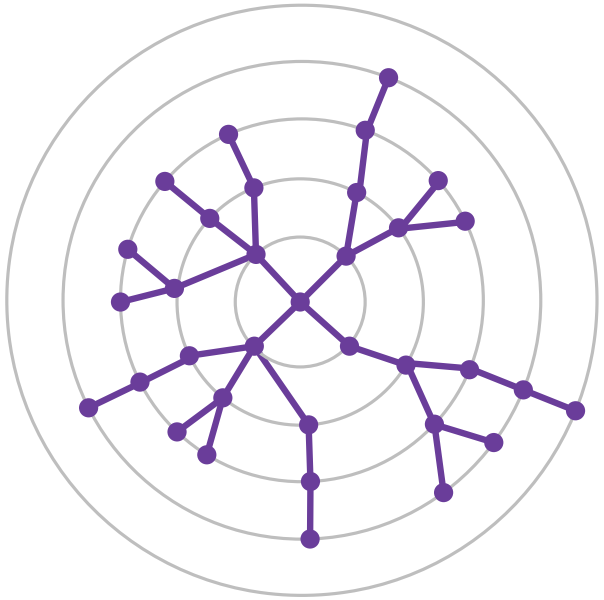


Flare Visualization Toolkit code structure
by Heer, Bostock and Ogievetsky, 2010



Greek Myth Family
by Ribecca, 2011

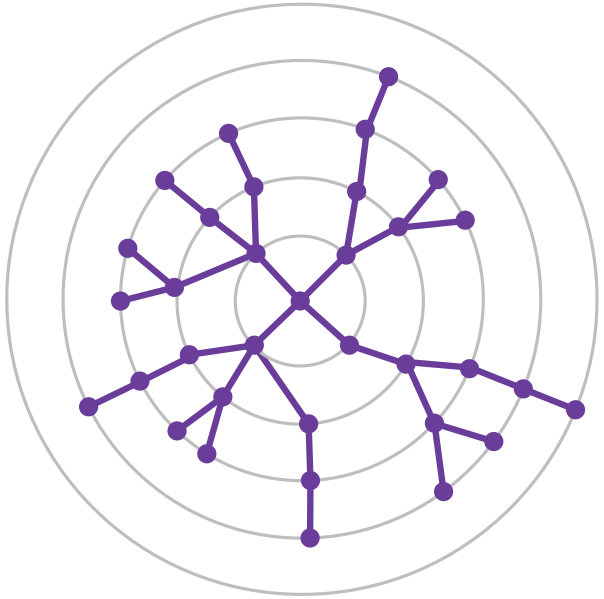
Radial Layouts – Drawing Style



Drawing conventions

Drawing aesthetics to optimize

Radial Layouts – Drawing Style

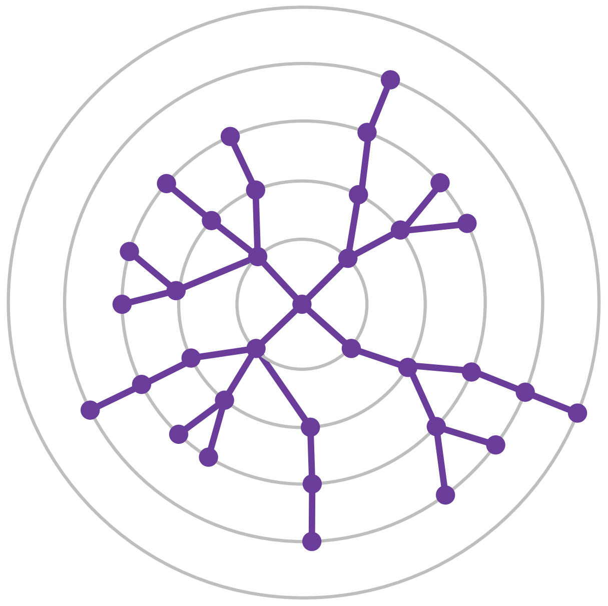


Drawing conventions

- Vertices lie on circular layers according to their depth

Drawing aesthetics to optimize

Radial Layouts – Drawing Style

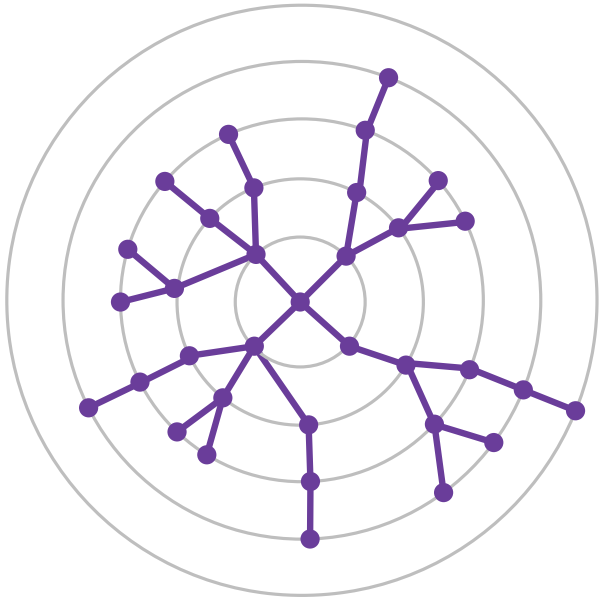


Drawing conventions

- Vertices lie on circular layers according to their depth
- Drawing is planar

Drawing aesthetics to optimize

Radial Layouts – Drawing Style



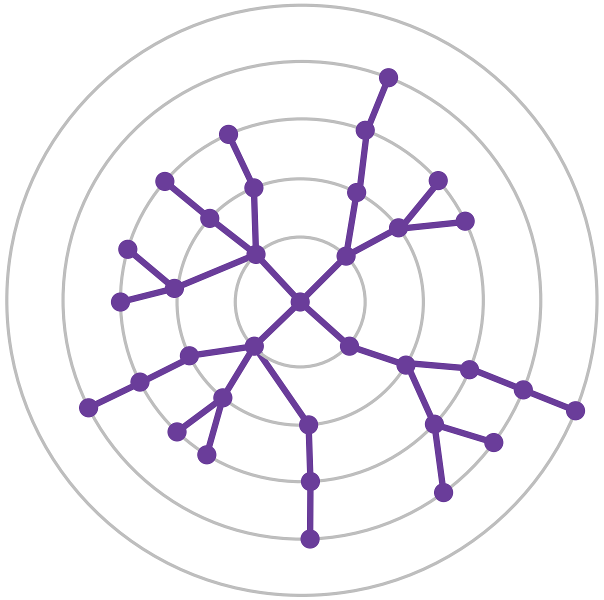
Drawing conventions

- Vertices lie on circular layers according to their depth
- Drawing is planar

Drawing aesthetics to optimize

- Balanced distribution of the vertices

Radial Layouts – Drawing Style



Drawing conventions

- Vertices lie on circular layers according to their depth
- Drawing is planar

Drawing aesthetics to optimize

- Balanced distribution of the vertices

How can an algorithm optimize the distribution of the vertices?

Radial Layouts – Algorithm Attempt

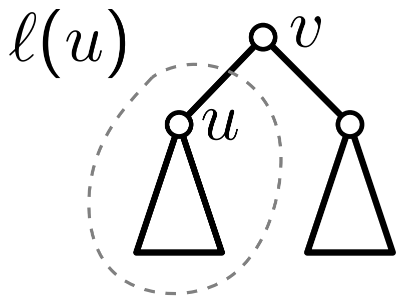
Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

Radial Layouts – Algorithm Attempt

Idea

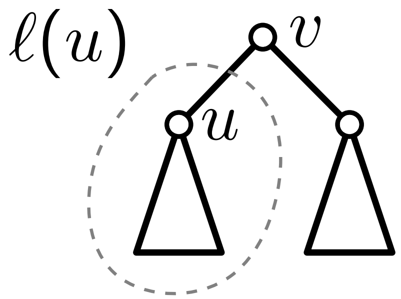
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

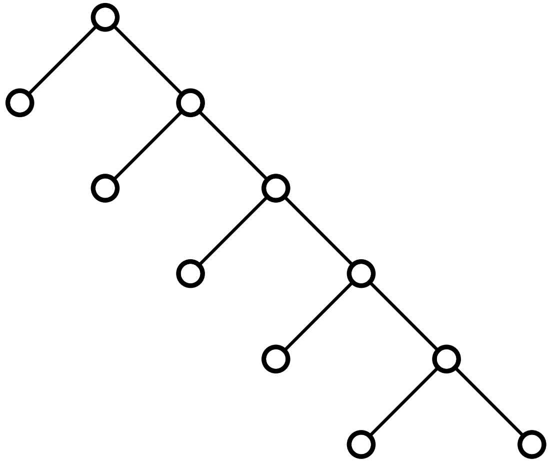


Radial Layouts – Algorithm Attempt

Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

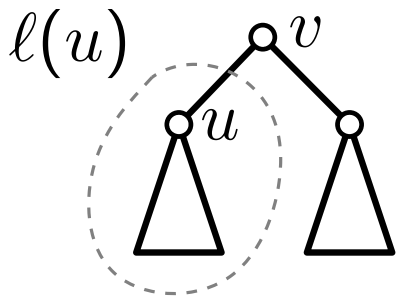


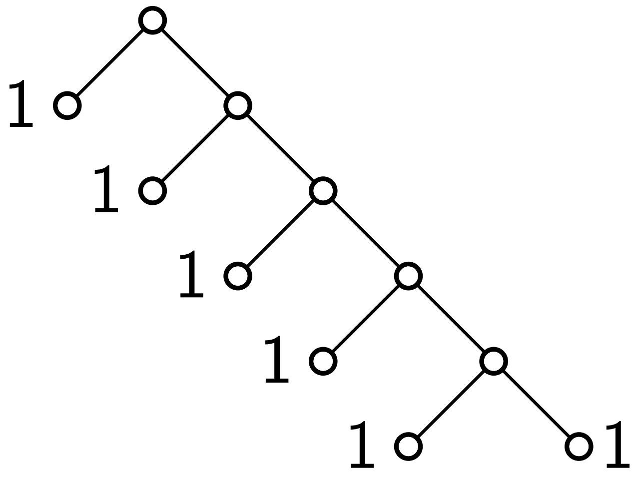


Radial Layouts – Algorithm Attempt

Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

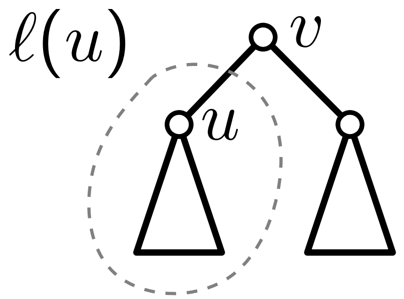


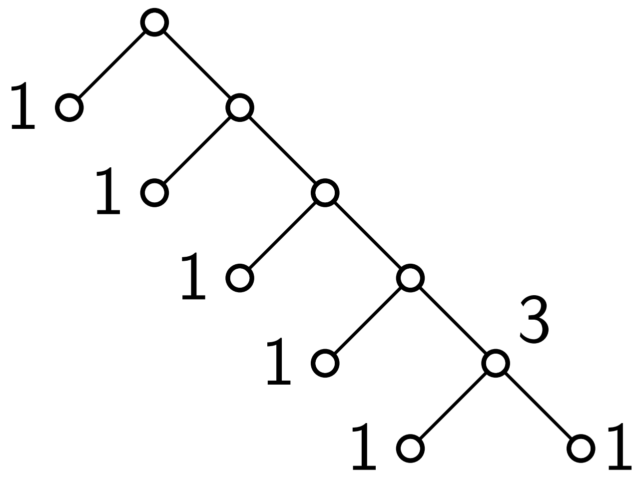


Radial Layouts – Algorithm Attempt

Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

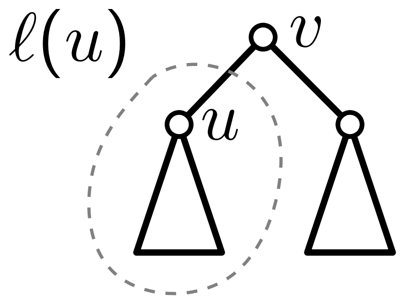


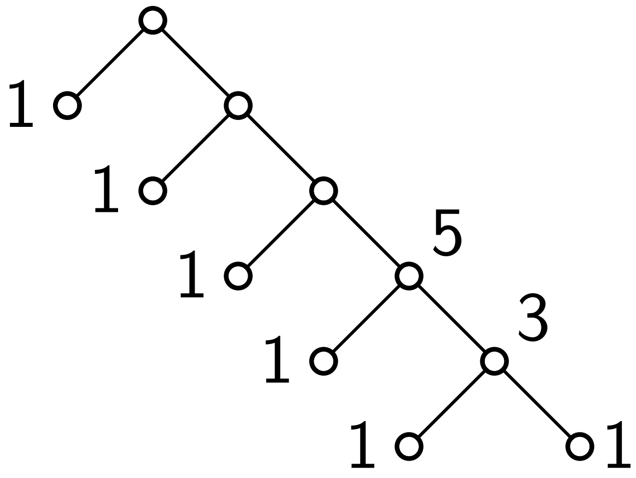


Radial Layouts – Algorithm Attempt

Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

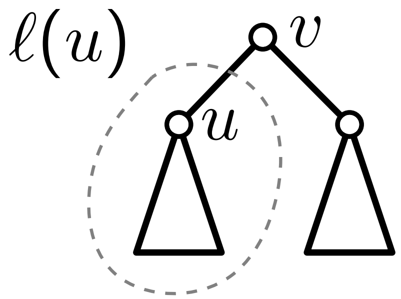


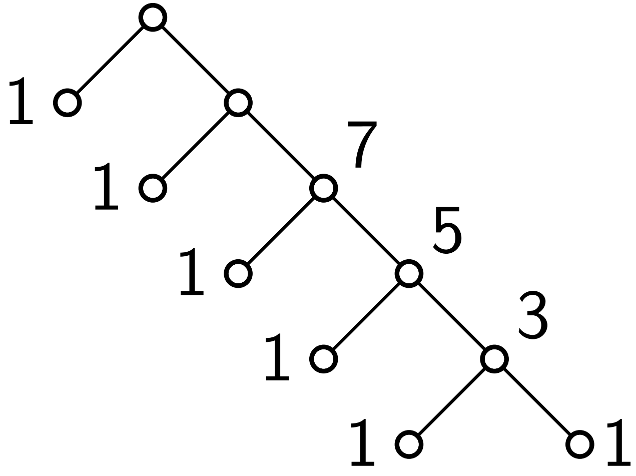


Radial Layouts – Algorithm Attempt

Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

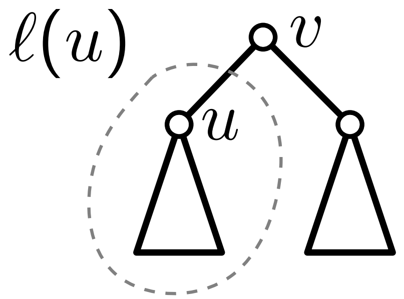


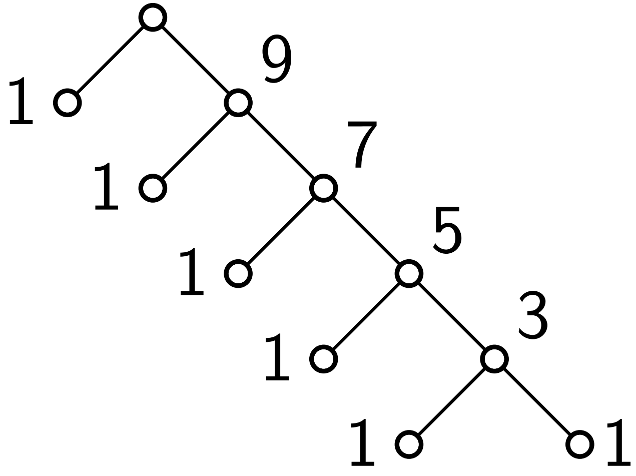


Radial Layouts – Algorithm Attempt

Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

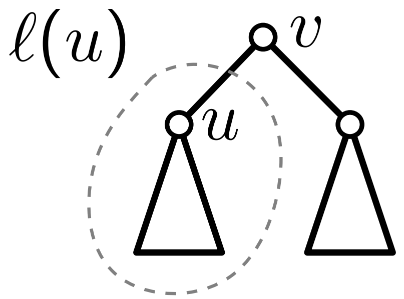


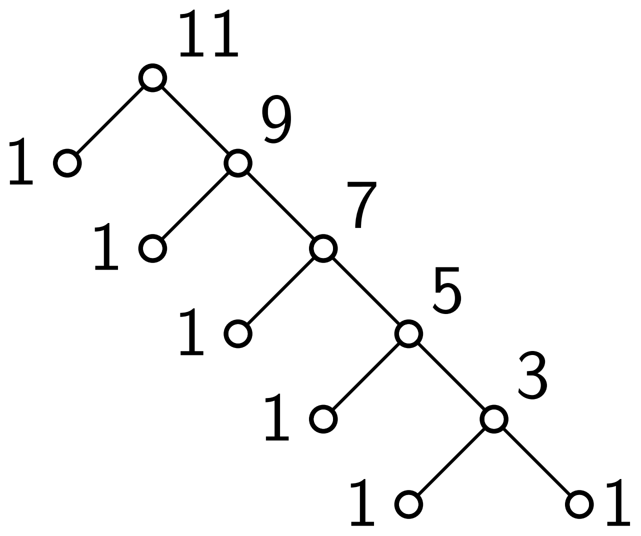


Radial Layouts – Algorithm Attempt

Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

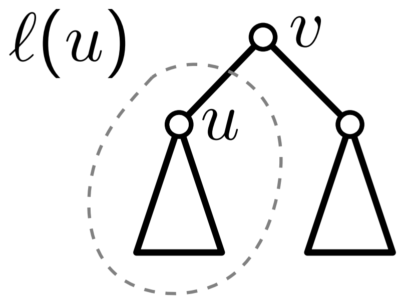




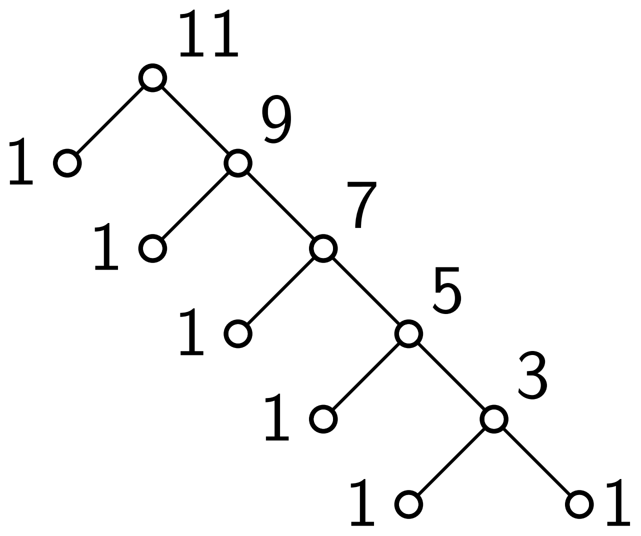
Radial Layouts – Algorithm Attempt

Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



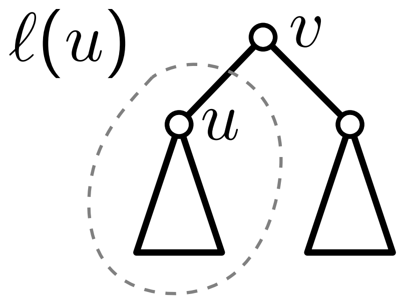
$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$



Radial Layouts – Algorithm Attempt

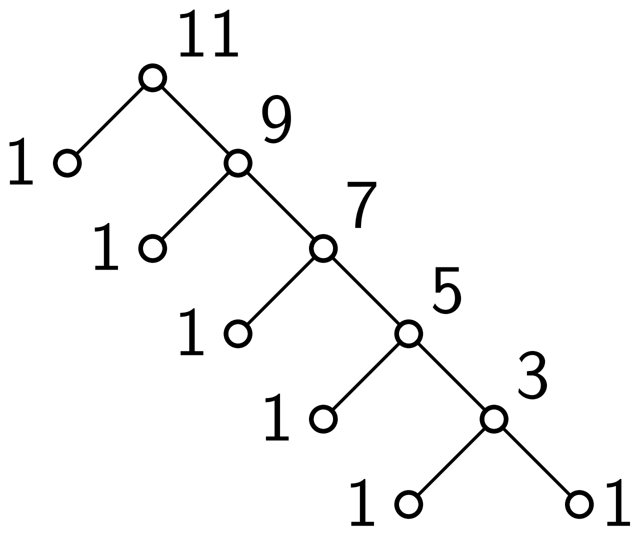
Idea

- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$

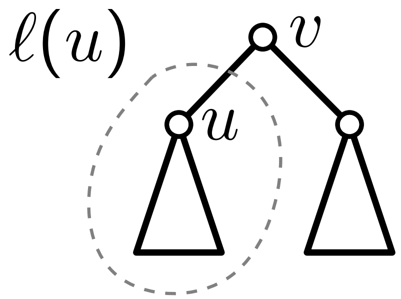
- Place u in the middle of its area



Radial Layouts – Algorithm Attempt

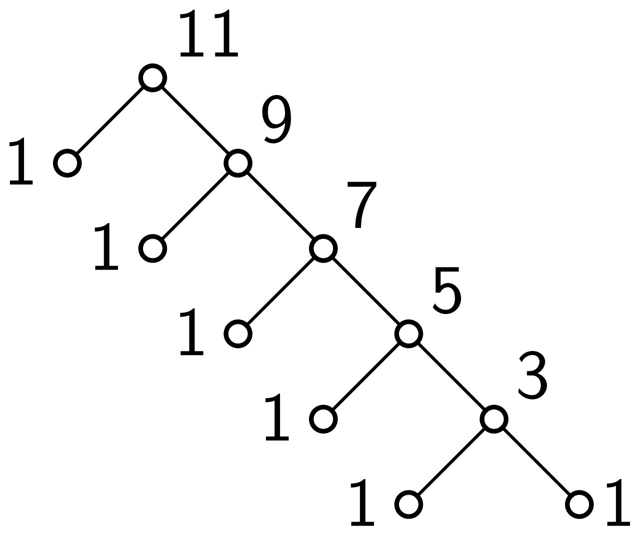
Idea

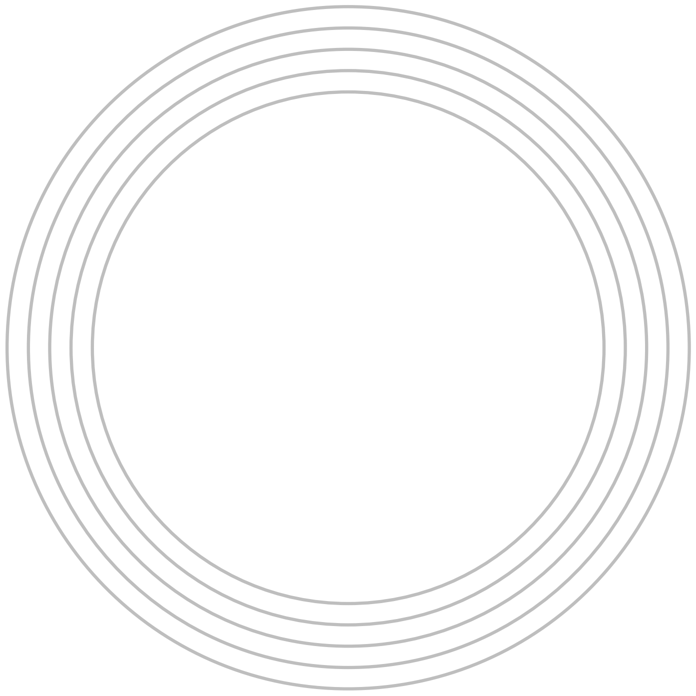
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$

- Place u in the middle of its area

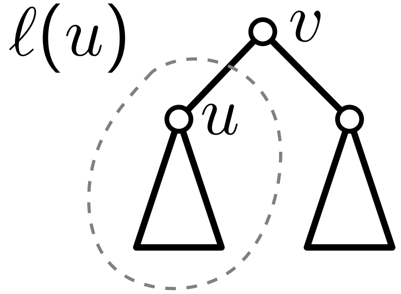




Radial Layouts – Algorithm Attempt

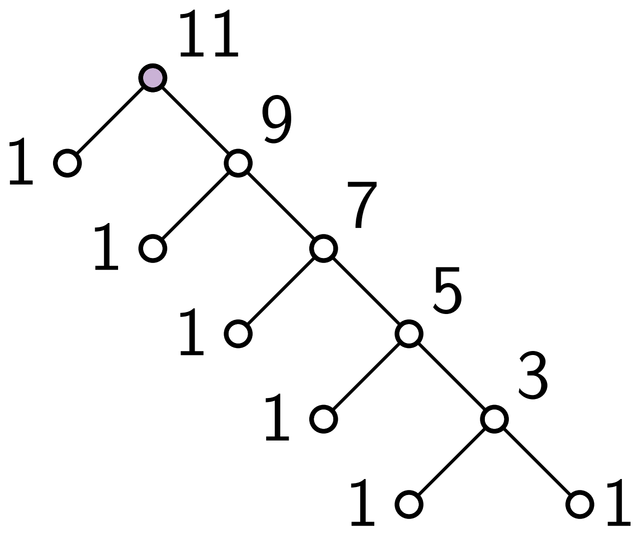
Idea

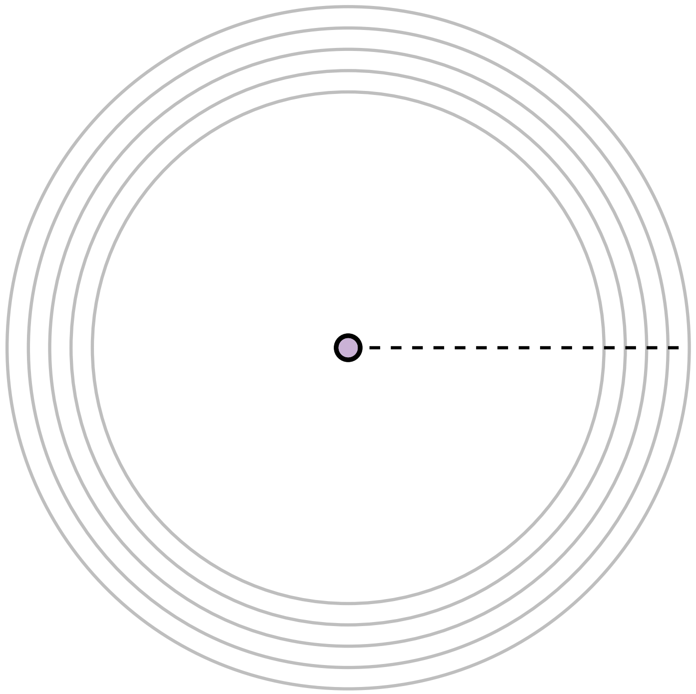
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$

- Place u in the middle of its area



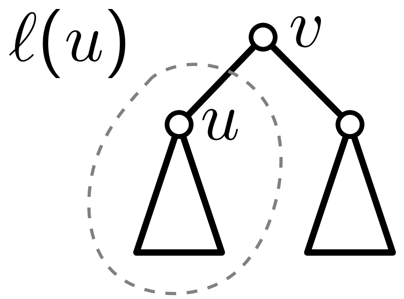


Radial Layouts – Algorithm Attempt

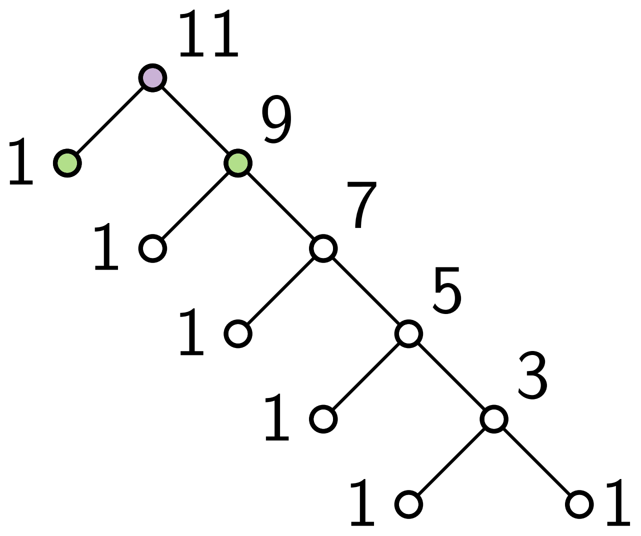
Idea

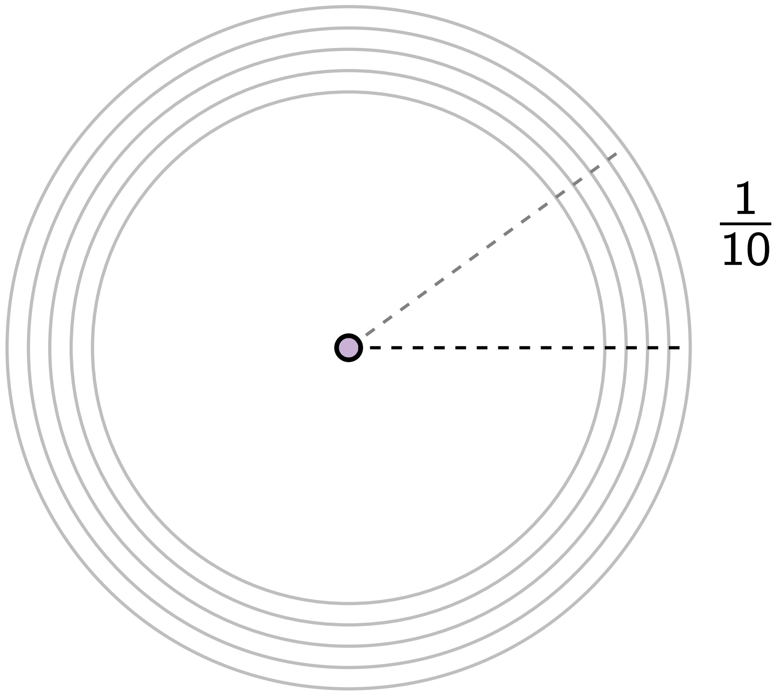
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$



- Place u in the middle of its area

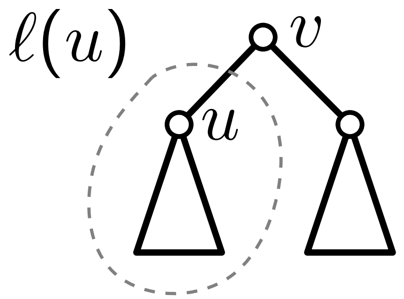




Radial Layouts – Algorithm Attempt

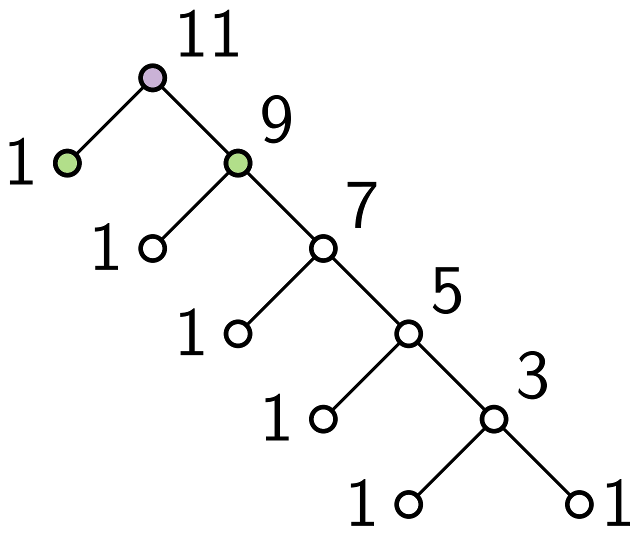
Idea

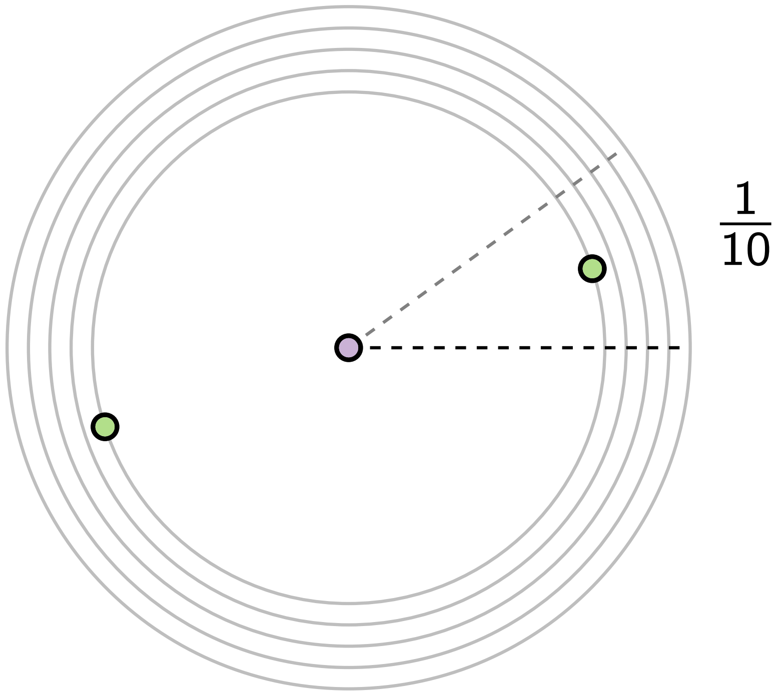
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$

- Place u in the middle of its area

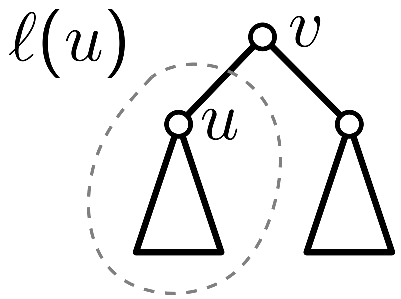




Radial Layouts – Algorithm Attempt

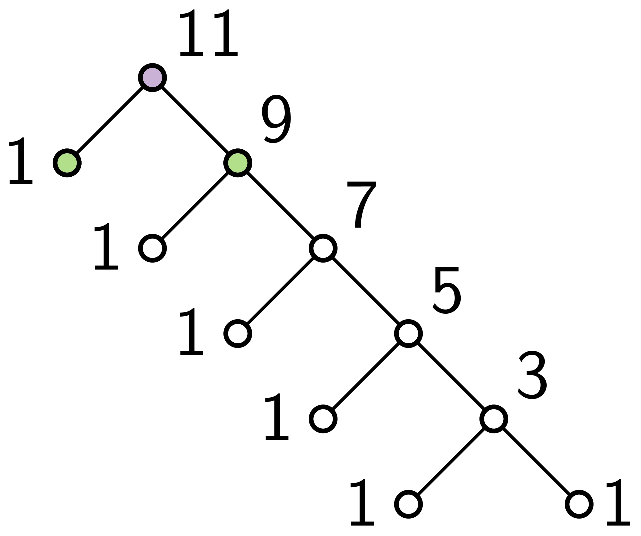
Idea

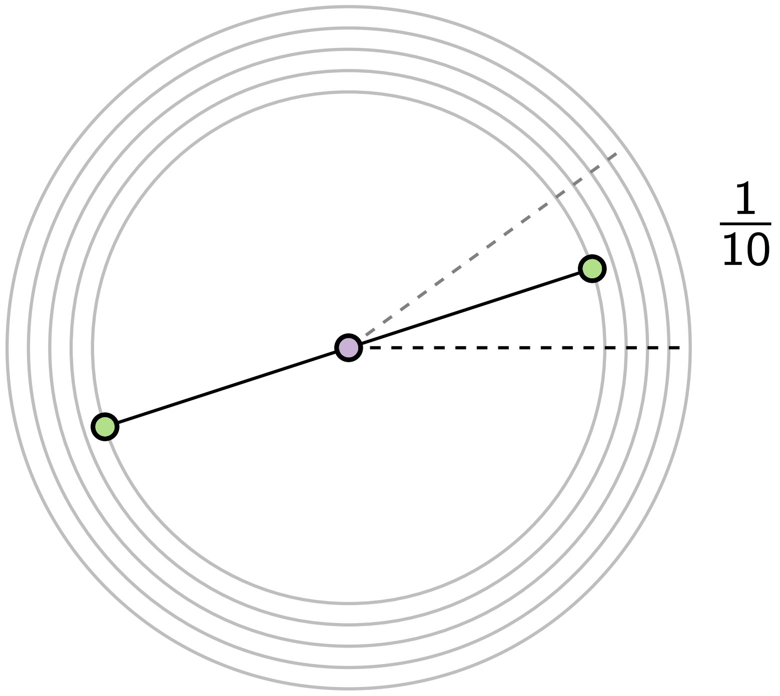
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$

- Place u in the middle of its area



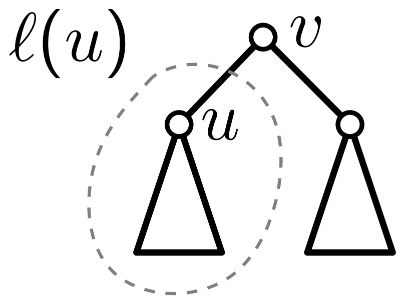


Radial Layouts – Algorithm Attempt

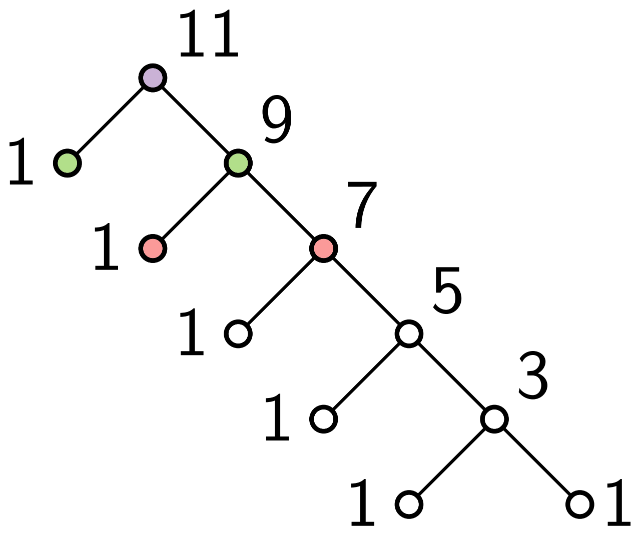
Idea

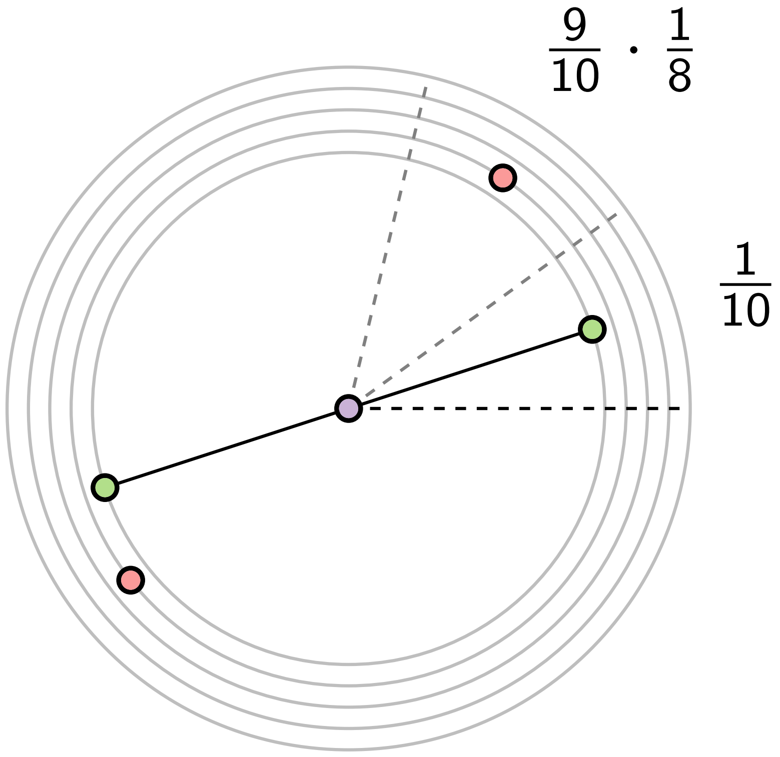
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$



- Place u in the middle of its area

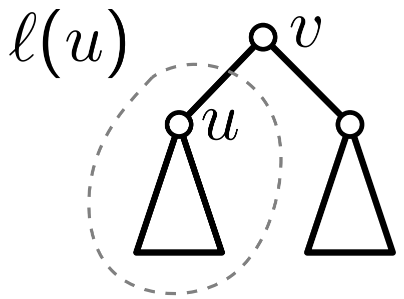




Radial Layouts – Algorithm Attempt

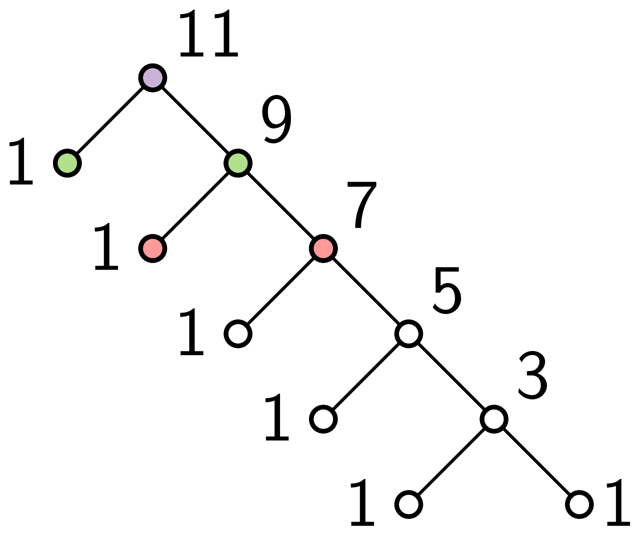
Idea

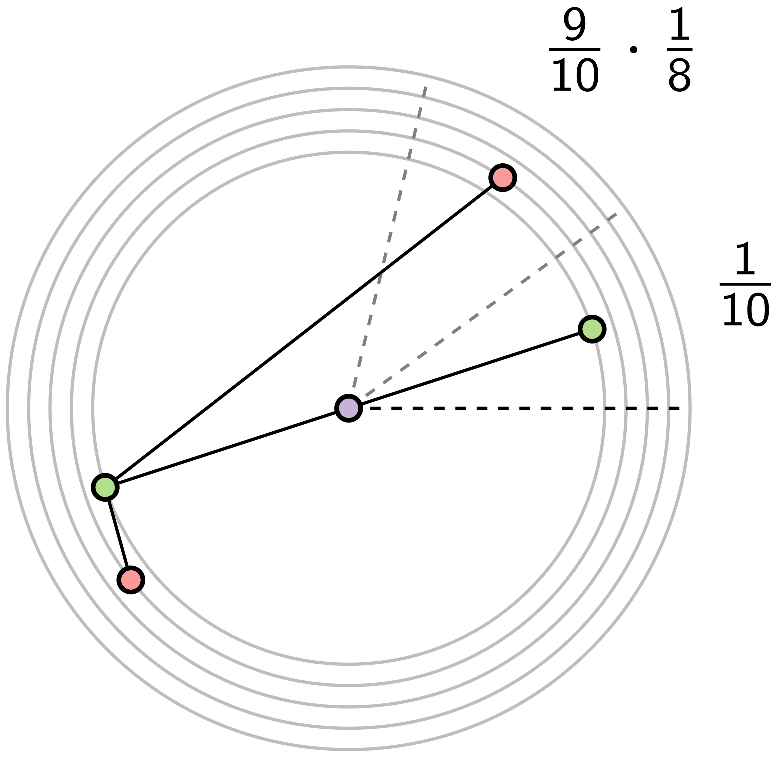
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$

- Place u in the middle of its area

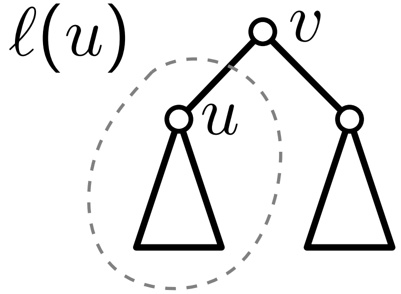




Radial Layouts – Algorithm Attempt

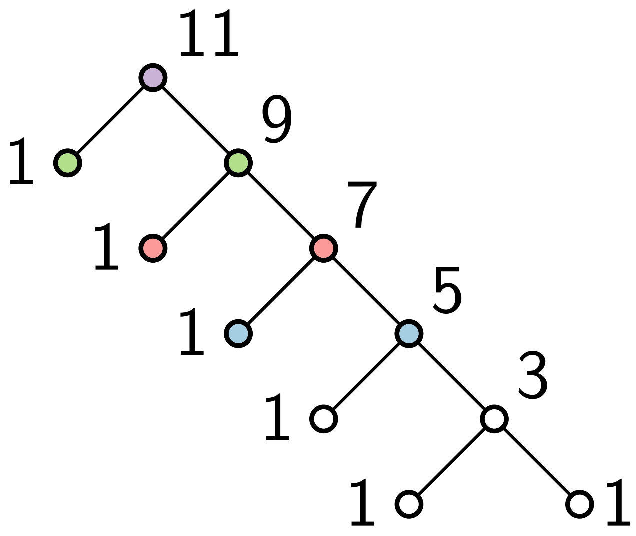
Idea

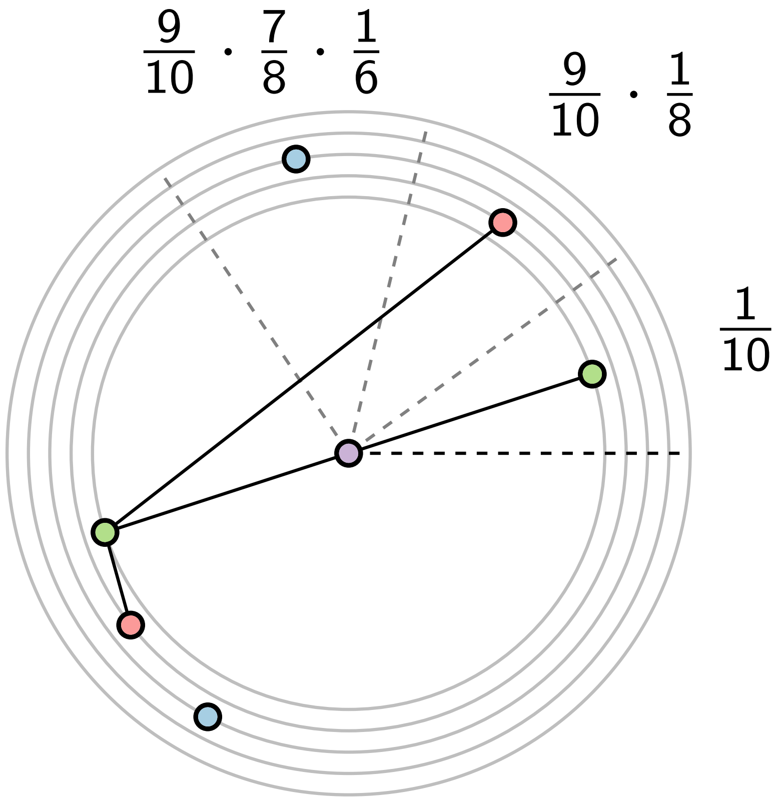
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$

- Place u in the middle of its area

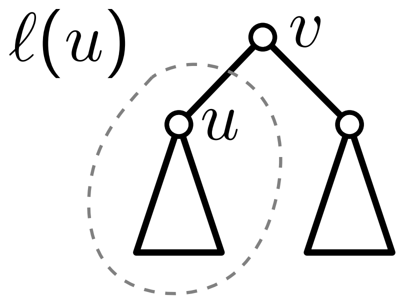




Radial Layouts – Algorithm Attempt

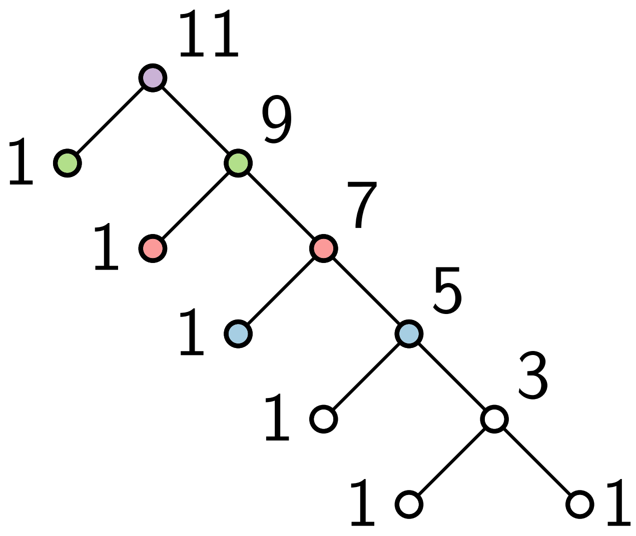
Idea

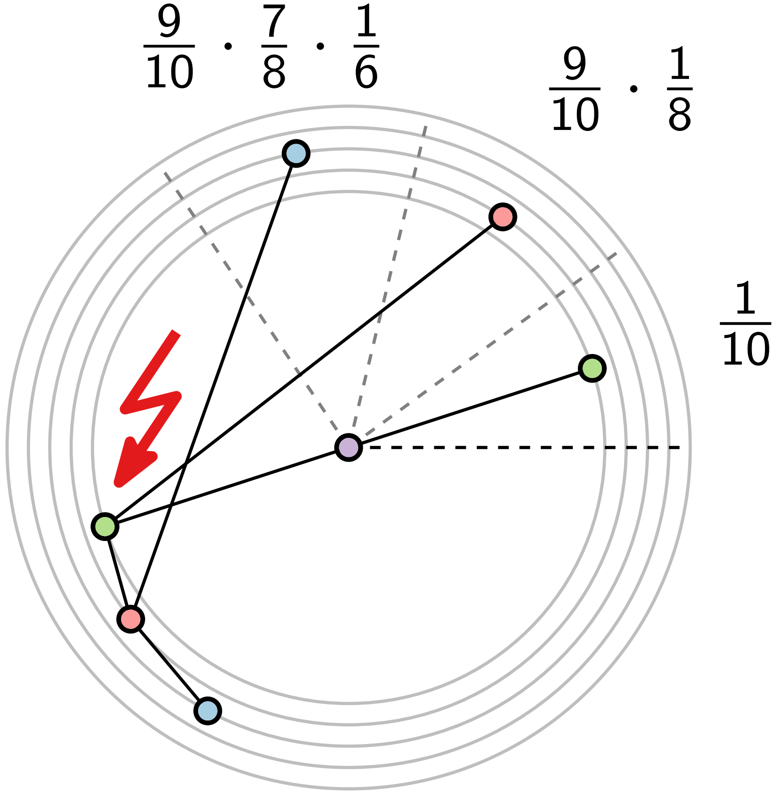
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



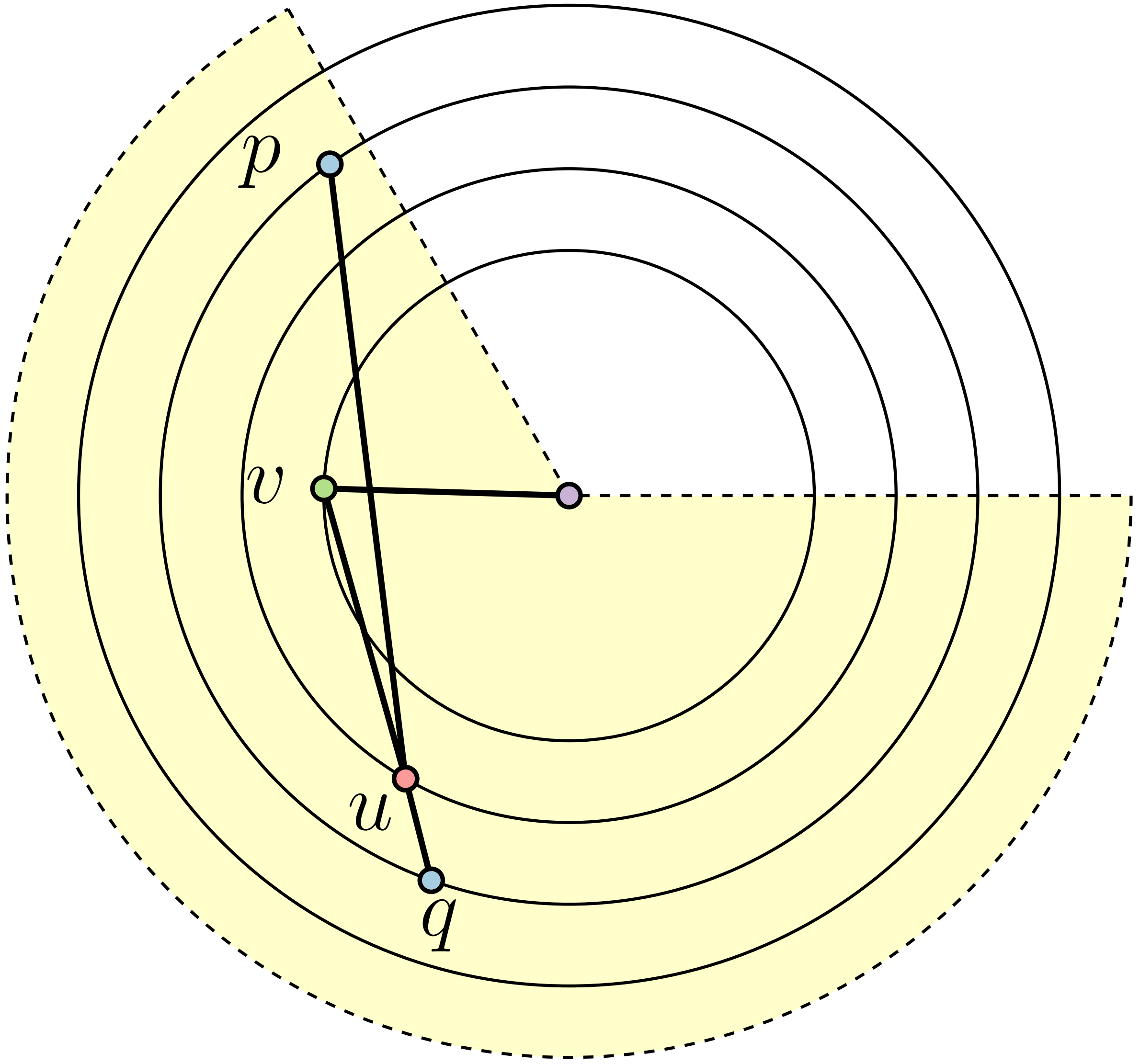
$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$

- Place u in the middle of its area

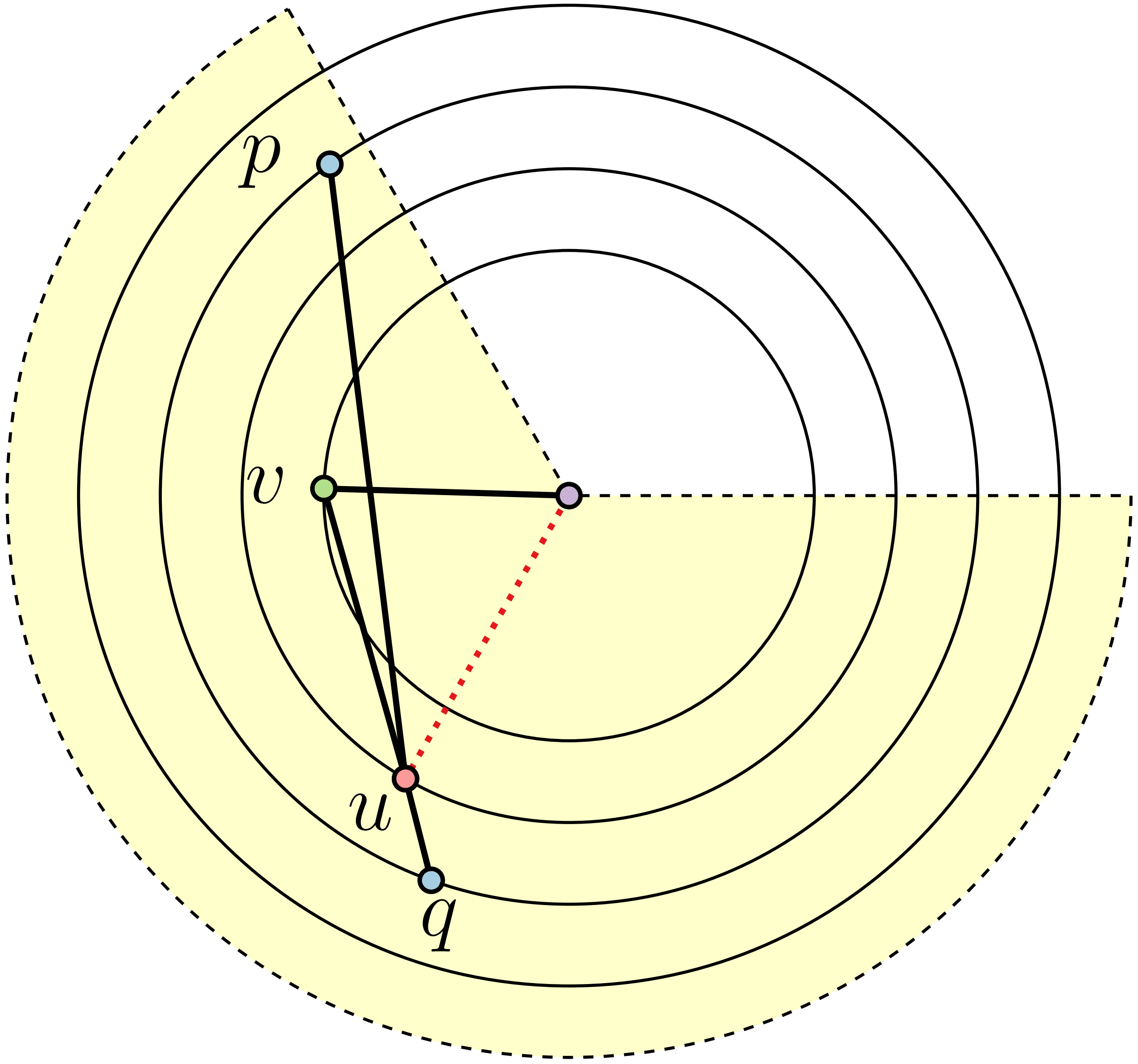




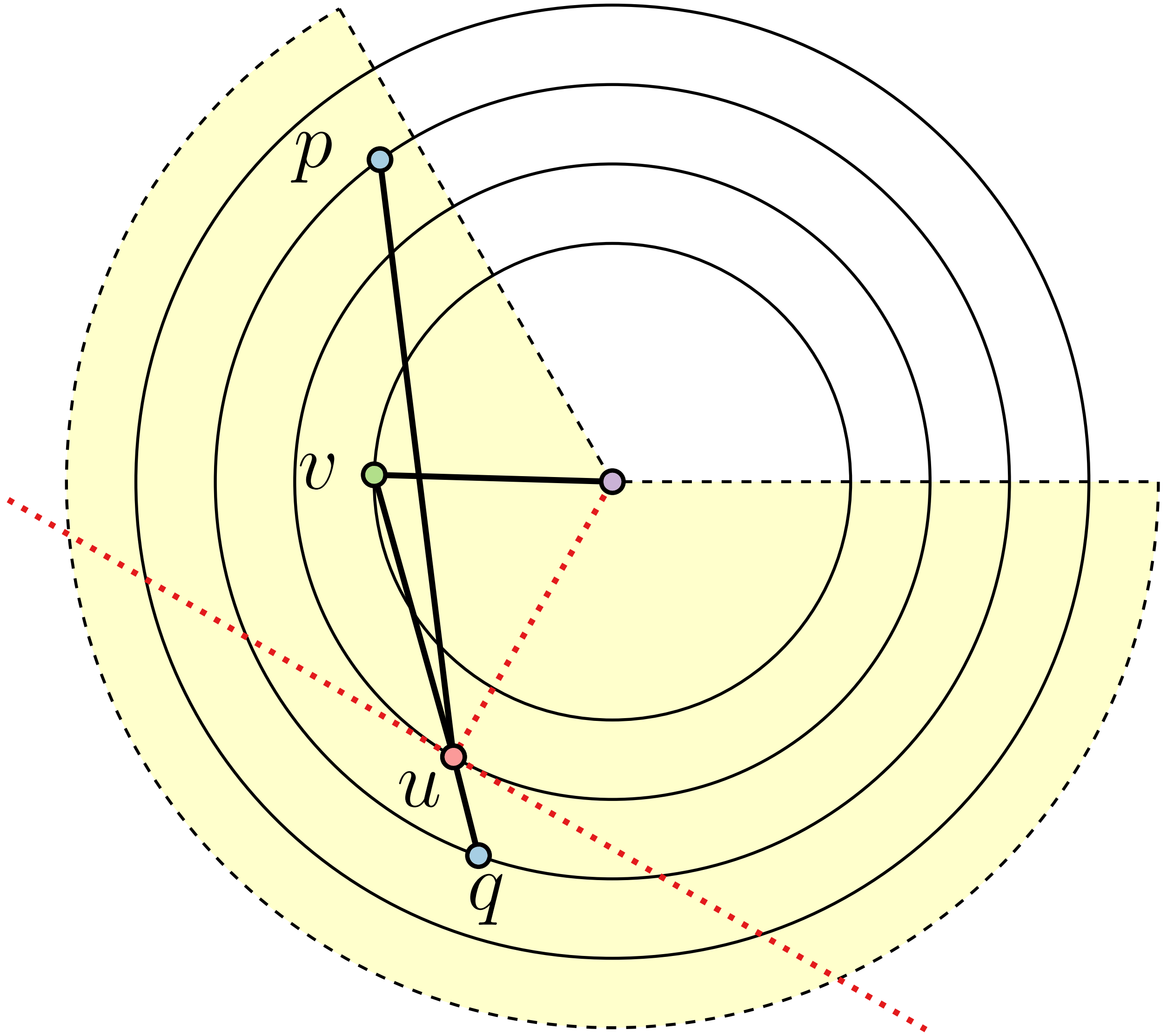
Radial Layouts – How To Avoid Crossings



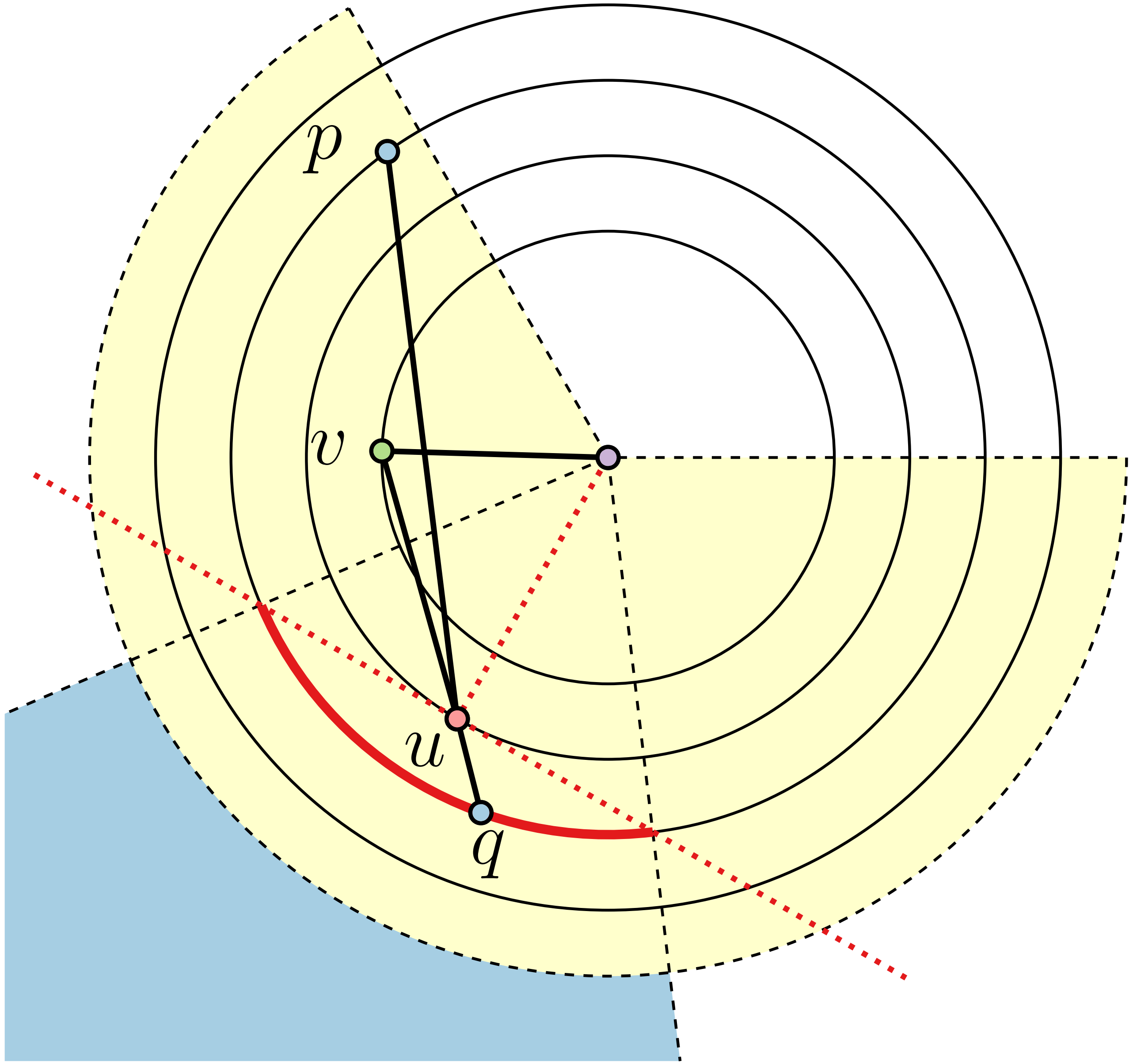
Radial Layouts – How To Avoid Crossings



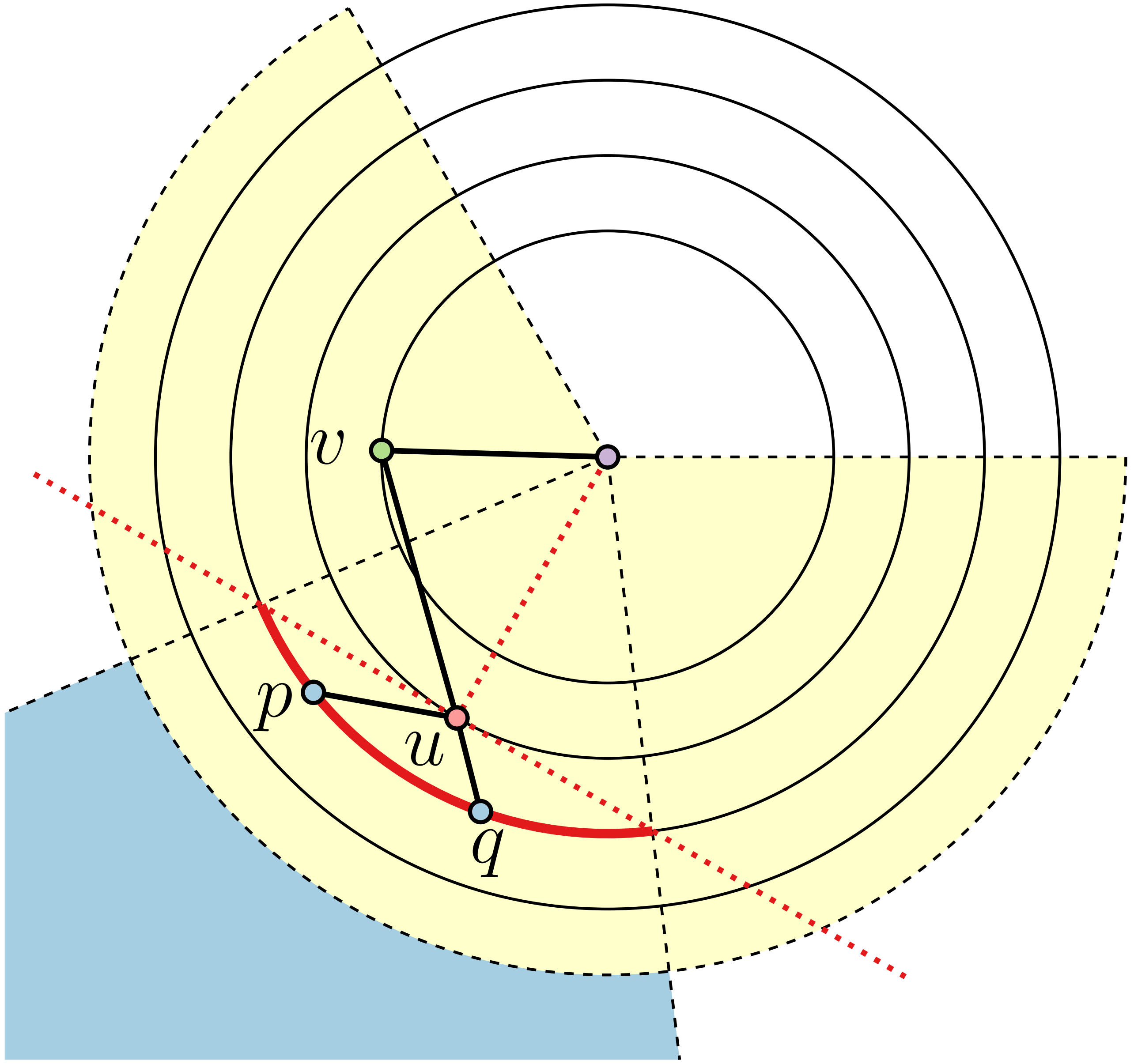
Radial Layouts – How To Avoid Crossings



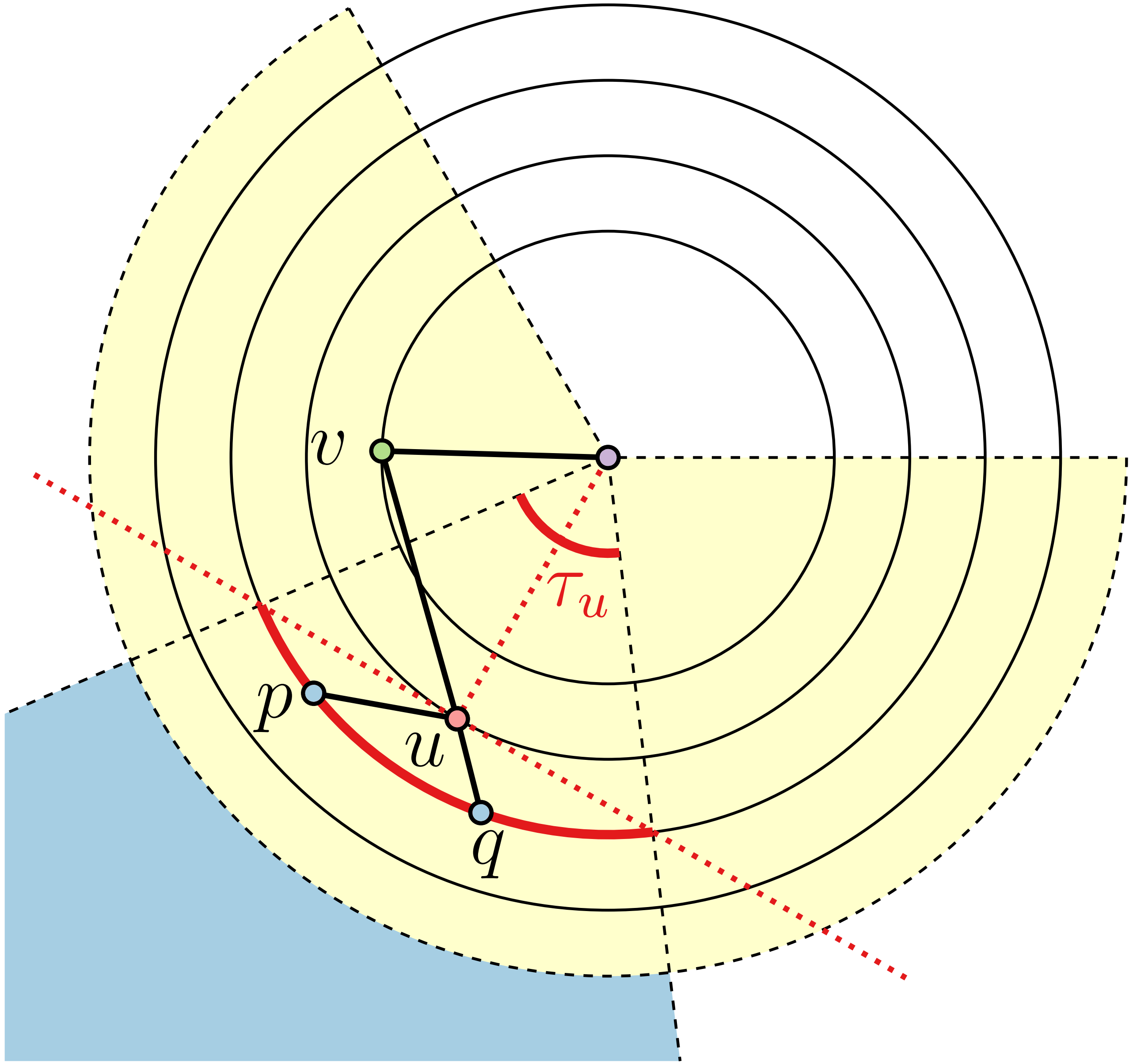
Radial Layouts – How To Avoid Crossings



Radial Layouts – How To Avoid Crossings

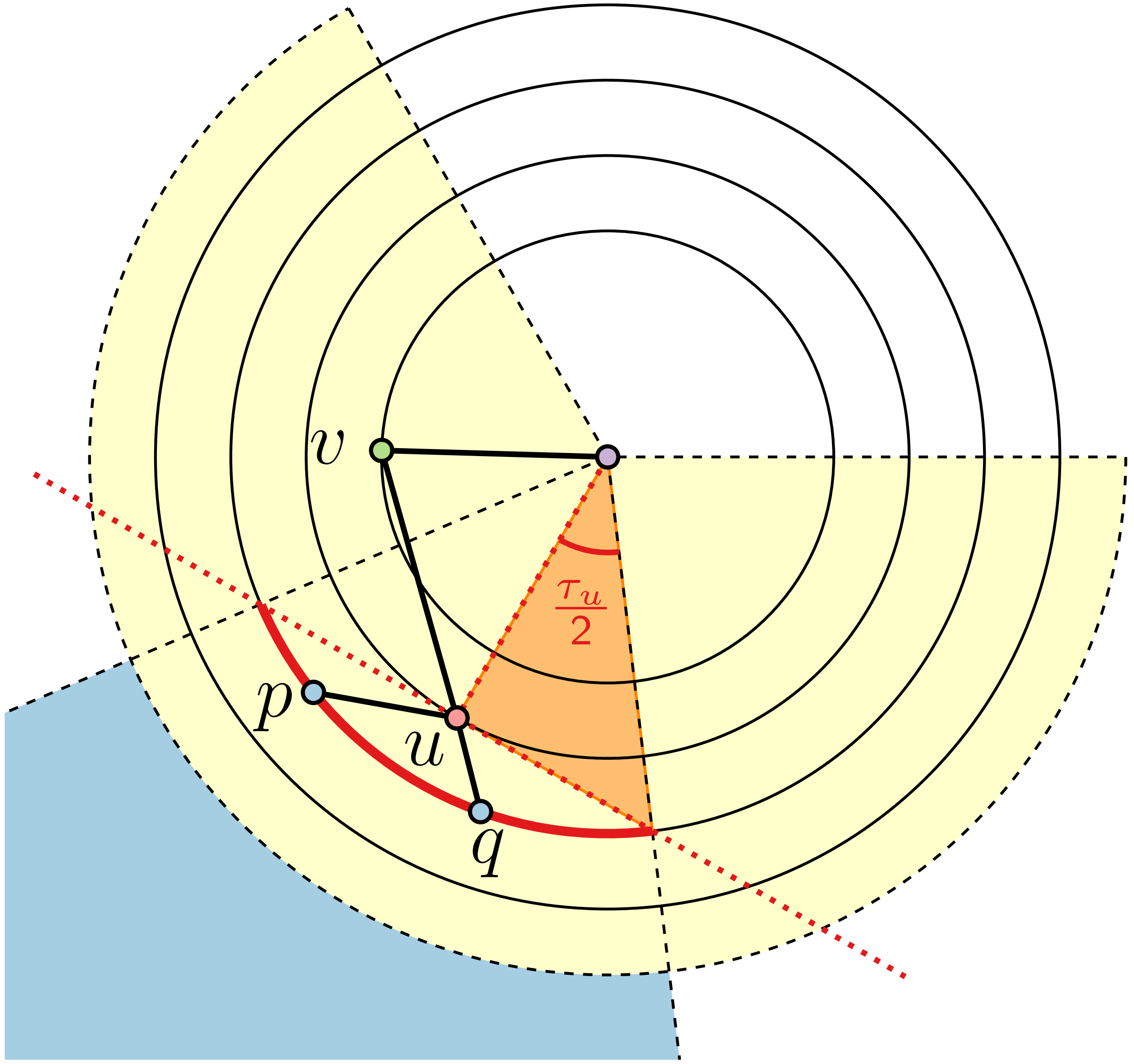


Radial Layouts – How To Avoid Crossings



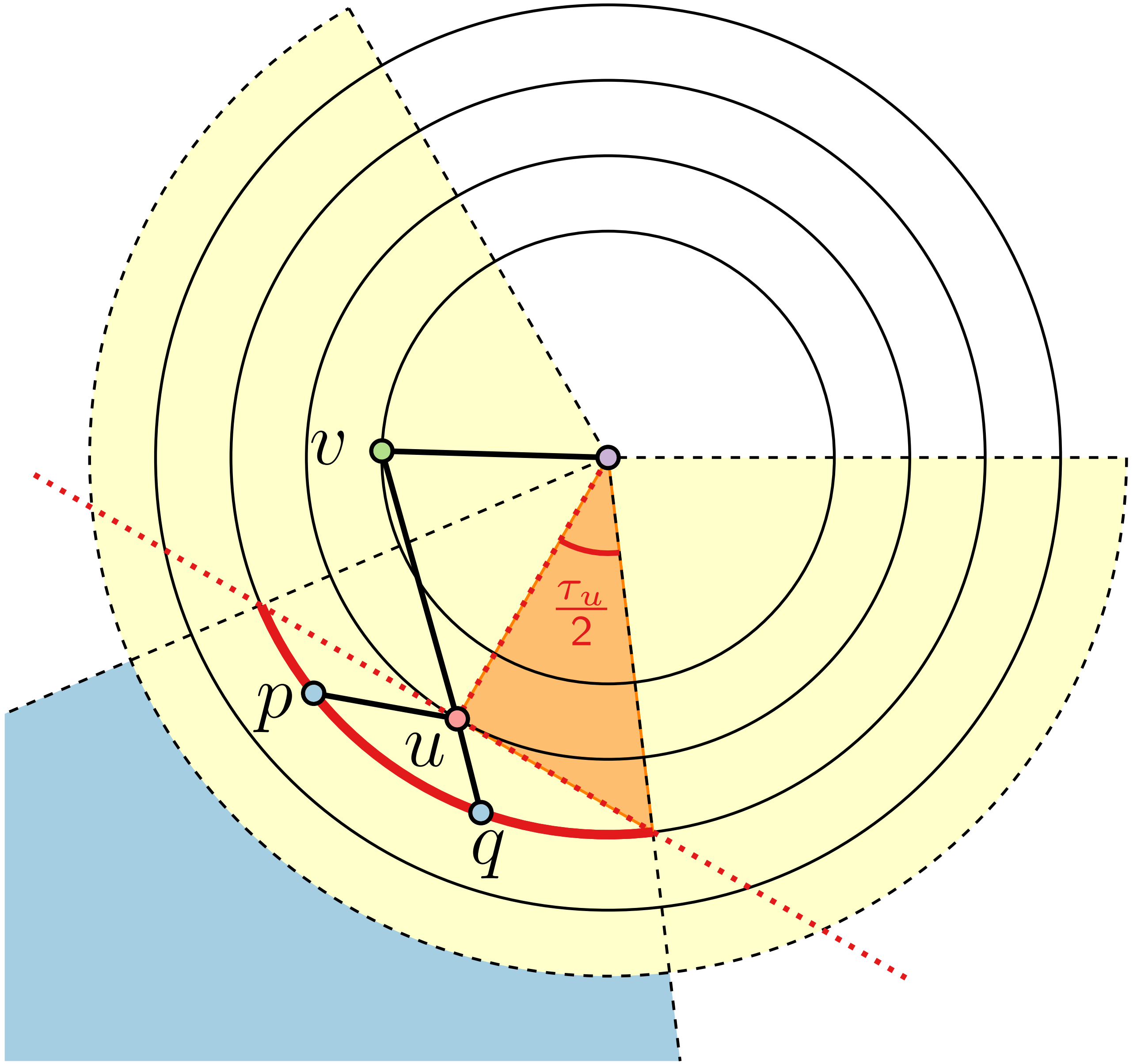
- τ_u – angle of the wedge corresponding to vertex u

Radial Layouts – How To Avoid Crossings



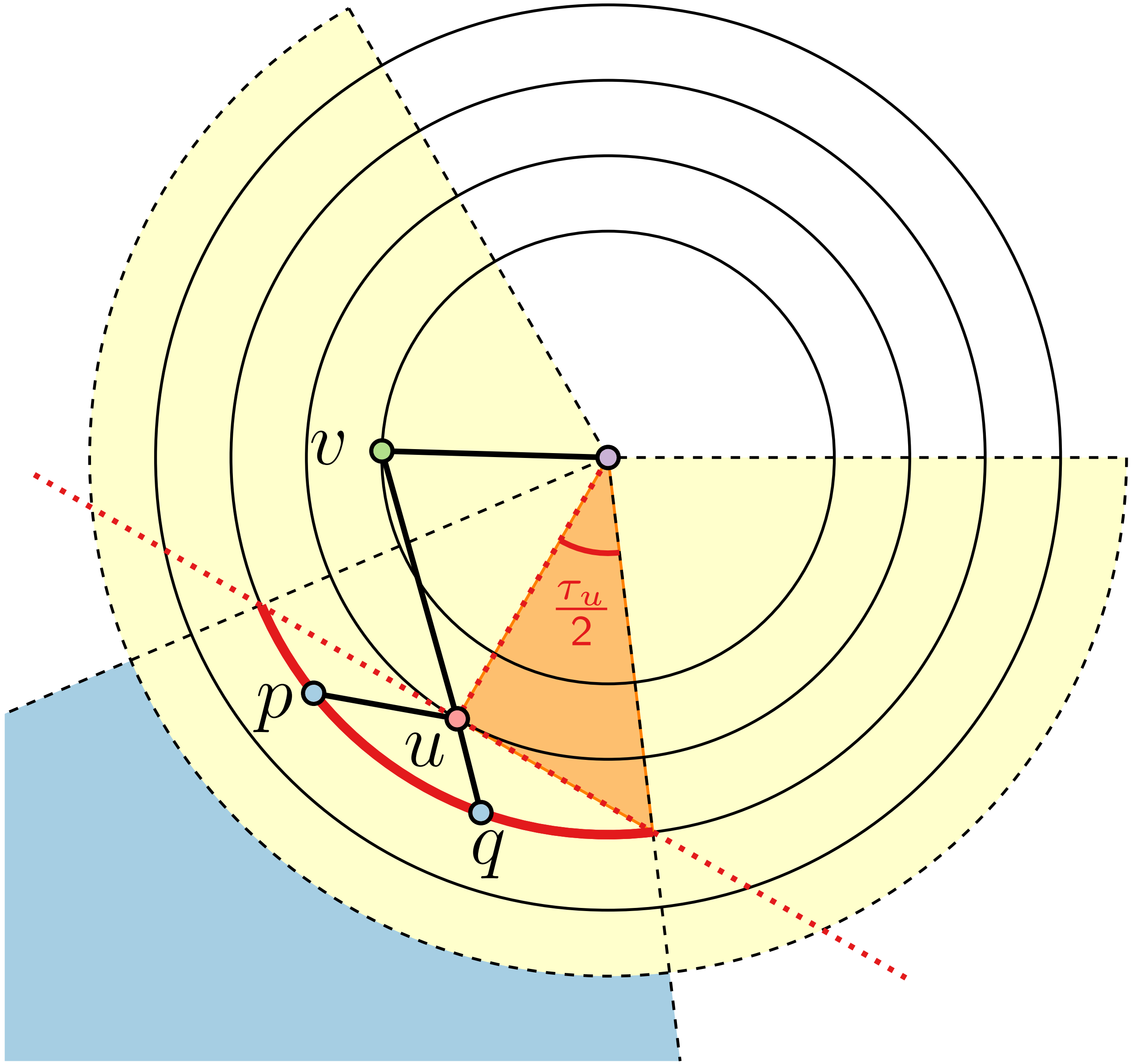
- τ_u – angle of the wedge corresponding to vertex u

Radial Layouts – How To Avoid Crossings



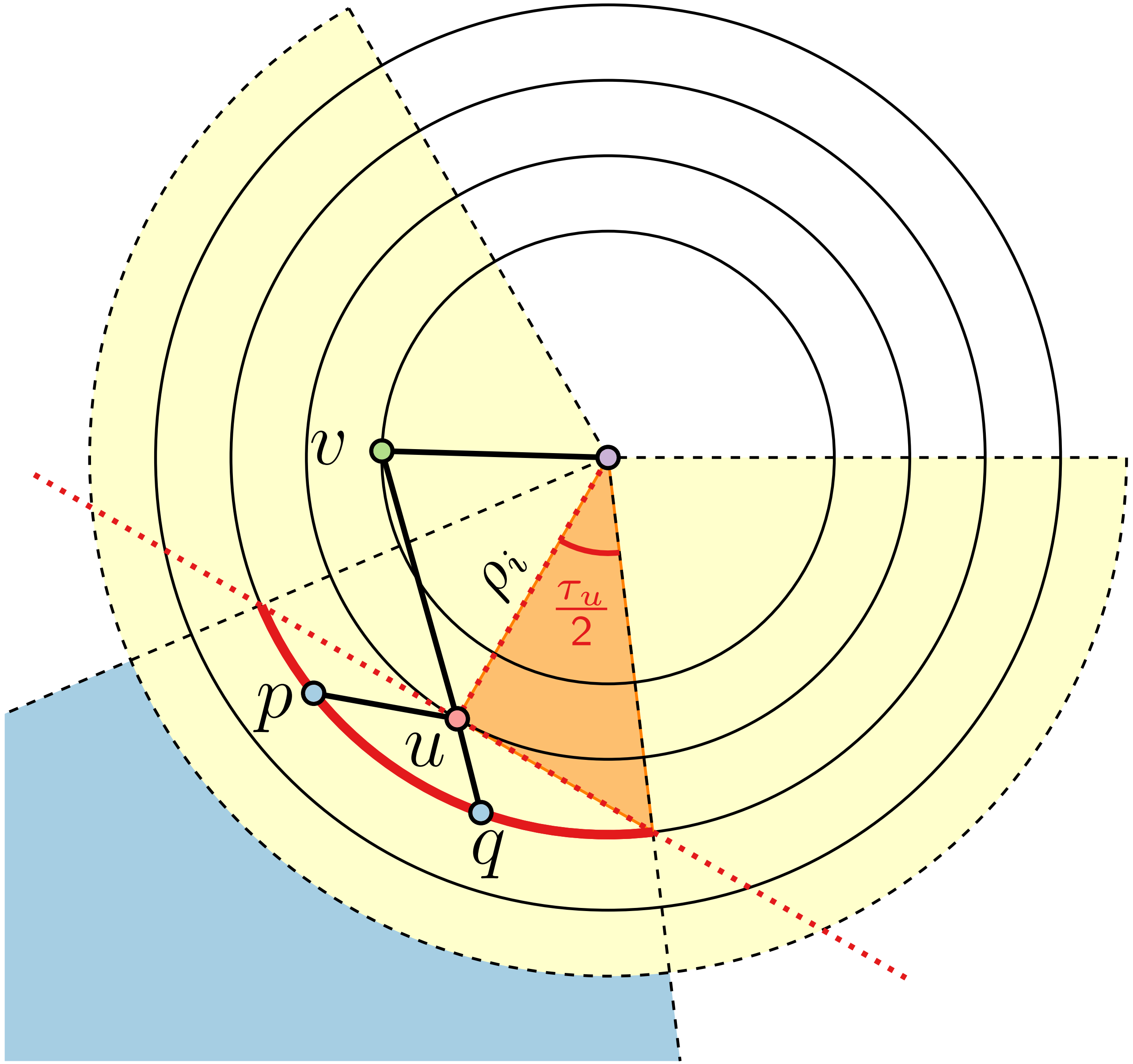
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u

Radial Layouts – How To Avoid Crossings



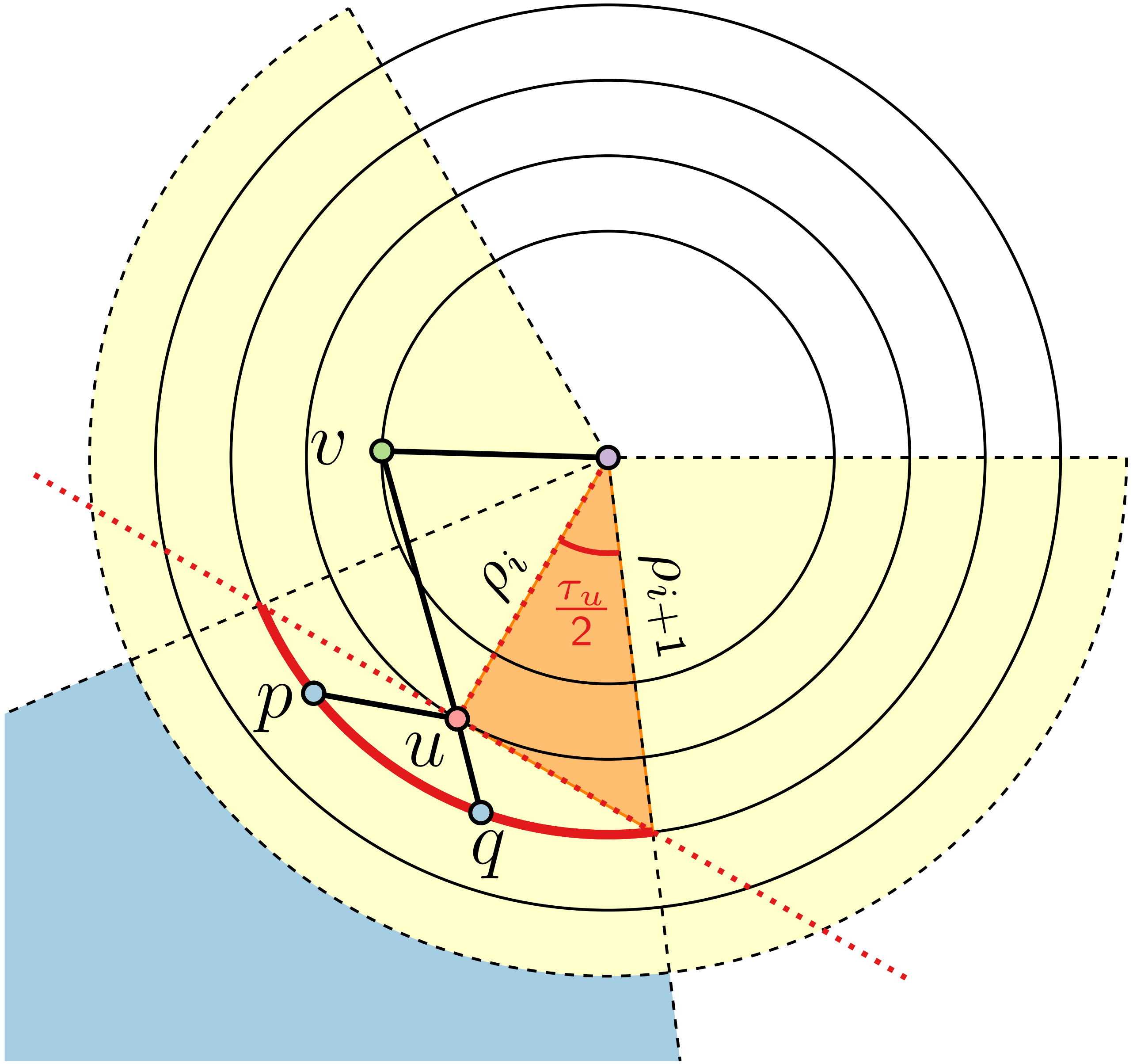
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i

Radial Layouts – How To Avoid Crossings



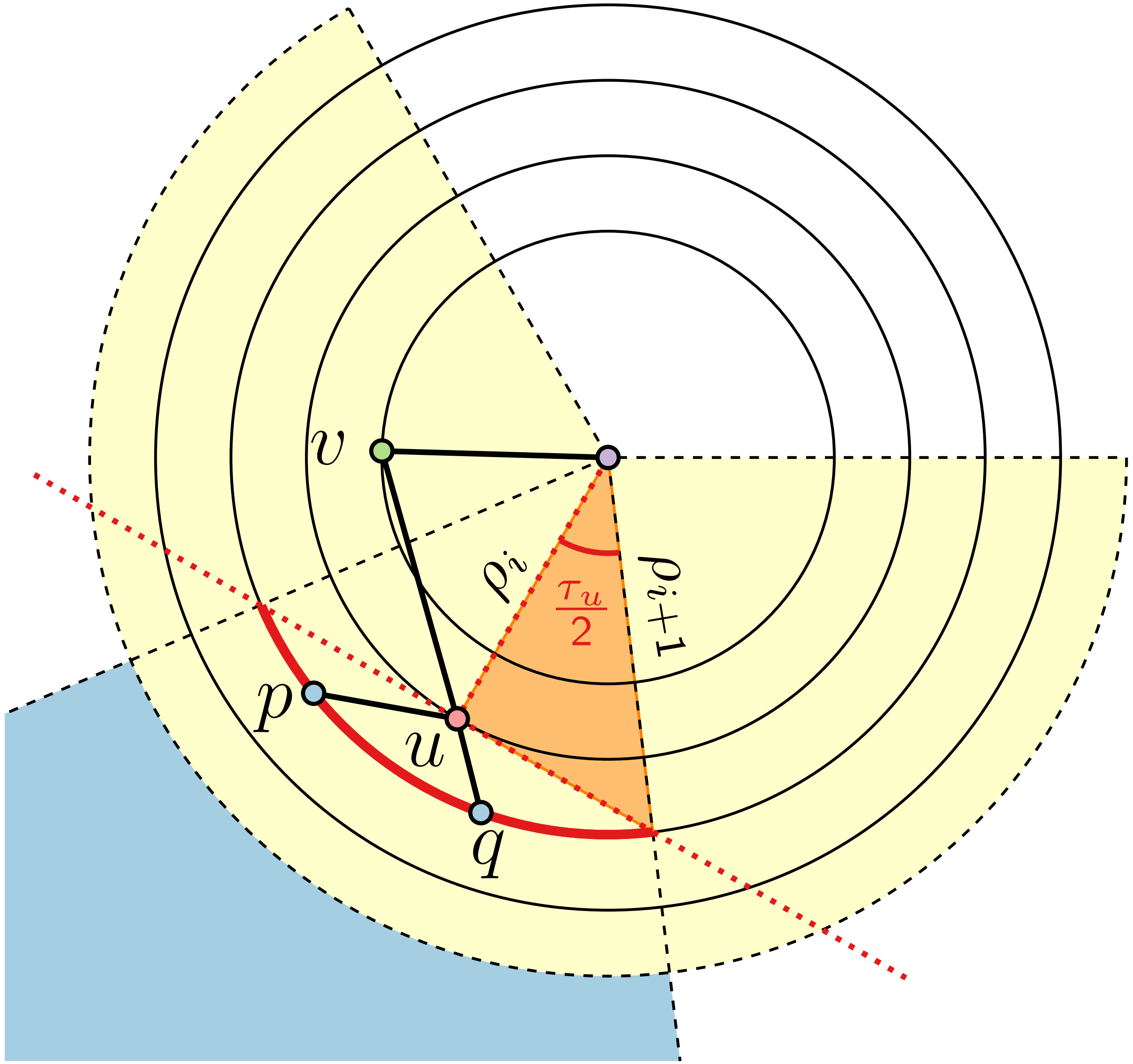
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i

Radial Layouts – How To Avoid Crossings



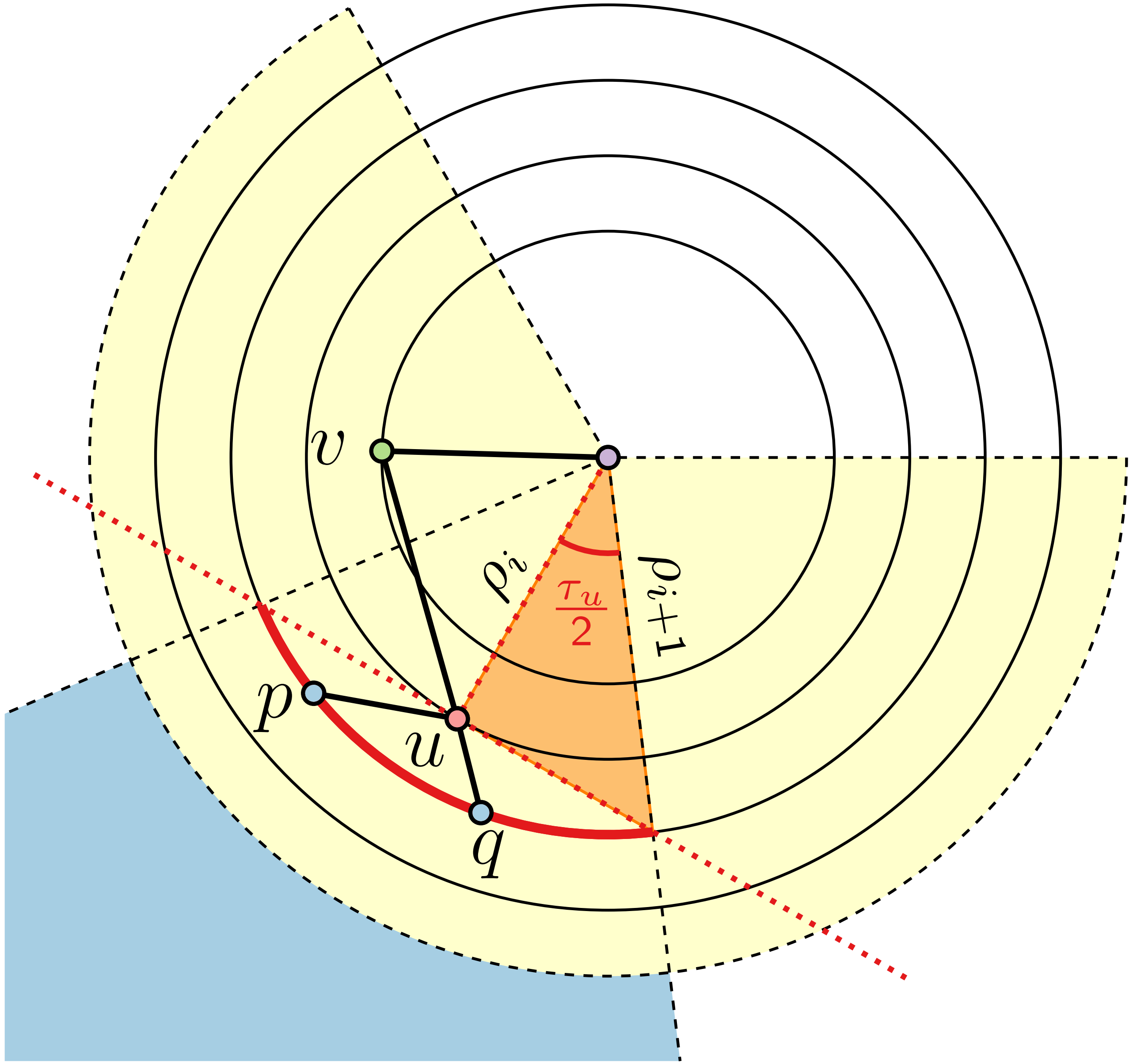
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i

Radial Layouts – How To Avoid Crossings



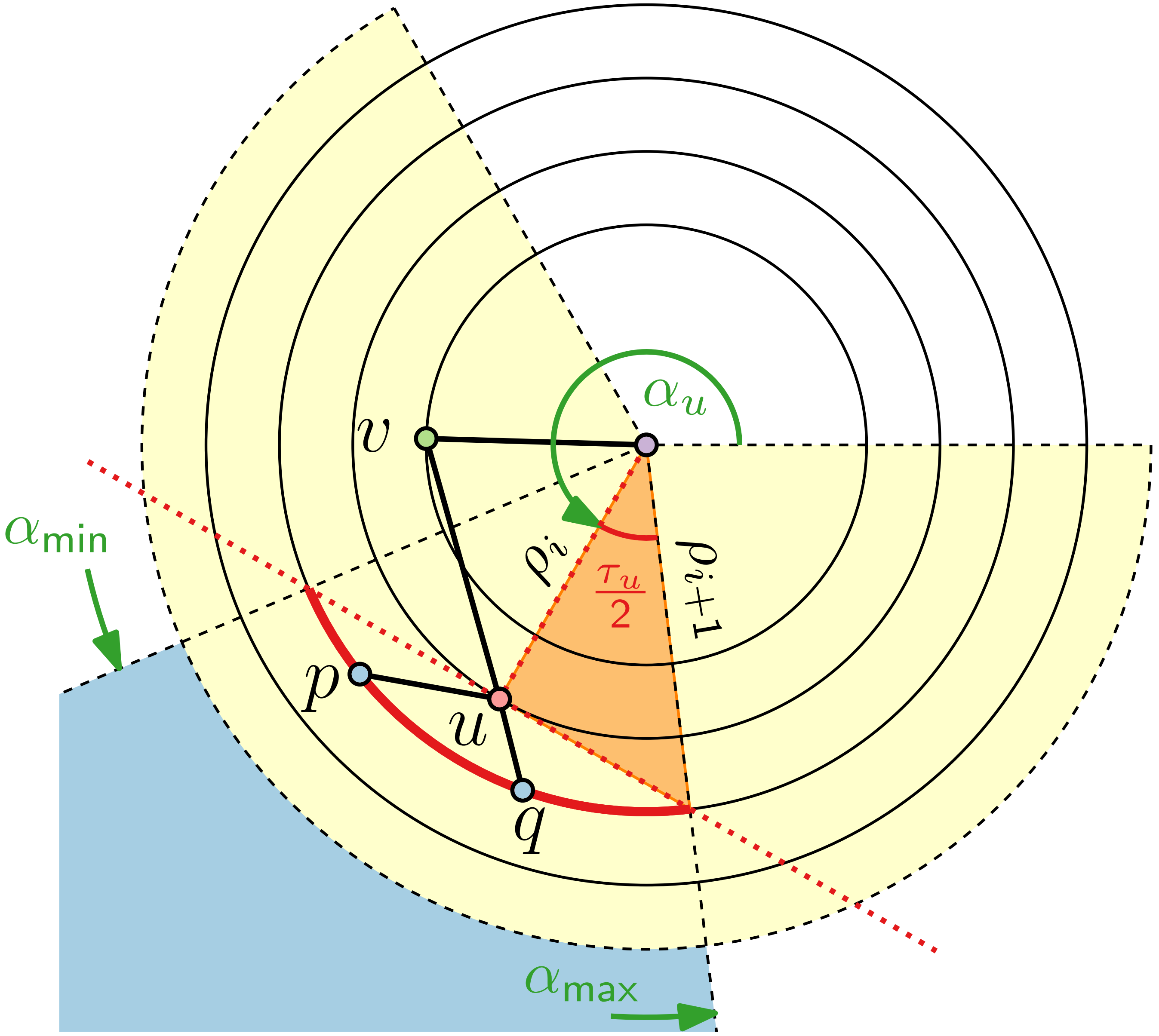
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i
- $\cos(\tau_u/2) = \rho_i/\rho_{i+1}$

Radial Layouts – How To Avoid Crossings



- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i
- $\cos(\tau_u/2) = \rho_i/\rho_{i+1}$
- $\tau_u = \min\left\{\frac{\ell(u)}{\ell(v)-1} \cdot \tau_v,\ 2\arccos\frac{\rho_i}{\rho_{i+1}}\right\}$

Radial Layouts – How To Avoid Crossings



- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i
- $\cos(\tau_u/2) = \rho_i/\rho_{i+1}$
- $\tau_u = \min\left\{\frac{\ell(u)}{\ell(v)-1} \cdot \tau_v,\ 2\arccos\frac{\rho_i}{\rho_{i+1}}\right\}$
- Alternative:

  $\alpha_{\text{min}} = \alpha_u - \arccos\frac{\rho_i}{\rho_{i+1}}$

  $\alpha_{\text{max}} = \alpha_u + \arccos\frac{\rho_i}{\rho_{i+1}}$

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

postorder(r)
preorder($r, 0, 0, 2\pi$)
return $(d_v, \alpha_v)_{v \in V(T)}$
// vertex positions in polar coordinates

Radial Layouts – Pseudocode

```
RadialTreeLayout(tree T, root r ∈ T, radii ρ_1 < ··· < ρ_k)
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex positions in polar coordinates
```

```
postorder(vertex v)
    ℓ(v) ← 1
    foreach child w of v do
        compute the size of
        the subtree recursively.
```

Radial Layouts – Pseudocode

```
RadialTreeLayout(tree T, root r ∈ T, radii ρ_1 < ··· < ρ_k)
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex positions in polar coordinates
```

```
postorder(vertex v)
    ℓ(v) ← 1
    foreach child w of v do
        postorder(w)
        ℓ(v) ← ℓ(v) + ℓ(w)
```

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

```
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex positions in polar coordinates
```

postorder(vertex v)

```
    ℓ(v) ← 1
    foreach child w of v do
        postorder(w)
        ℓ(v) ← ℓ(v) + ℓ(w)
```

preorder(vertex v, $t, \alpha_{\mathrm{min}}, \alpha_{\mathrm{max}}$)

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

```
postorder(r)
preorder(r, 0, 0, 2π)
return (d_v, α_v)_{v∈V(T)}
// vertex positions in polar coordinates
```

postorder(vertex v)

```
ℓ(v) ← 1
foreach child w of v do
    postorder(w)
    ℓ(v) ← ℓ(v) + ℓ(w)
```

preorder(vertex v, $t, \alpha_{\mathsf{min}}, \alpha_{\mathsf{max}}$)

```
d_v ← max{0, ρ_t}
```

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

```
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex positions in polar coordinates
```

postorder(vertex v)

```
    ℓ(v) ← 1
    foreach child w of v do
        postorder(w)
        ℓ(v) ← ℓ(v) + ℓ(w)
```

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)

```
    d_v ← max{0, ρ_t}
    α_v ← (α_min + α_max)/2
```

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

```
postorder(r)
preorder(r, 0, 0, 2π)
return (d_v, α_v)_{v∈V(T)}
// vertex positions in polar coordinates
```

postorder(vertex v)

```
ℓ(v) ← 1
foreach child w of v do
    postorder(w)
    ℓ(v) ← ℓ(v) + ℓ(w)
```

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)

```
d_v ← max{0, ρ_t}
α_v ← (α_min + α_max)/2
```

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

- postorder(r)
- preorder($r, 0, 0, 2\pi$)
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex positions in polar coordinates

postorder(vertex v)

- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - postorder(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)

- $d_v \leftarrow \max\{0, \rho_t\}$
- $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ // output

Radial Layouts – Pseudocode

```
RadialTreeLayout(tree T, root r ∈ T, radii ρ_1 < ··· < ρ_k)
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex positions in polar coordinates
```

```
postorder(vertex v)
    ℓ(v) ← 1
    foreach child w of v do
        postorder(w)
        ℓ(v) ← ℓ(v) + ℓ(w)
```

```
preorder(vertex v, t, α_min, α_max)
    d_v ← max{0, ρ_t}                        // output
    α_v ← (α_min + α_max)/2
    if t > 0 then
```

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \dots < \rho_k$)

```
postorder(r)
preorder(r, 0, 0, 2π)
return (d_v, α_v)_{v ∈ V(T)}
// vertex positions in polar coordinates
```

postorder(vertex v)

```
ℓ(v) ← 1
foreach child w of v do
    postorder(w)
    ℓ(v) ← ℓ(v) + ℓ(w)
```

preorder(vertex v, $t, \alpha_{\text{min}}, \alpha_{\text{max}}$)

```
d_v ← max{0, ρ_t}                      // output
α_v ← (α_min + α_max)/2
if t > 0 then
    α_min ← max{α_min, α_v − arccos (ρ_t / ρ_{t+1})}
    α_max ← min{α_max, α_v + arccos (ρ_t / ρ_{t+1})}
```

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

- postorder(r)
- preorder($r, 0, 0, 2\pi$)
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex positions in polar coordinates

postorder(vertex v)

- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - postorder(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)

- $d_v \leftarrow \max\{0, \rho_t\}$
- $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ // output
- **if** $t > 0$ **then**
  - $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  - $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
- *left* $\leftarrow \alpha_{\min}$

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)
- postorder(r)
- preorder($r, 0, 0, 2\pi$)
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex positions in polar coordinates

postorder(vertex v)
- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - postorder(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)
- $d_v \leftarrow \max\{0, \rho_t\}$
- $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ // output
- **if** $t > 0$ **then**
  - $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  - $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
- *left* $\leftarrow \alpha_{\min}$
- **foreach** child w of v **do**

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

```
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex positions in polar coordinates
```

postorder(vertex v)

```
    ℓ(v) ← 1
    foreach child w of v do
        postorder(w)
        ℓ(v) ← ℓ(v) + ℓ(w)
```

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)

```
    d_v ← max{0, ρ_t}                    // output
    α_v ← (α_min + α_max)/2
    if t > 0 then
        α_min ← max{α_min, α_v − arccos ρ_t/ρ_{t+1}}
        α_max ← min{α_max, α_v + arccos ρ_t/ρ_{t+1}}
    left ← α_min
    foreach child w of v do
        right ← left + ℓ(w)/(ℓ(v)−1) · (α_max − α_min)
```

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)
- postorder(r)
- preorder($r, 0, 0, 2\pi$)
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex positions in polar coordinates

postorder(vertex v)
- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - postorder(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)
- $d_v \leftarrow \max\{0, \rho_t\}$ // output
- $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$
- **if** $t > 0$ **then**
  - $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  - $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
- $\mathit{left} \leftarrow \alpha_{\min}$
- **foreach** child w of v **do**
  - $\mathit{right} \leftarrow \mathit{left} + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$
  - preorder($w, t+1, \mathit{left}, \mathit{right}$)

Radial Layouts – Pseudocode

```
RadialTreeLayout(tree T, root r ∈ T, radii ρ_1 < ··· < ρ_k)
    postorder(r)
    preorder(r, 0, 0, 2π)
    return (d_v, α_v)_{v∈V(T)}
    // vertex positions in polar coordinates
```

```
postorder(vertex v)
    ℓ(v) ← 1
    foreach child w of v do
        postorder(w)
        ℓ(v) ← ℓ(v) + ℓ(w)
```

```
preorder(vertex v, t, α_min, α_max)
    d_v ← max{0, ρ_t}                         // output
    α_v ← (α_min + α_max)/2
    if t > 0 then
        α_min ← max{α_min, α_v − arccos ρ_t/ρ_{t+1}}
        α_max ← min{α_max, α_v + arccos ρ_t/ρ_{t+1}}
    left ← α_min
    foreach child w of v do
        right ← left + ℓ(w)/(ℓ(v)−1) · (α_max − α_min)
        preorder(w, t + 1, left, right)
        left ← right
```

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

- postorder(r)
- preorder($r, 0, 0, 2\pi$)
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex positions in polar coordinates

postorder(vertex v)

- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - postorder(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)

- $d_v \leftarrow \max\{0, \rho_t\}$
- $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ // output
- **if** $t > 0$ **then**
  - $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  - $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
- $\mathit{left} \leftarrow \alpha_{\min}$
- **foreach** child w of v **do**
  - $\mathit{right} \leftarrow \mathit{left} + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$
  - preorder($w, t+1, \mathit{left}, \mathit{right}$)
  - $\mathit{left} \leftarrow \mathit{right}$

Runtime?

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)
- postorder(r)
- preorder($r, 0, 0, 2\pi$)
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex positions in polar coordinates

postorder(vertex v)
- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - postorder(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)
- $d_v \leftarrow \max\{0, \rho_t\}$ // output
- $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$
- **if** $t > 0$ **then**
  - $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  - $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
- $\mathit{left} \leftarrow \alpha_{\min}$
- **foreach** child w of v **do**
  - $\mathit{right} \leftarrow \mathit{left} + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$
  - preorder($w, t+1, \mathit{left}, \mathit{right}$)
  - $\mathit{left} \leftarrow \mathit{right}$

Runtime? $\mathcal{O}(n)$

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)
- postorder(r)
- preorder($r, 0, 0, 2\pi$)
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex positions in polar coordinates

postorder(vertex v)
- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - postorder(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, $t, \alpha_{\min}, \alpha_{\max}$)
- $d_v \leftarrow \max\{0, \rho_t\}$
- $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ // output
- **if** $t > 0$ **then**
  - $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  - $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
- $\mathit{left} \leftarrow \alpha_{\min}$
- **foreach** child w of v **do**
  - $\mathit{right} \leftarrow \mathit{left} + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$
  - preorder($w, t+1, \mathit{left}, \mathit{right}$)
  - $\mathit{left} \leftarrow \mathit{right}$

Runtime? $\mathcal{O}(n)$

Correctness?

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)

- postorder(r)
- preorder($r, 0, 0, 2\pi$)
- **return** $(d_v, \alpha_v)_{v \in V(T)}$
- // vertex positions in polar coordinates

postorder(vertex v)

- $\ell(v) \leftarrow 1$
- **foreach** child w of v **do**
  - postorder(w)
  - $\ell(v) \leftarrow \ell(v) + \ell(w)$

preorder(vertex v, $t, \alpha_{\text{min}}, \alpha_{\text{max}}$)

- $d_v \leftarrow \max\{0, \rho_t\}$
- $\alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2$ // output
- **if** $t > 0$ **then**
  - $\alpha_{\text{min}} \leftarrow \max\{\alpha_{\text{min}}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$
  - $\alpha_{\text{max}} \leftarrow \min\{\alpha_{\text{max}}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$
- $\mathit{left} \leftarrow \alpha_{\text{min}}$
- **foreach** child w of v **do**
  - $\mathit{right} \leftarrow \mathit{left} + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\text{max}} - \alpha_{\text{min}})$
  - preorder($w, t+1, \mathit{left}, \mathit{right}$)
  - $\mathit{left} \leftarrow \mathit{right}$

Runtime? $\mathcal{O}(n)$

Correctness? ✓

Radial Layouts – Result

Theorem.
Let T be a rooted tree with n vertices. The algorithm RadialTreeLayout constructs in $O(n)$ time a drawing Γ of T s.t.:

Radial Layouts – Result

Theorem.
Let T be a rooted tree with n vertices. The algorithm RadialTreeLayout constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is a radial, crossing-free drawing,

Radial Layouts – Result

Theorem.
Let T be a rooted tree with n vertices. The algorithm RadialTreeLayout constructs in $O(n)$ time a drawing Γ of T s.t.:

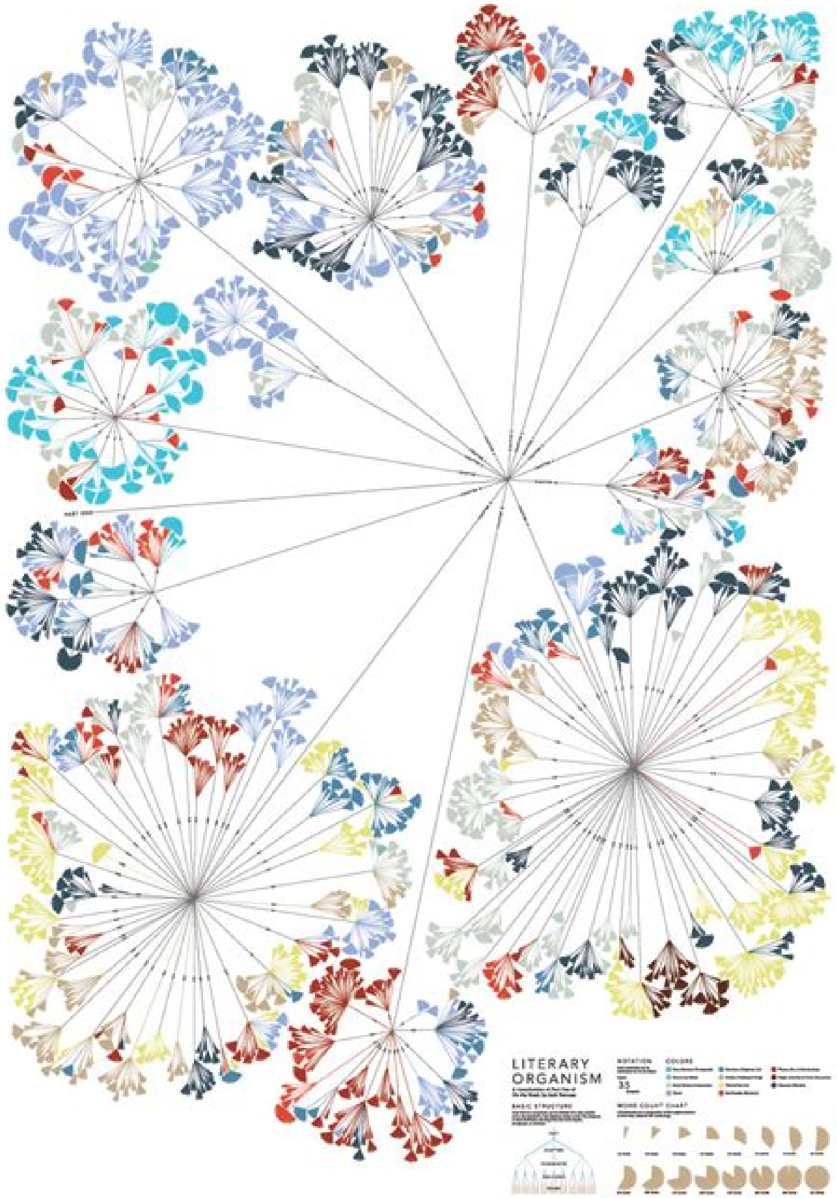
- Γ is a radial, crossing-free drawing,
- vertices lie on circles according to their depth, and

Radial Layouts – Result

Theorem.
Let T be a rooted tree with n vertices. The algorithm RadialTreeLayout constructs in $O(n)$ time a drawing Γ of T s.t.:

- Γ is a radial, crossing-free drawing,
- vertices lie on circles according to their depth, and
- the area of Γ is quadratic in max-degree(T) $\times$ height(T) (see [GD Ch. 3.1.3] for the details).

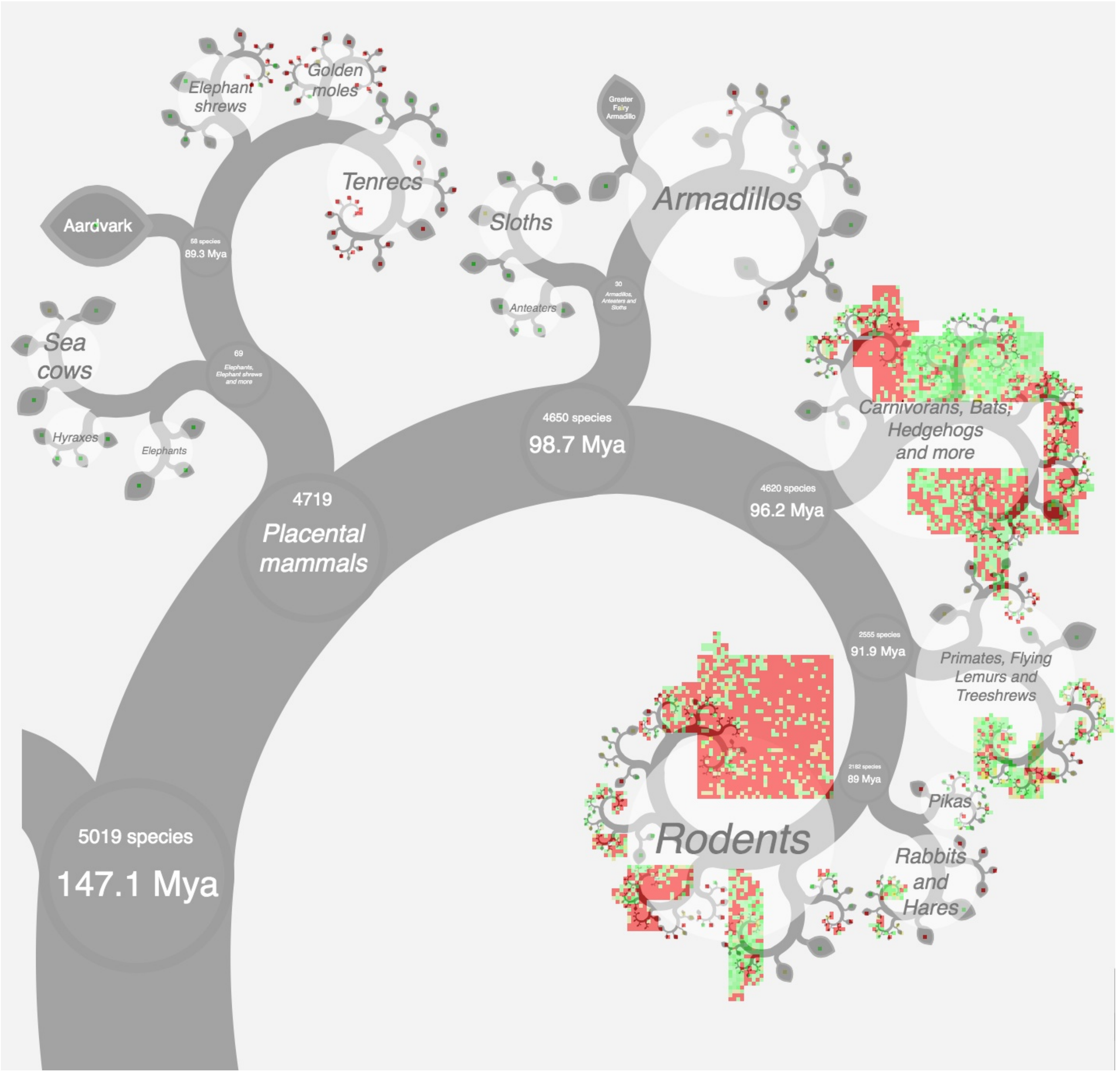
Other Tree Visualization Styles



Writing Without Words:
The project explores methods to visualize the differences in writing styles of different authors.

Similar to balloon layout

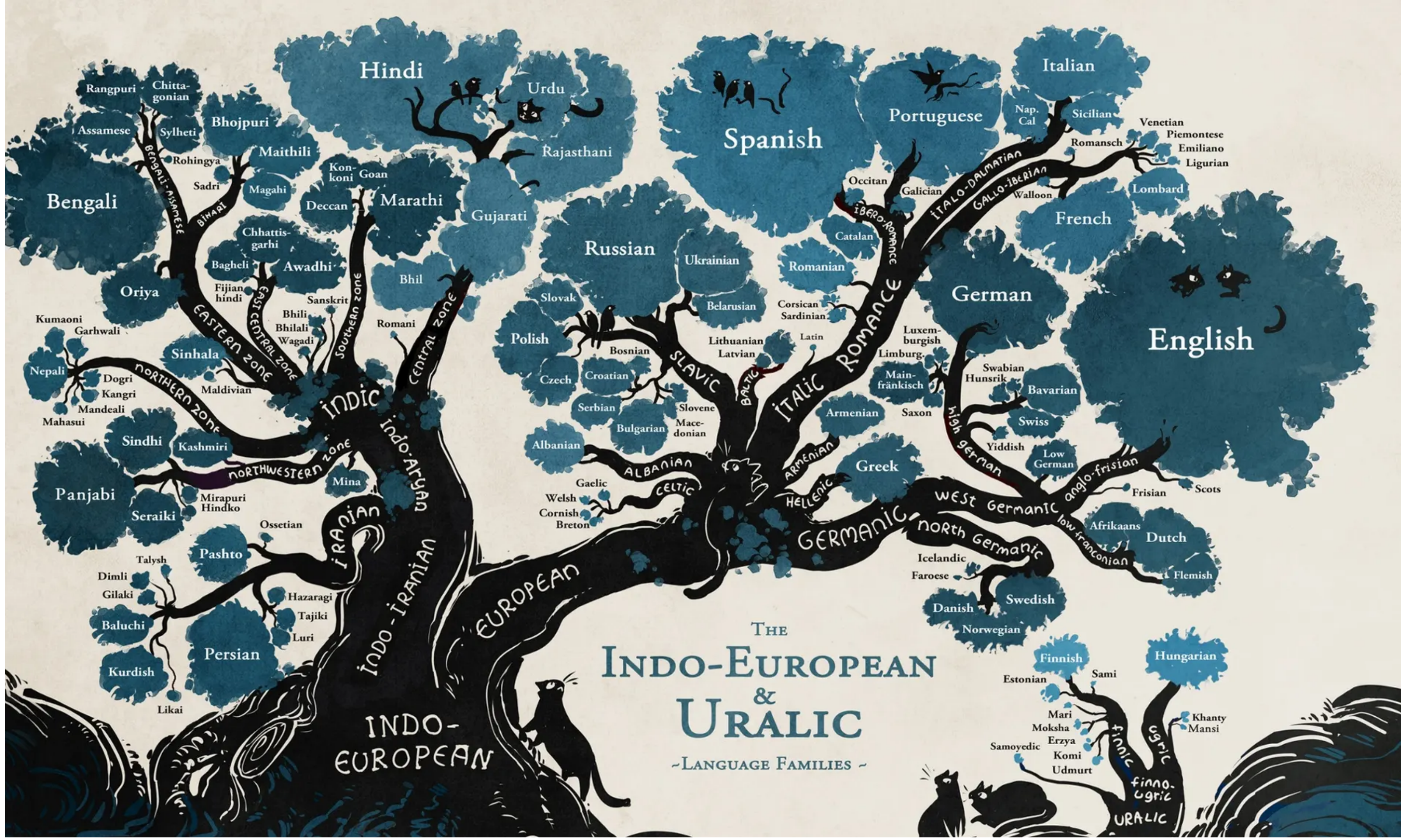
Other Tree Visualization Styles



A phylogenetically organized display of data for all placental mammal species.

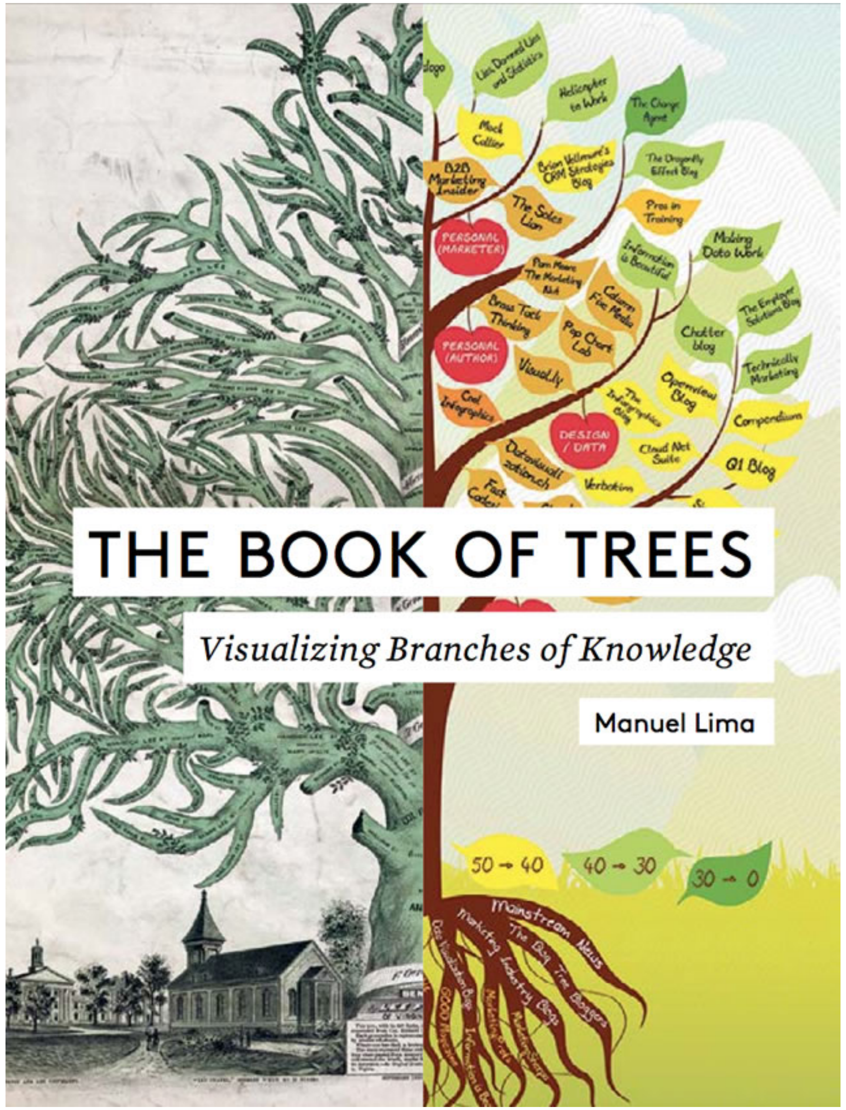
Fractal layout

Other Tree Visualization Styles

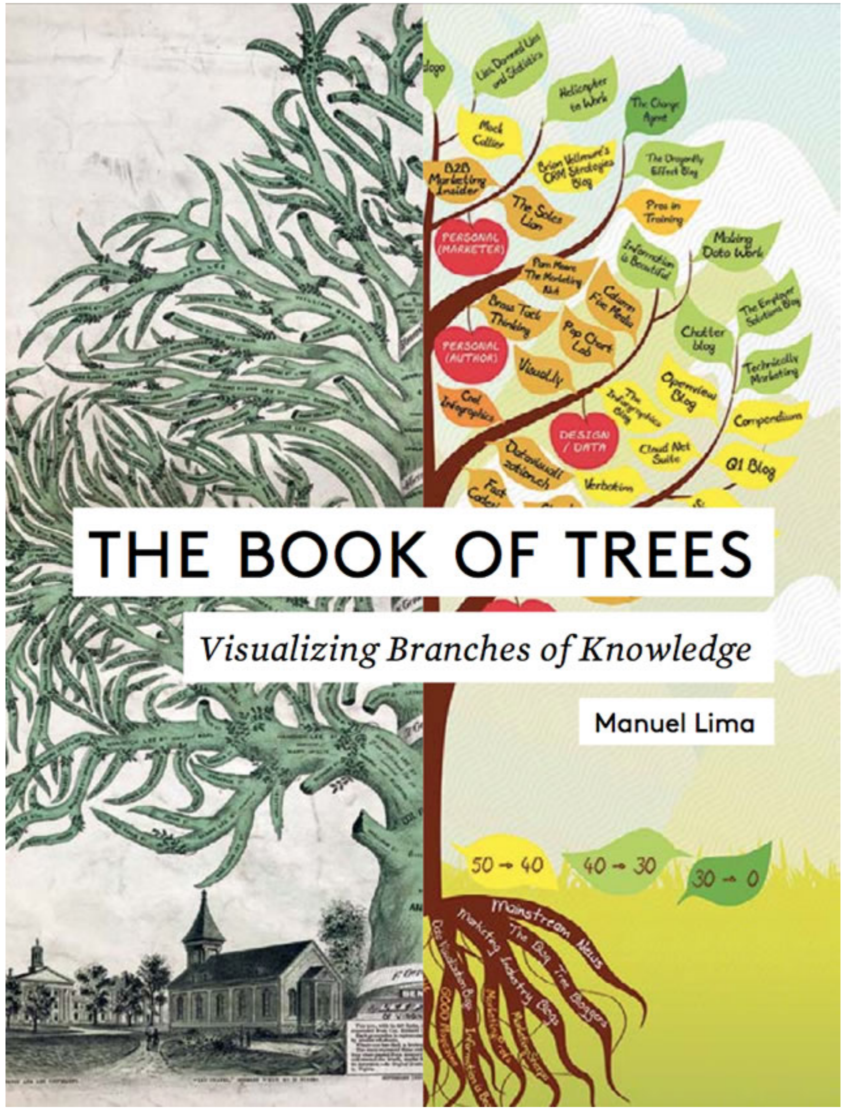


A language family tree – in pictures

Other Tree Visualization Styles



Other Tree Visualization Styles



treevis.net

Literature

- [GD, Chapter 3] divide and conquer methods for rooted trees and series-parallel graphs
- [Reingold, Tilford '81] "Tidier Drawings of Trees"
  – original paper for level-based layout algo
- [Reingold, Supowit '83] "The complexity of drawing trees nicely"
  – linear program and NP-hardness proof for area minimization
- `treevis.net` – compendium of drawing methods for trees