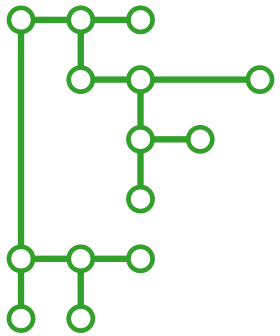
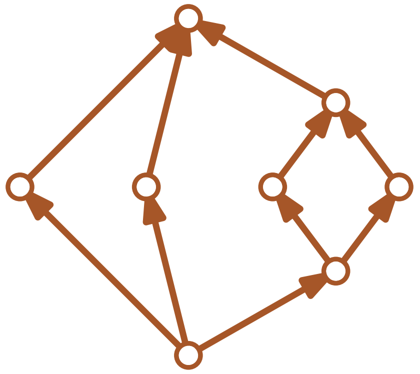
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

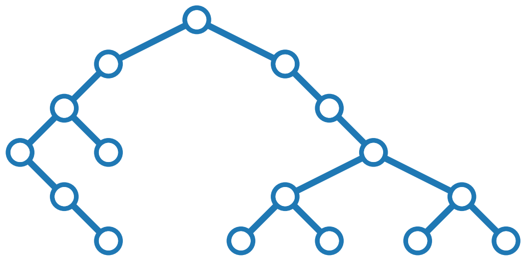
Visualization of Graphs

Lecture 1b:
Drawing Trees and Series-Parallel Graphs

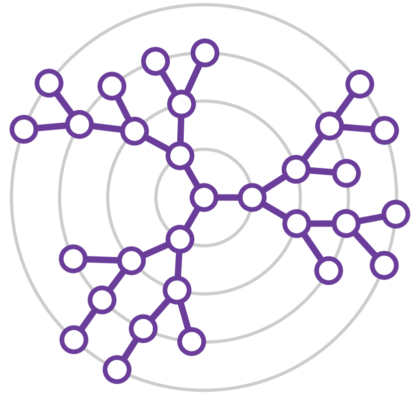


Part I:
Layered Drawings

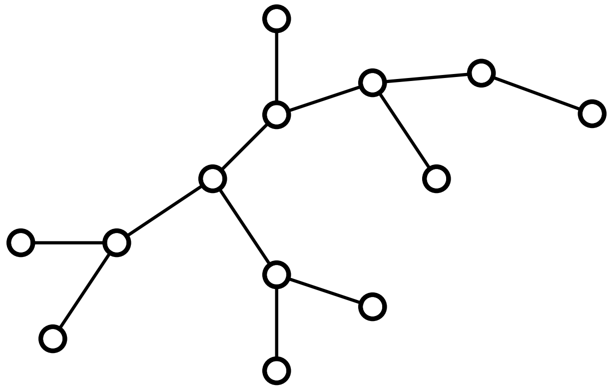




Alexander Wolff

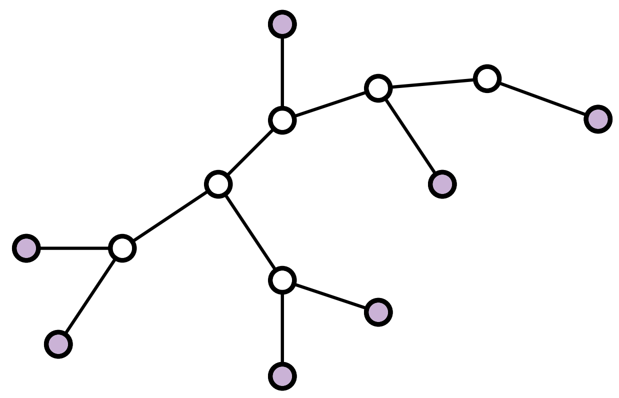


(Rooted) Trees



(Rooted) Trees

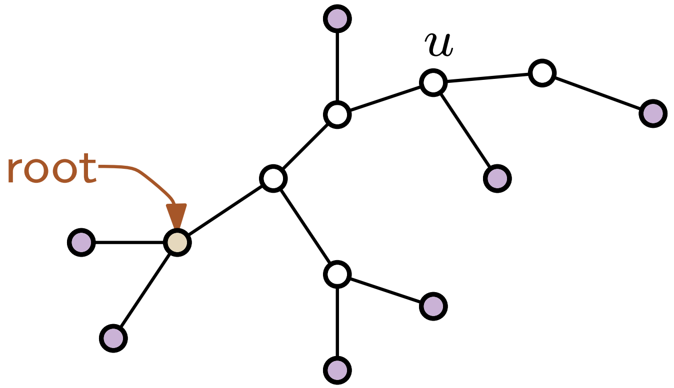
Leaf: Vertex of degree 1



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

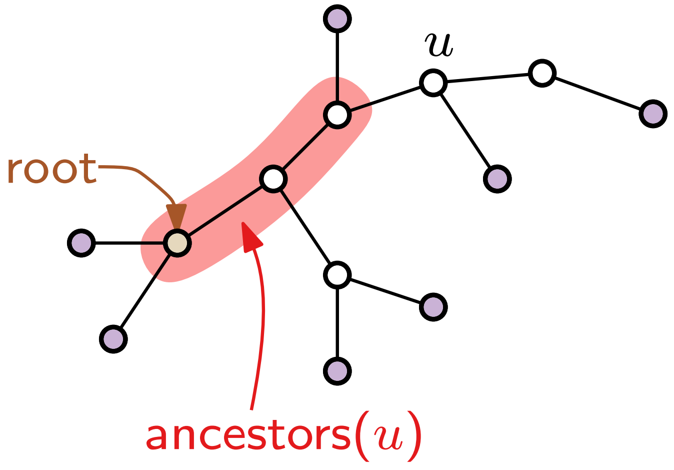


(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root



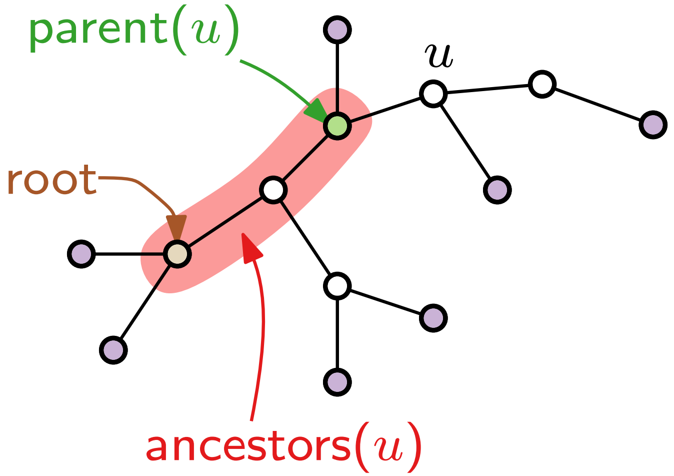
(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root



(Rooted) Trees

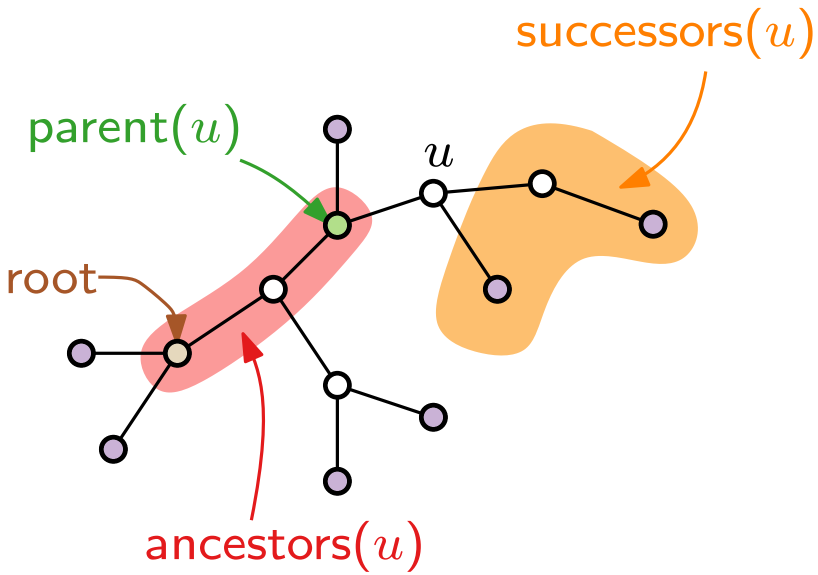
Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

Successor: Vertex on path away from root



(Rooted) Trees

Leaf: Vertex of degree 1

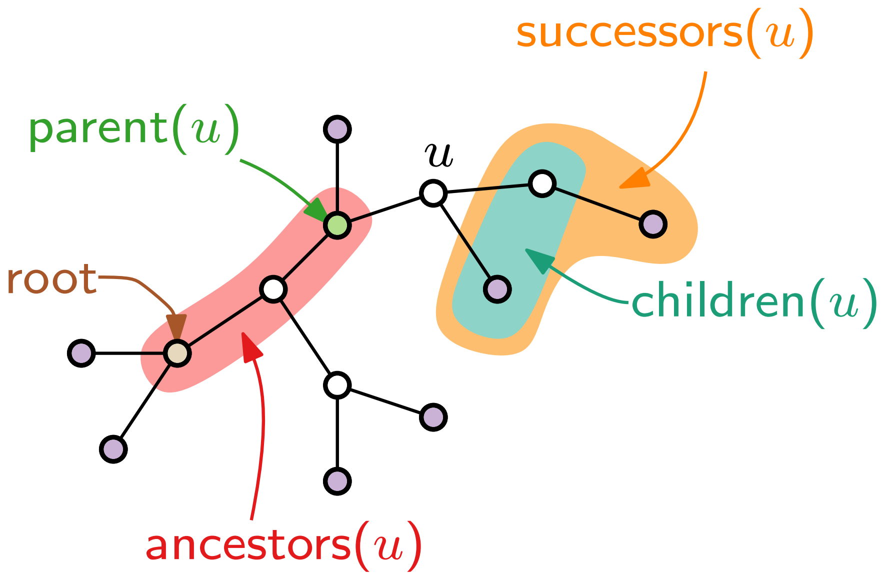
Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

Successor: Vertex on path away from root

Child: Neighbor not on path to root



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

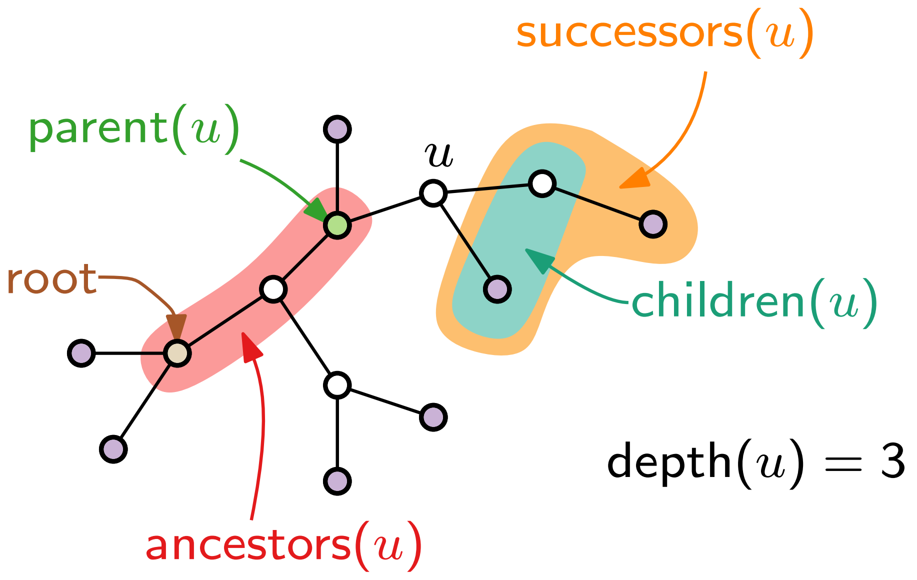
Ancestor: Vertex on path to root

Parent: Neighbor on path to root

Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

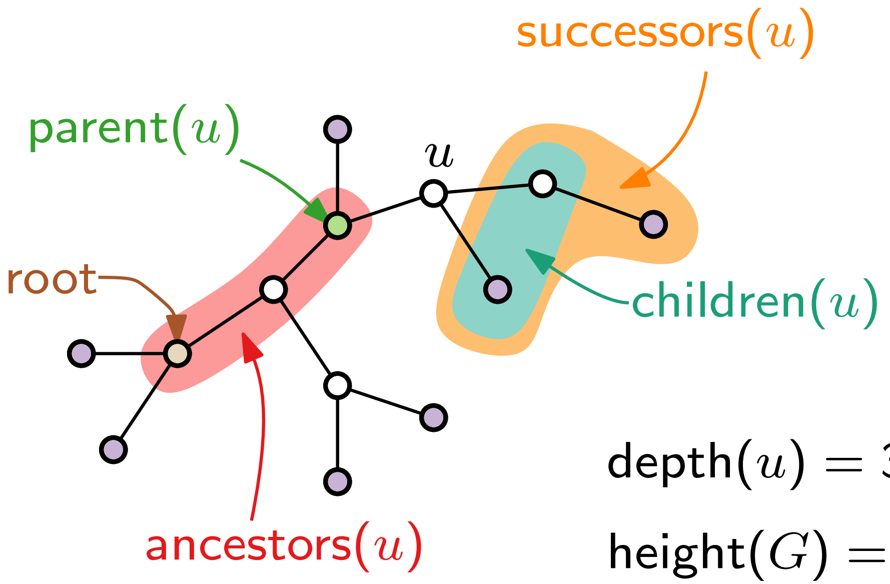
Parent: Neighbor on path to root

Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf



depth(u) $= 3$

height(G) $= 5$

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

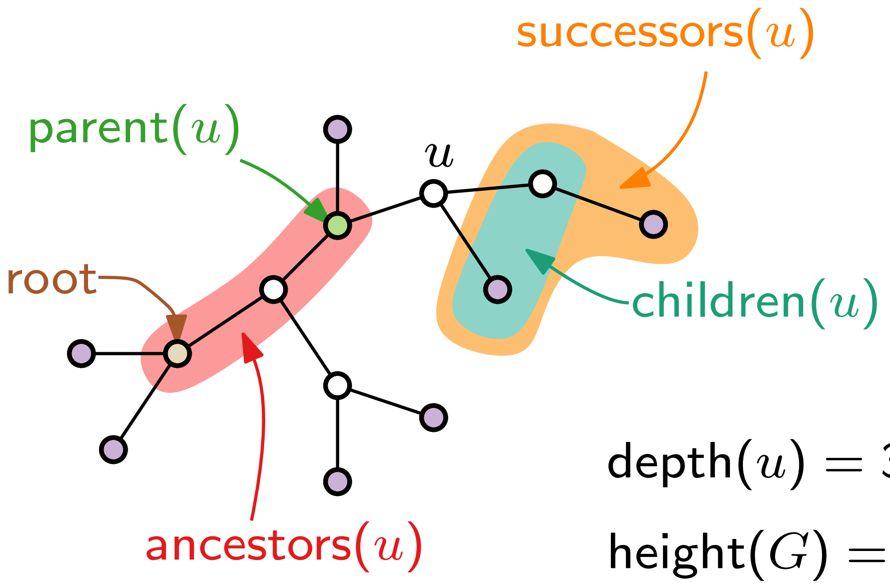
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



depth$(u) = 3$

height$(G) = 5$

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

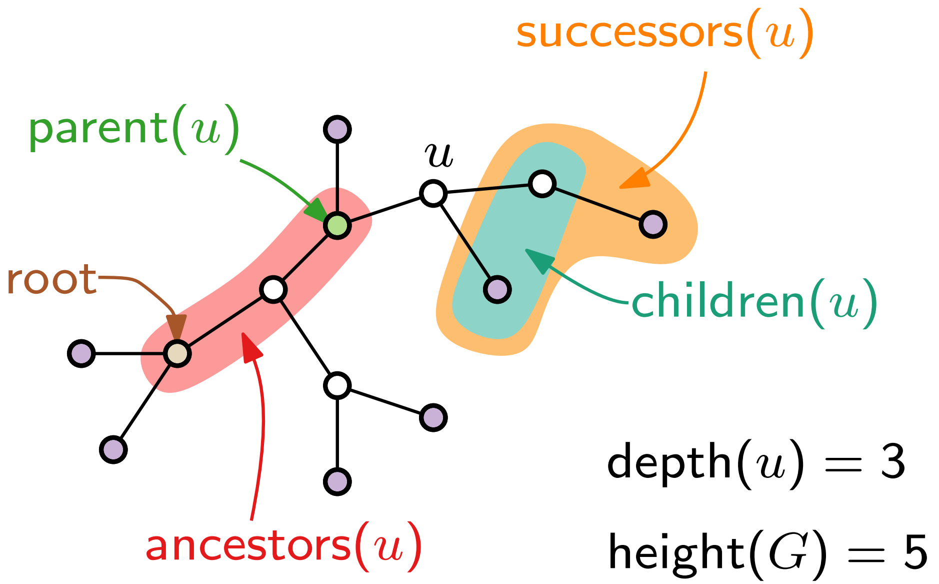
Successor: Vertex on path away from root

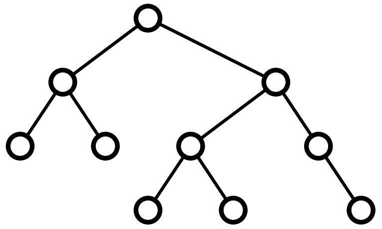
Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)





(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

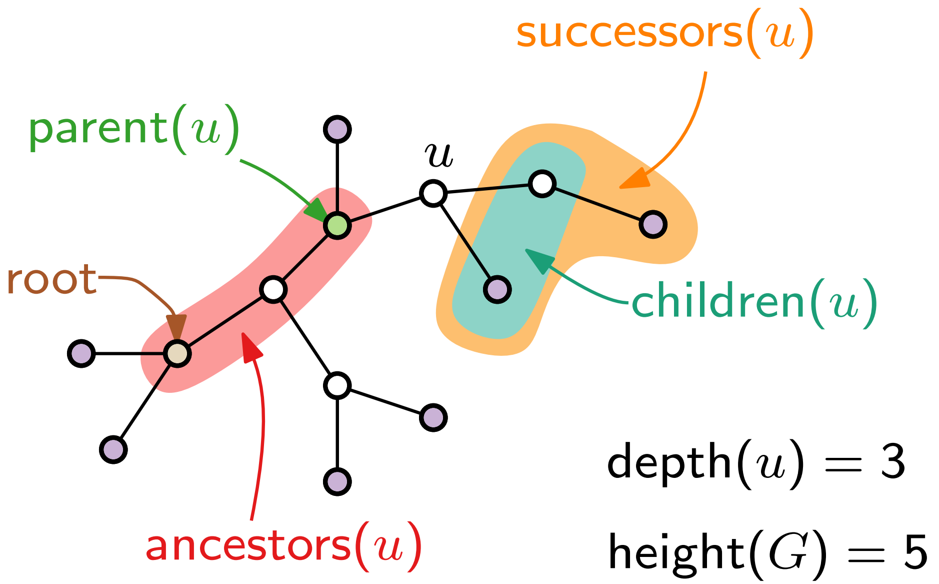
Successor: Vertex on path away from root

Child: Neighbor not on path to root

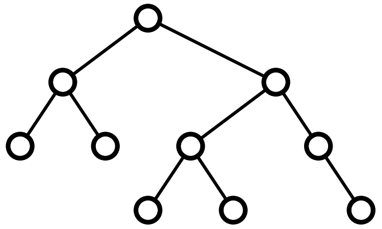
Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



3 traversals:



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

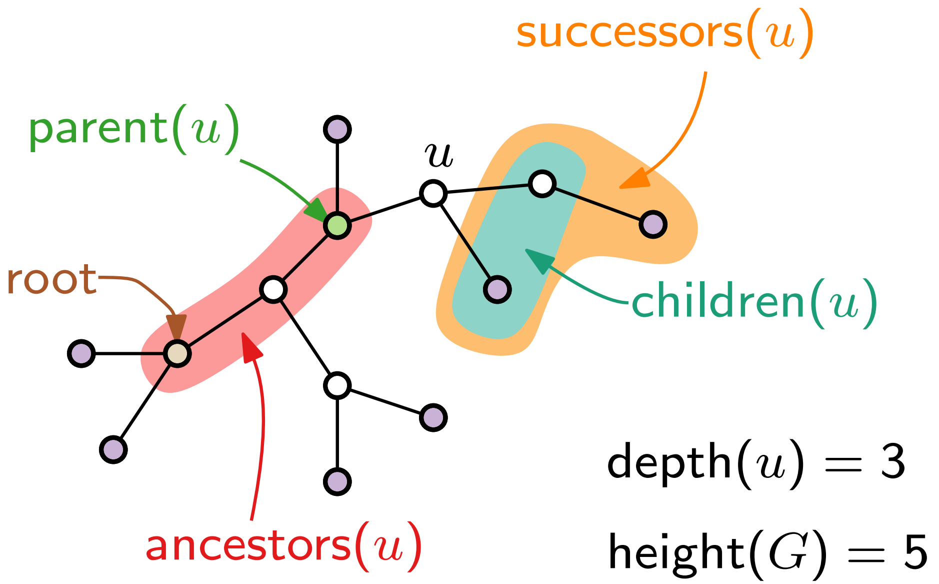
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

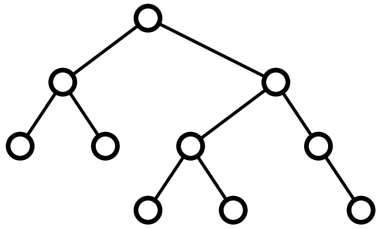
Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



3 traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

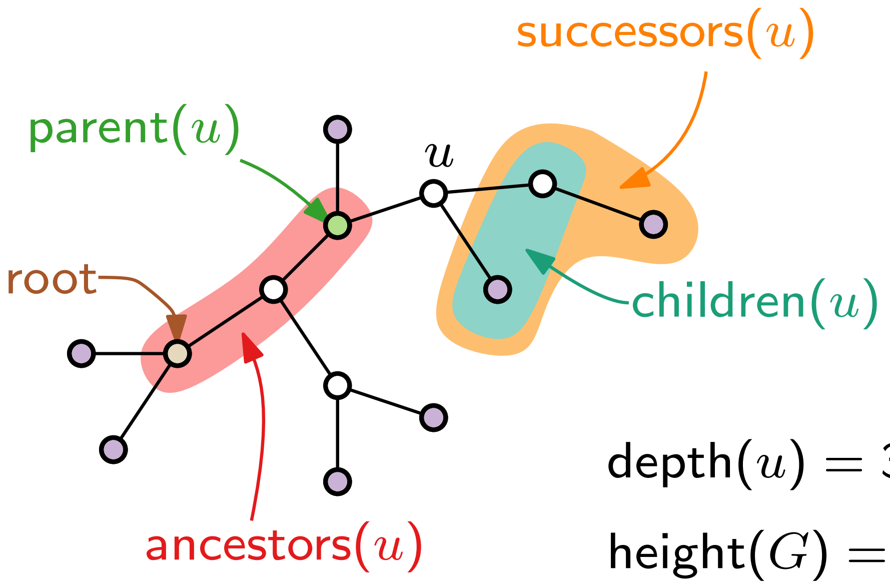
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

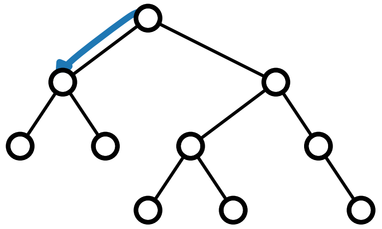


depth(u) = 3

height(G) = 5

3 traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

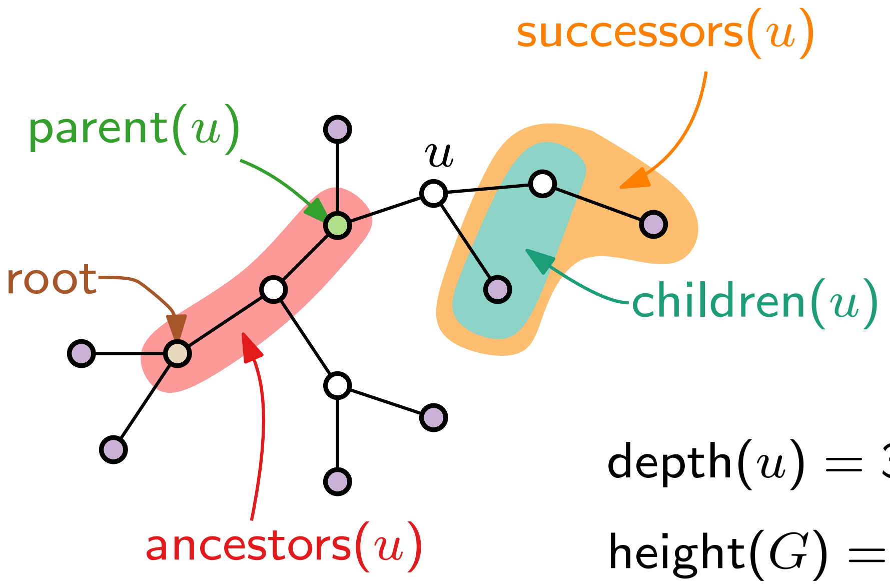
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

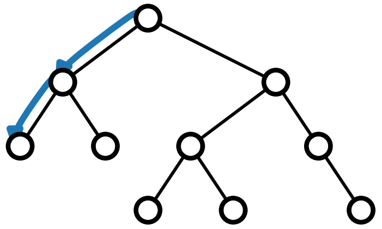


depth(u) $= 3$

height(G) $= 5$

3 traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

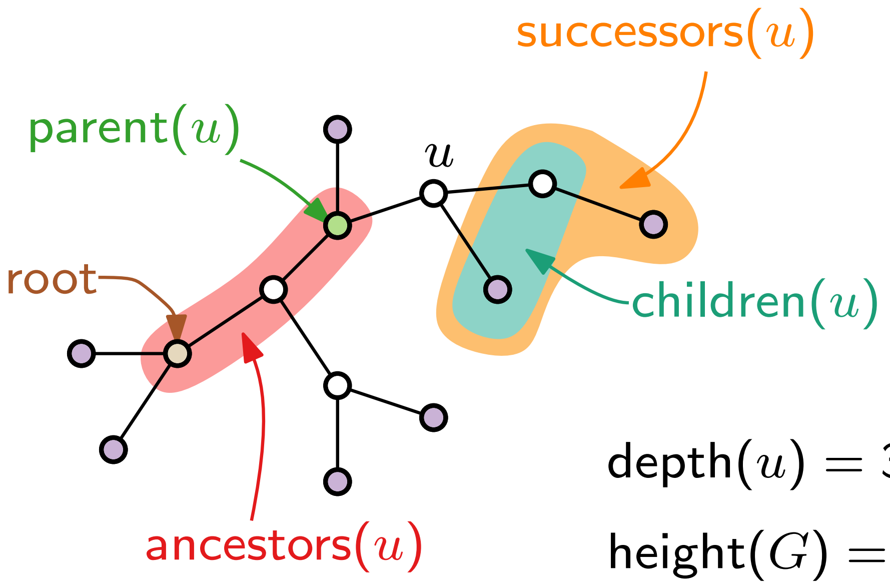
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

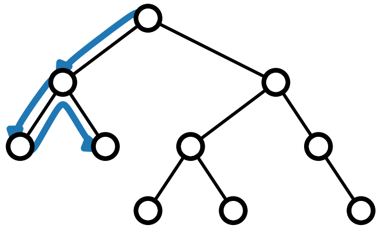


depth(u) $= 3$

height(G) $= 5$

3 traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

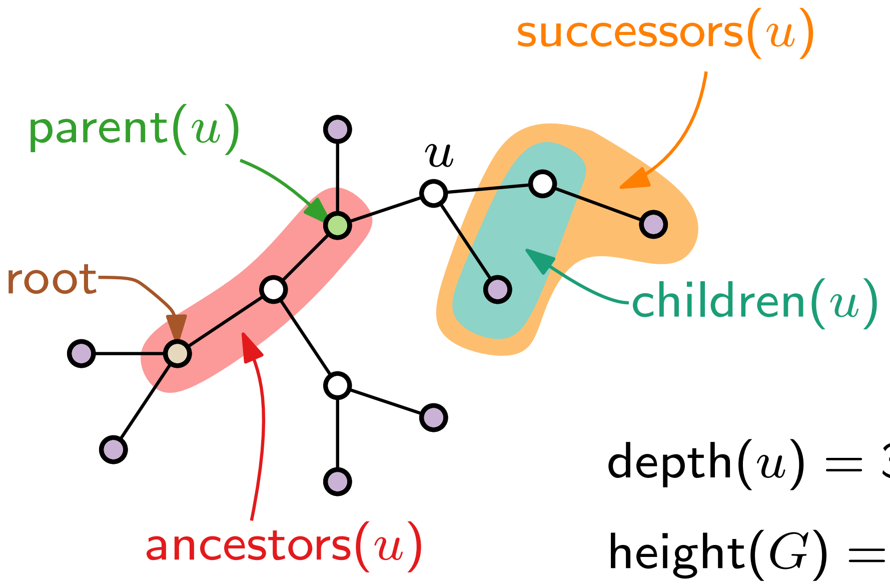
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

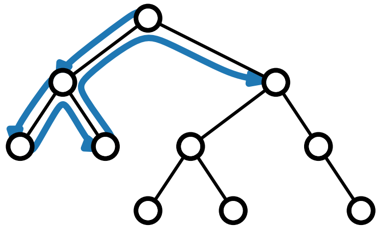


depth(u) $= 3$

height(G) $= 5$

3 traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

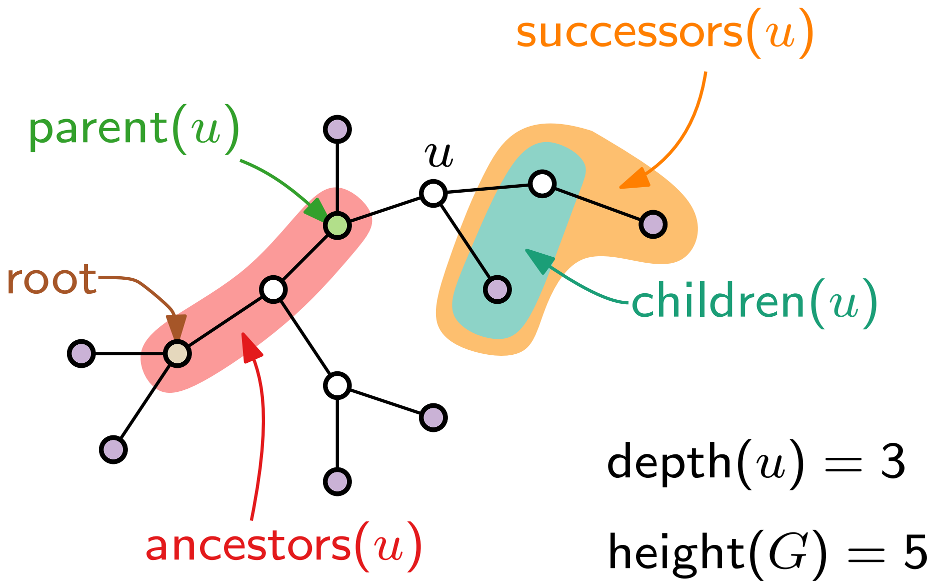
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

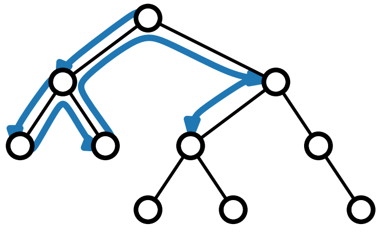
Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



3 traversals:

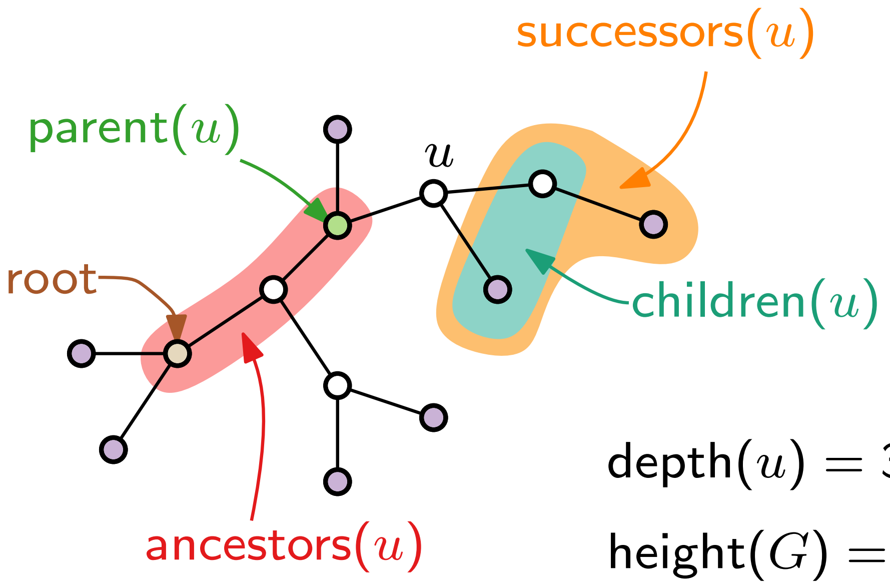
preorder



node – left – right

(Rooted) Trees

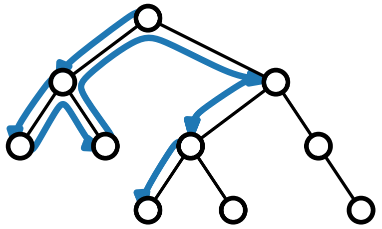
Leaf: Vertex of degree 1
Rooted tree: tree with designated **root**
Ancestor: Vertex on path to root
Parent: Neighbor on path to root
Successor: Vertex on path away from root
Child: Neighbor not on path to root
Depth: Length of path to root
Height: Maximum depth of a leaf
Binary Tree: At most two children per vertex (left / right child)



depth(u) $= 3$
height(G) $= 5$

3 traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

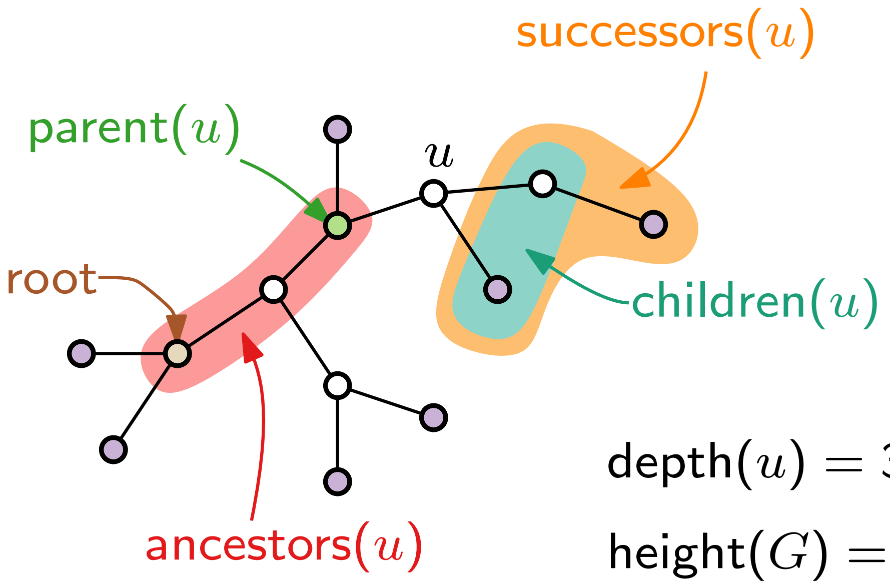
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

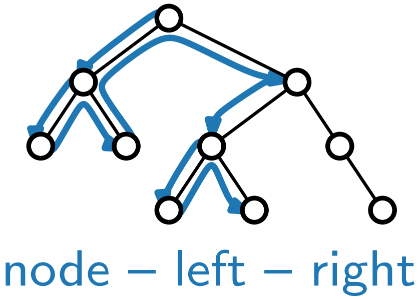


depth(u) $= 3$

height(G) $= 5$

3 traversals:

preorder



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

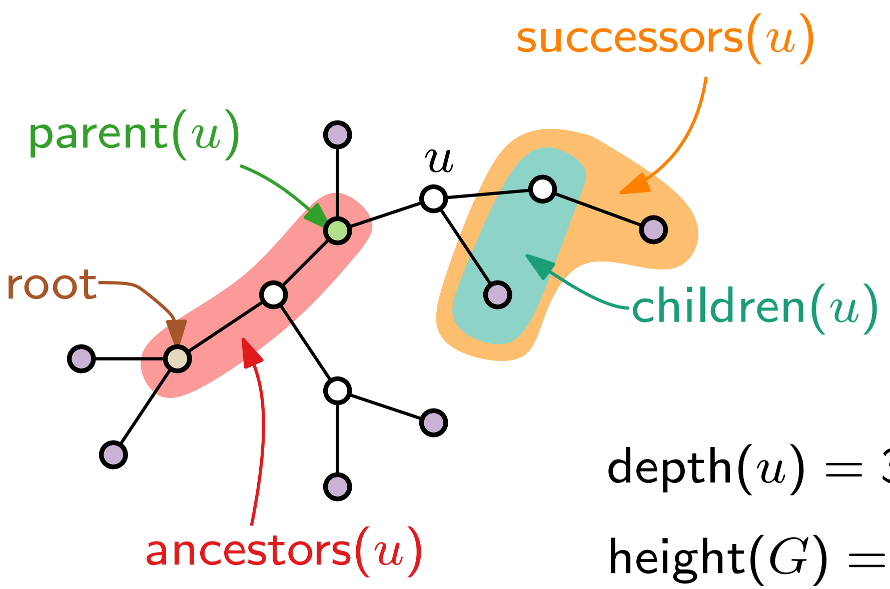
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

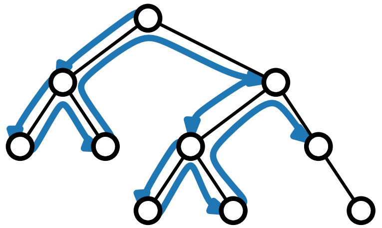


depth(u) $= 3$

height(G) $= 5$

3 traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

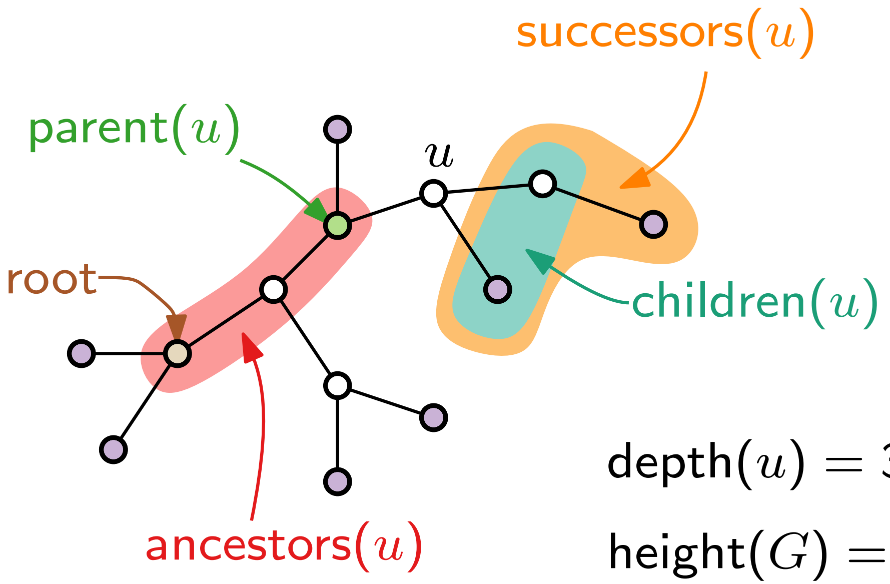
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

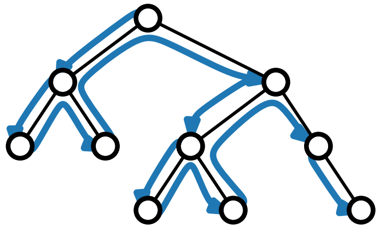


depth(u) $= 3$

height(G) $= 5$

3 traversals:

preorder



node – left – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

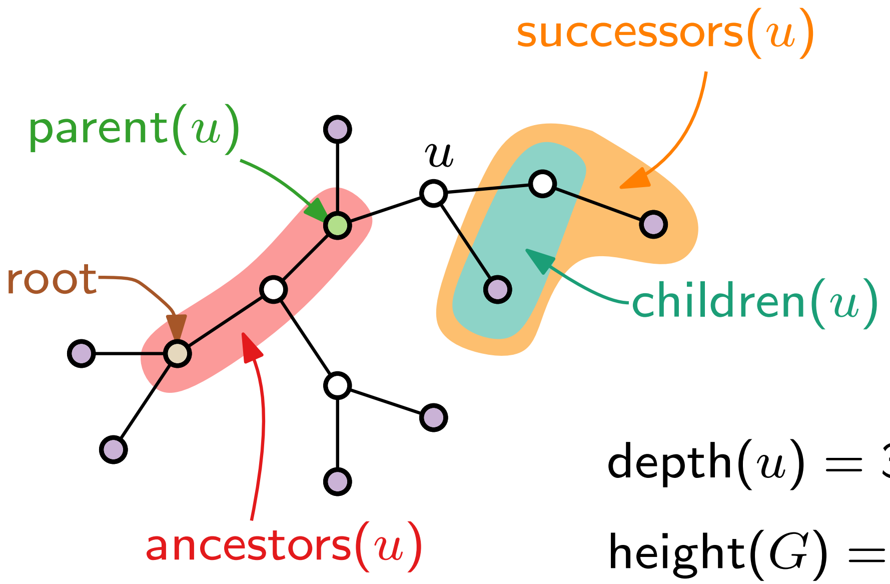
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

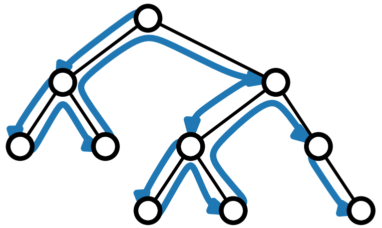


depth(u) $= 3$

height(G) $= 5$

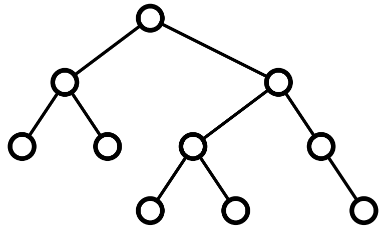
3 traversals:

preorder



node – left – right

inorder



left – node – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

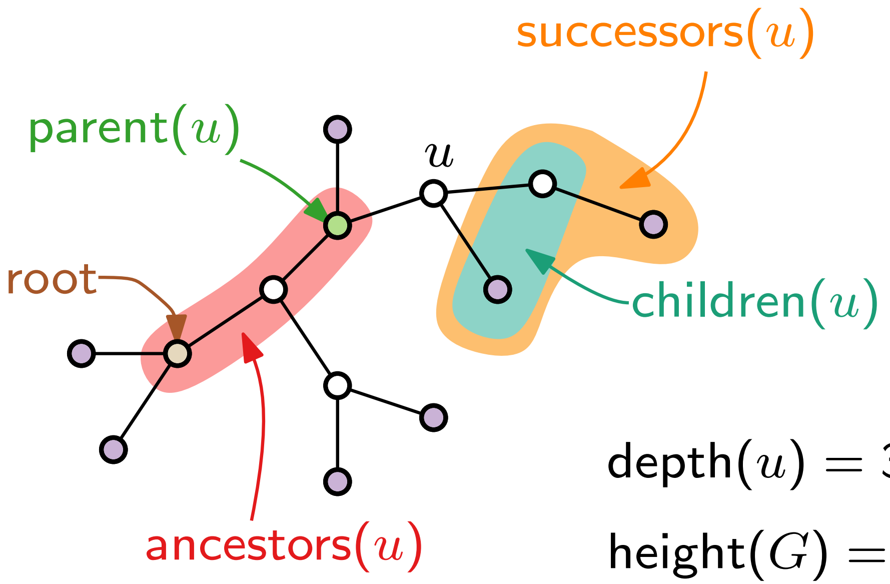
Parent: Neighbor on path to root

Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf



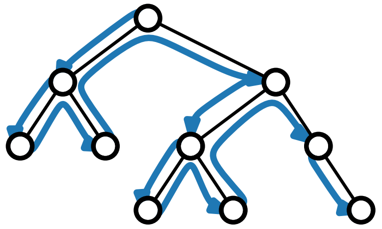
depth(u) $= 3$

height(G) $= 5$

Binary Tree: At most two children per vertex (left / right child)

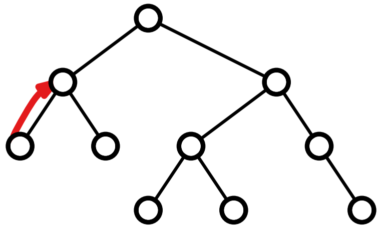
3 traversals:

preorder



node – left – right

inorder



left – node – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

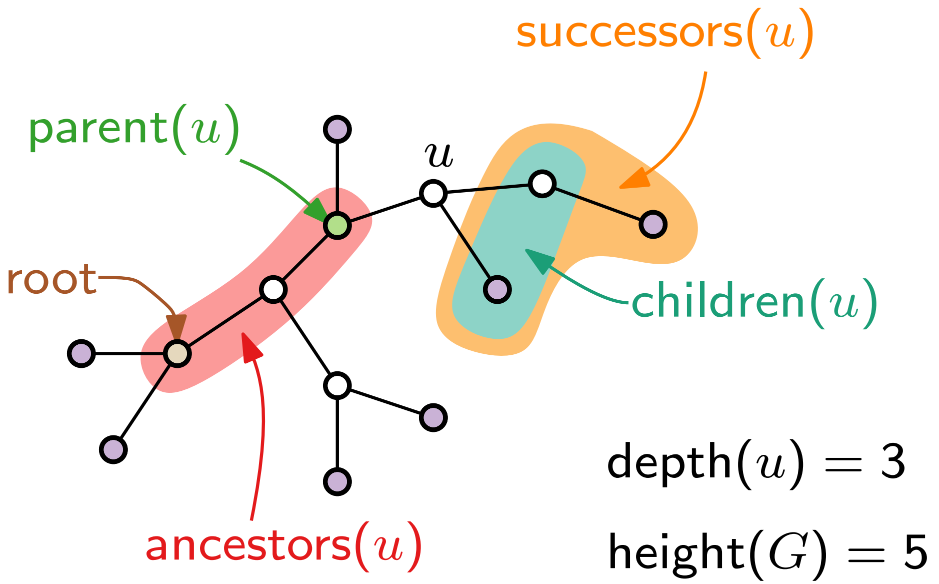
Successor: Vertex on path away from root

Child: Neighbor not on path to root

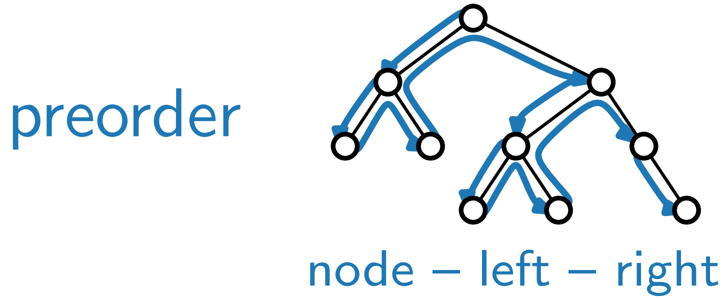
Depth: Length of path to root

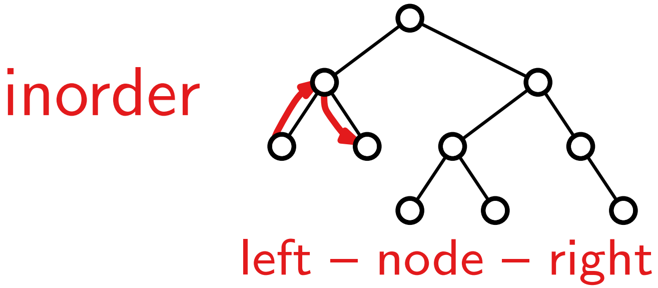
Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



3 traversals:





(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

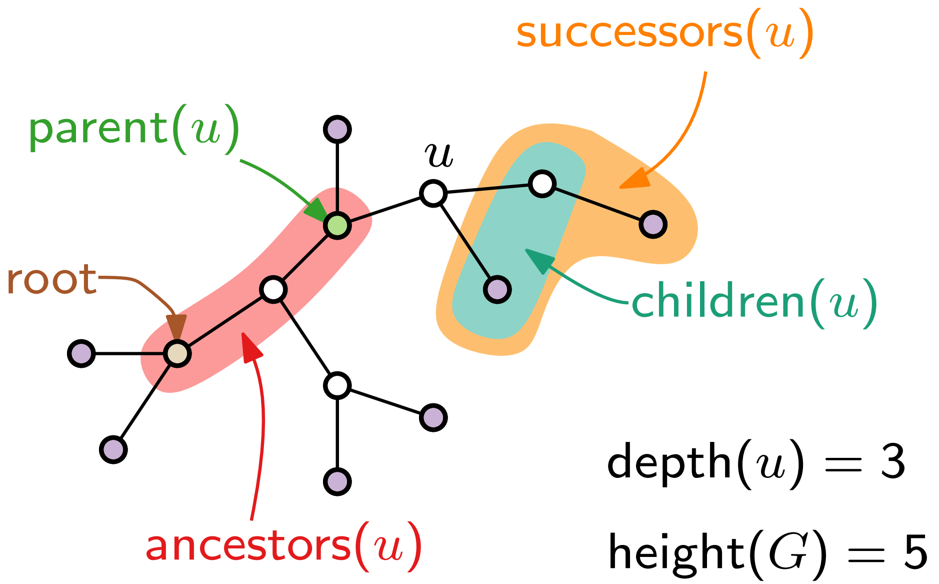
Successor: Vertex on path away from root

Child: Neighbor not on path to root

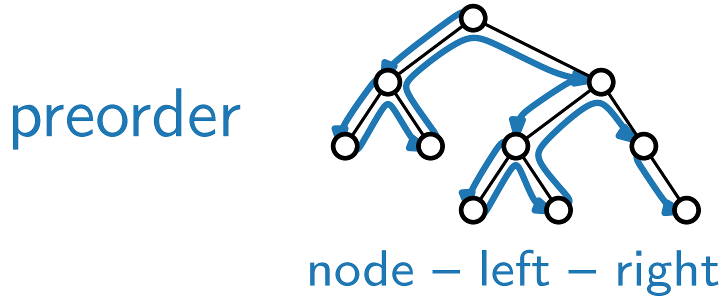
Depth: Length of path to root

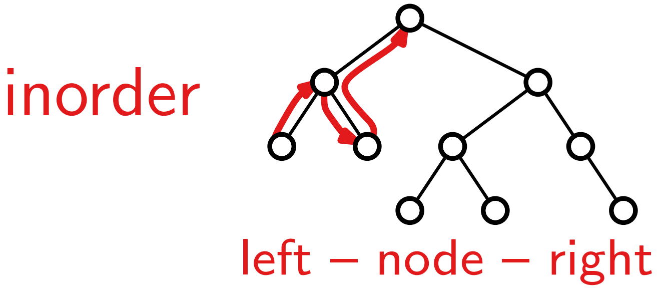
Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



3 traversals:





(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

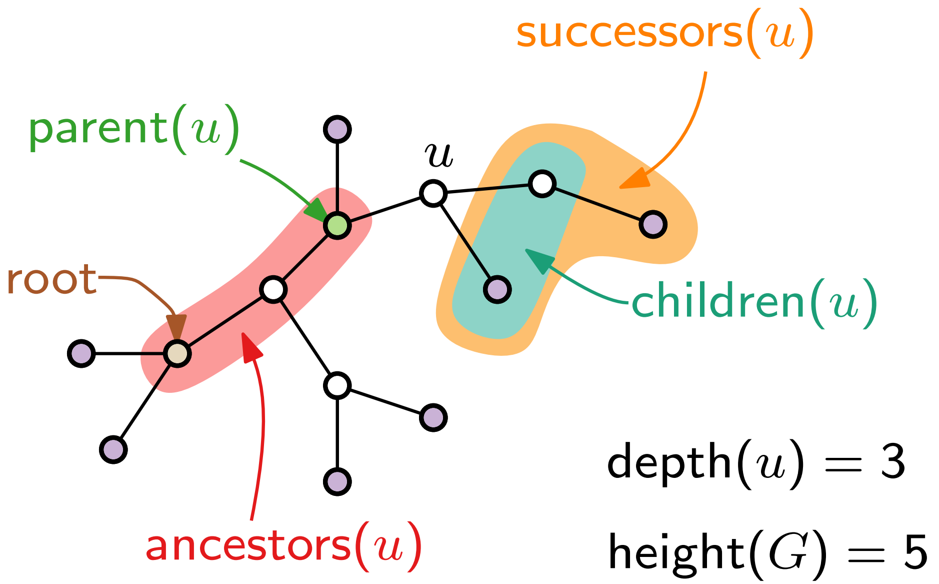
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

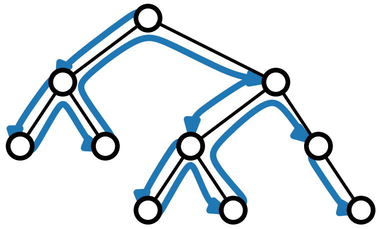
Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



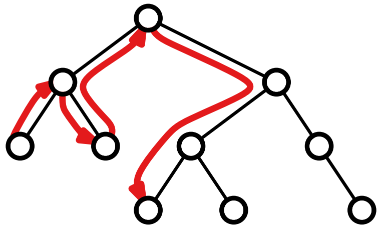
3 traversals:

preorder



node – left – right

inorder



left – node – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

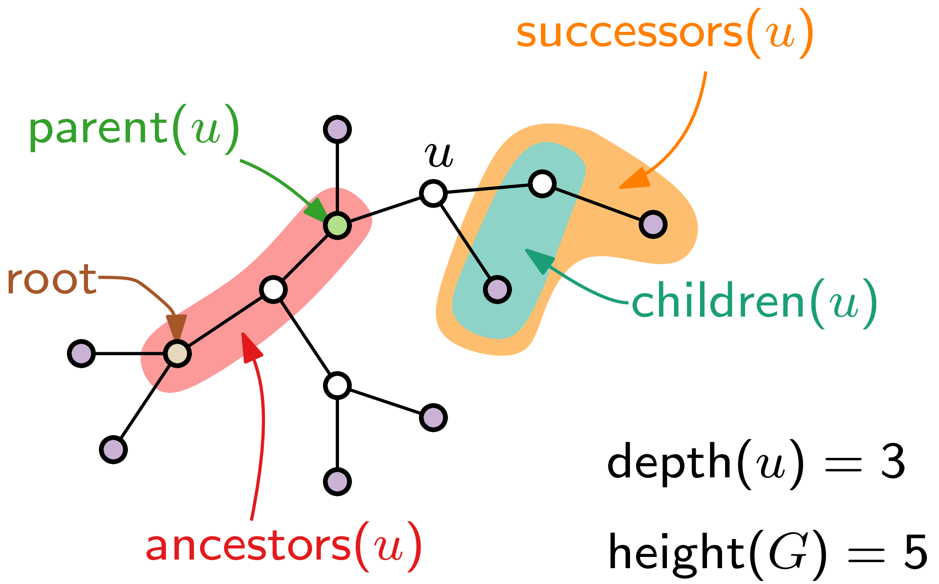
Parent: Neighbor on path to root

Successor: Vertex on path away from root

Child: Neighbor not on path to root

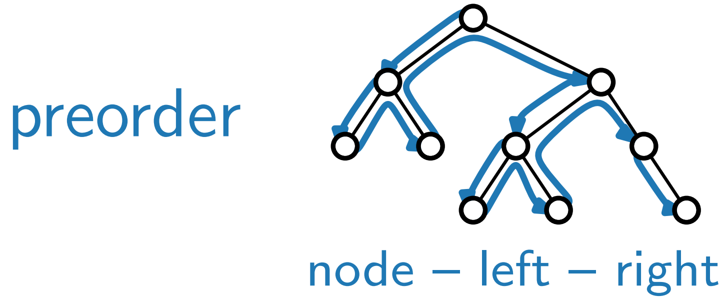
Depth: Length of path to root

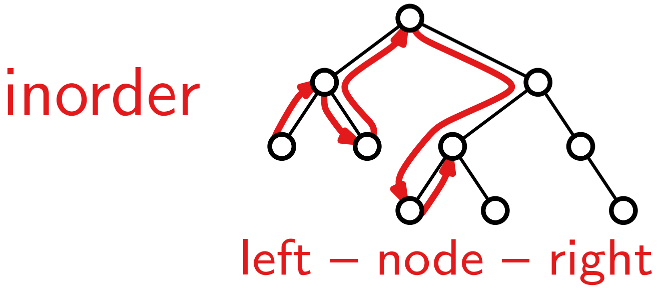
Height: Maximum depth of a leaf



Binary Tree: At most two children per vertex (left / right child)

3 traversals:





(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

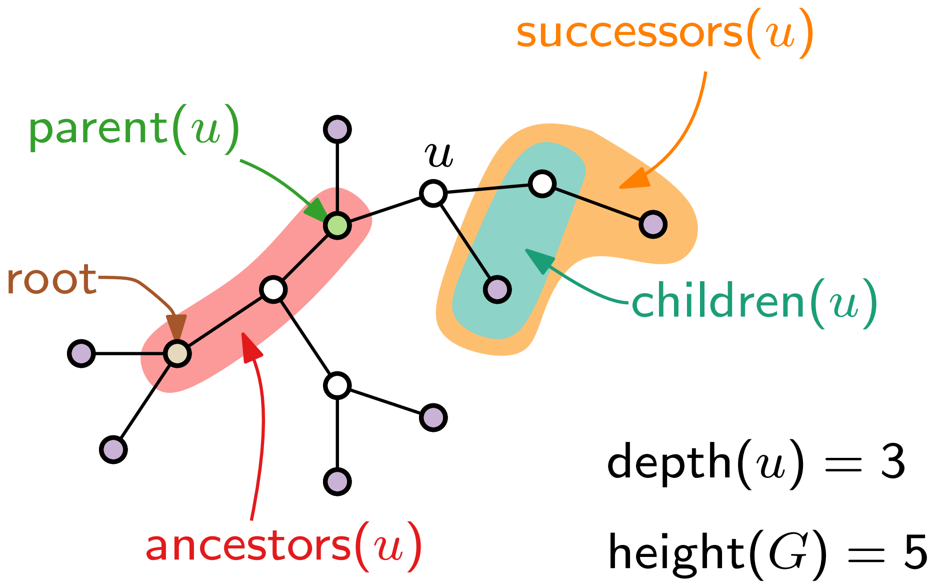
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

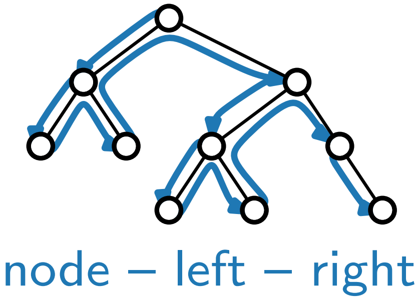
Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

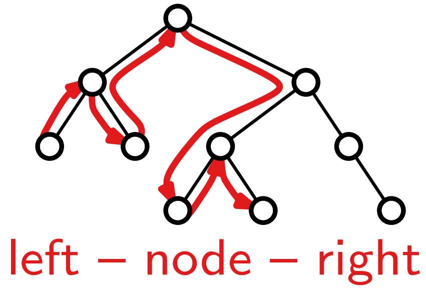


3 traversals:

preorder



inorder



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

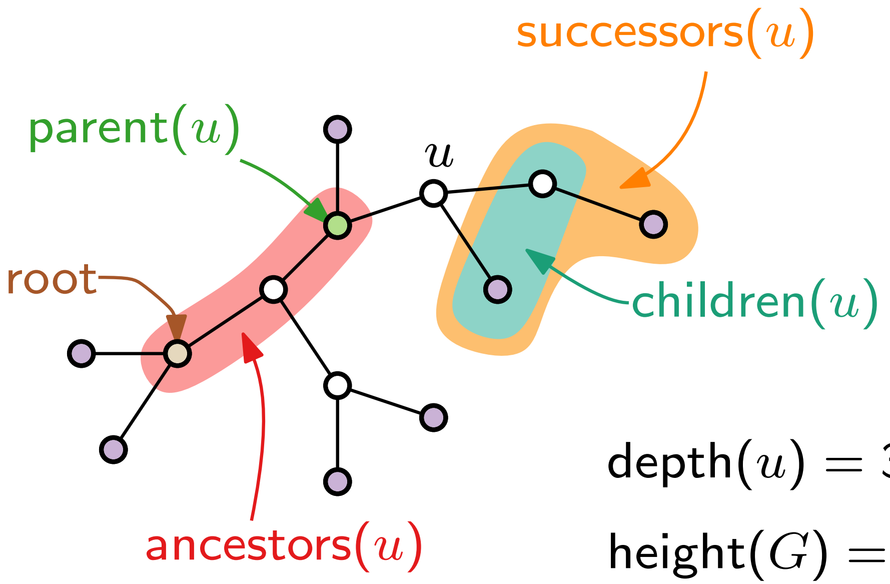
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

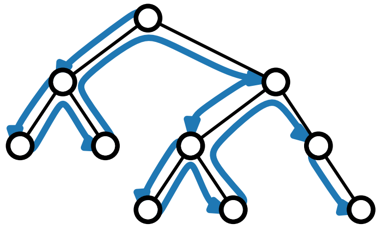


depth(u) $= 3$

height(G) $= 5$

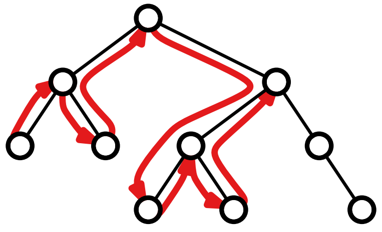
3 traversals:

preorder



node – left – right

inorder



left – node – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

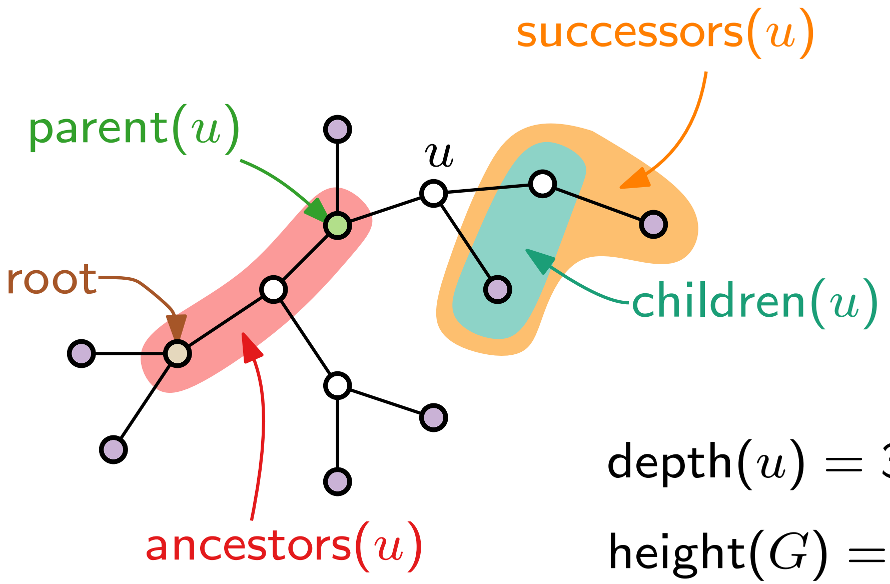
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

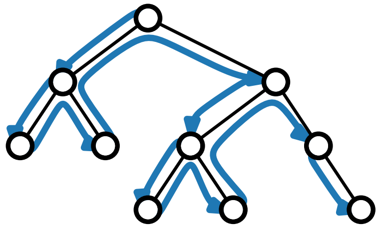


depth(u) = 3

height(G) = 5

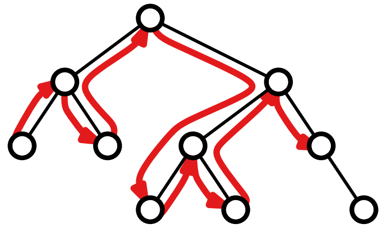
3 traversals:

preorder



node – left – right

inorder



left – node – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

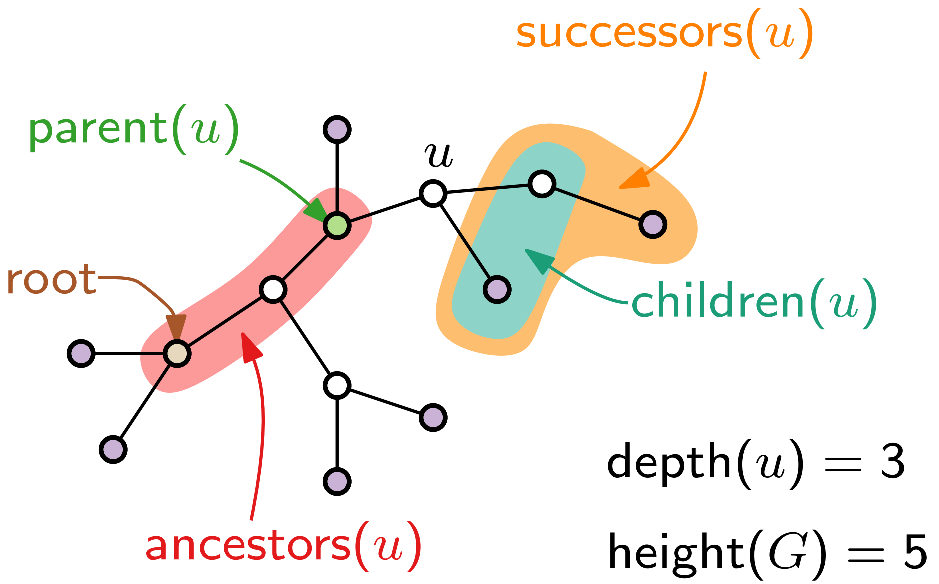
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

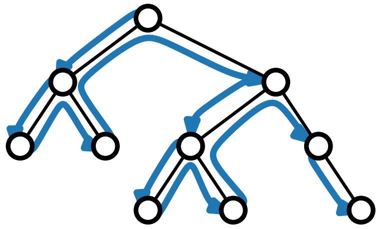
Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



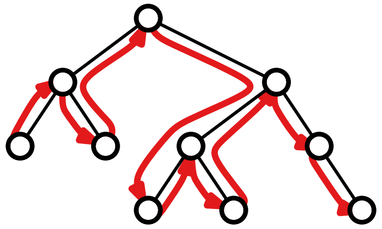
3 traversals:

preorder



node – left – right

inorder



left – node – right

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

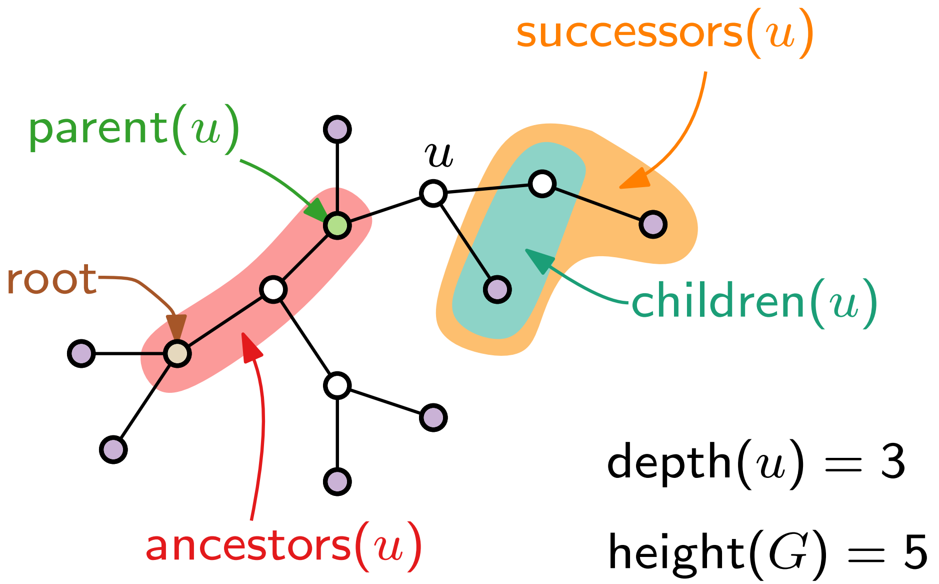
Parent: Neighbor on path to root

Successor: Vertex on path away from root

Child: Neighbor not on path to root

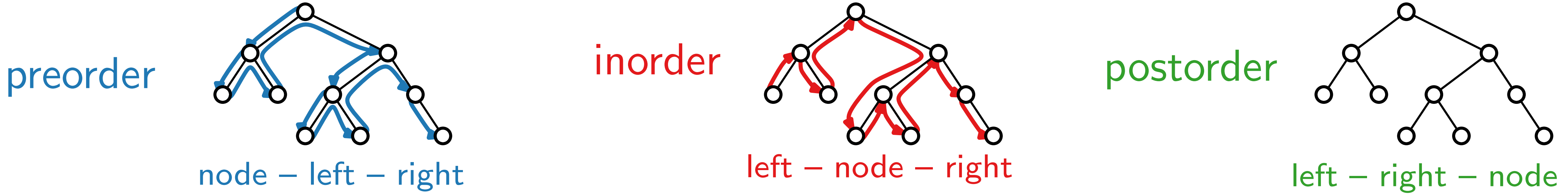
Depth: Length of path to root

Height: Maximum depth of a leaf



Binary Tree: At most two children per vertex (left / right child)

3 traversals:



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

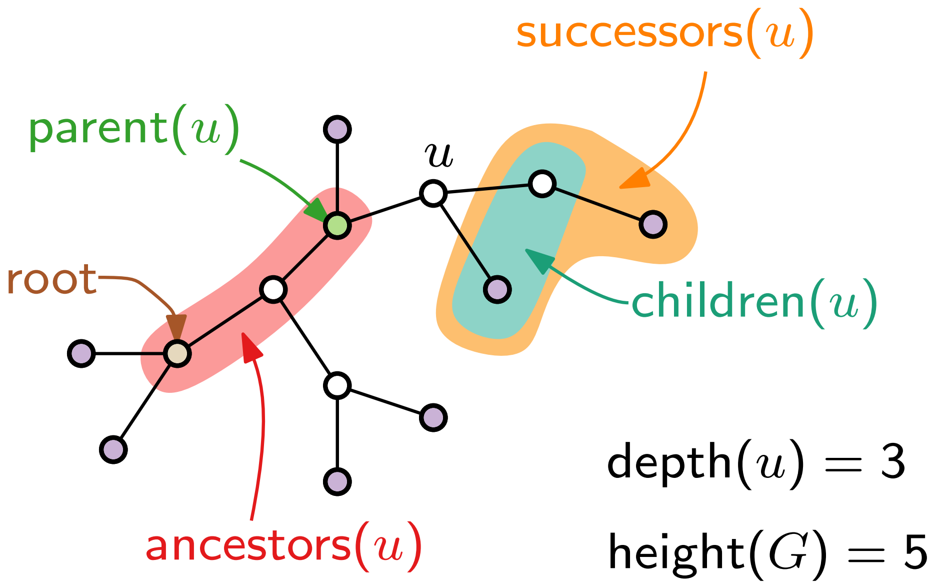
Parent: Neighbor on path to root

Successor: Vertex on path away from root

Child: Neighbor not on path to root

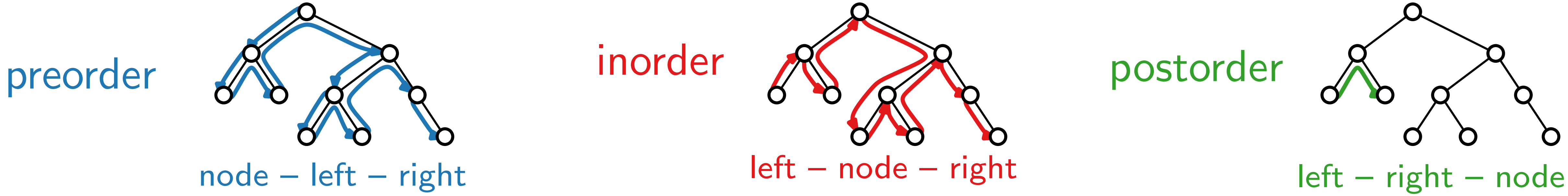
Depth: Length of path to root

Height: Maximum depth of a leaf



Binary Tree: At most two children per vertex (left / right child)

3 traversals:



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

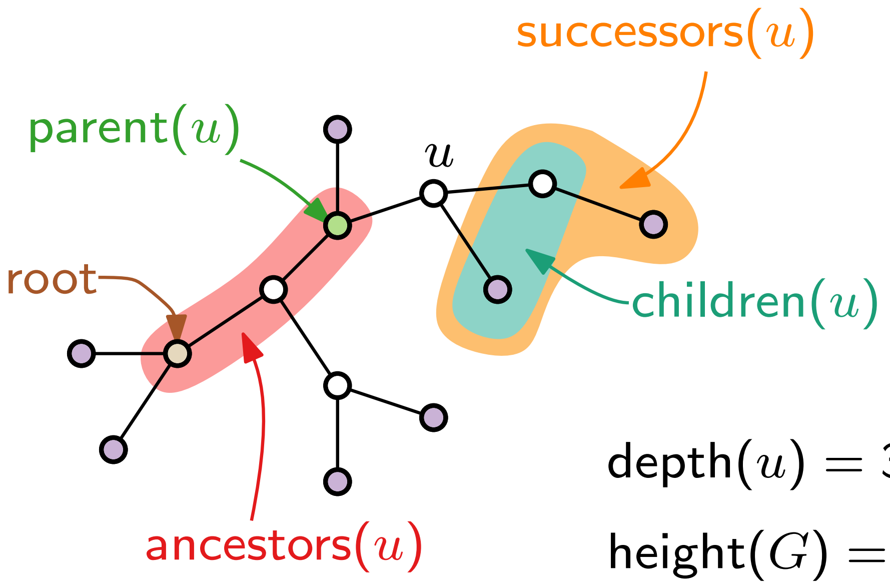
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

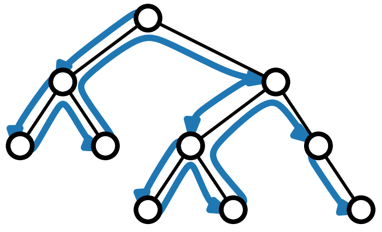


depth(u) $= 3$

height(G) $= 5$

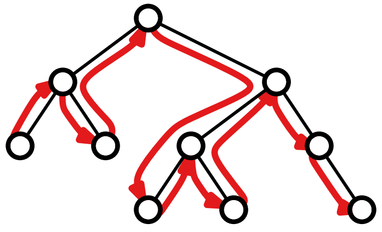
3 traversals:

preorder



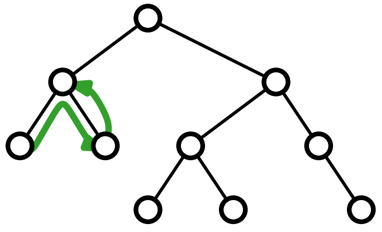
node – left – right

inorder



left – node – right

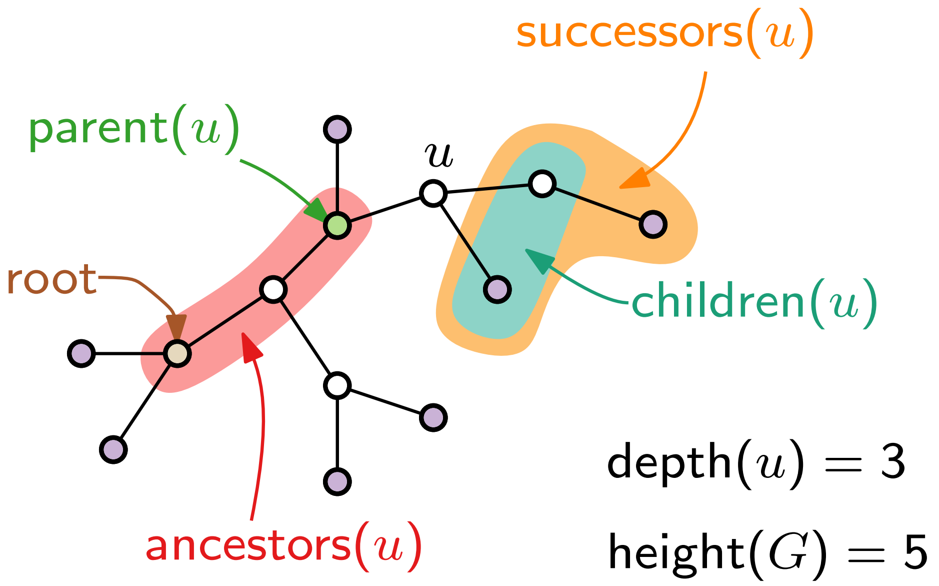
postorder



left – right – node

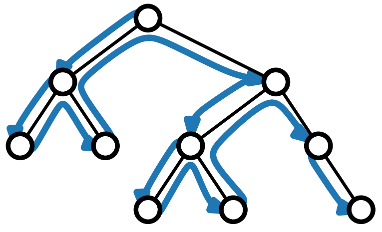
(Rooted) Trees

Leaf: Vertex of degree 1
Rooted tree: tree with designated **root**
Ancestor: Vertex on path to root
Parent: Neighbor on path to root
Successor: Vertex on path away from root
Child: Neighbor not on path to root
Depth: Length of path to root
Height: Maximum depth of a leaf
Binary Tree: At most two children per vertex (left / right child)



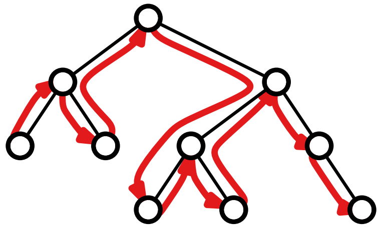
3 traversals:

preorder



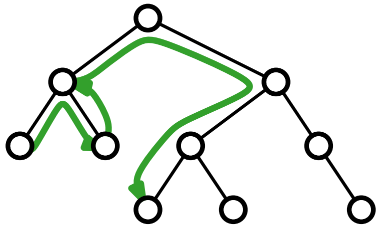
node – left – right

inorder



left – node – right

postorder



left – right – node

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

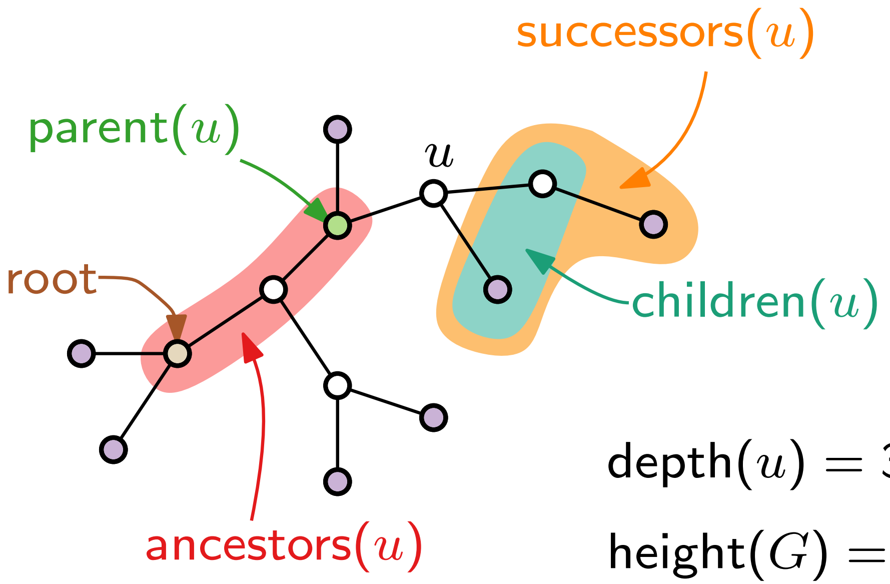
Successor: Vertex on path away from root

Child: Neighbor not on path to root

Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

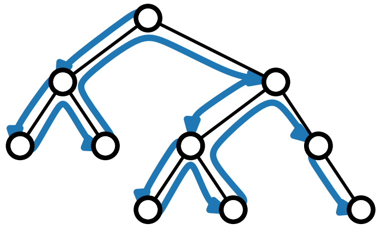


depth(u) $= 3$

height(G) $= 5$

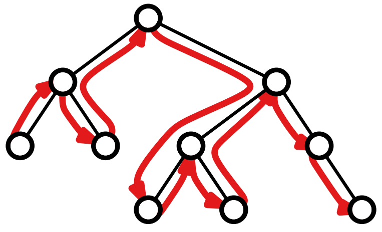
3 traversals:

preorder



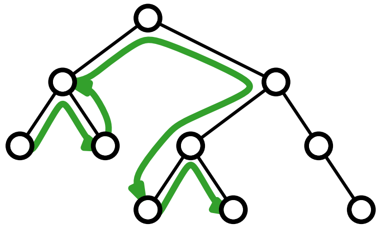
node – left – right

inorder



left – node – right

postorder



left – right – node

(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

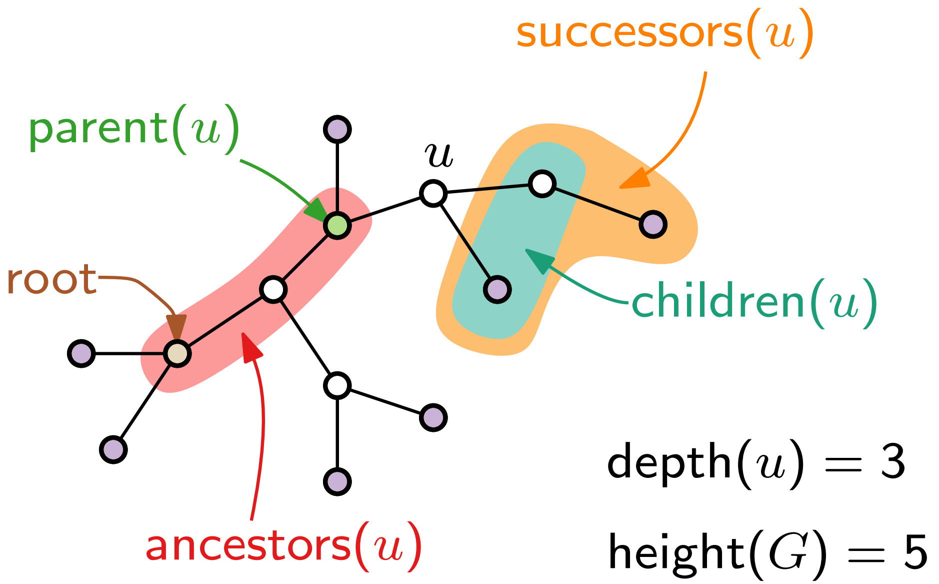
Successor: Vertex on path away from root

Child: Neighbor not on path to root

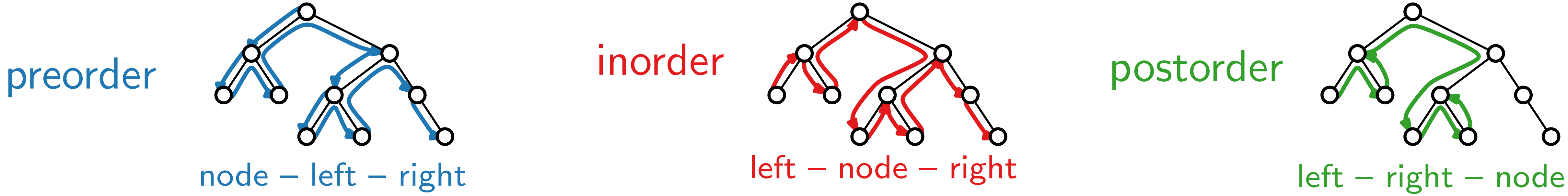
Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



3 traversals:



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

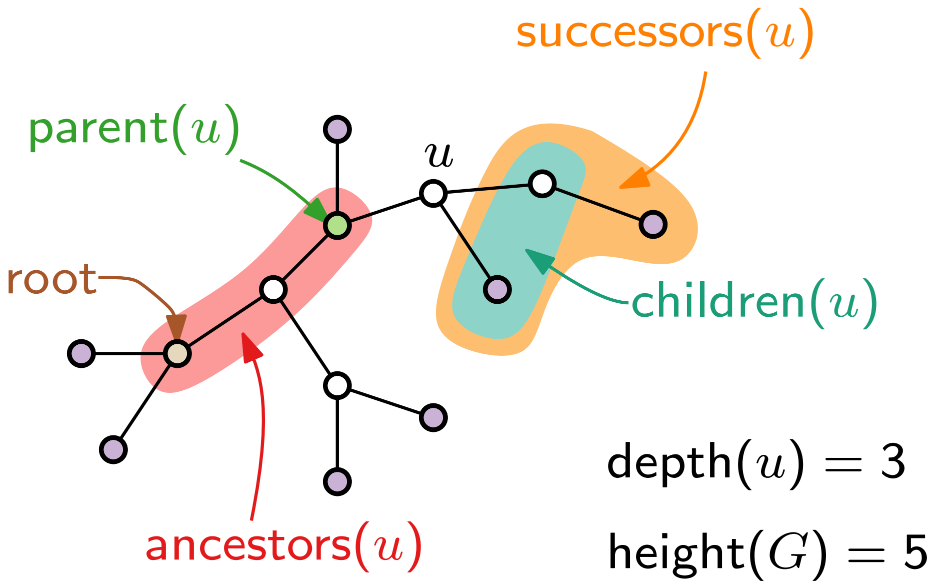
Successor: Vertex on path away from root

Child: Neighbor not on path to root

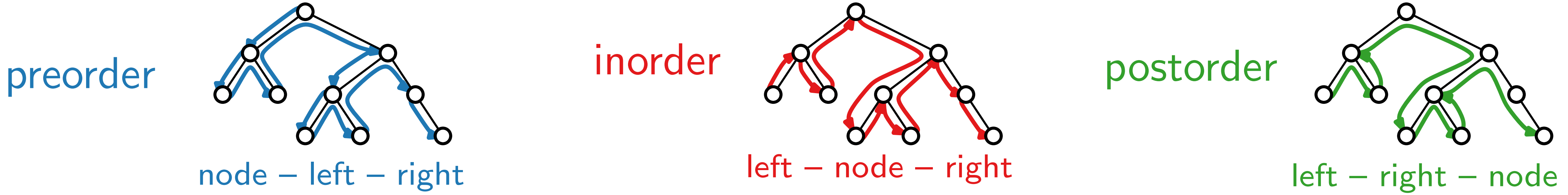
Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)

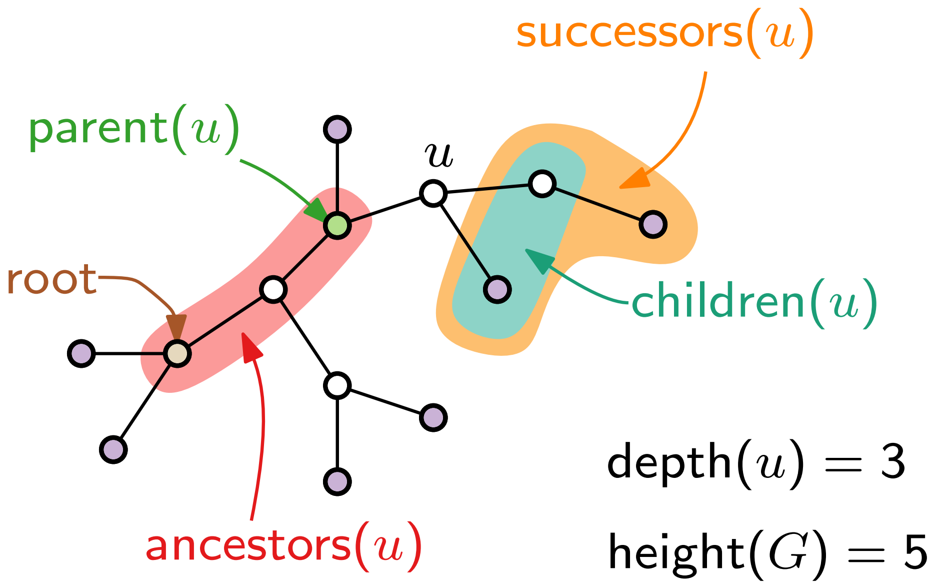


3 traversals:

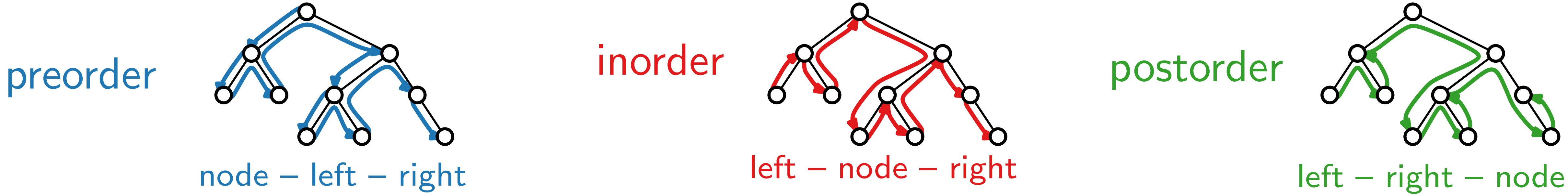


(Rooted) Trees

Leaf: Vertex of degree 1
Rooted tree: tree with designated **root**
Ancestor: Vertex on path to root
Parent: Neighbor on path to root
Successor: Vertex on path away from root
Child: Neighbor not on path to root
Depth: Length of path to root
Height: Maximum depth of a leaf
Binary Tree: At most two children per vertex (left / right child)



3 traversals:



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

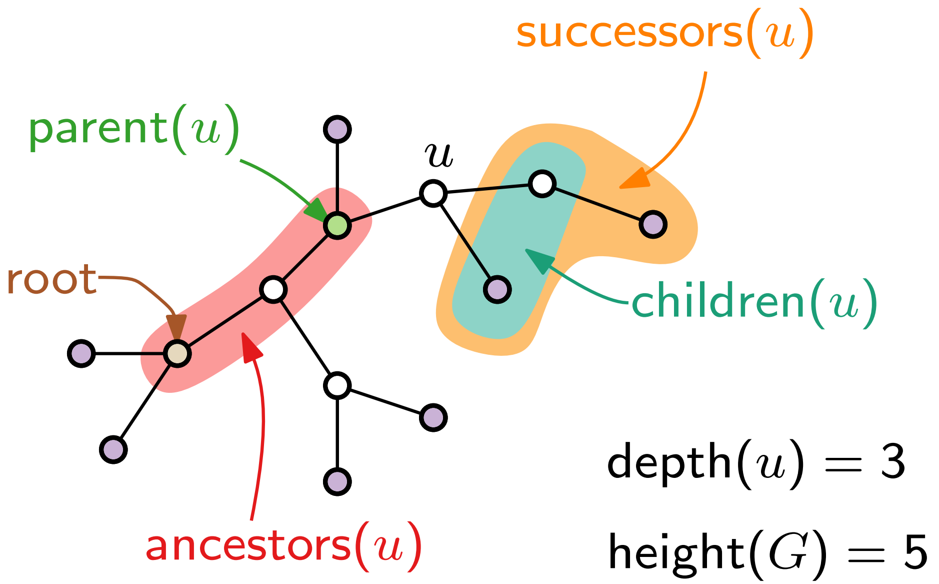
Successor: Vertex on path away from root

Child: Neighbor not on path to root

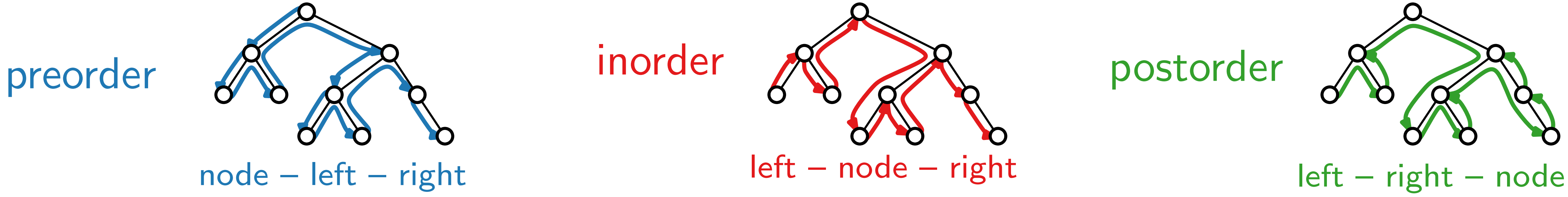
Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



3 traversals:



(Rooted) Trees

Leaf: Vertex of degree 1

Rooted tree: tree with designated **root**

Ancestor: Vertex on path to root

Parent: Neighbor on path to root

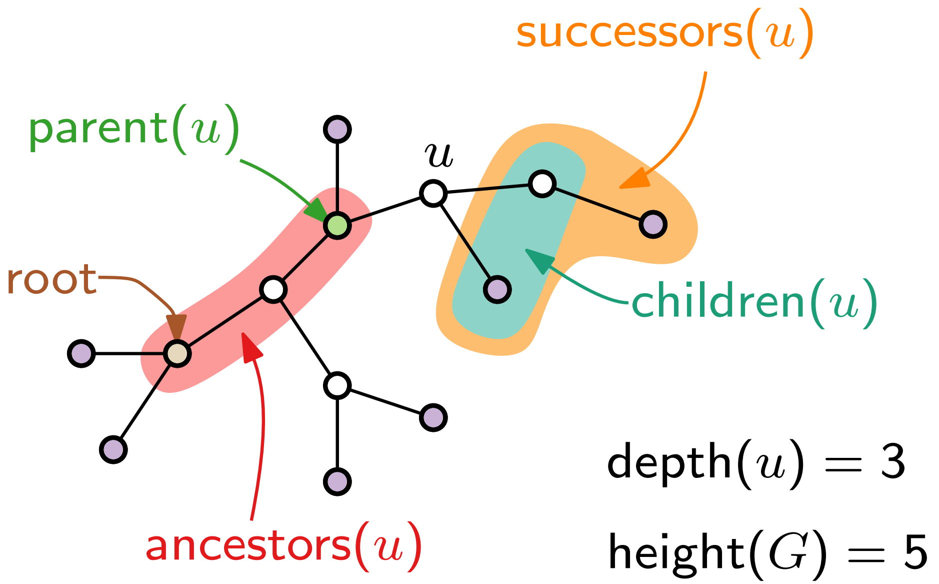
Successor: Vertex on path away from root

Child: Neighbor not on path to root

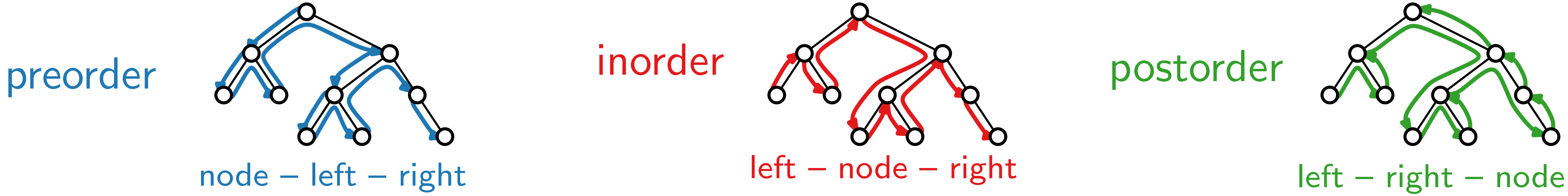
Depth: Length of path to root

Height: Maximum depth of a leaf

Binary Tree: At most two children per vertex (left / right child)



3 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:

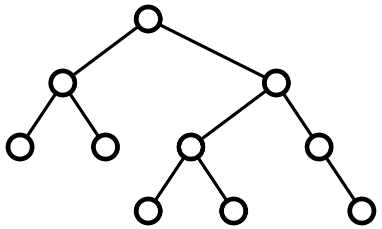
First Grid Layout of Binary Trees

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

2. Choose x-coordinates:

First Grid Layout of Binary Trees

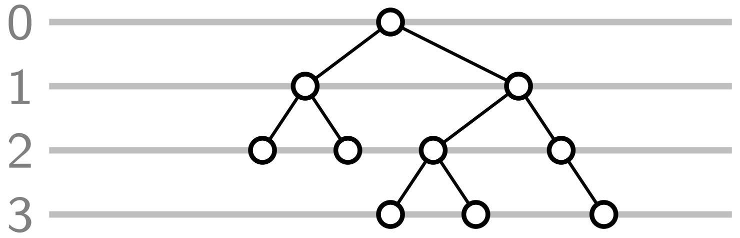
1. Choose y-coordinates: $\quad y(u) = \mathsf{depth}(u)$



2. Choose x-coordinates:

First Grid Layout of Binary Trees

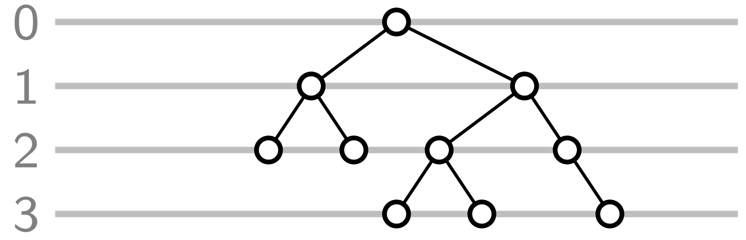
1. Choose y-coordinates: $y(u) = \text{depth}(u)$



2. Choose x-coordinates:

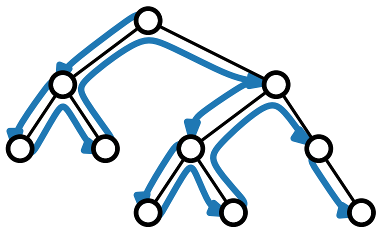
First Grid Layout of Binary Trees

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

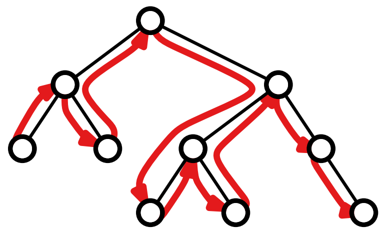


2. Choose x-coordinates:

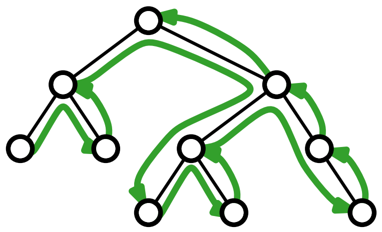
preorder



inorder

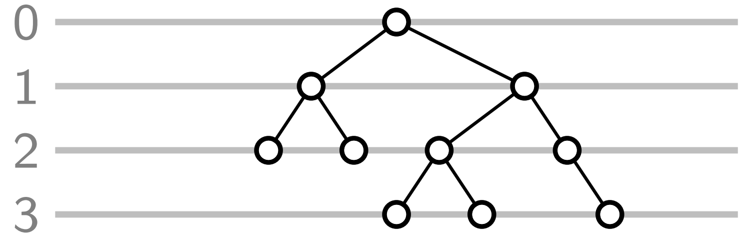


postorder

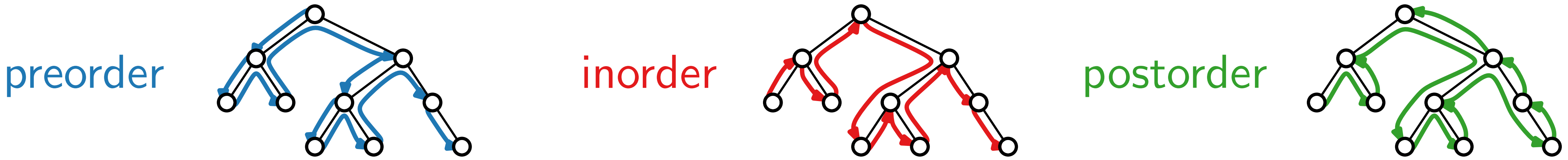


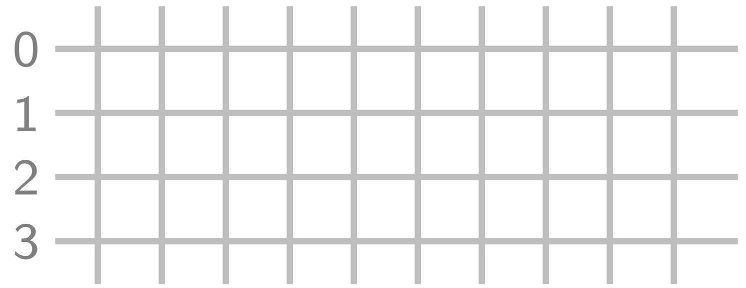
First Grid Layout of Binary Trees

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



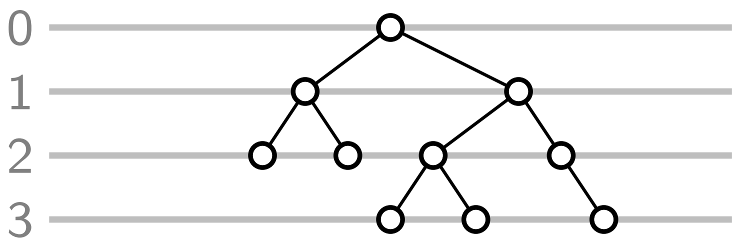
2. Choose x-coordinates:





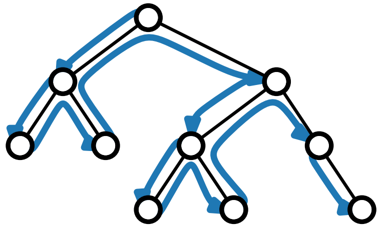
First Grid Layout of Binary Trees

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

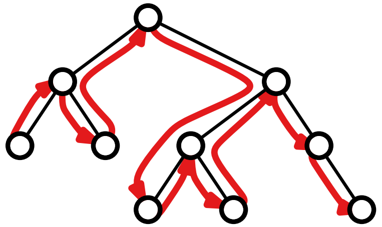


2. Choose x-coordinates:

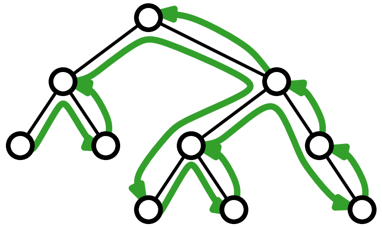
preorder

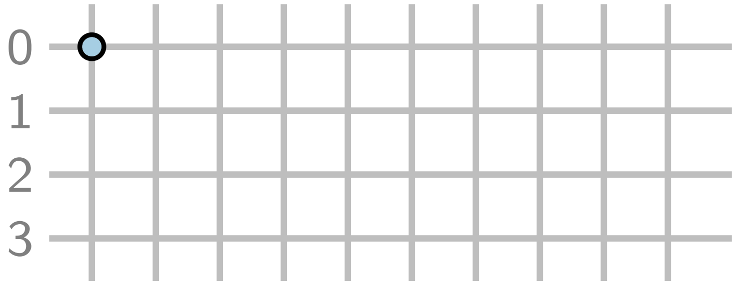


inorder



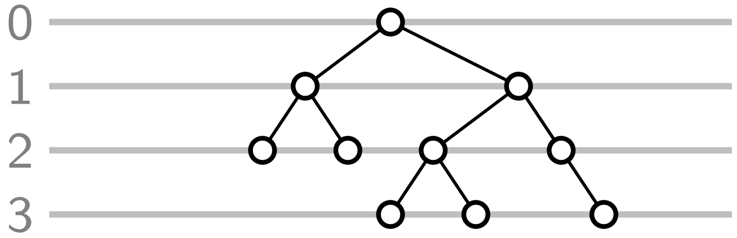
postorder





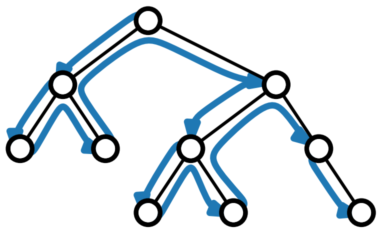
First Grid Layout of Binary Trees

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

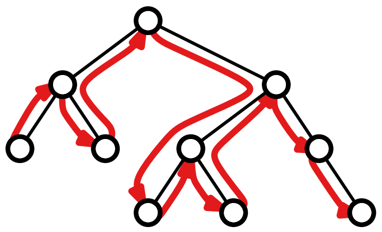


2. Choose x-coordinates:

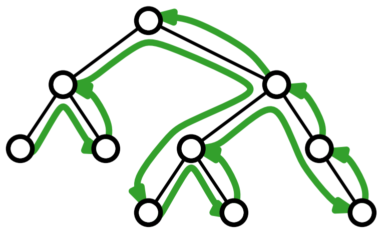
preorder

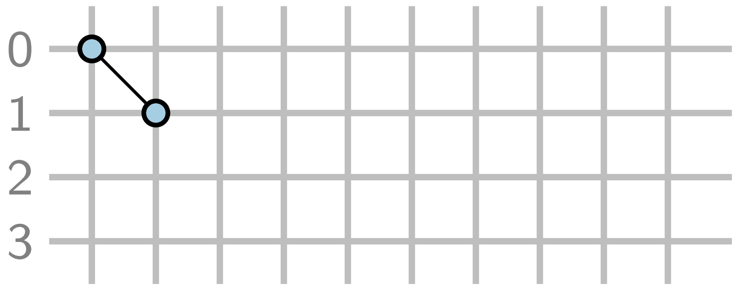


inorder



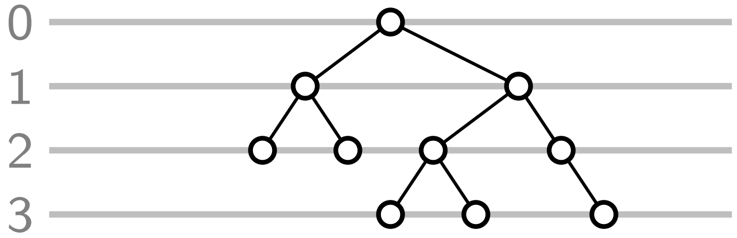
postorder



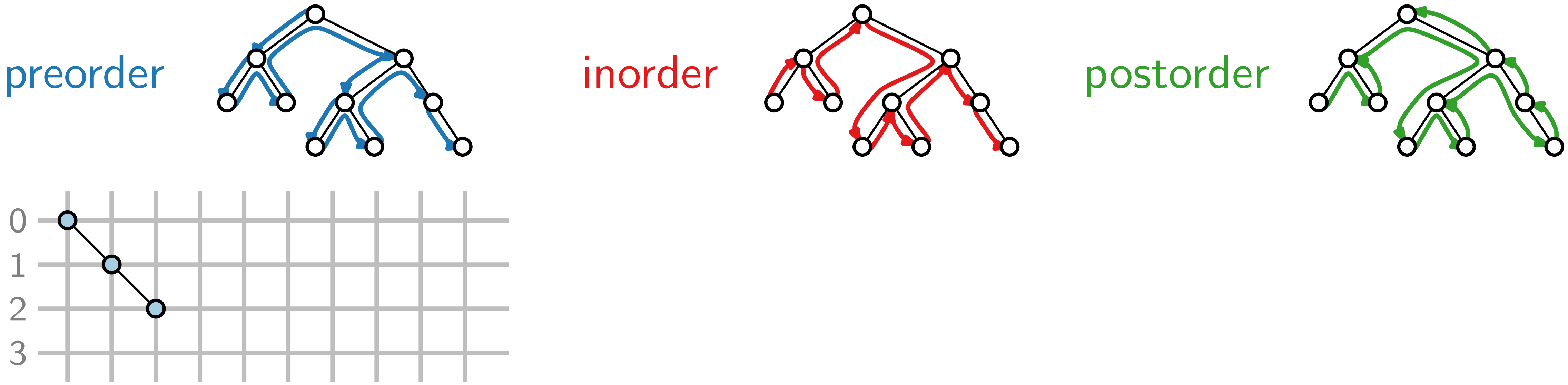


First Grid Layout of Binary Trees

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

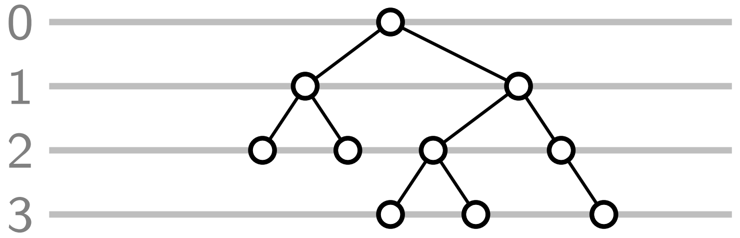


2. Choose x-coordinates:

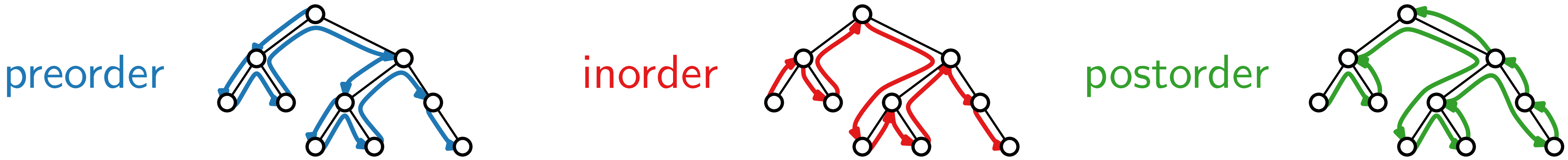


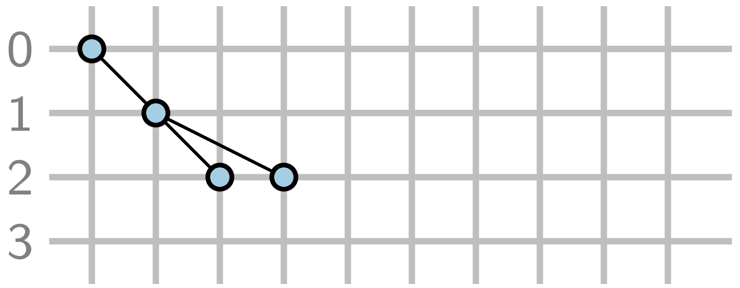
First Grid Layout of Binary Trees

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



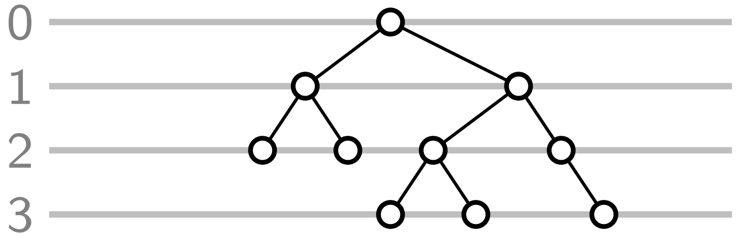
2. Choose x-coordinates:



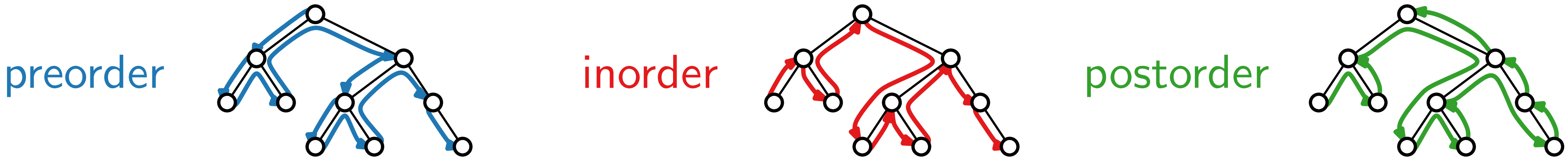


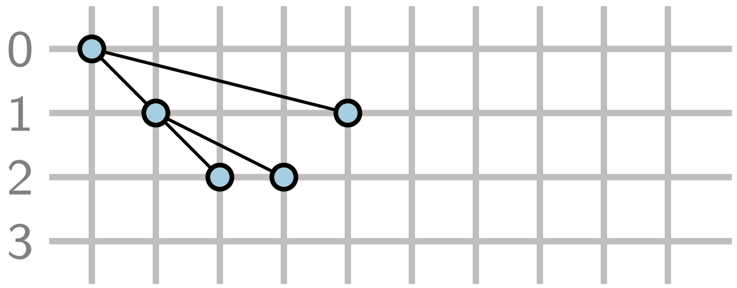
First Grid Layout of Binary Trees

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



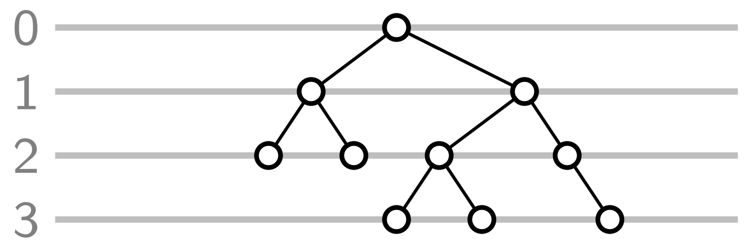
2. Choose x-coordinates:



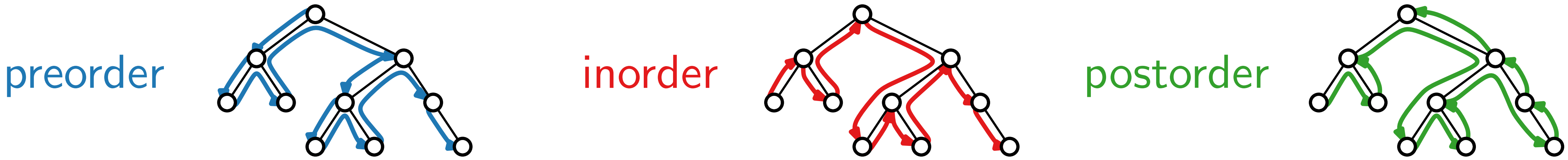


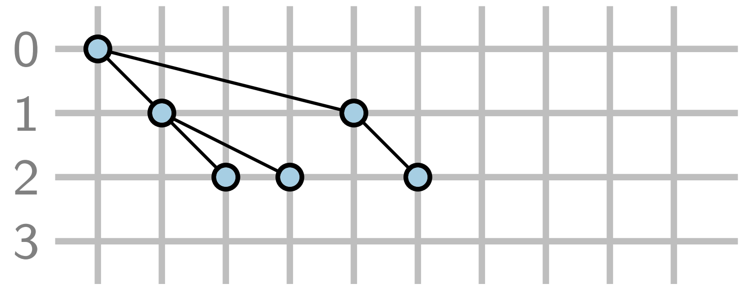
First Grid Layout of Binary Trees

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



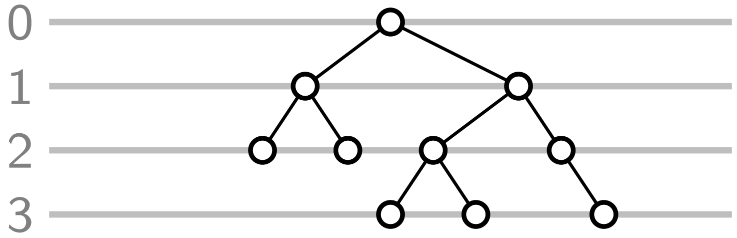
2. Choose x-coordinates:





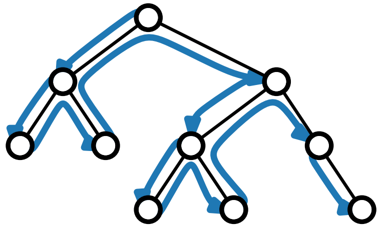
First Grid Layout of Binary Trees

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

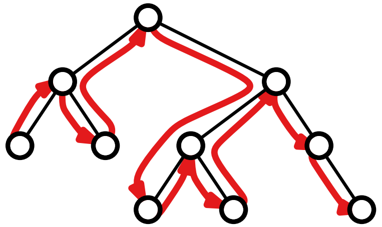


2. Choose x-coordinates:

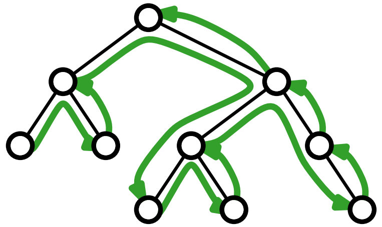
preorder

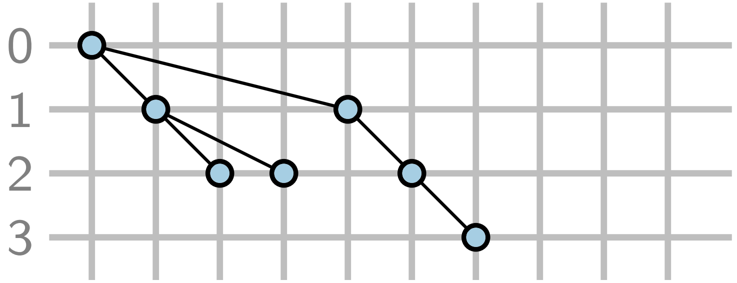


inorder



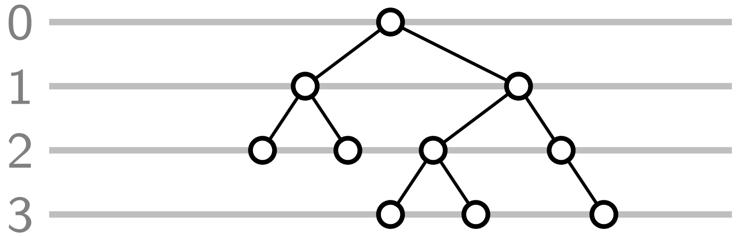
postorder





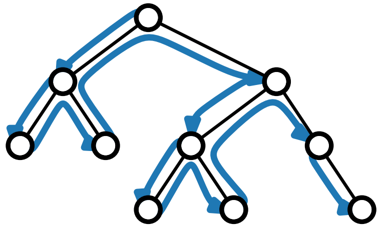
First Grid Layout of Binary Trees

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

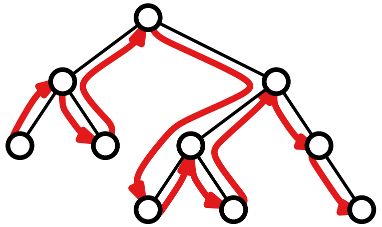


2. Choose x-coordinates:

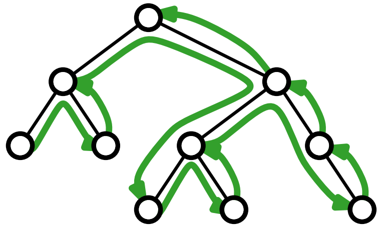
preorder

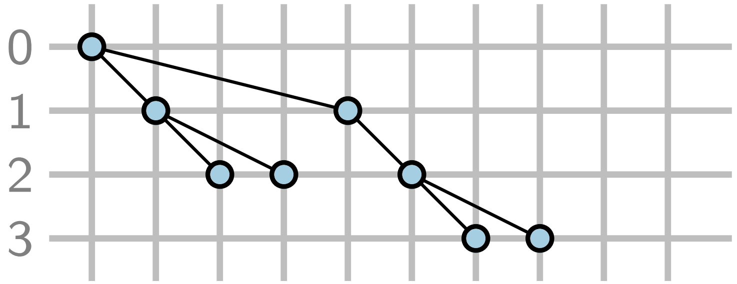


inorder



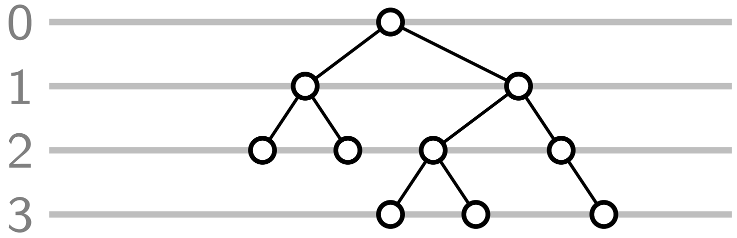
postorder





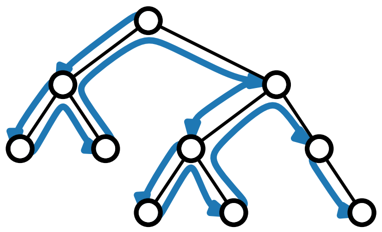
First Grid Layout of Binary Trees

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

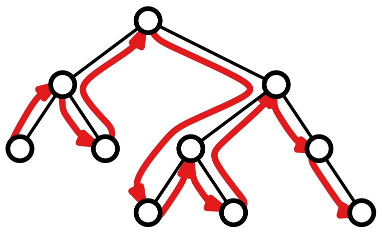


2. Choose x-coordinates:

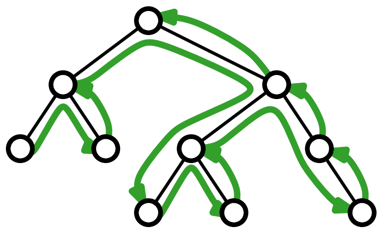
preorder

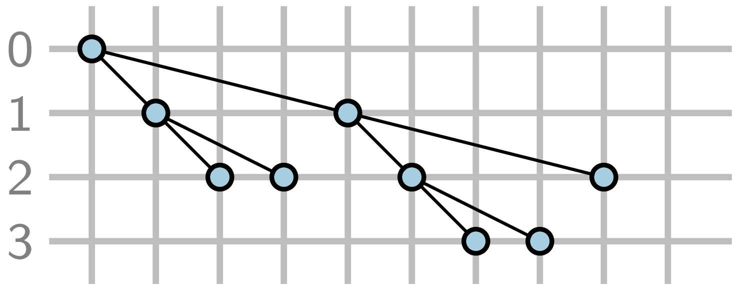


inorder



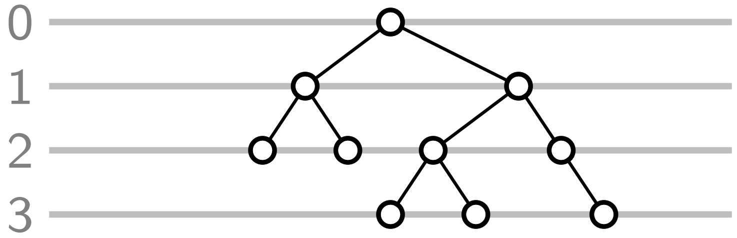
postorder



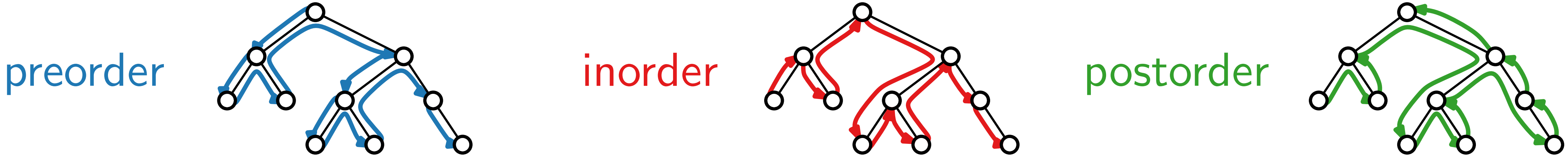


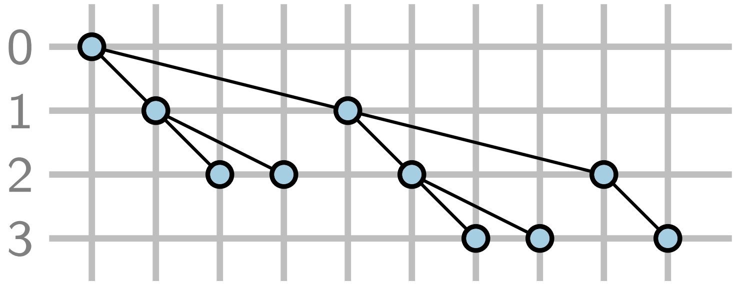
First Grid Layout of Binary Trees

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



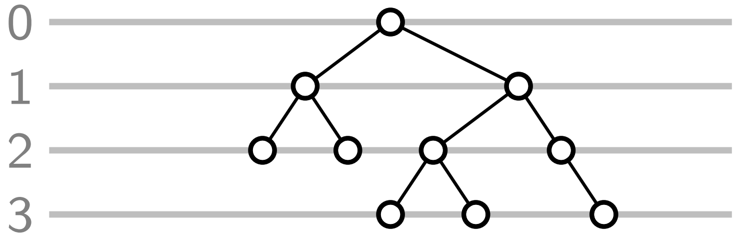
2. Choose x-coordinates:



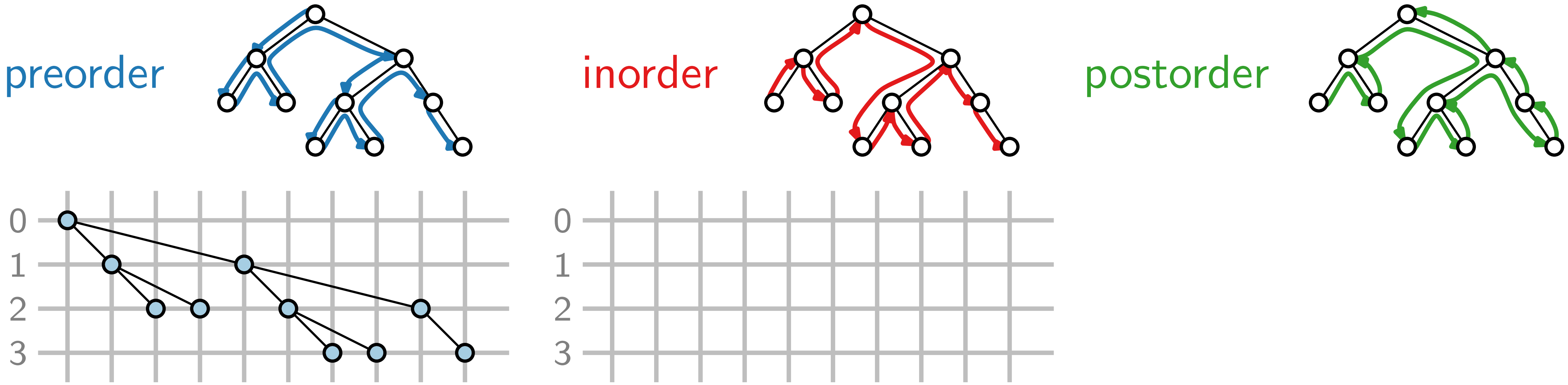


First Grid Layout of Binary Trees

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

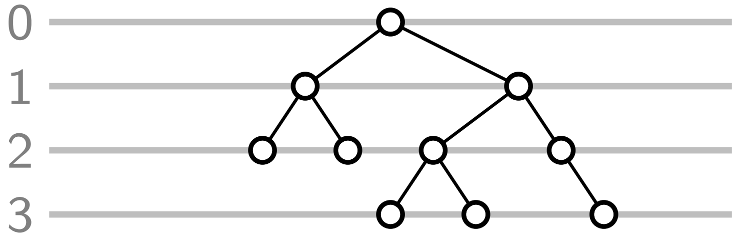


2. Choose x-coordinates:



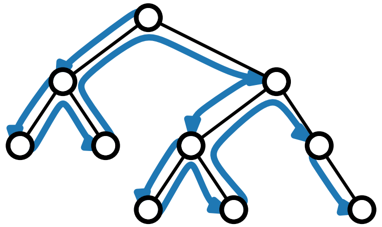
First Grid Layout of Binary Trees

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

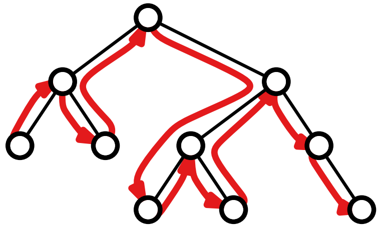


2. Choose x-coordinates:

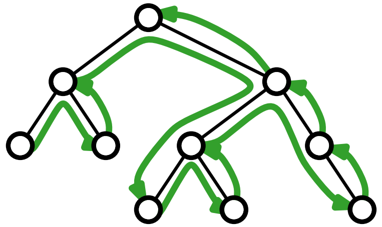
preorder

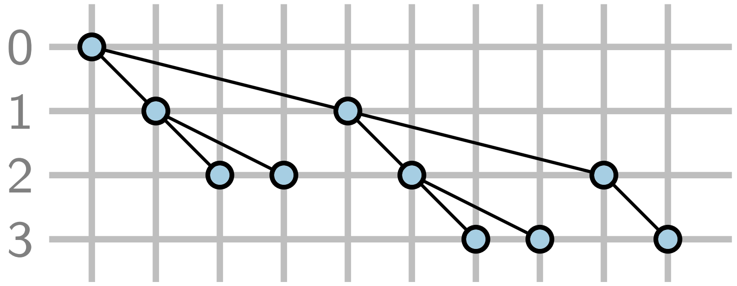


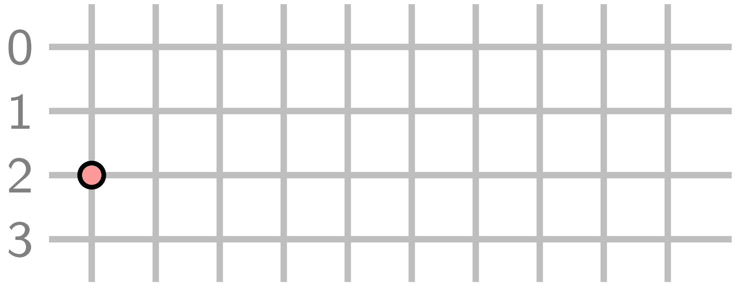
inorder



postorder

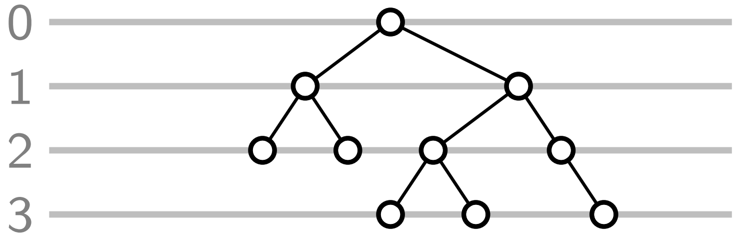






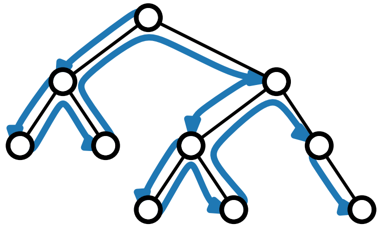
First Grid Layout of Binary Trees

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

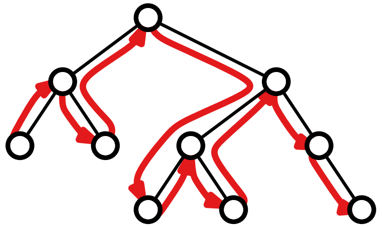


2. Choose x-coordinates:

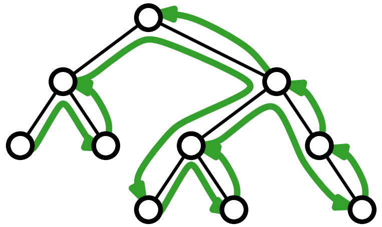
preorder

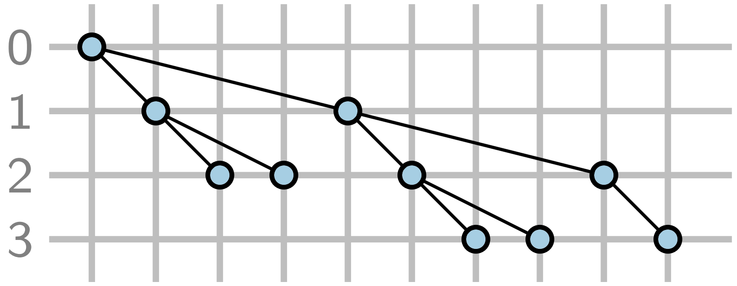


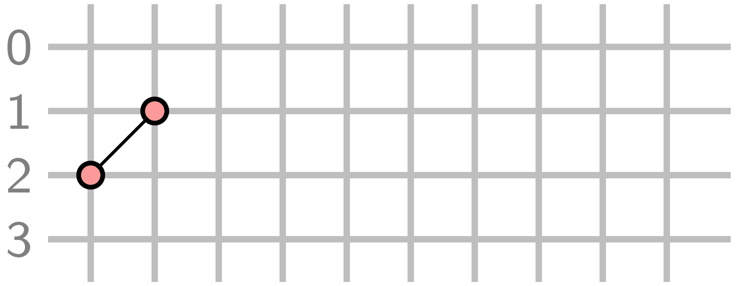
inorder



postorder

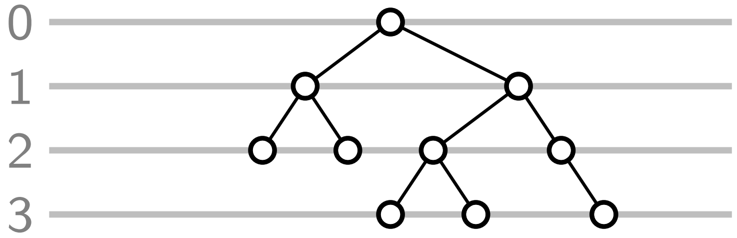




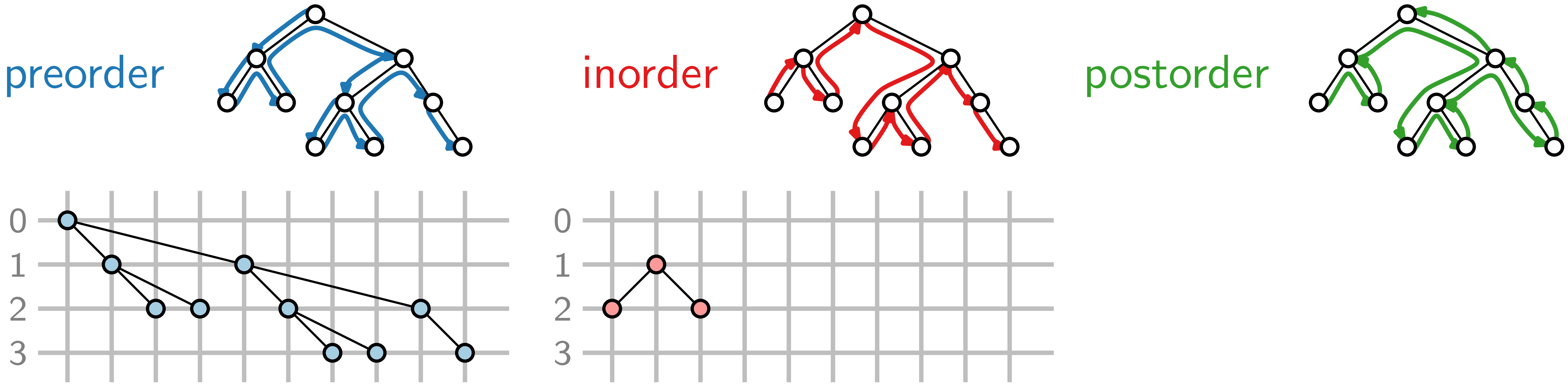


First Grid Layout of Binary Trees

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

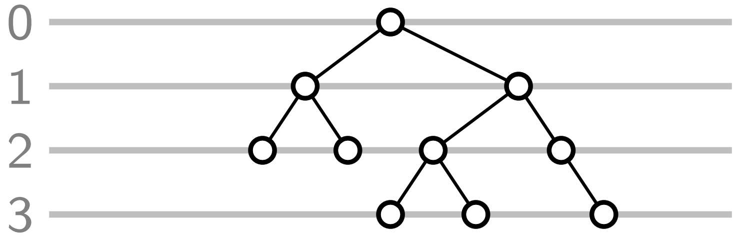


2. Choose x-coordinates:

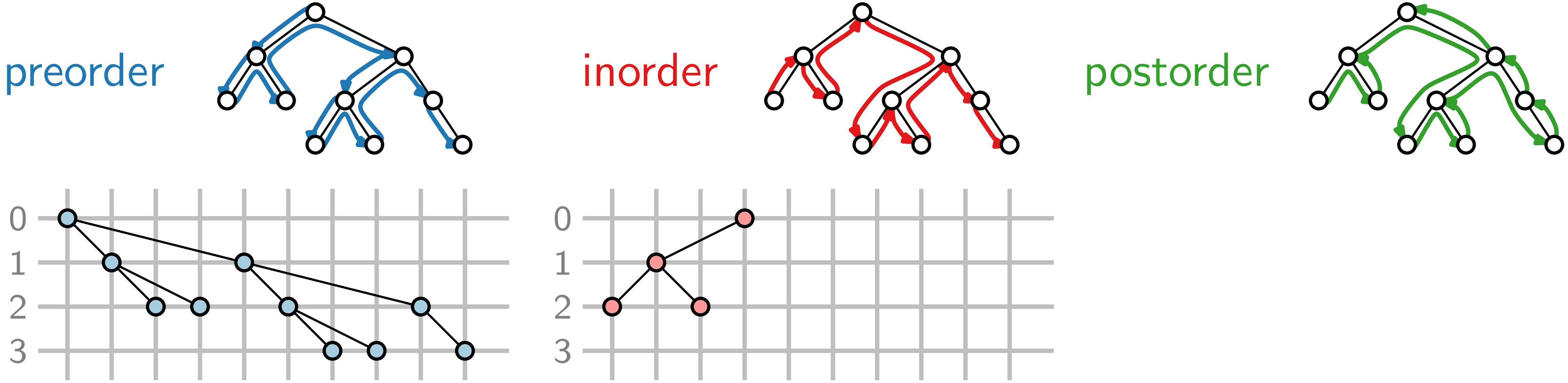


First Grid Layout of Binary Trees

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

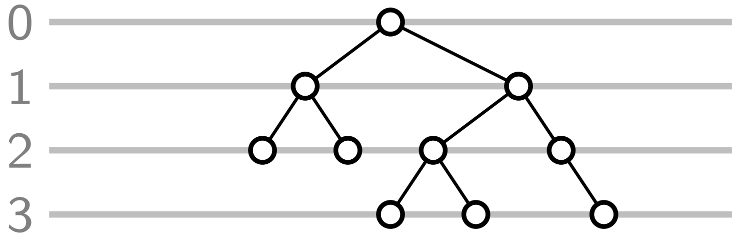


2. Choose x-coordinates:

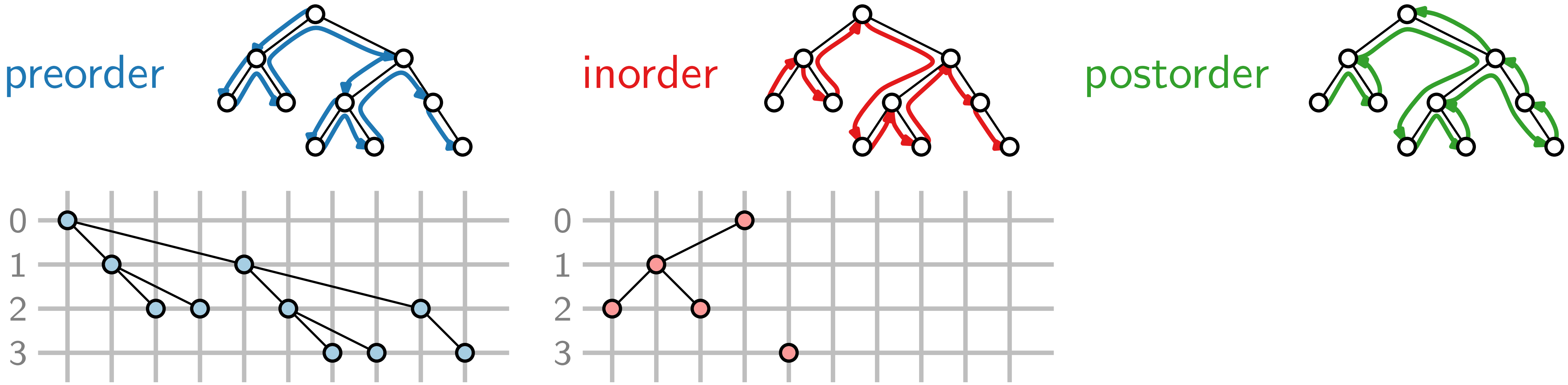


First Grid Layout of Binary Trees

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

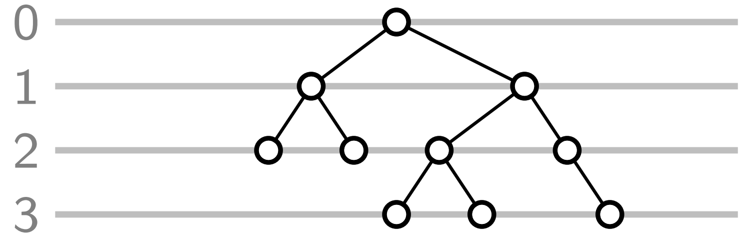


2. Choose x-coordinates:

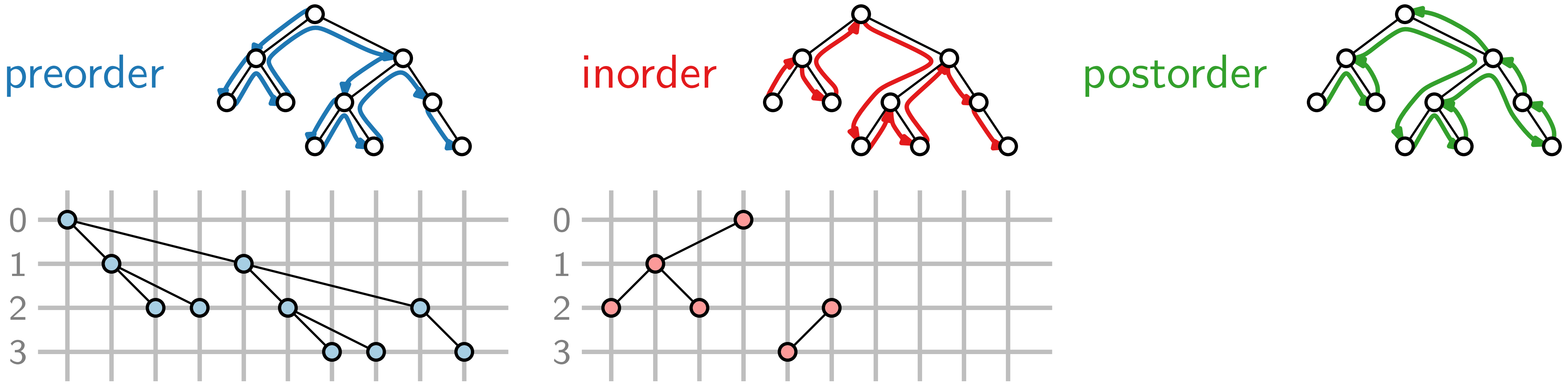


First Grid Layout of Binary Trees

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

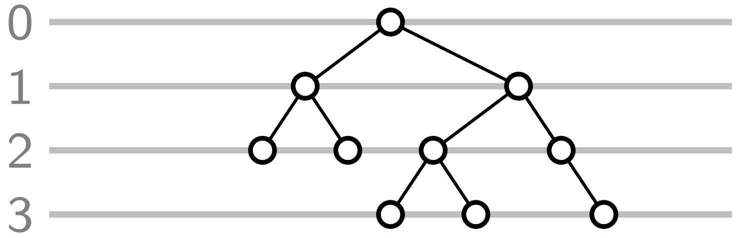


2. Choose x-coordinates:



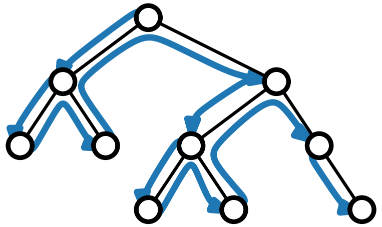
First Grid Layout of Binary Trees

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

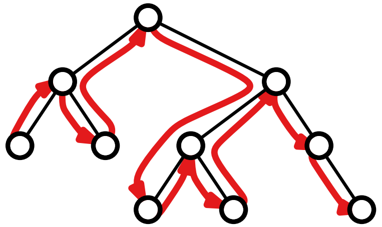


2. Choose x-coordinates:

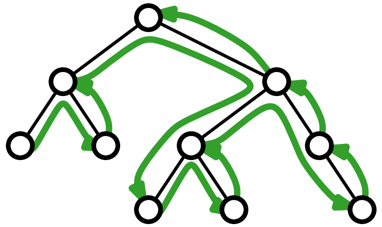
preorder

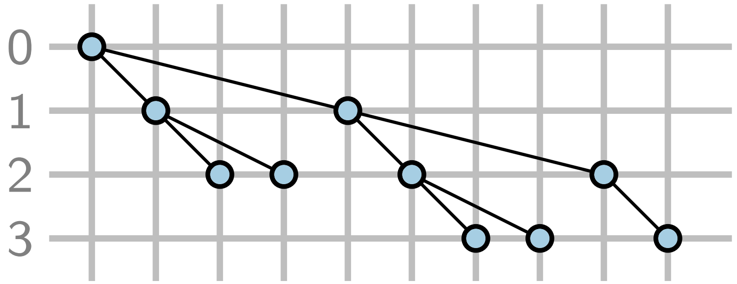


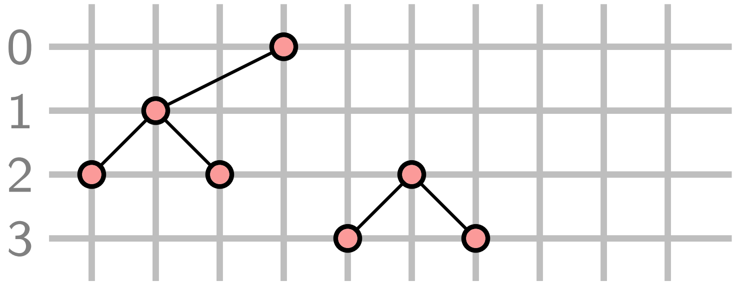
inorder



postorder

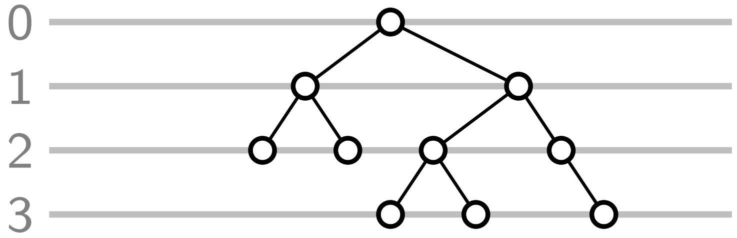




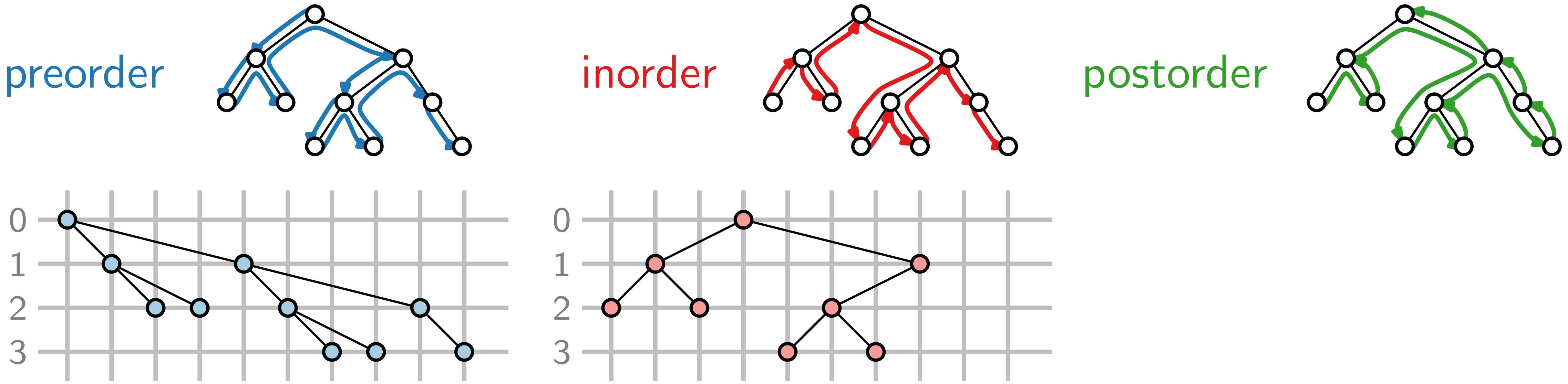


First Grid Layout of Binary Trees

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

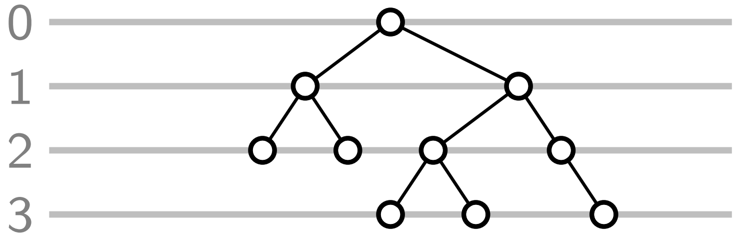


2. Choose x-coordinates:

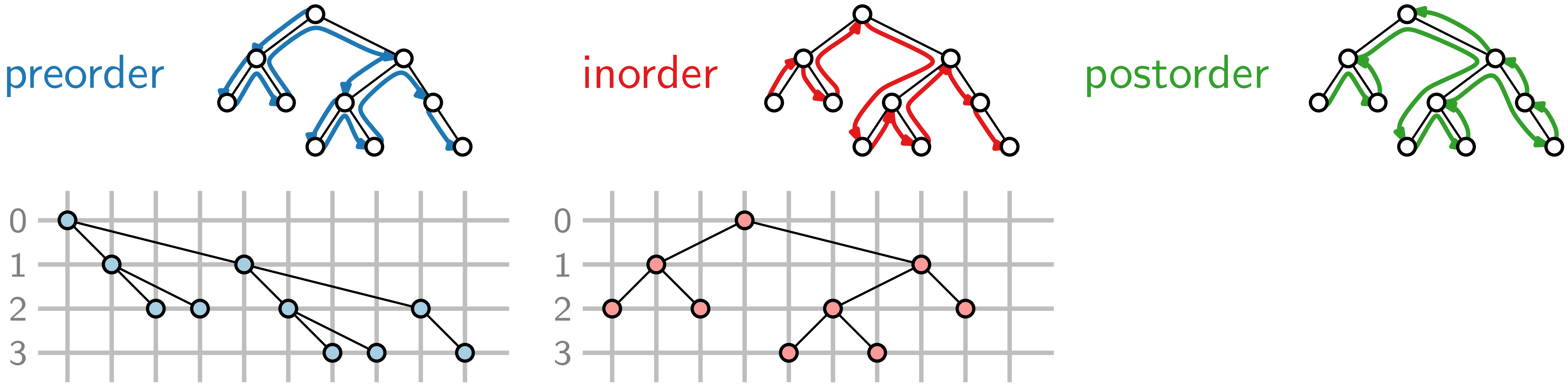


First Grid Layout of Binary Trees

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

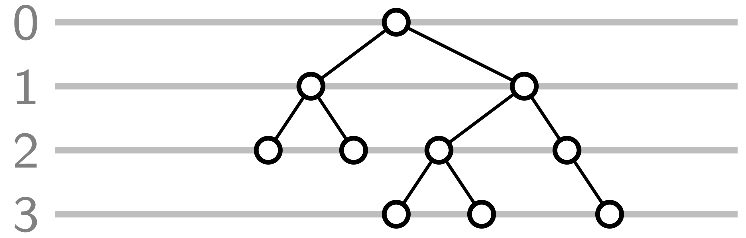


2. Choose x-coordinates:

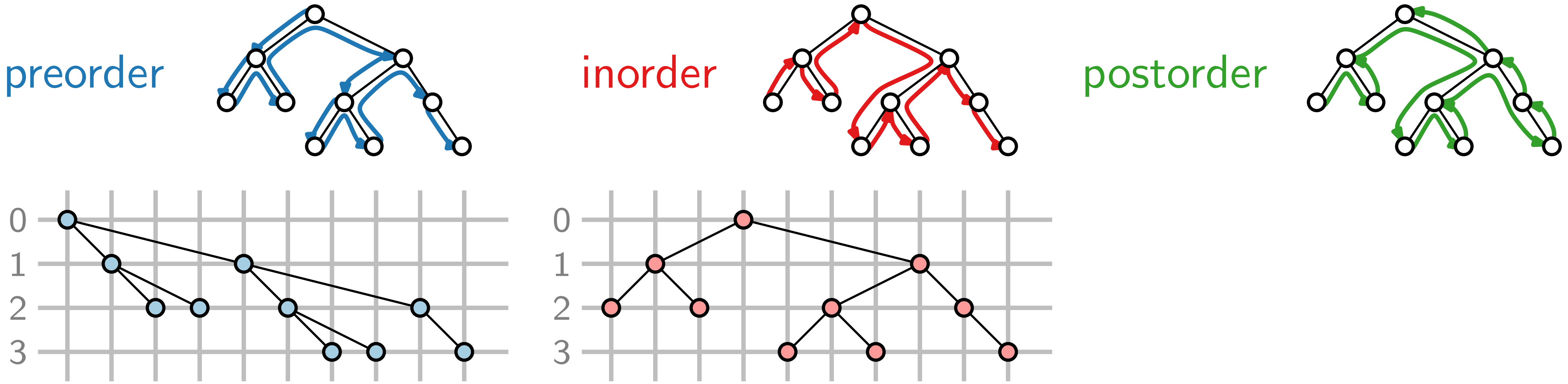


First Grid Layout of Binary Trees

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

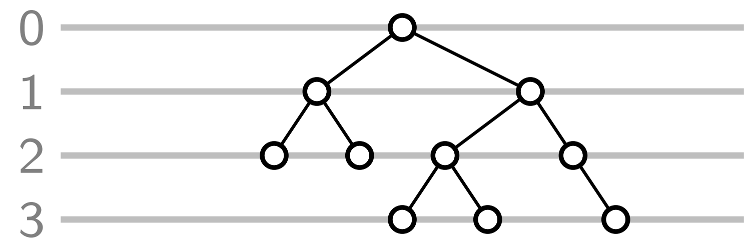


2. Choose x-coordinates:

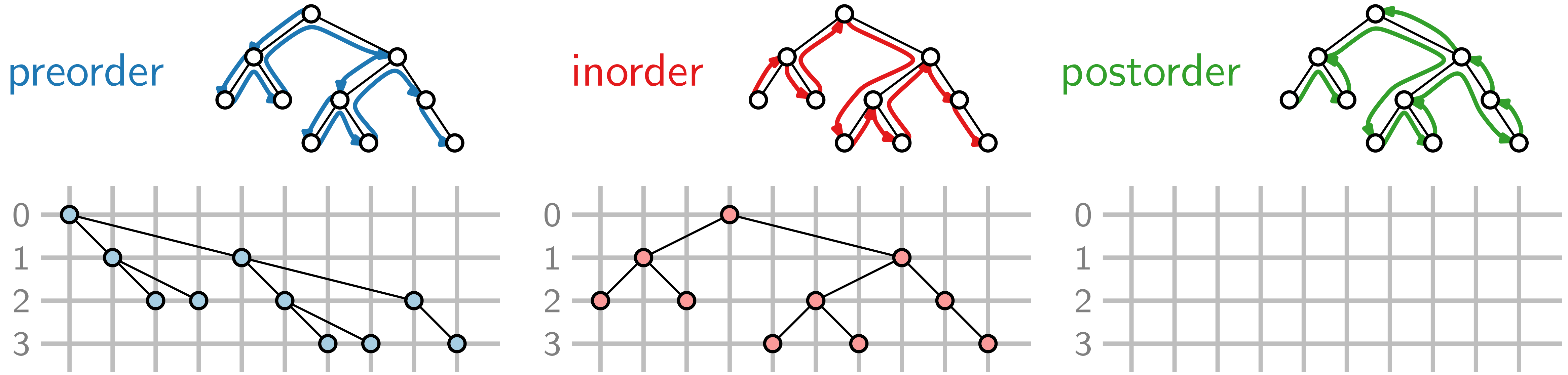


First Grid Layout of Binary Trees

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

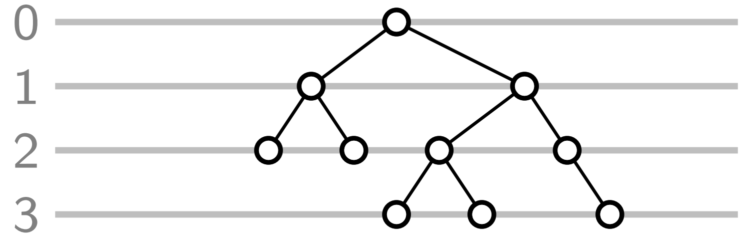


2. Choose x-coordinates:

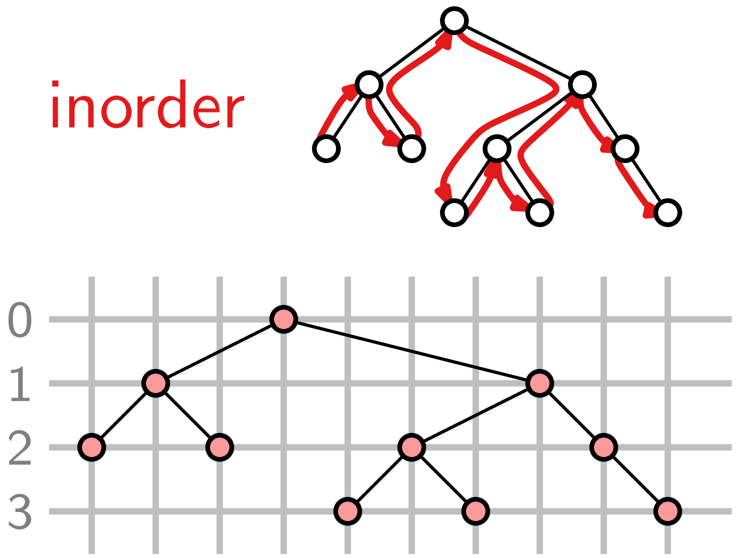


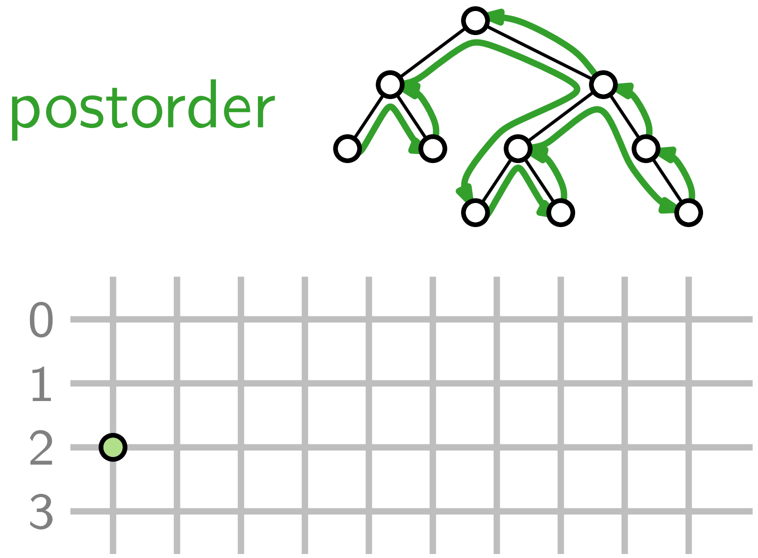
First Grid Layout of Binary Trees

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



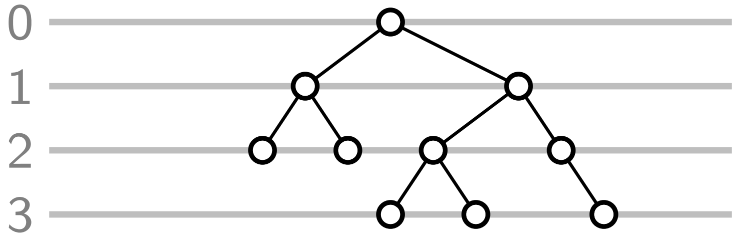
2. Choose x-coordinates:



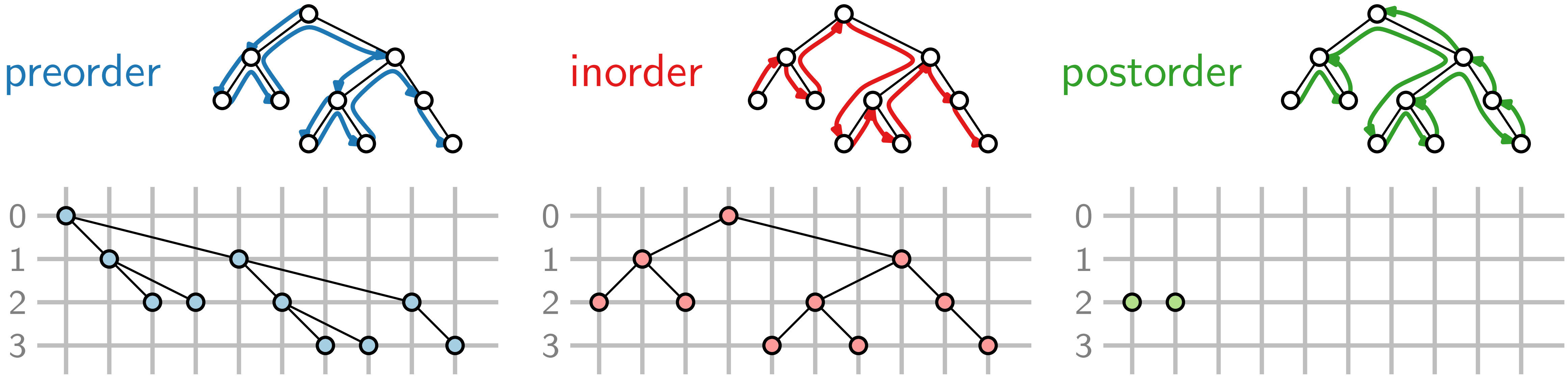


First Grid Layout of Binary Trees

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

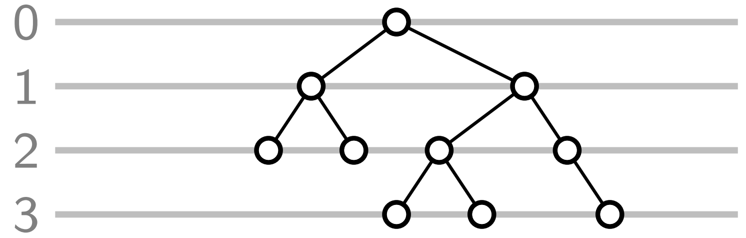


2. Choose x-coordinates:

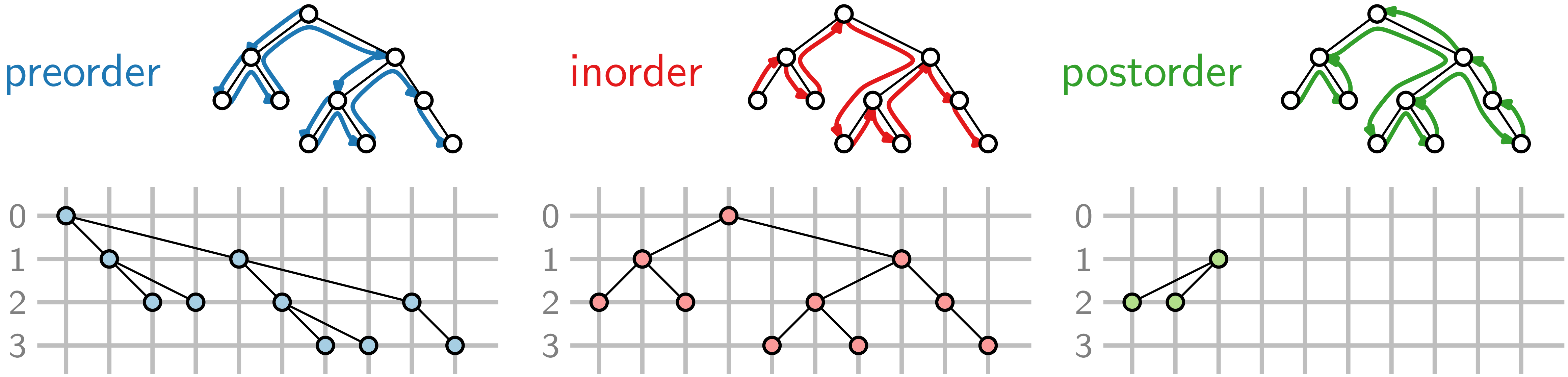


First Grid Layout of Binary Trees

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

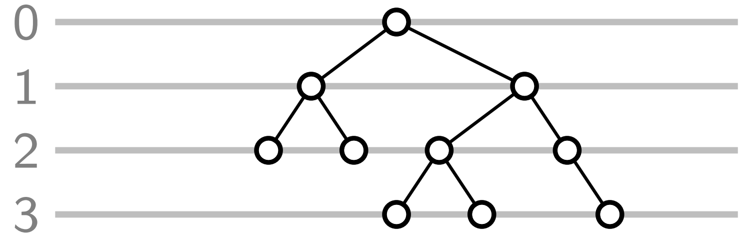


2. Choose x-coordinates:

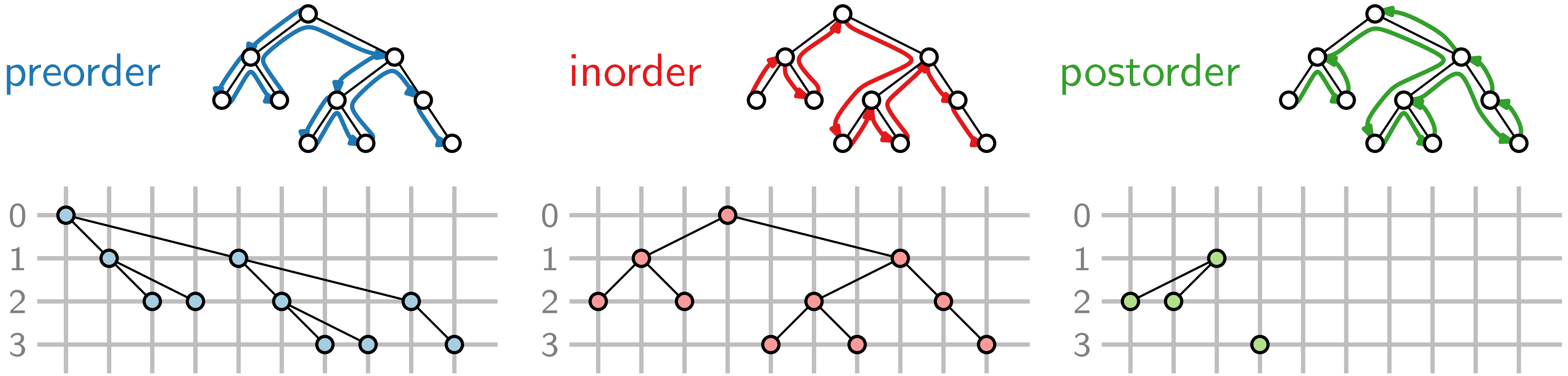


First Grid Layout of Binary Trees

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

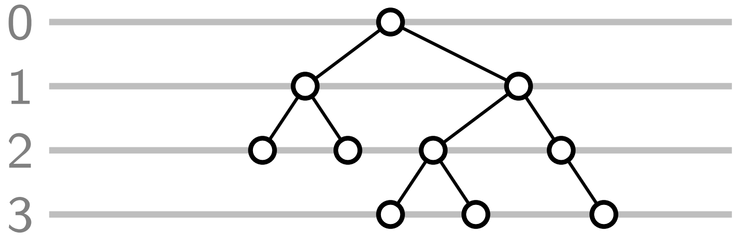


2. Choose x-coordinates:

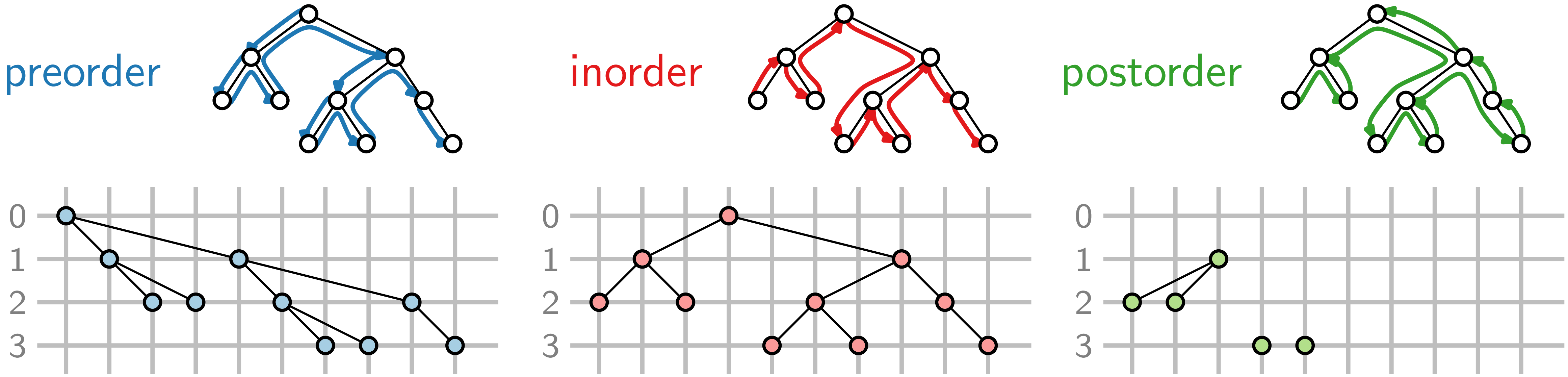


First Grid Layout of Binary Trees

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

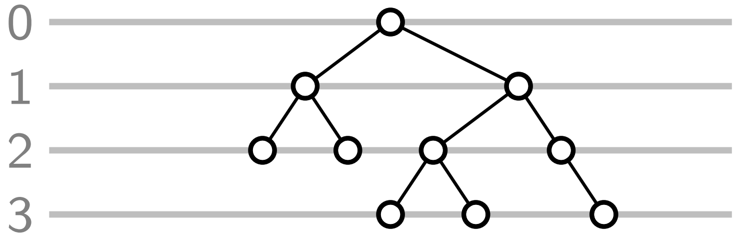


2. Choose x-coordinates:

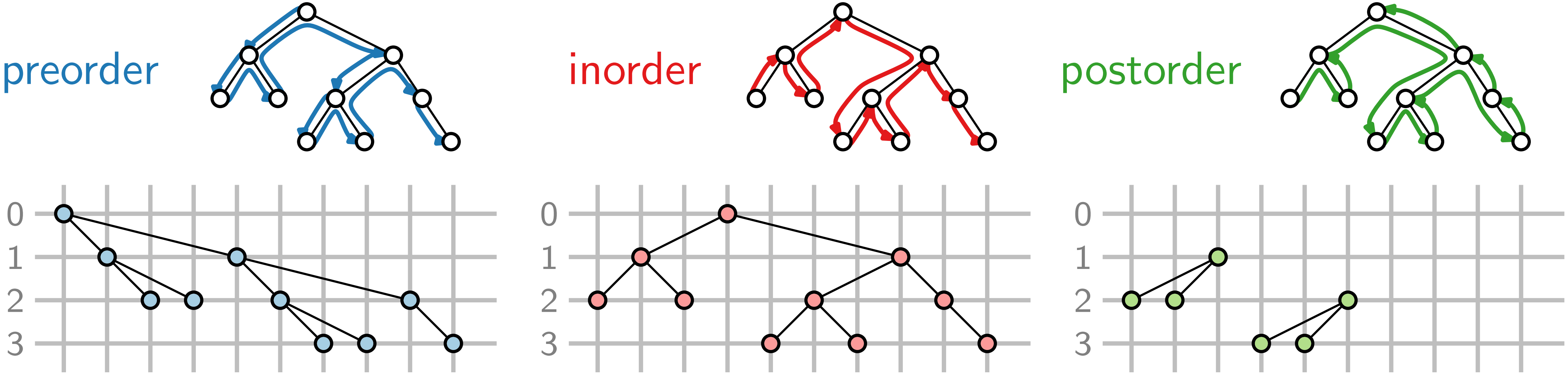


First Grid Layout of Binary Trees

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

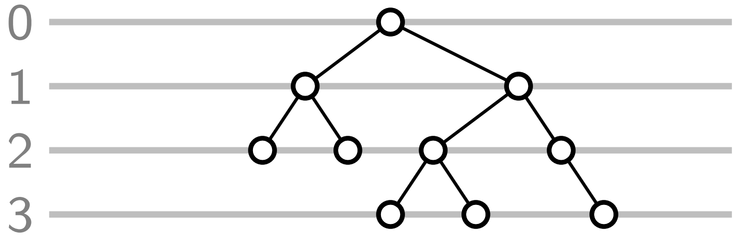


2. Choose x-coordinates:

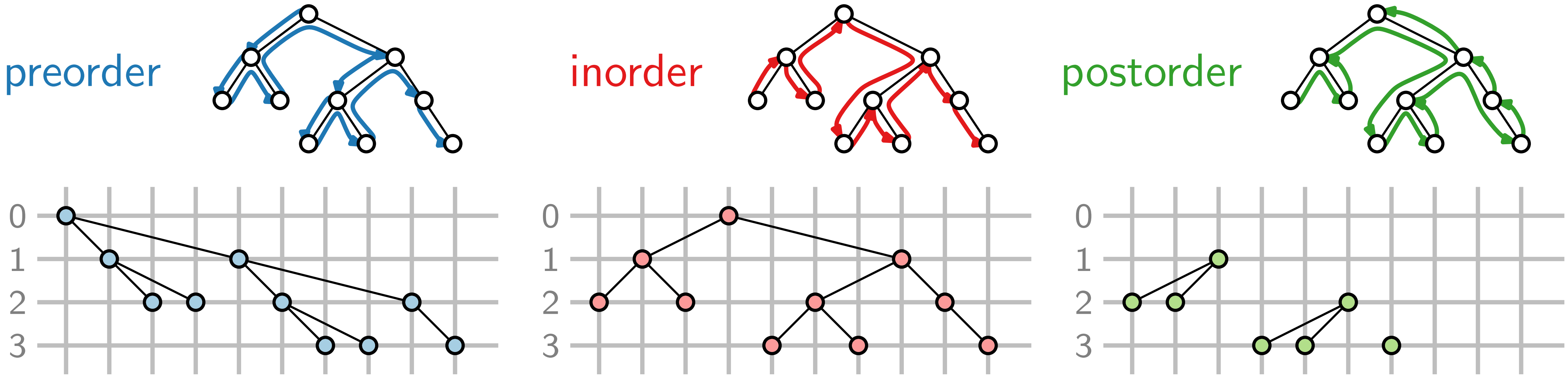


First Grid Layout of Binary Trees

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

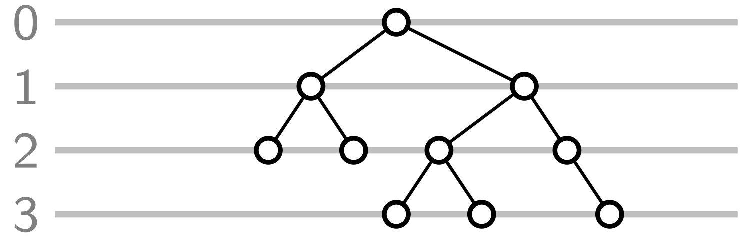


2. Choose x-coordinates:

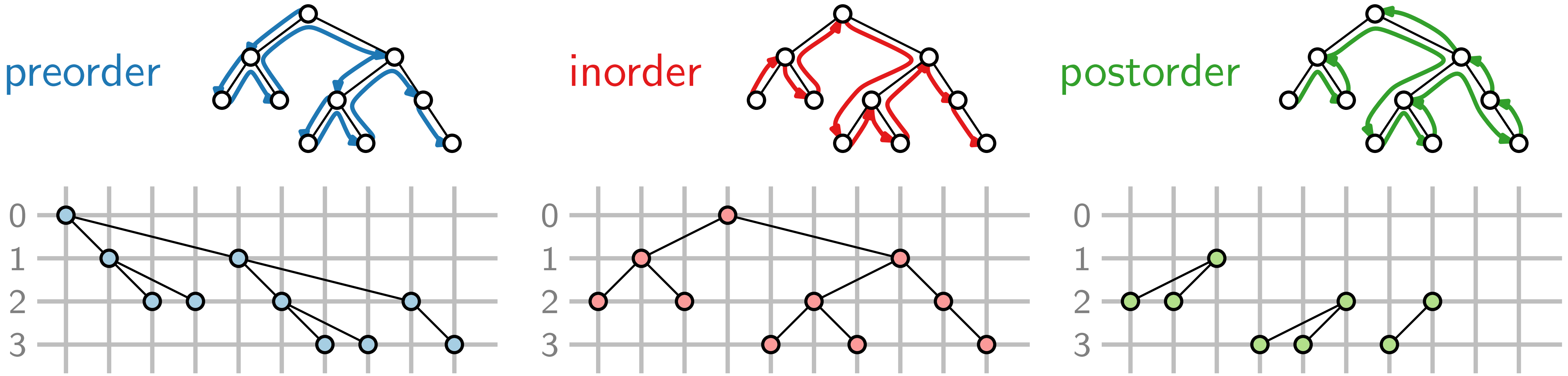


First Grid Layout of Binary Trees

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

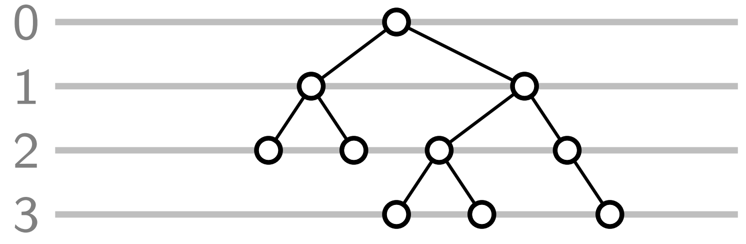


2. Choose x-coordinates:

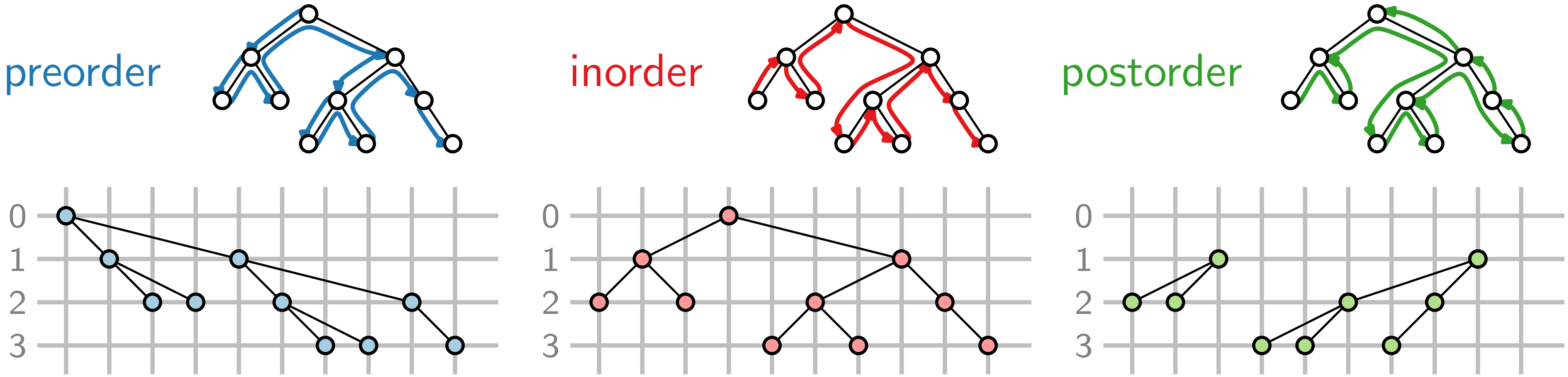


First Grid Layout of Binary Trees

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

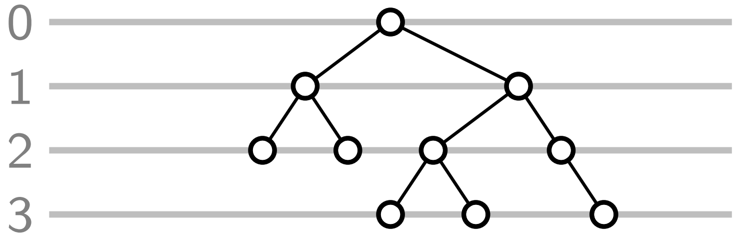


2. Choose x-coordinates:

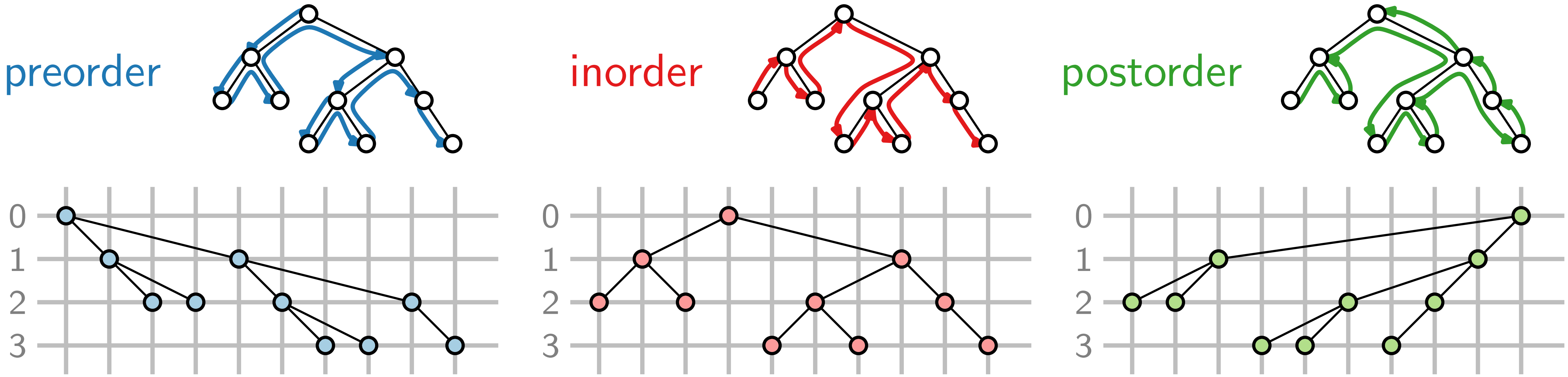


First Grid Layout of Binary Trees

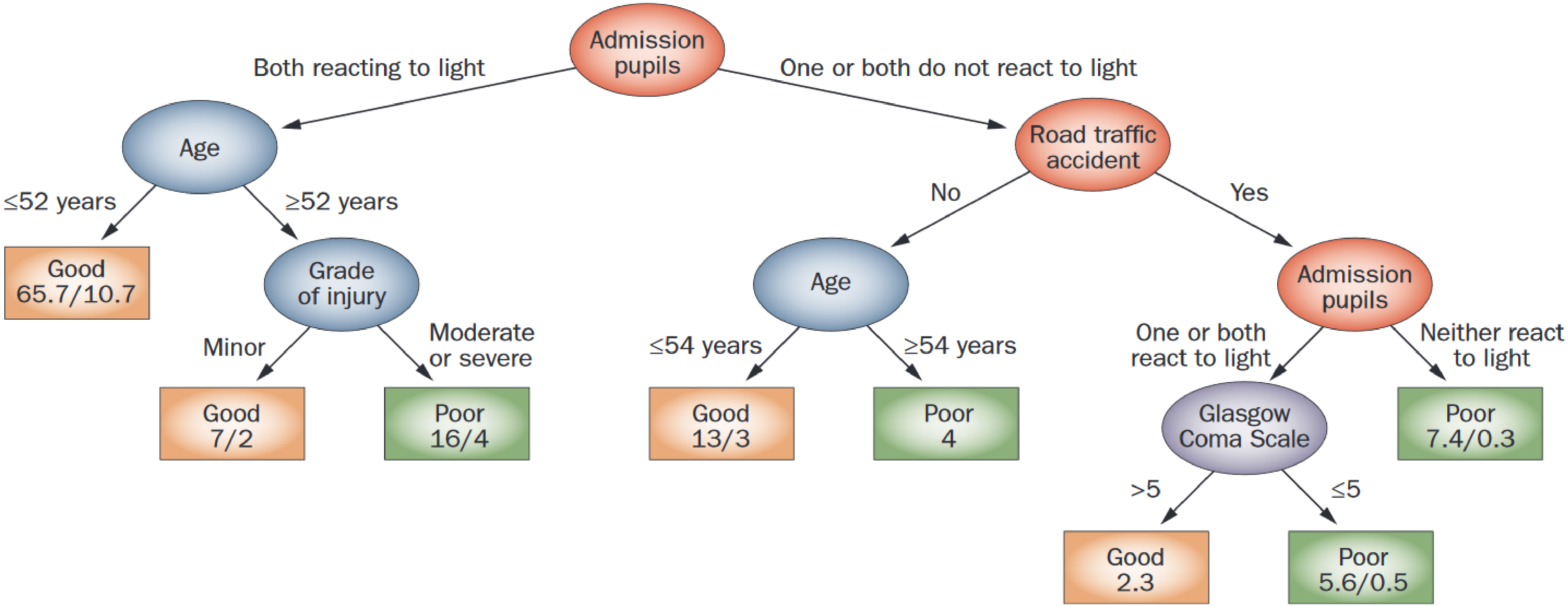
1. Choose y-coordinates: $y(u) = \text{depth}(u)$



2. Choose x-coordinates:



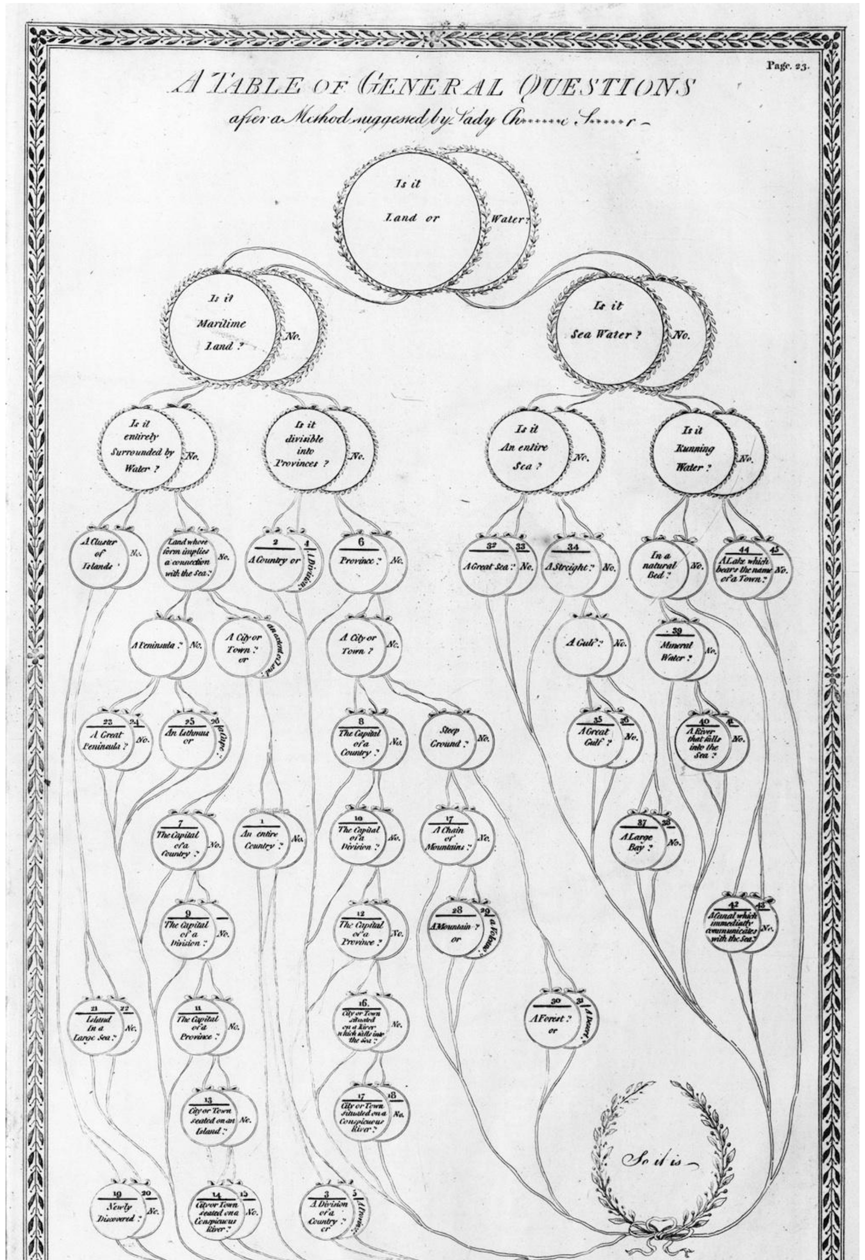
Layered Drawings – Applications



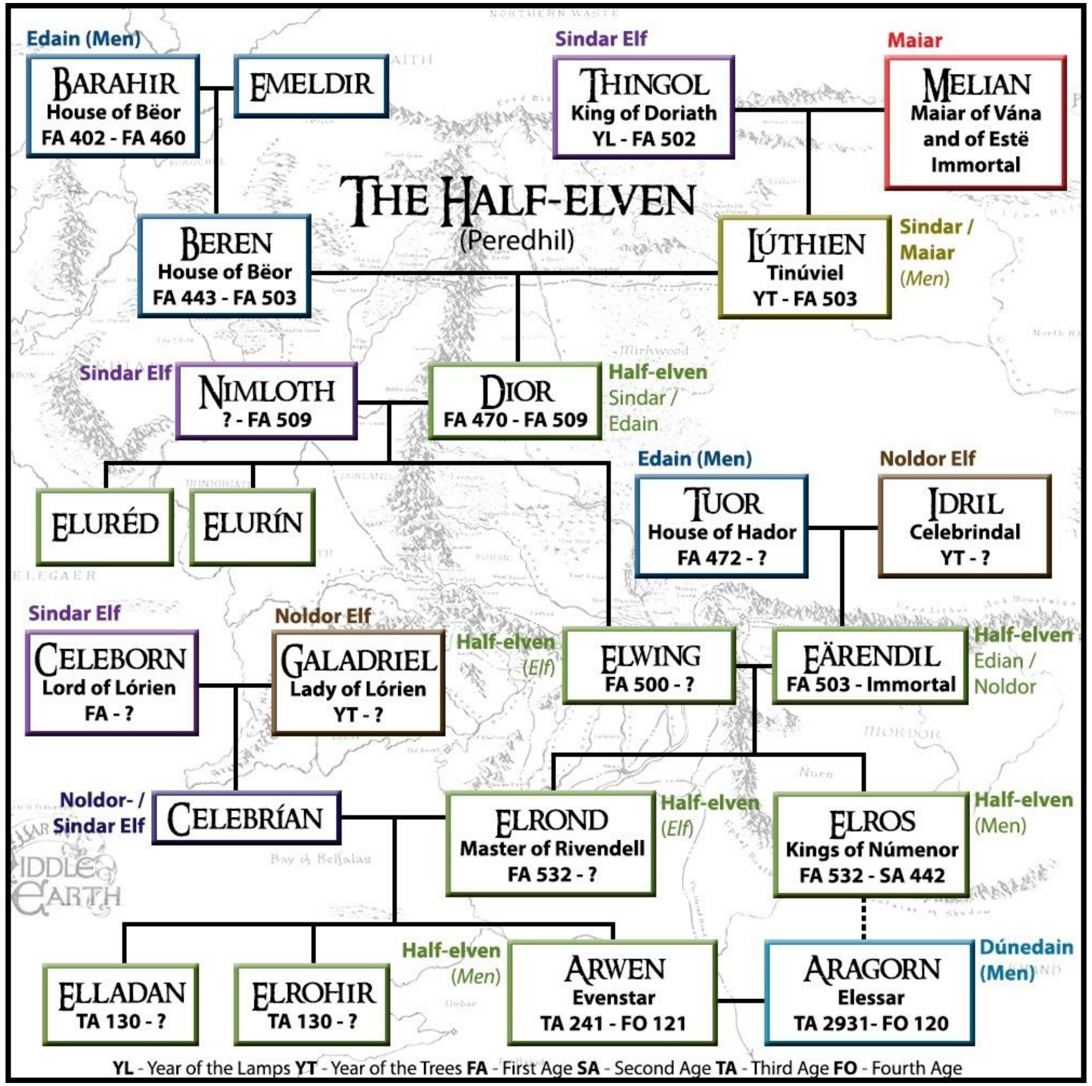
Decision tree for outcome prediction after traumatic brain injury

Source: Nature Reviews Neurology

Layered Drawings – Applications

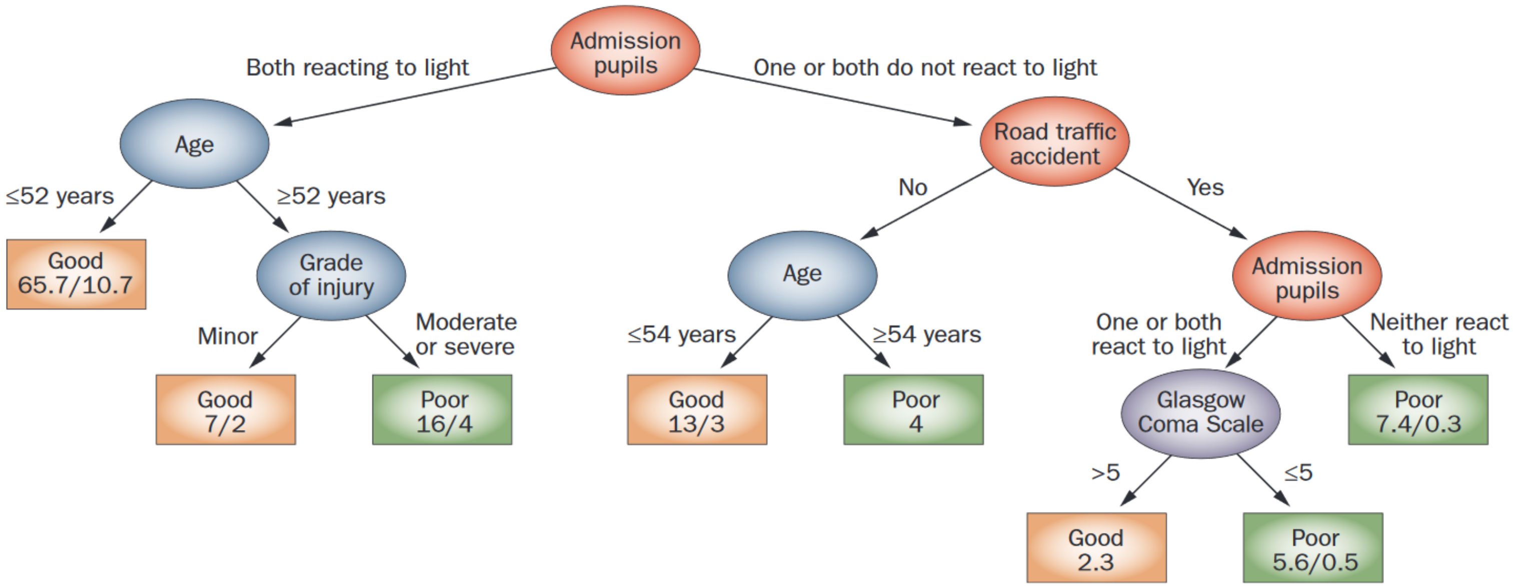


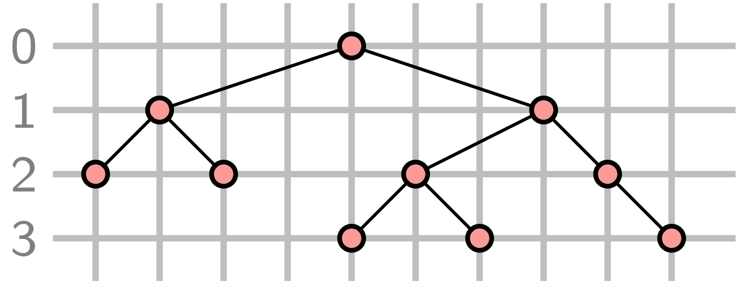
Aloisius Gaultier 1821



Family tree of LOTR elves and half-elves

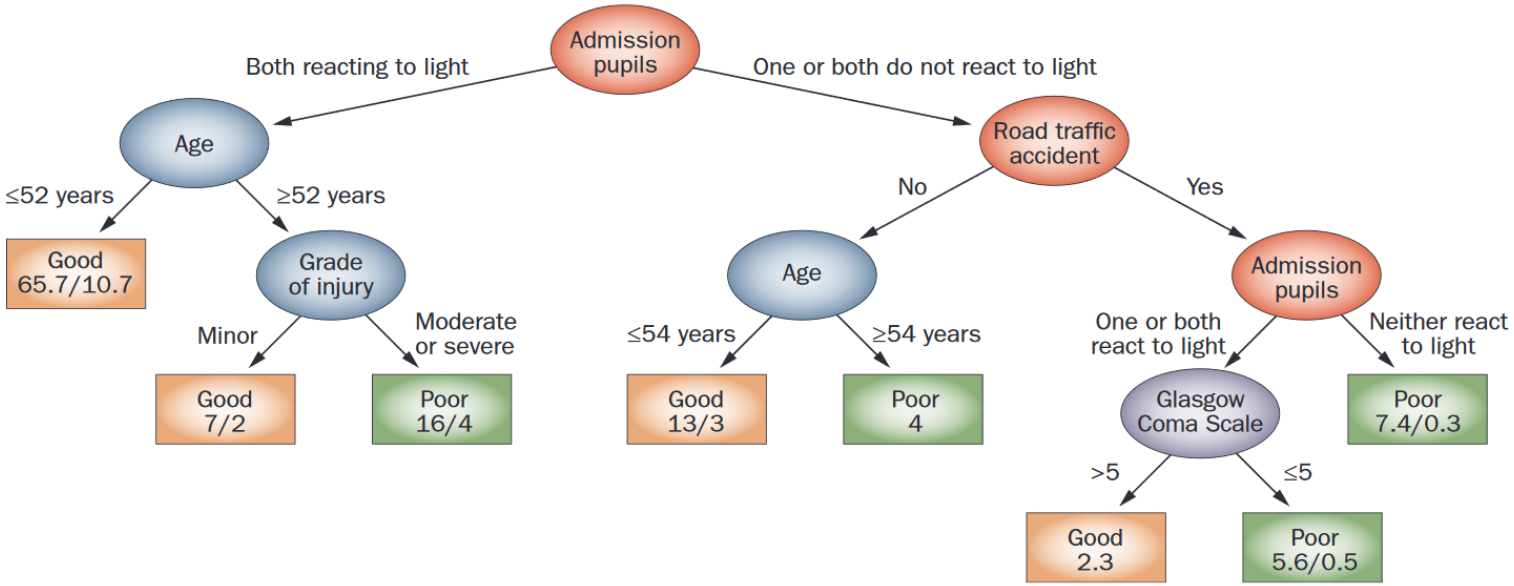
Layered Drawings – Drawing Style

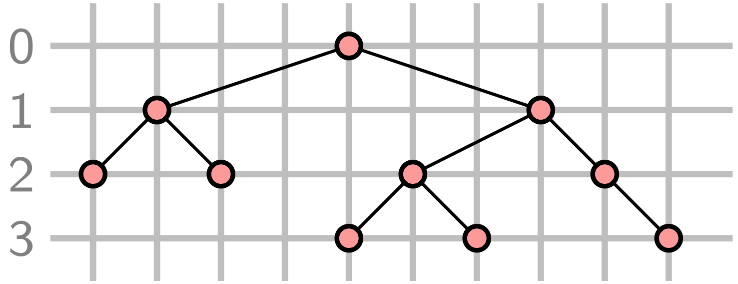




- What are properties of the layout?

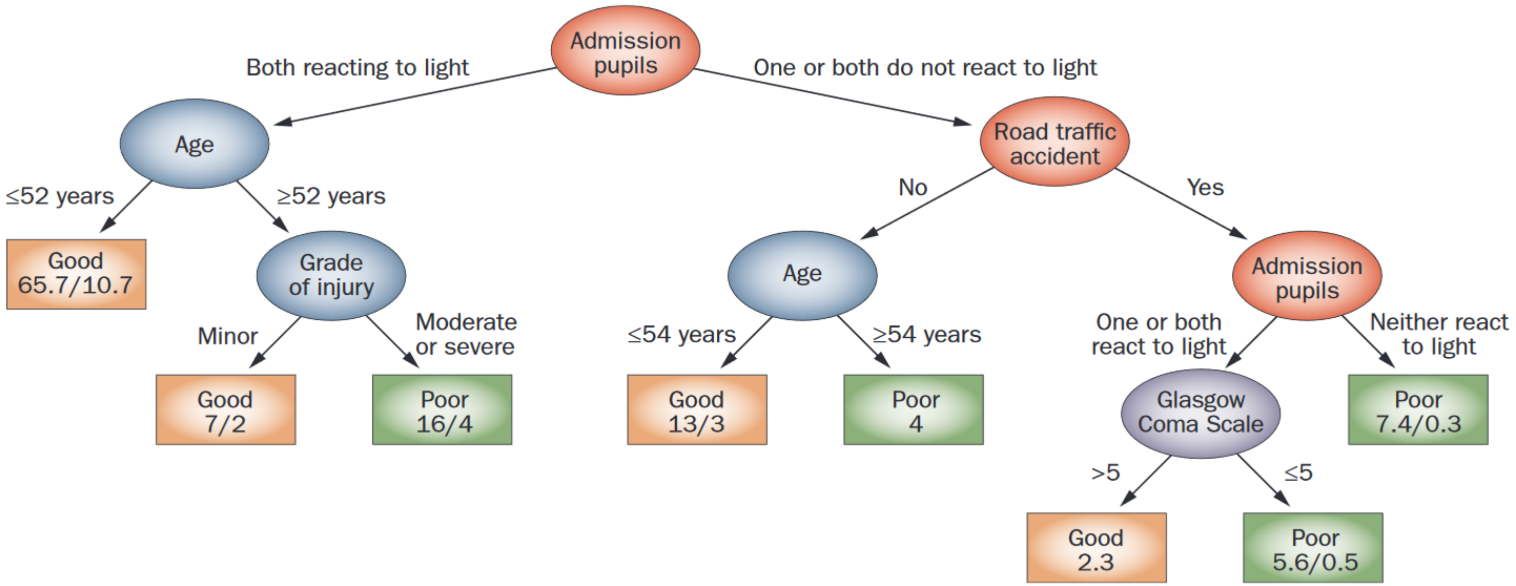
Layered Drawings – Drawing Style

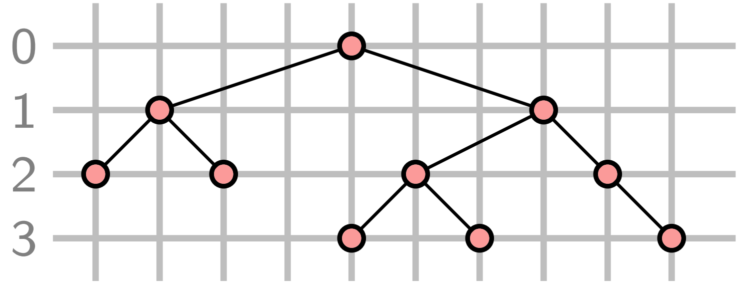




- What are properties of the layout?
- What are the drawing conventions?

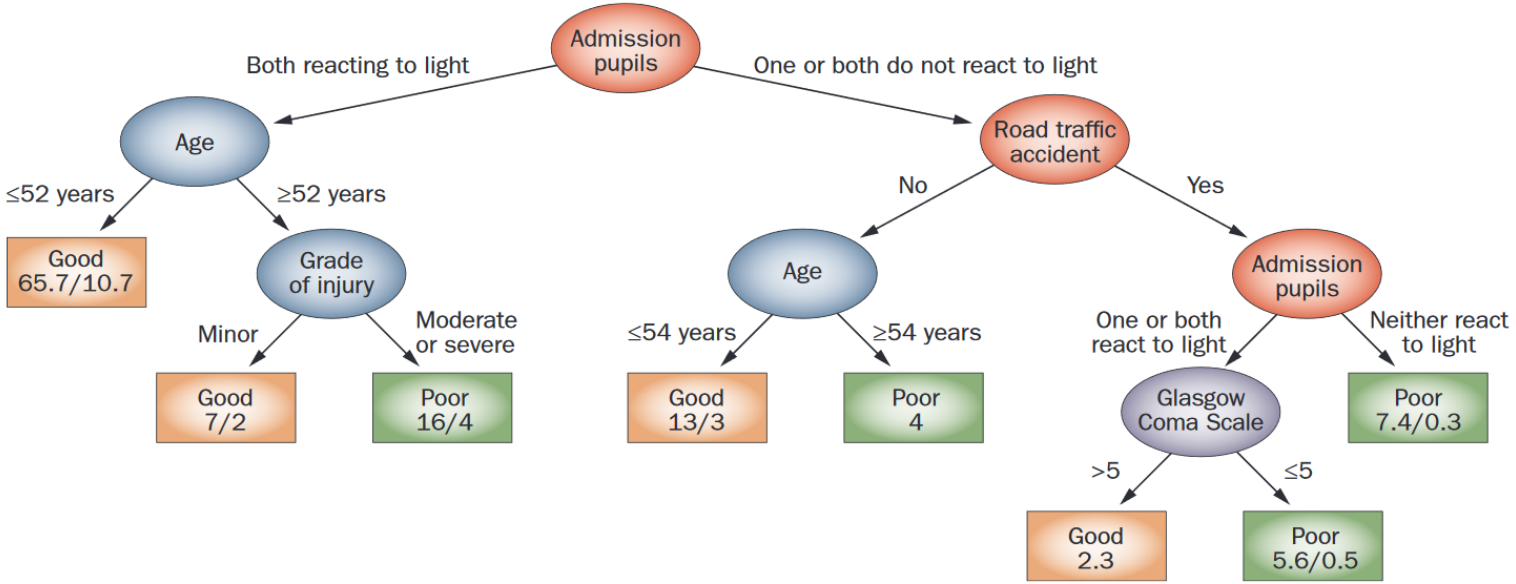
Layered Drawings – Drawing Style

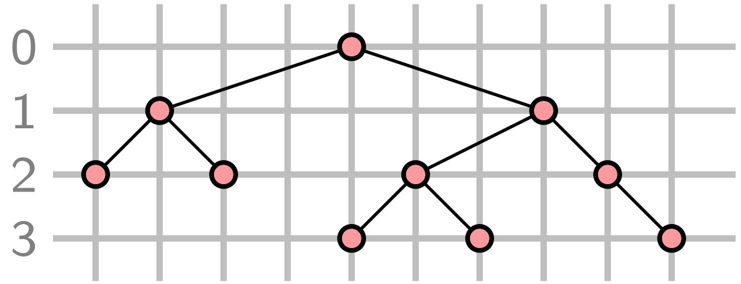




- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

Layered Drawings – Drawing Style

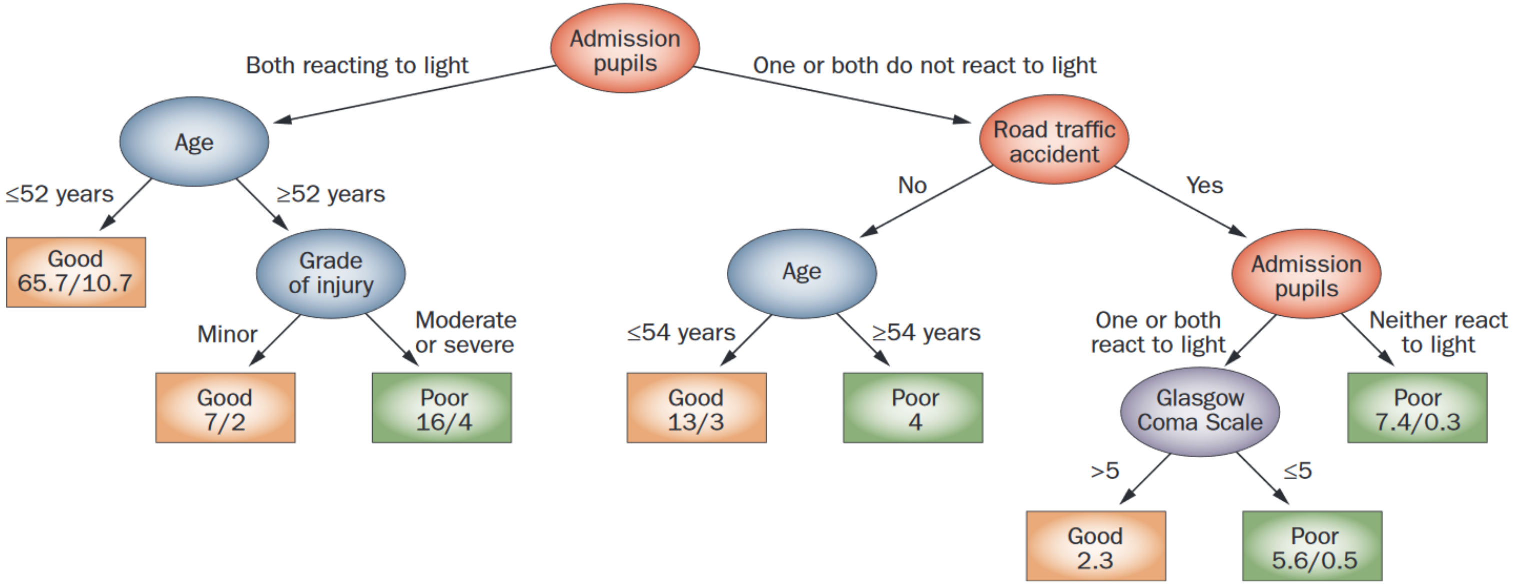




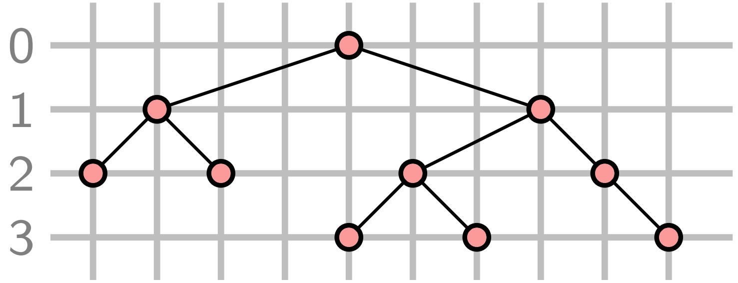
Drawing conventions

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

Layered Drawings – Drawing Style



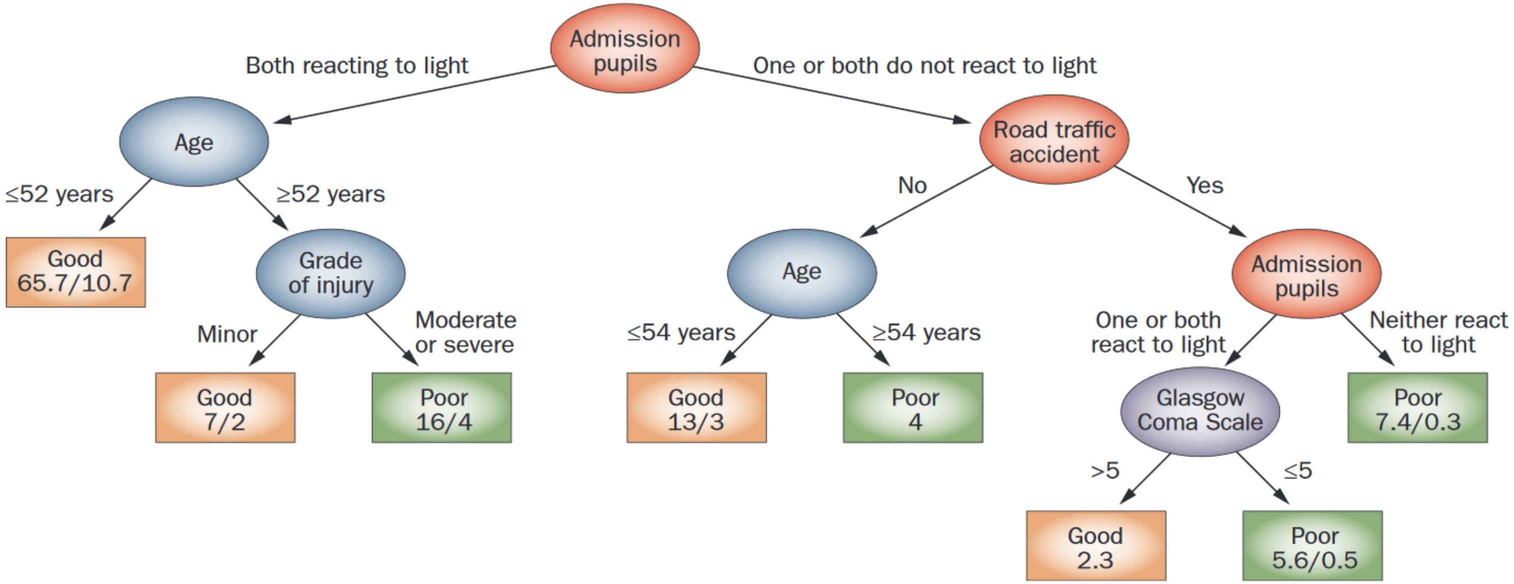
- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

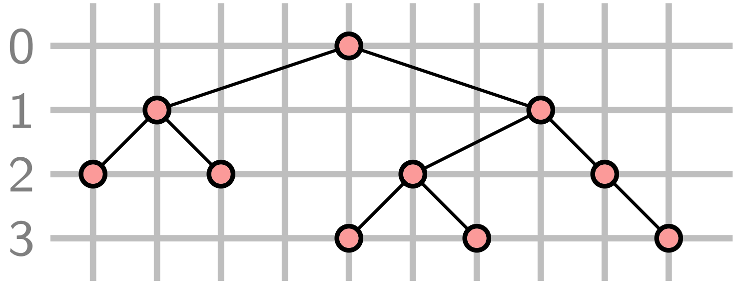


Drawing conventions

- Vertices lie on layers and have integer coordinates

Layered Drawings – Drawing Style



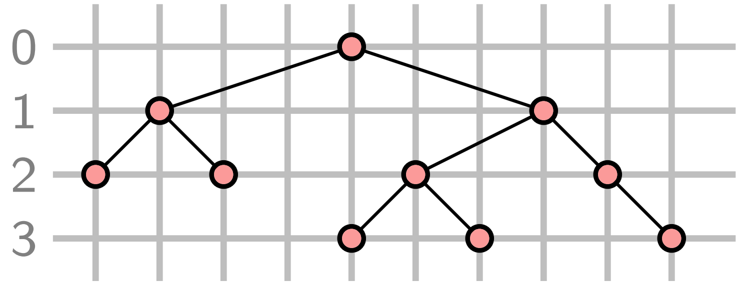


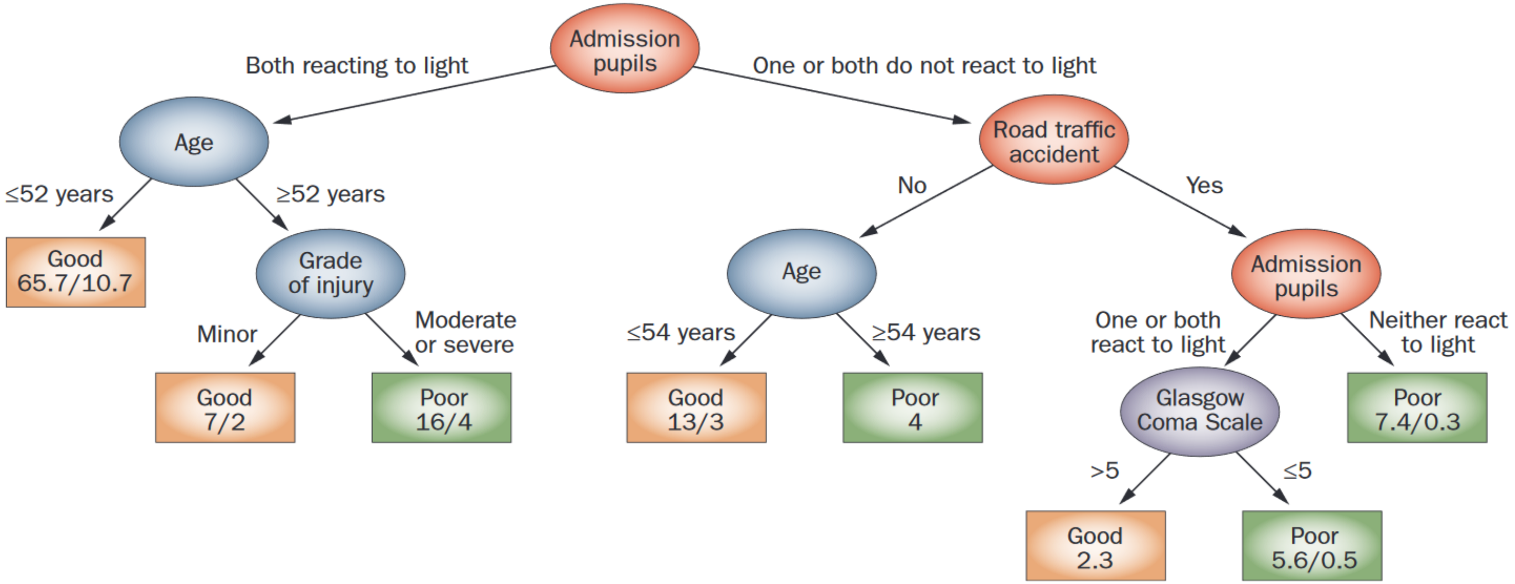
Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children

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

Layered Drawings – Drawing Style



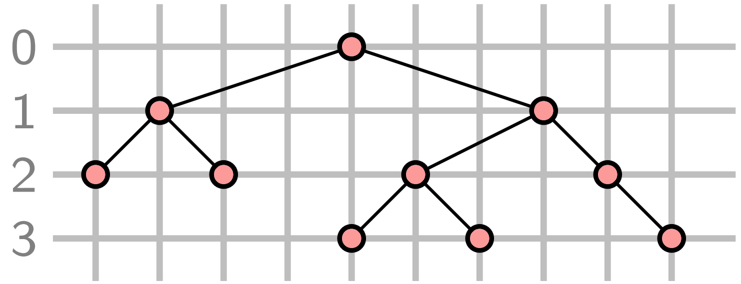


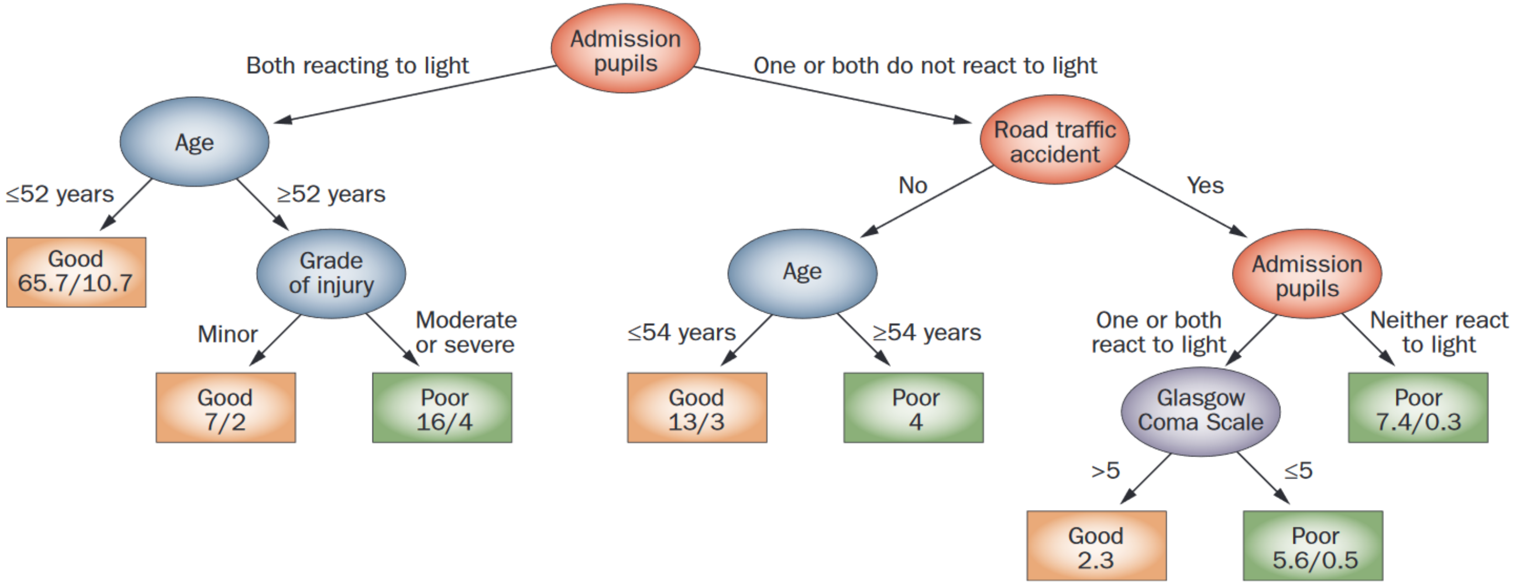
Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children
- Edges are straight-line segments

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

Layered Drawings – Drawing Style



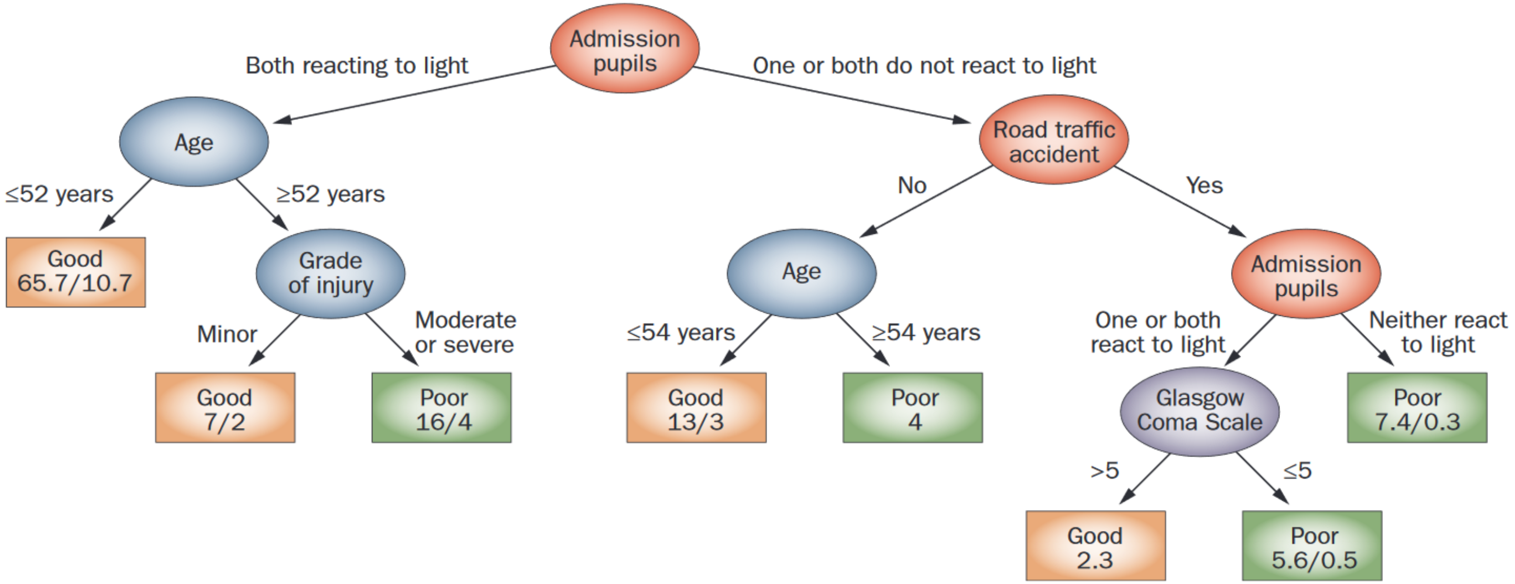


- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

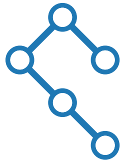
Drawing conventions

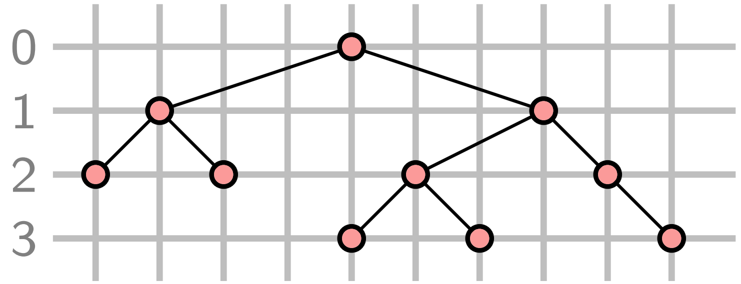
- Vertices lie on layers and have integer coordinates
- Parent centered above children
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

Layered Drawings – Drawing Style



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

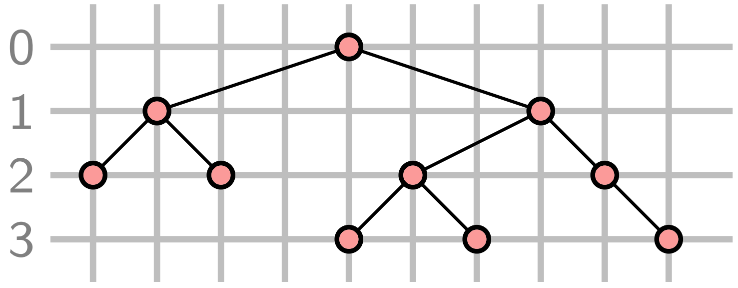


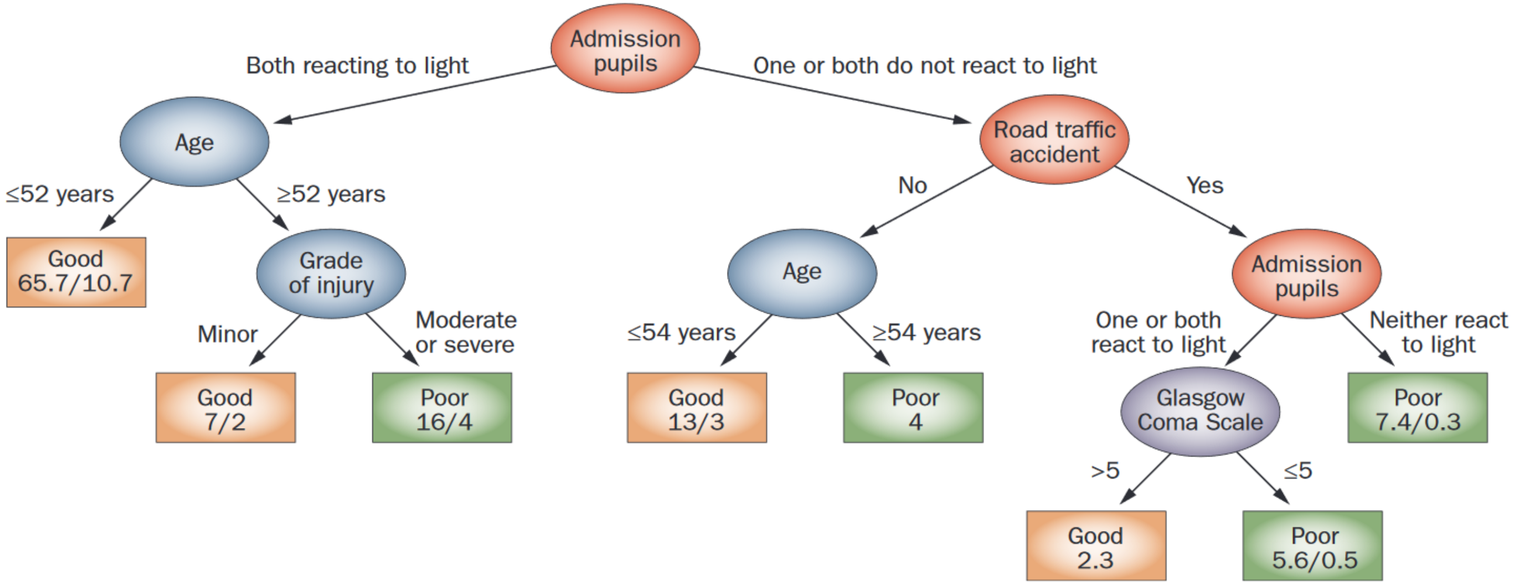


Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

Layered Drawings – Drawing Style

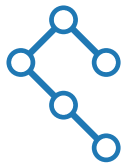


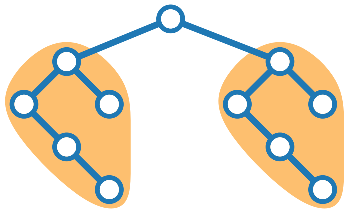


Drawing conventions

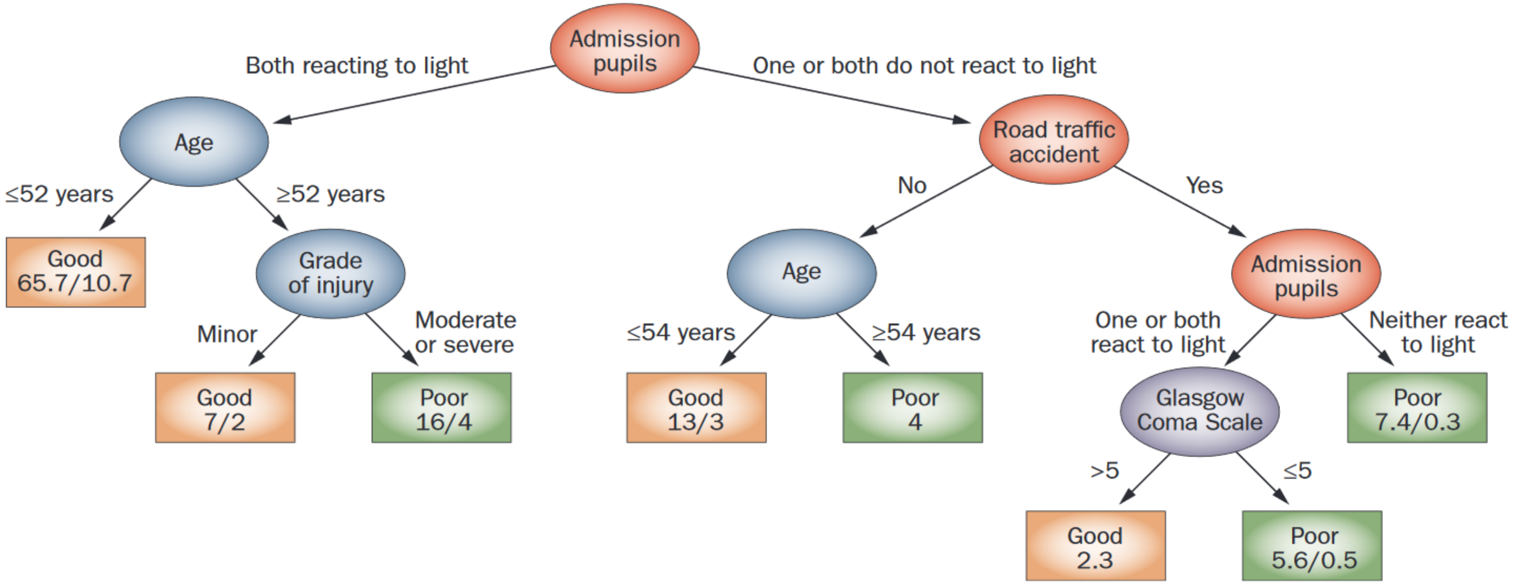
- Vertices lie on layers and have integer coordinates
- Parent centered above children
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

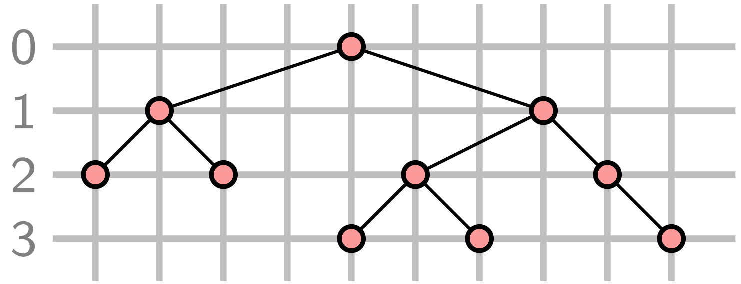
- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?





Layered Drawings – Drawing Style



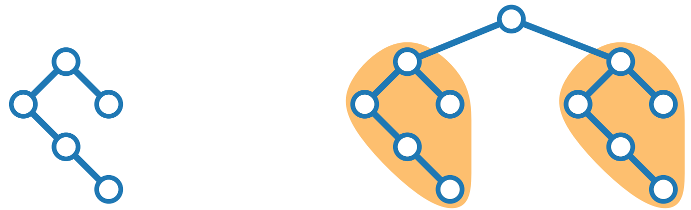


- What are properties of the layout?
- What are the drawing conventions?
- What are aesthetics to optimize?

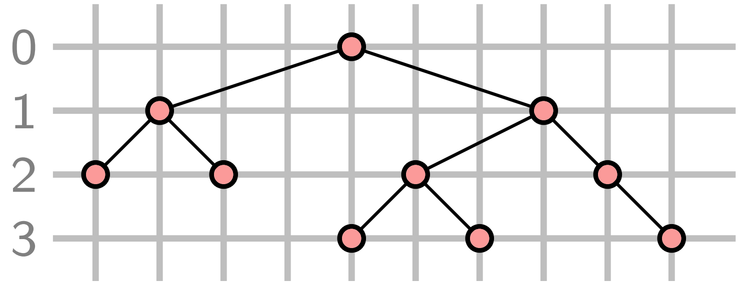
Drawing conventions

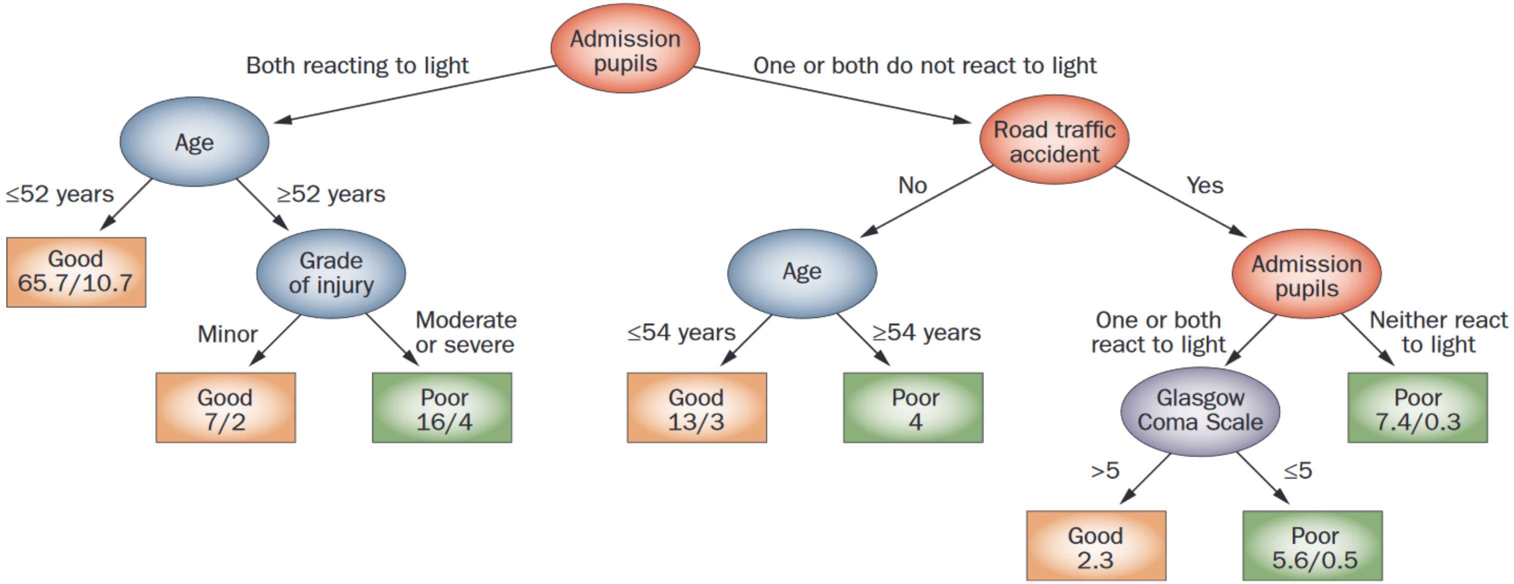
- Vertices lie on layers and have integer coordinates
- Parent centered above children
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

Drawing aesthetics

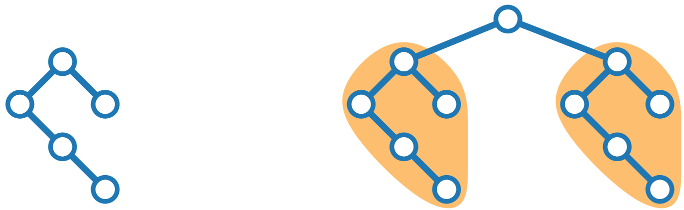


Layered Drawings – Drawing Style





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



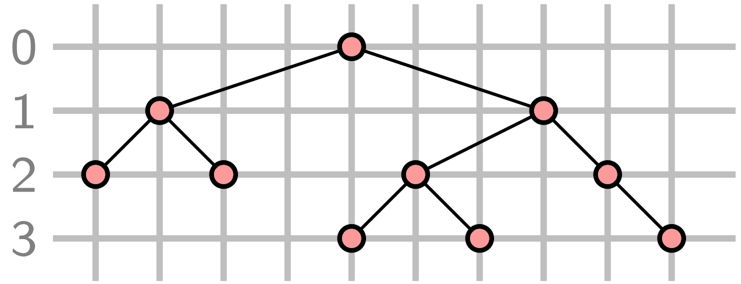
Drawing conventions

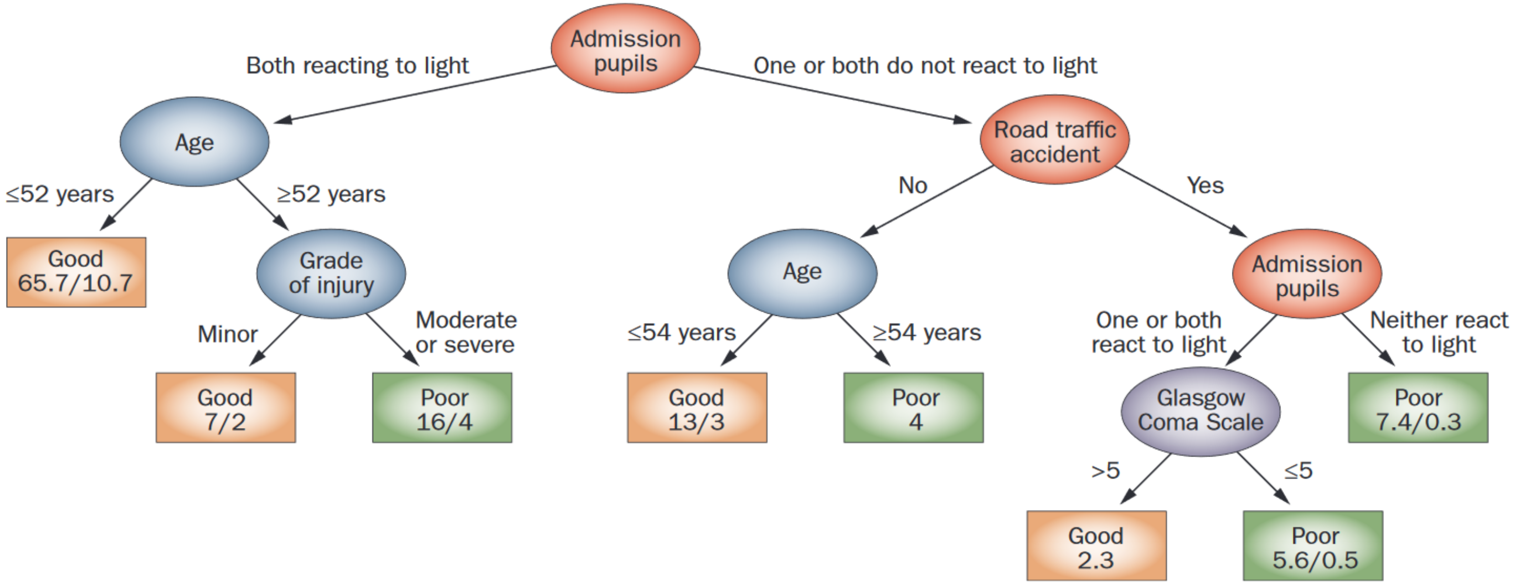
- Vertices lie on layers and have integer coordinates
- Parent centered above children
- Edges are straight-line segments
- Isomorphic subtrees have identical drawings

Drawing aesthetics

- Area

Layered Drawings – Drawing Style

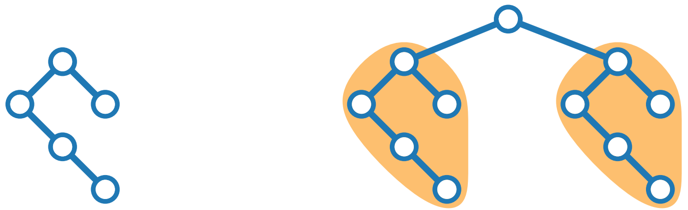




Drawing conventions

- Vertices lie on layers and have integer coordinates
- Parent centered above children
- 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?

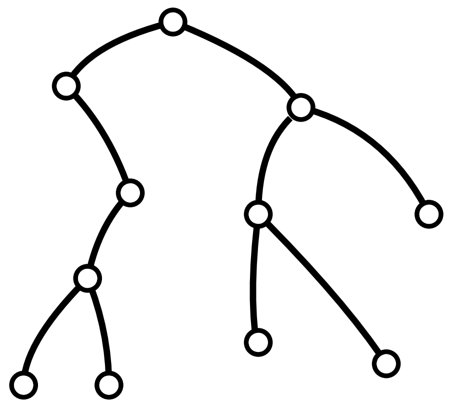


Drawing aesthetics

- Area
- Symmetries

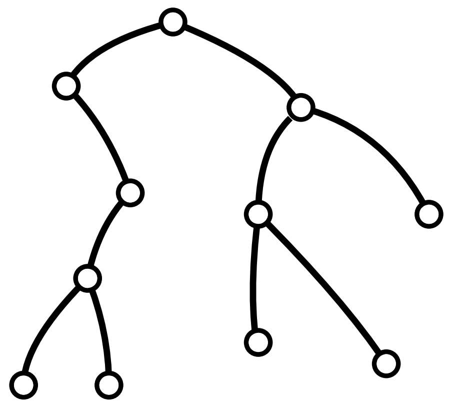
Layered Drawings – Algorithm

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

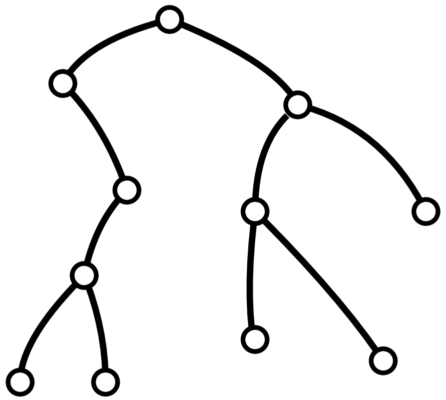


Base case:
Divide:

Conquer:

Layered Drawings – Algorithm

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



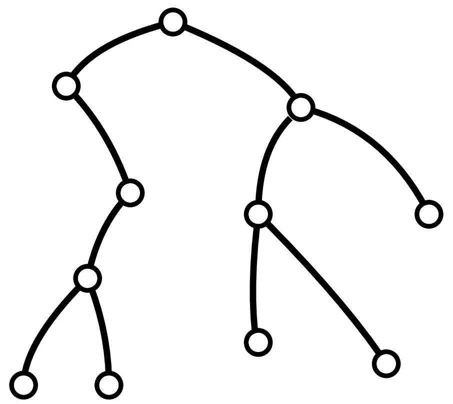
Base case: A single vertex

Divide:

Conquer:

Layered Drawings – Algorithm

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

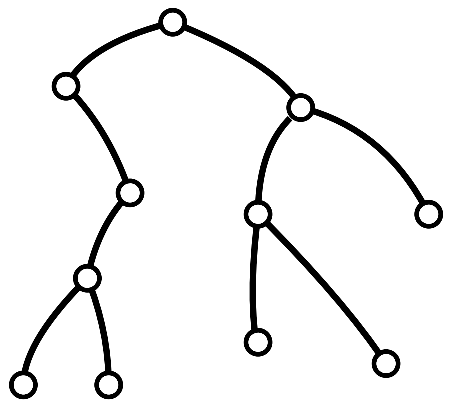


Base case: A single vertex ○
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:

Layered Drawings – Algorithm

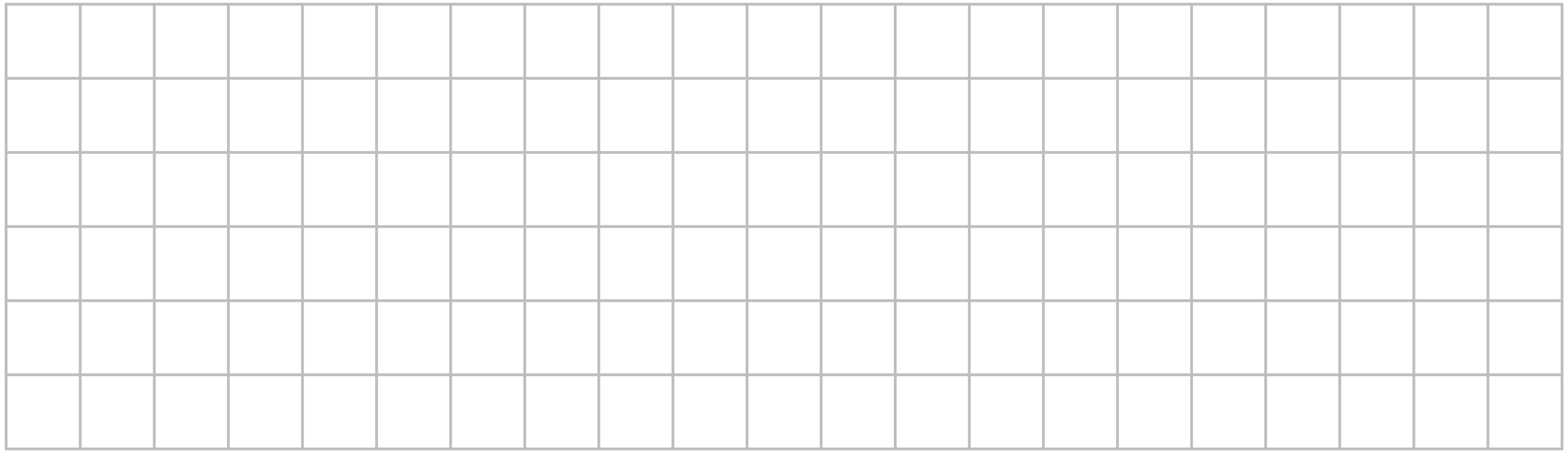
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

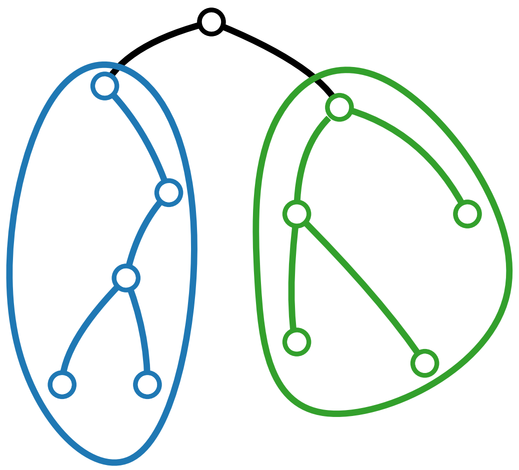
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



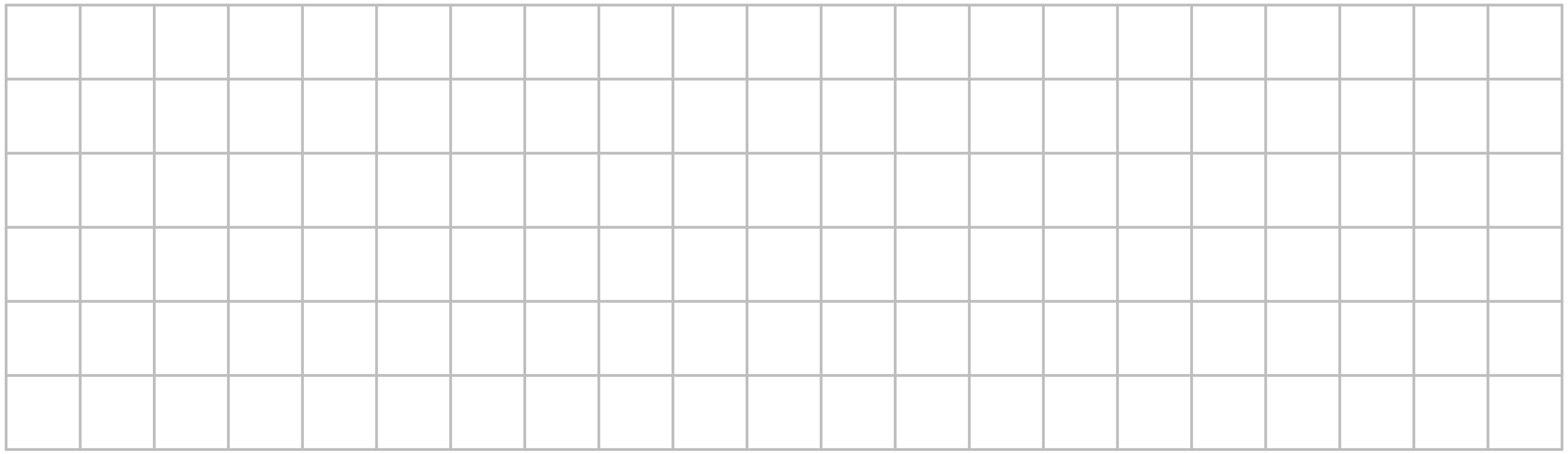
Layered Drawings – Algorithm

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



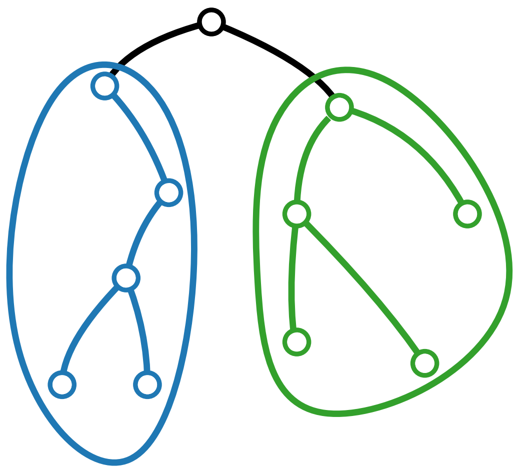
Base case: A single vertex ○
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

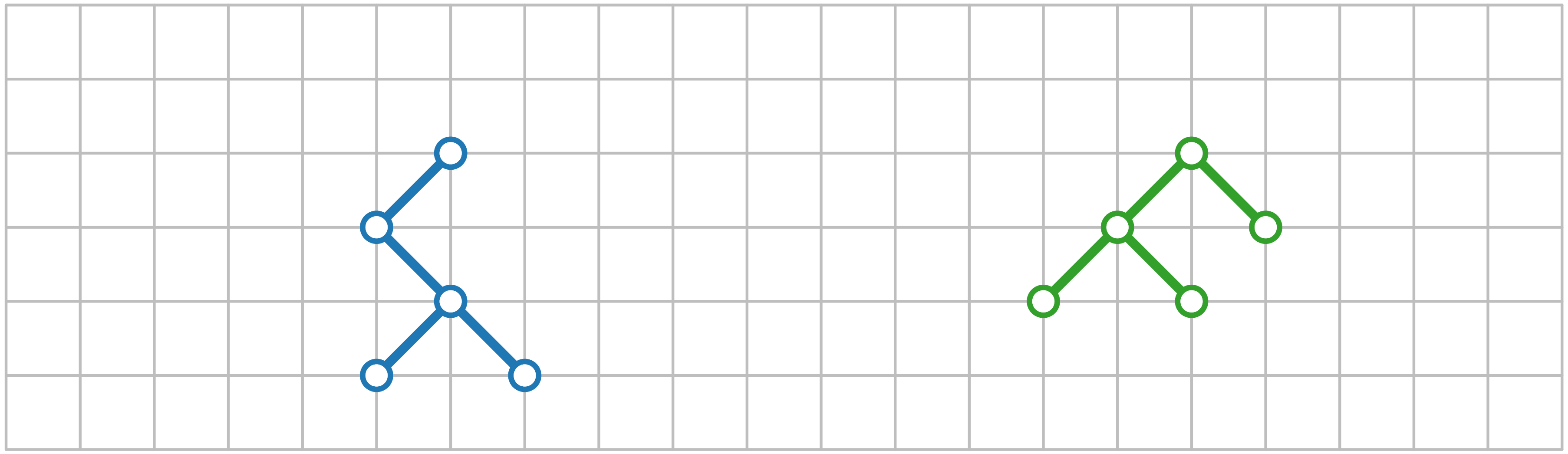
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

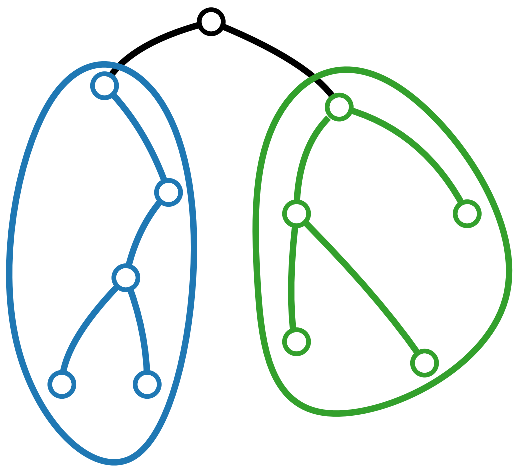
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

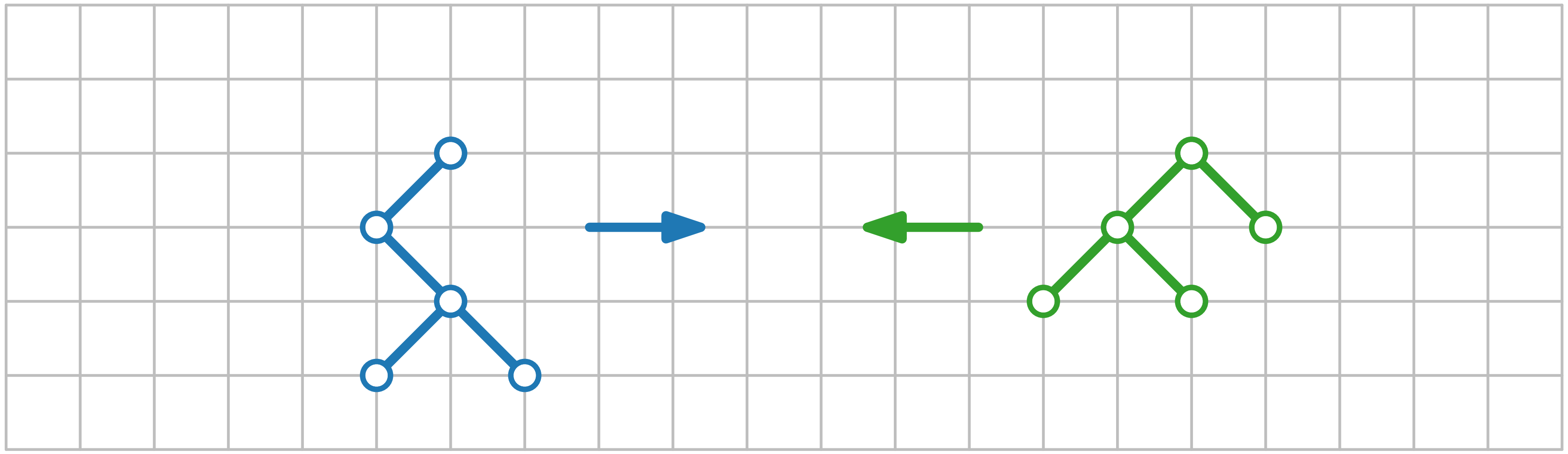
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

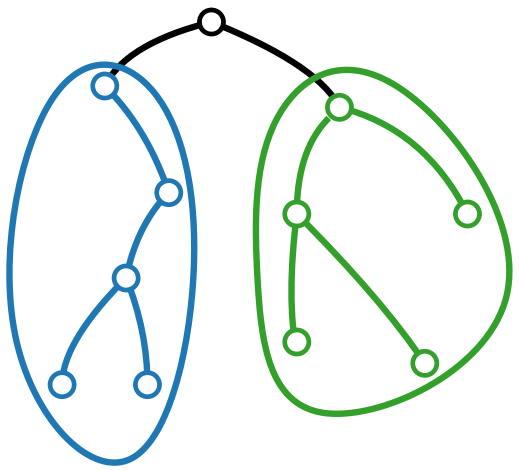
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

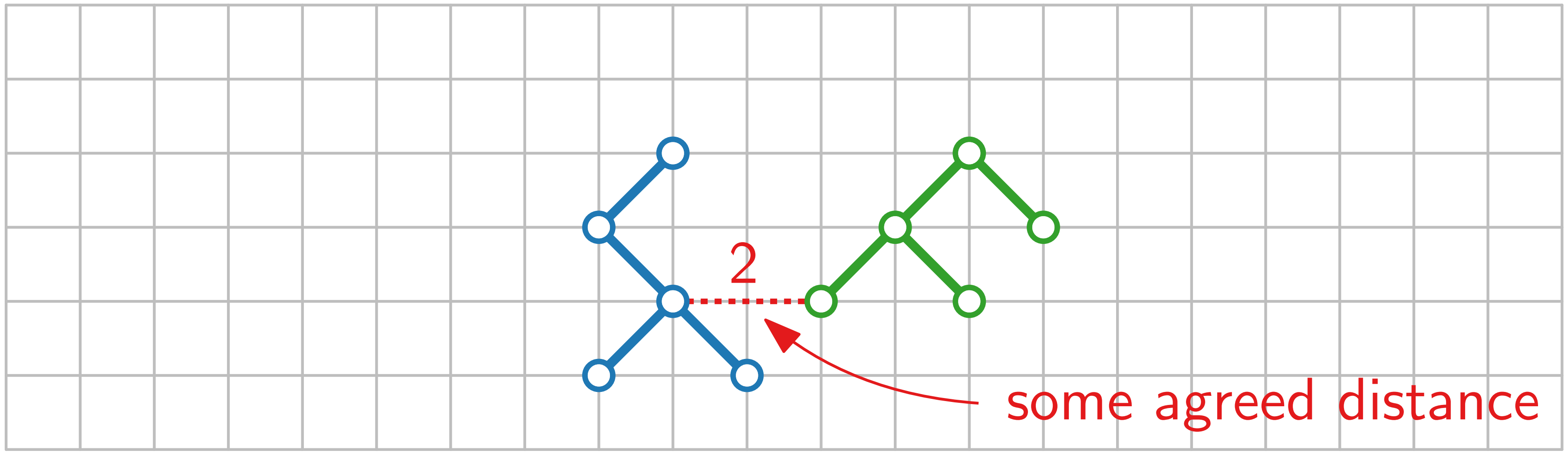
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

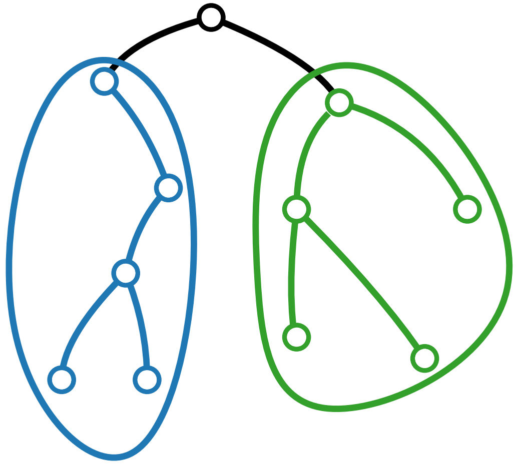
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

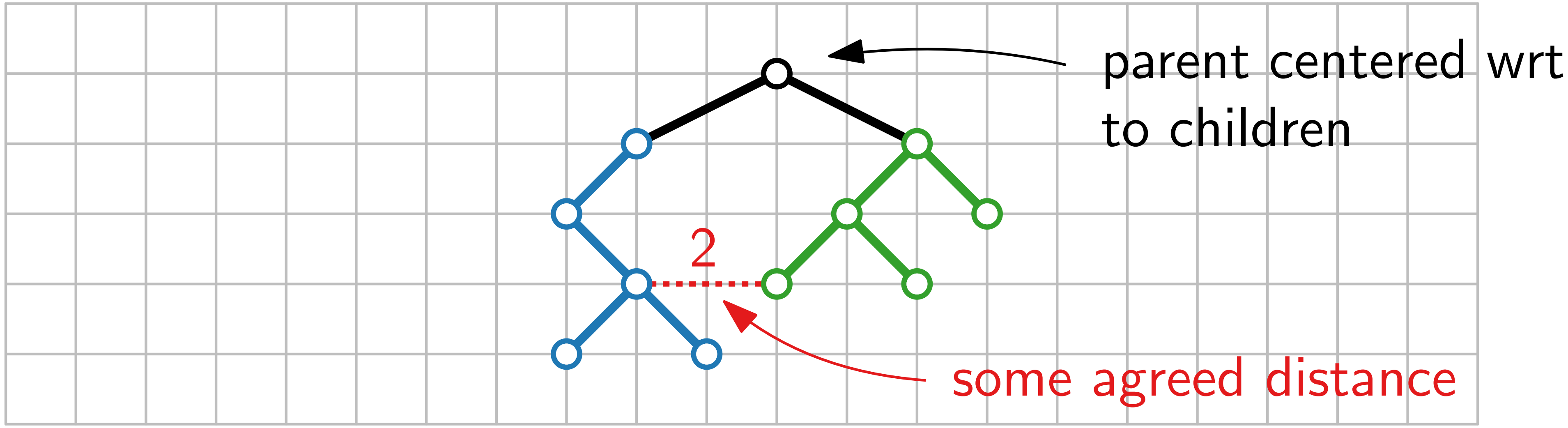
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

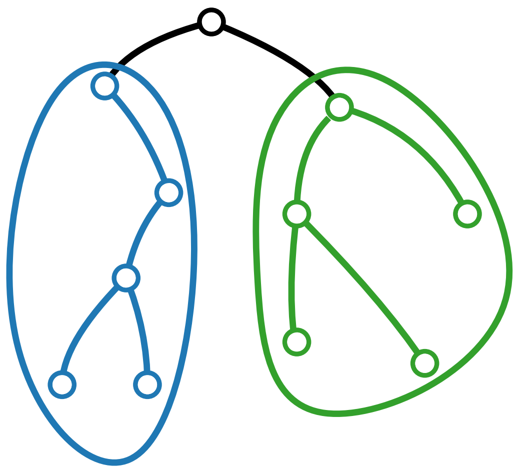
Divide: Recursively apply the algorithm to draw the left and right subtrees

Conquer:



Layered Drawings – Algorithm

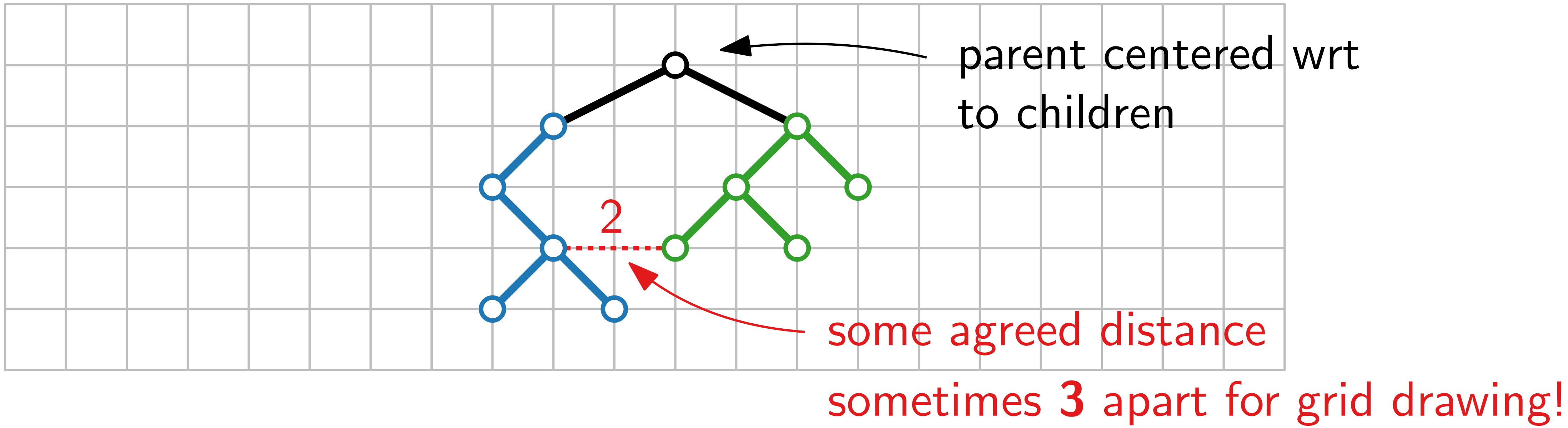
Input: A binary tree T
Output: A layered drawing of T



Base case: A single vertex

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

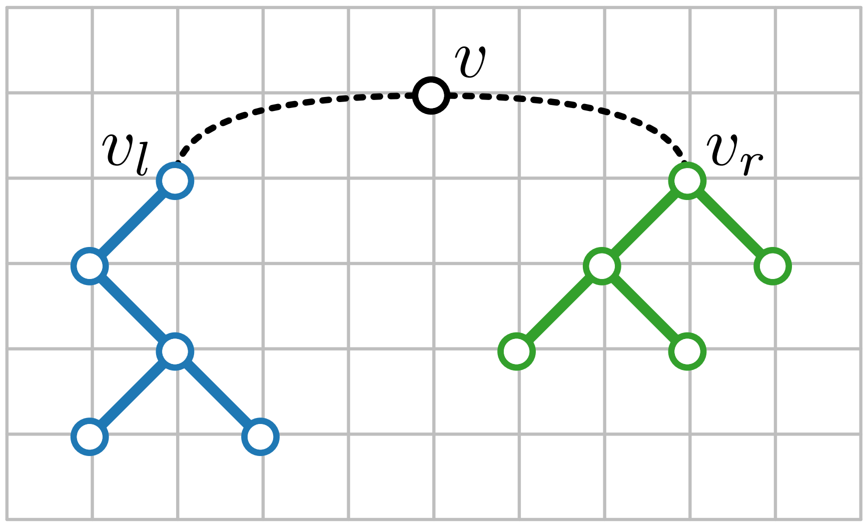
Conquer:



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

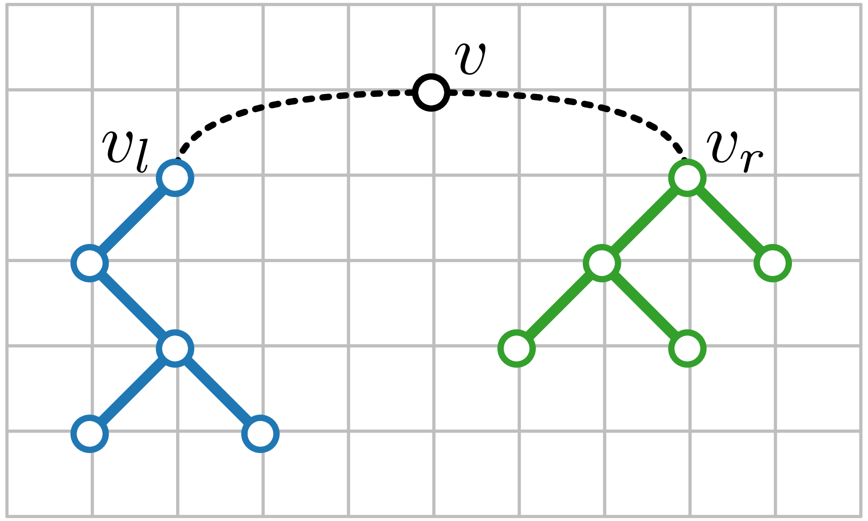
- For each vertex compute horizontal displacement of left and right child



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

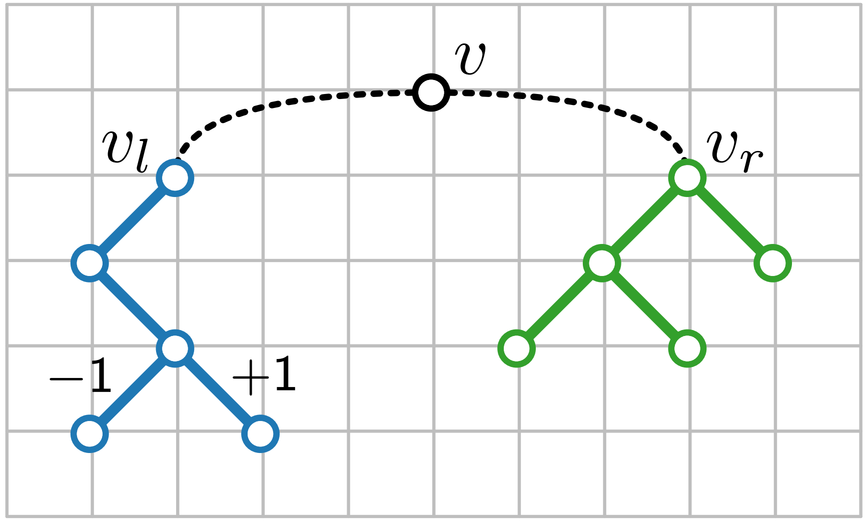
- For each vertex compute horizontal displacement of left and right child



Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child

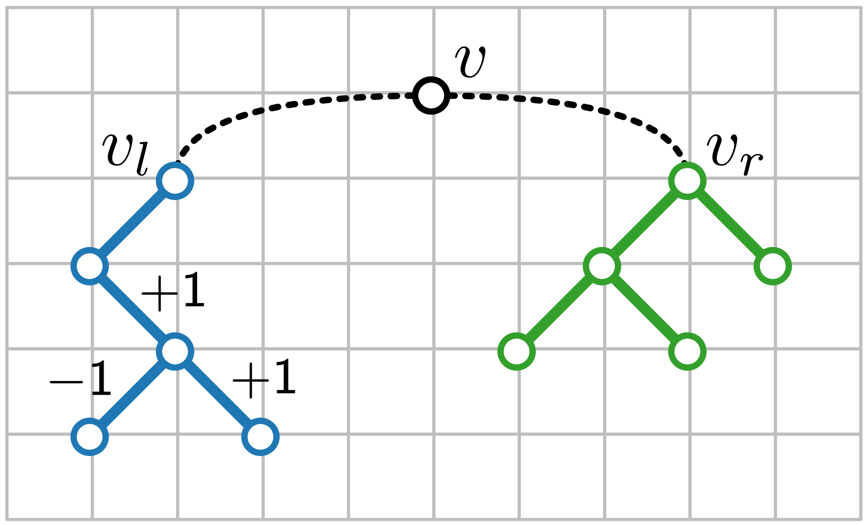
Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

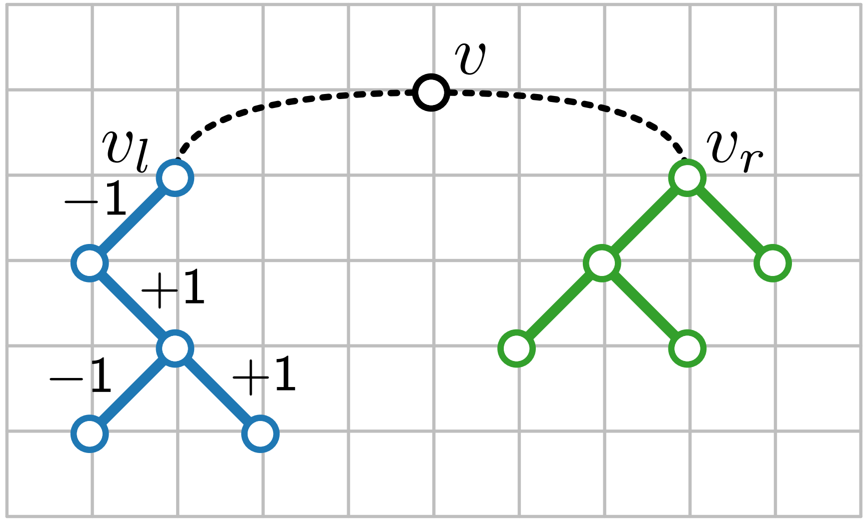
- For each vertex compute horizontal displacement of left and right child



Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details



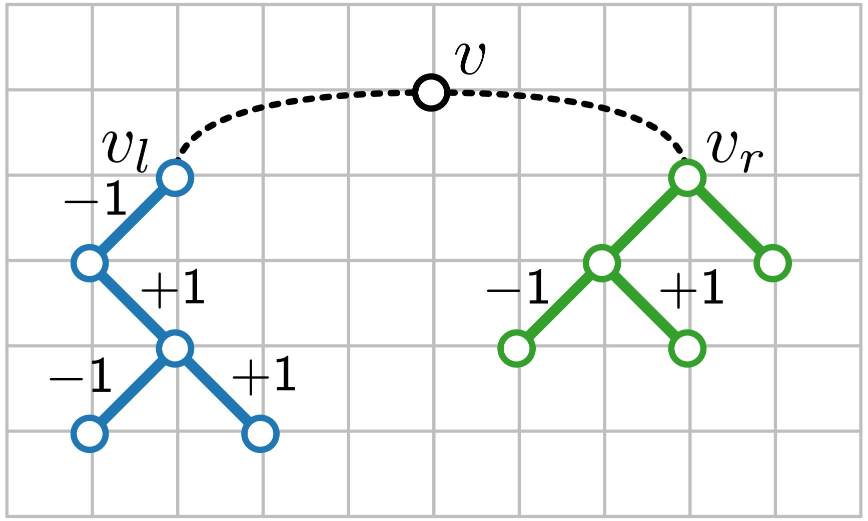
Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child

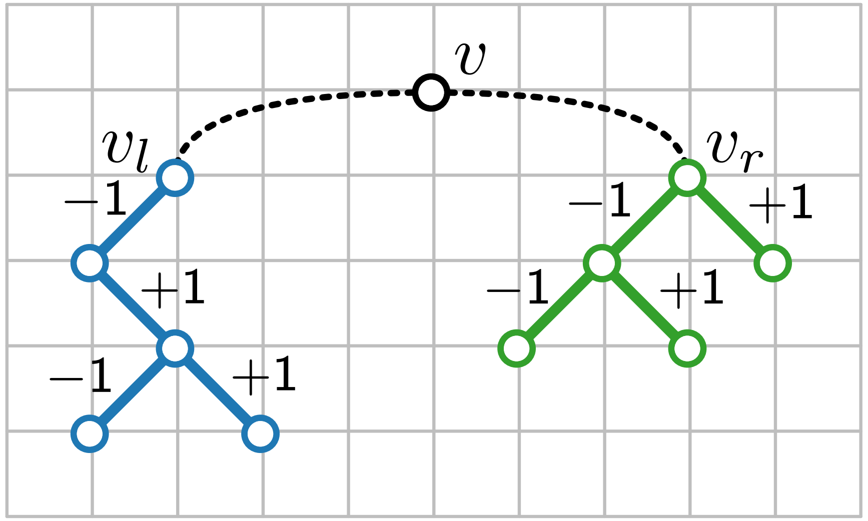
Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child



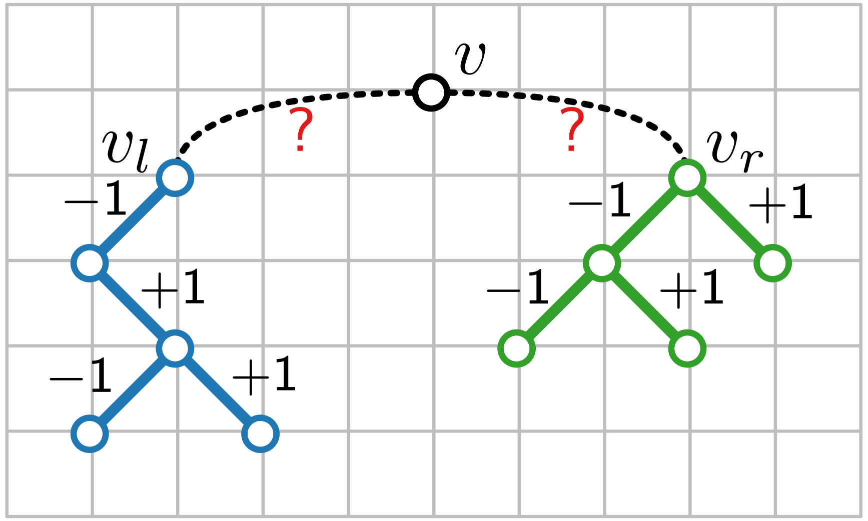
Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

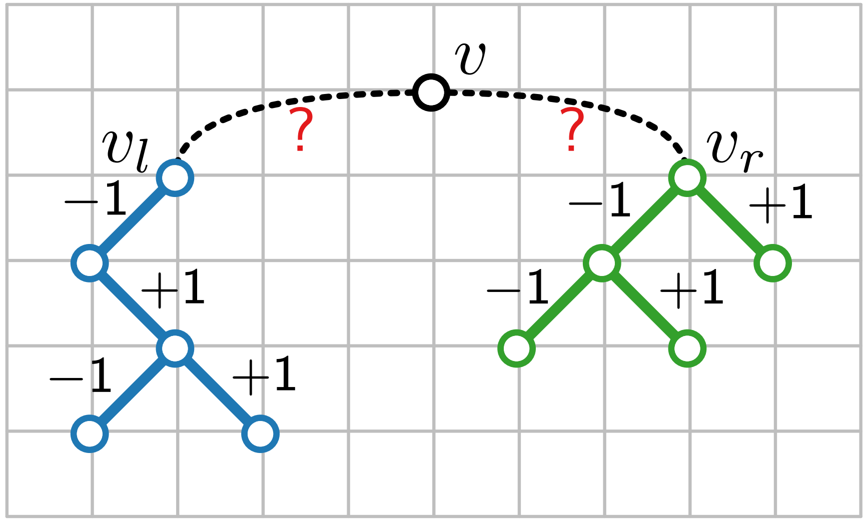
- For each vertex compute horizontal displacement of left and right child



Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is linked list of vertex coordinates/offsets

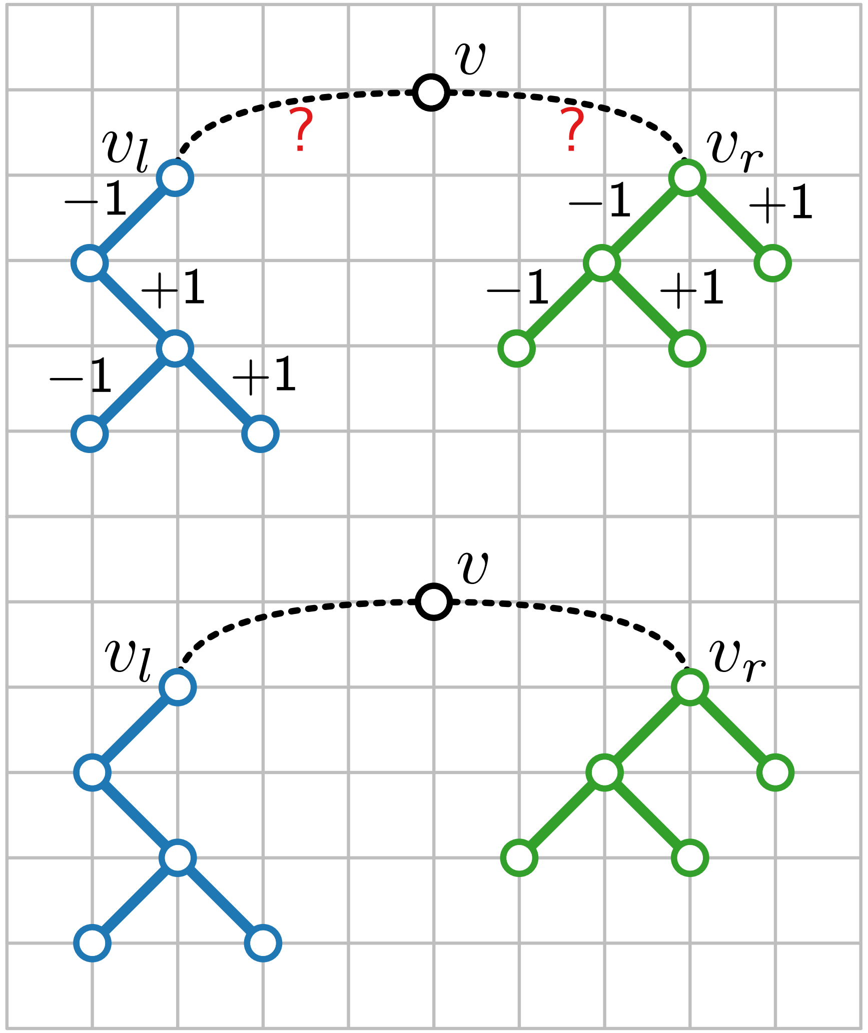
Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is linked list of vertex coordinates/offsets



Phase 2 – preorder traversal:

- Compute x- and y-coordinates

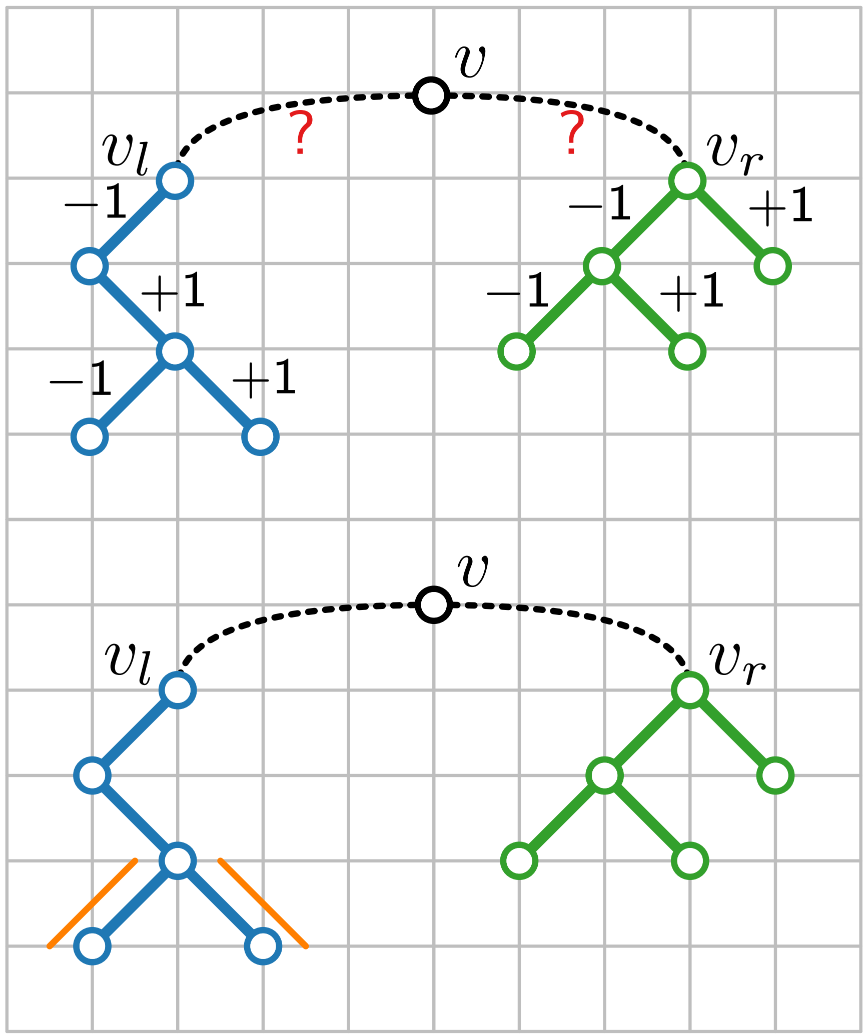
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is linked list of vertex coordinates/offsets

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



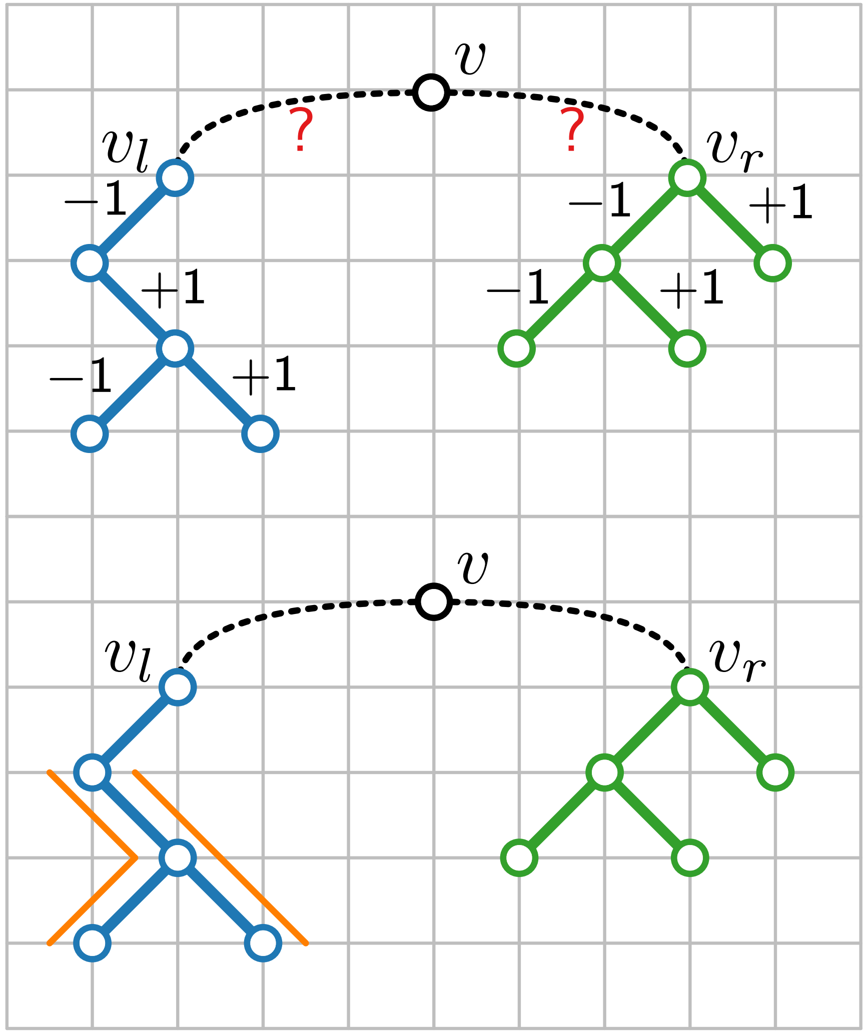
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

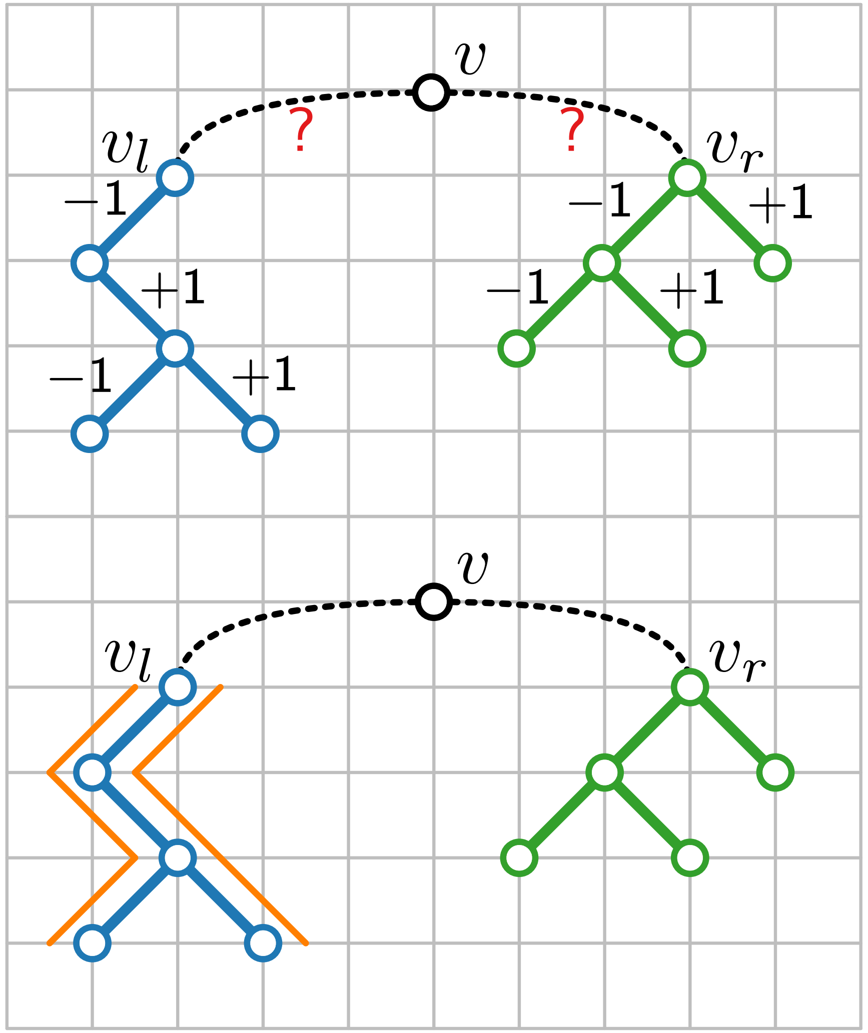
- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is linked list of vertex coordinates/offsets

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is linked list of vertex coordinates/offsets

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

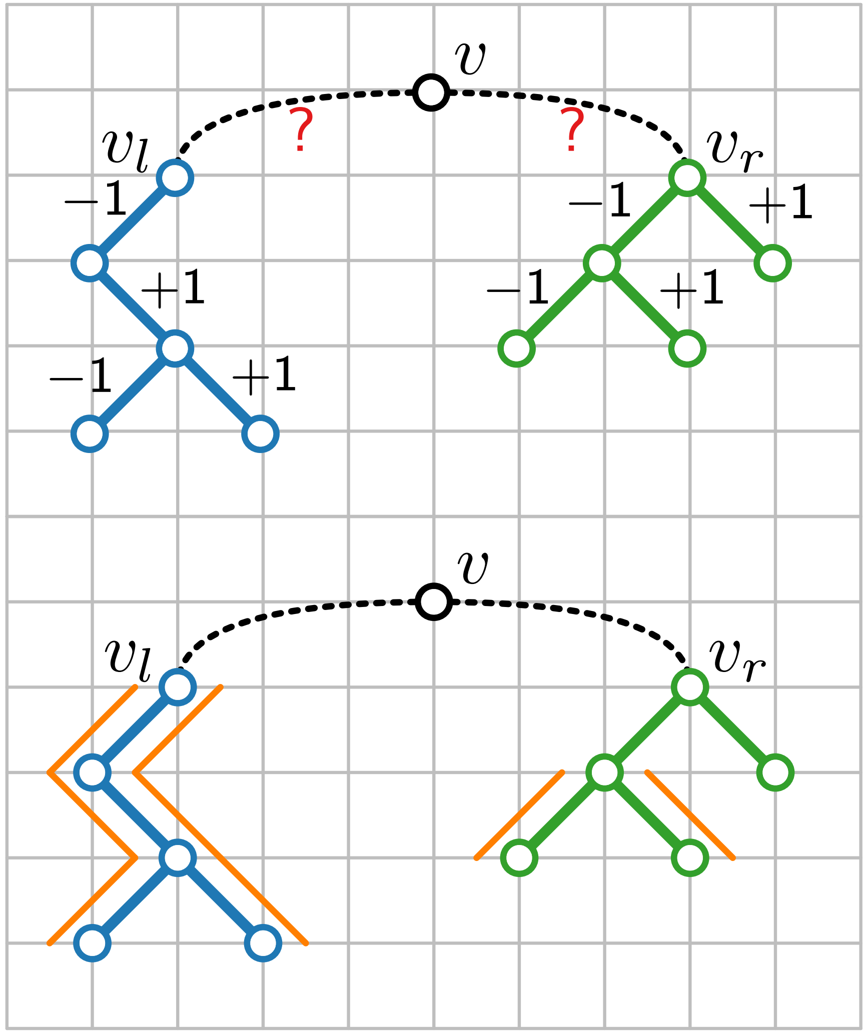
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is linked list of vertex coordinates/offsets

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



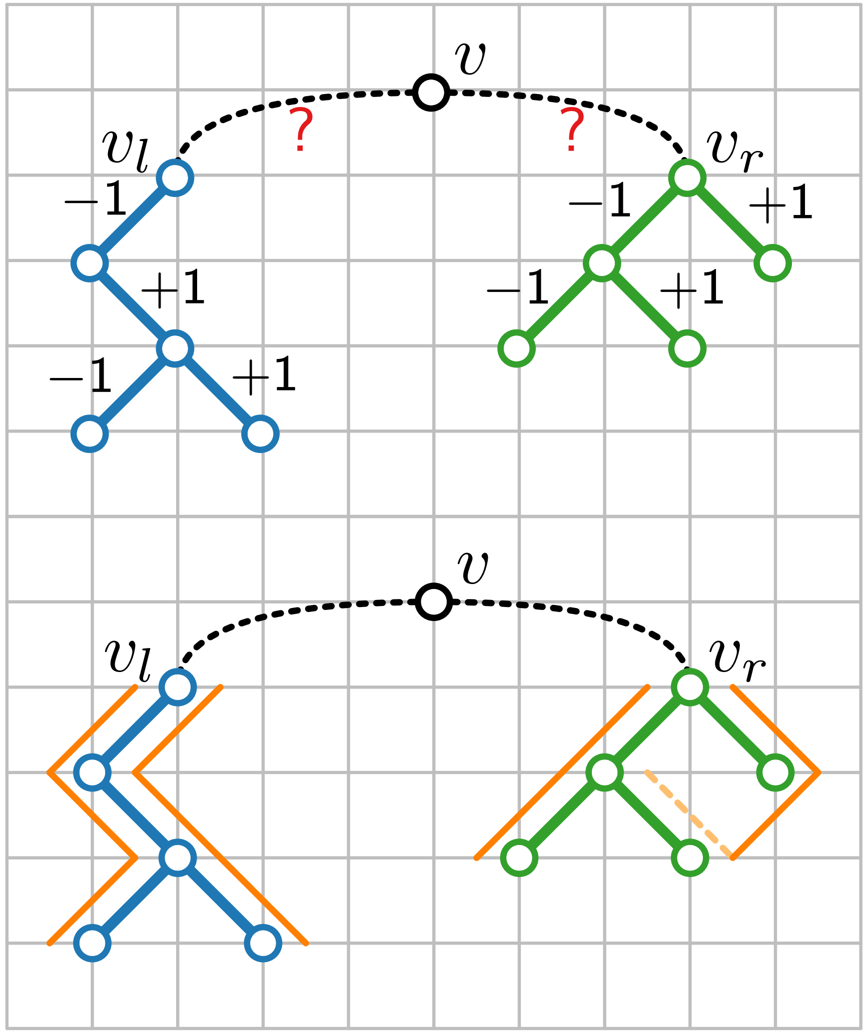
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is linked list of vertex coordinates/offsets

Phase 2 – preorder traversal:

- Compute x- and y-coordinates



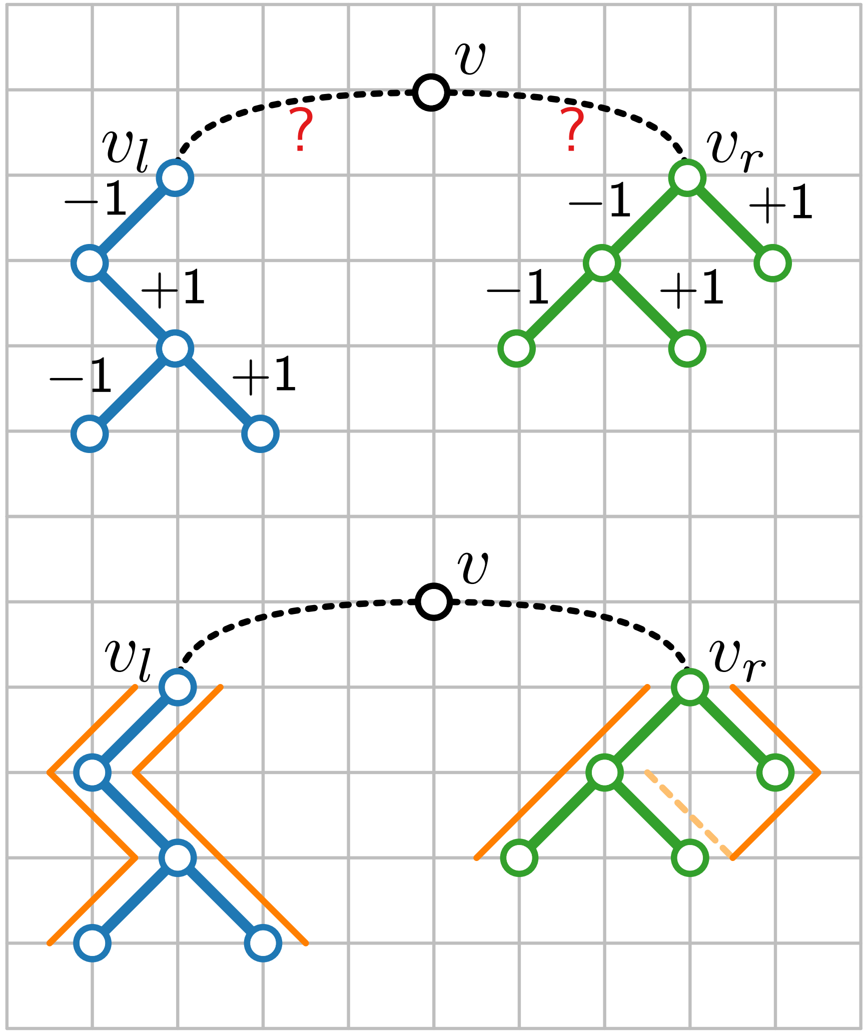
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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



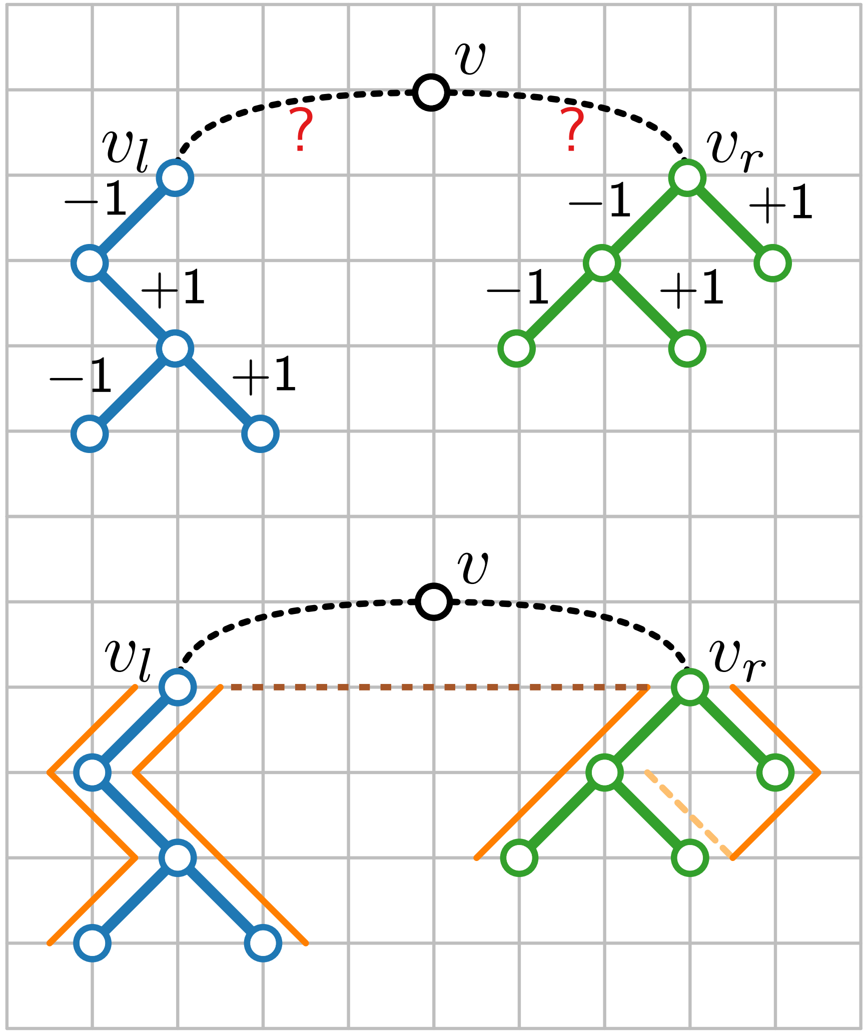
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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



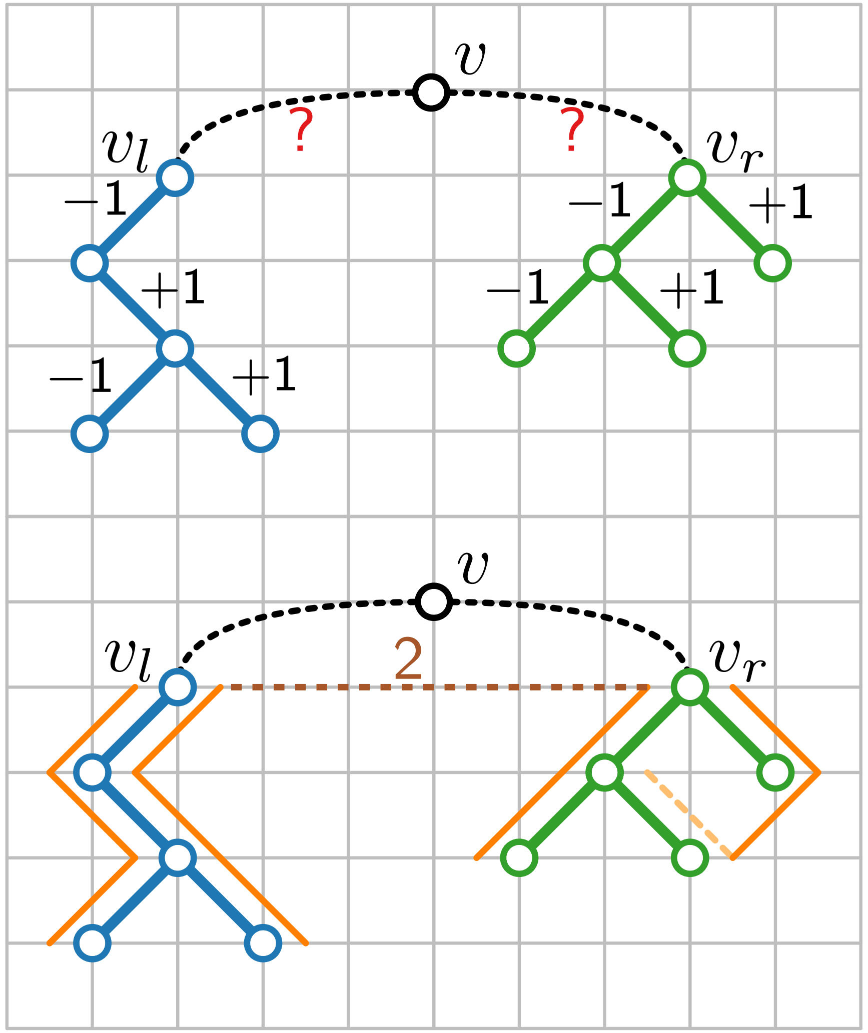
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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



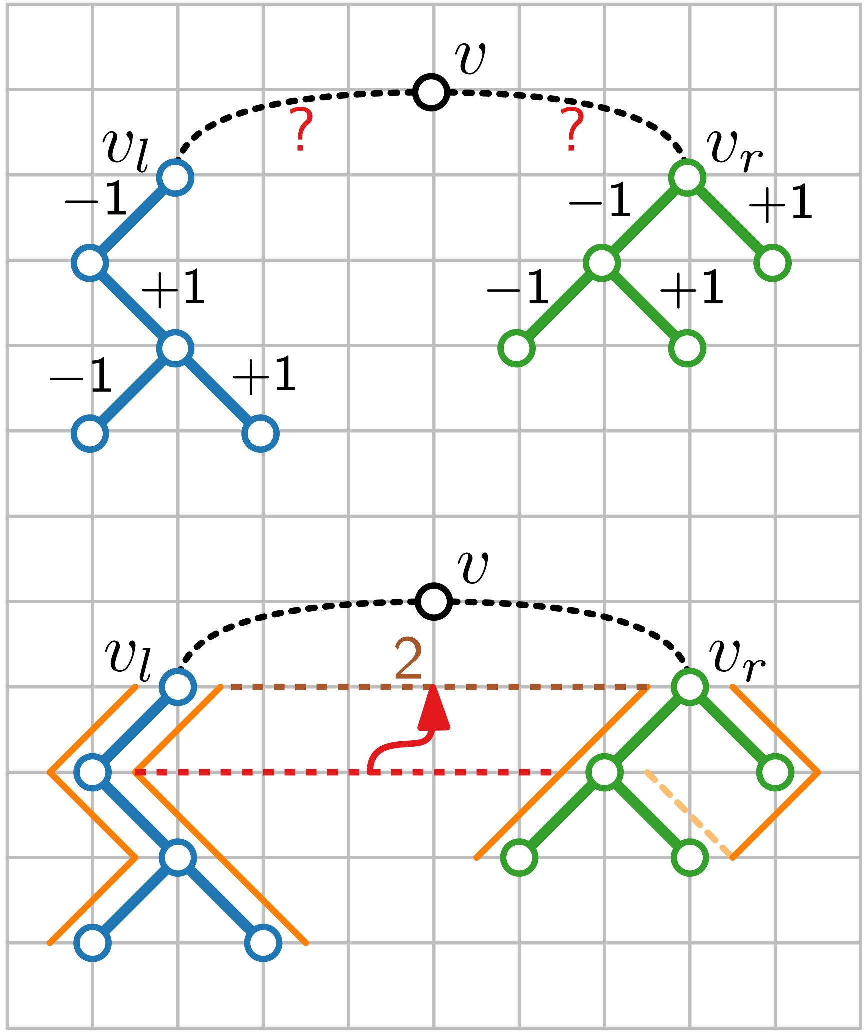
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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



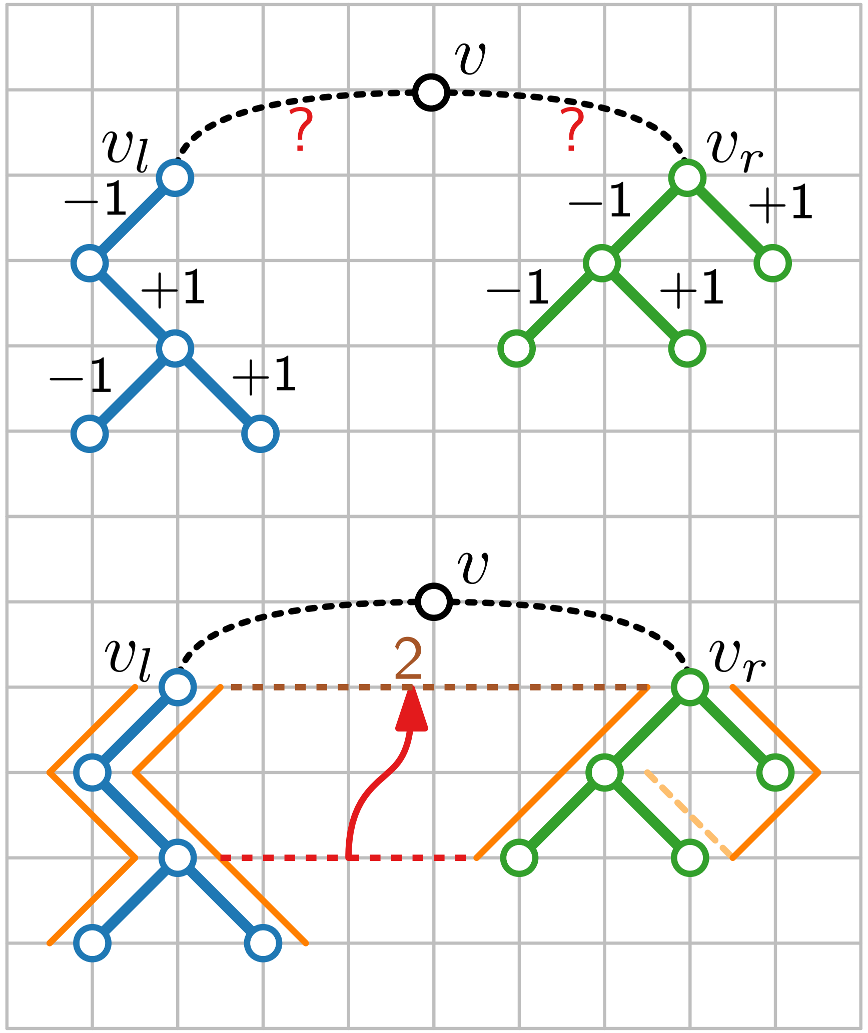
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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



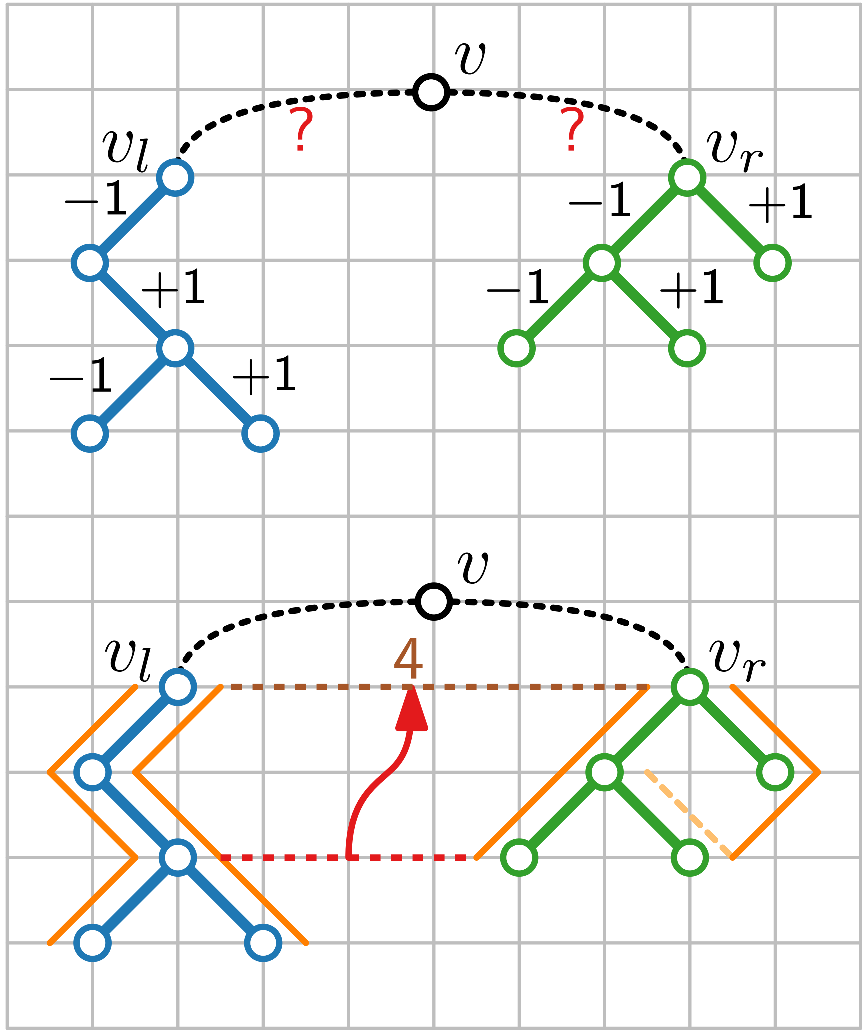
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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



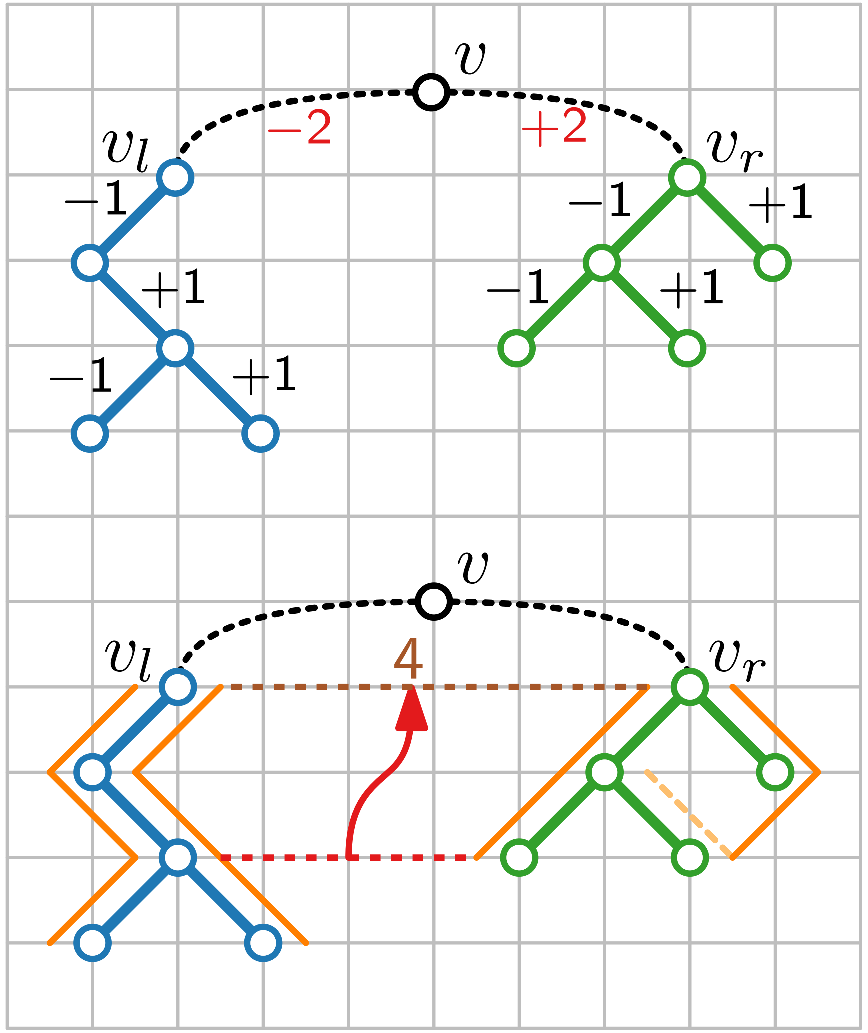
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of sub-tree $T(u)$
- Contour is 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



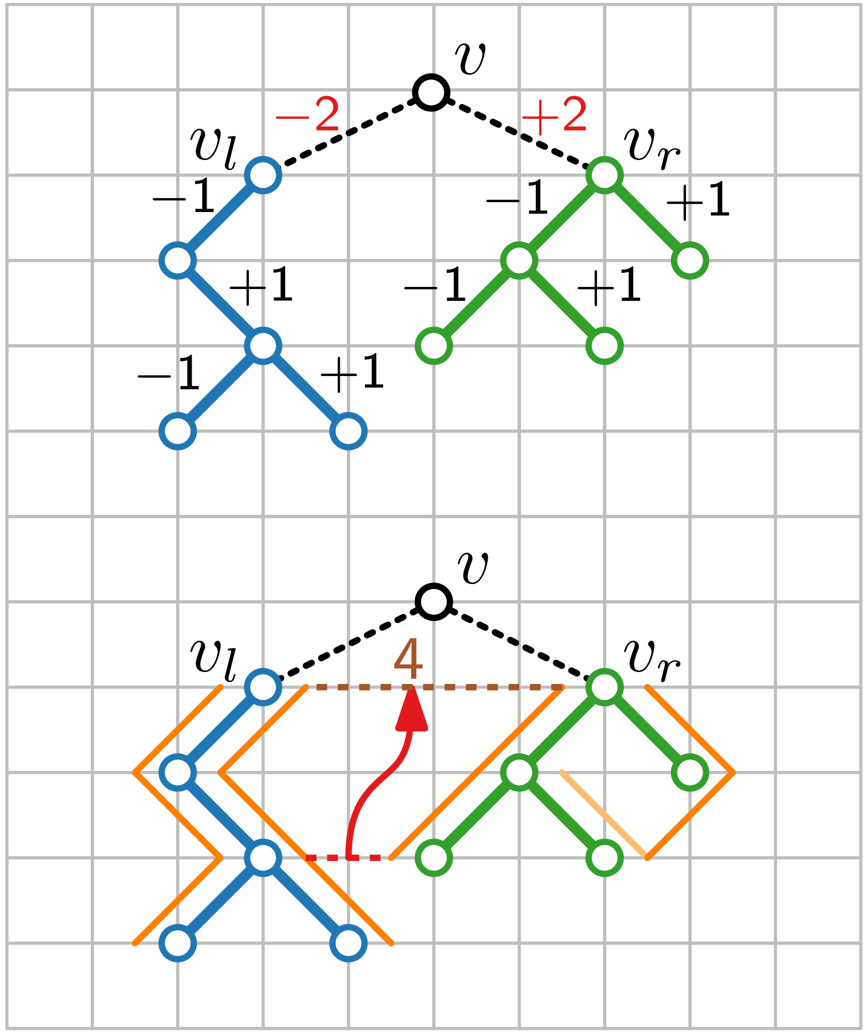
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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



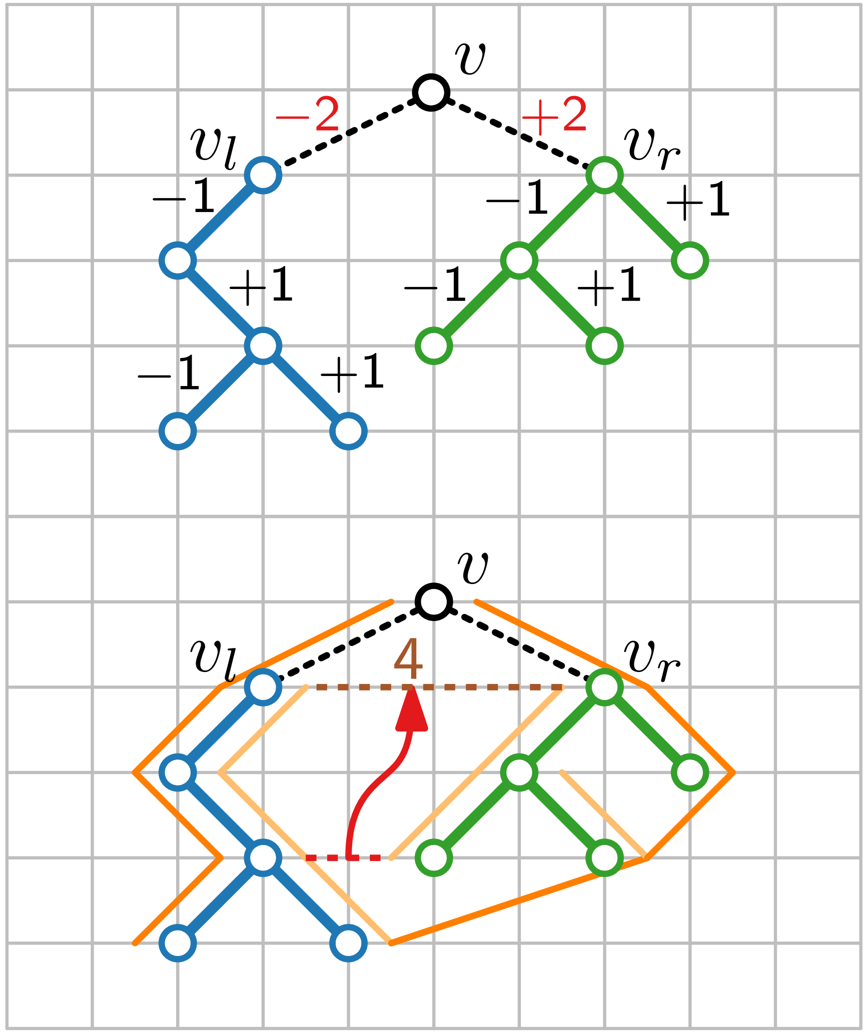
Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

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

Phase 2 – preorder traversal:

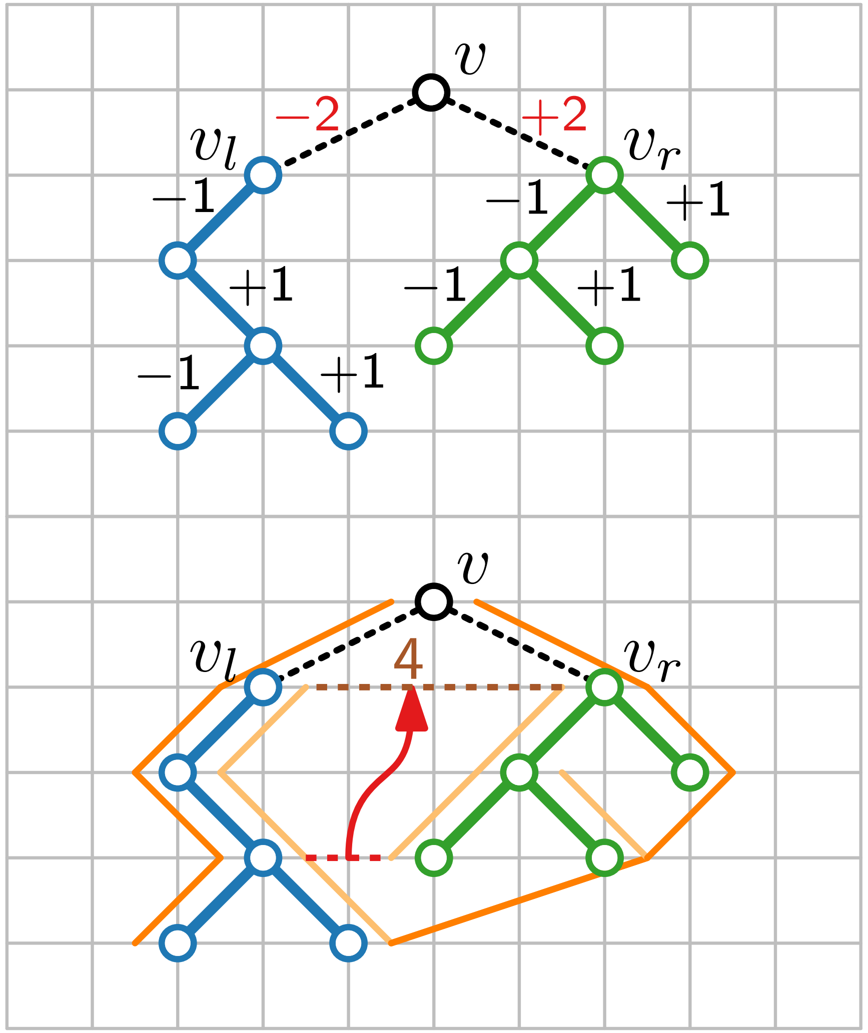
- Compute x- and y-coordinates



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of sub-tree $T(u)$
- Contour is 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:

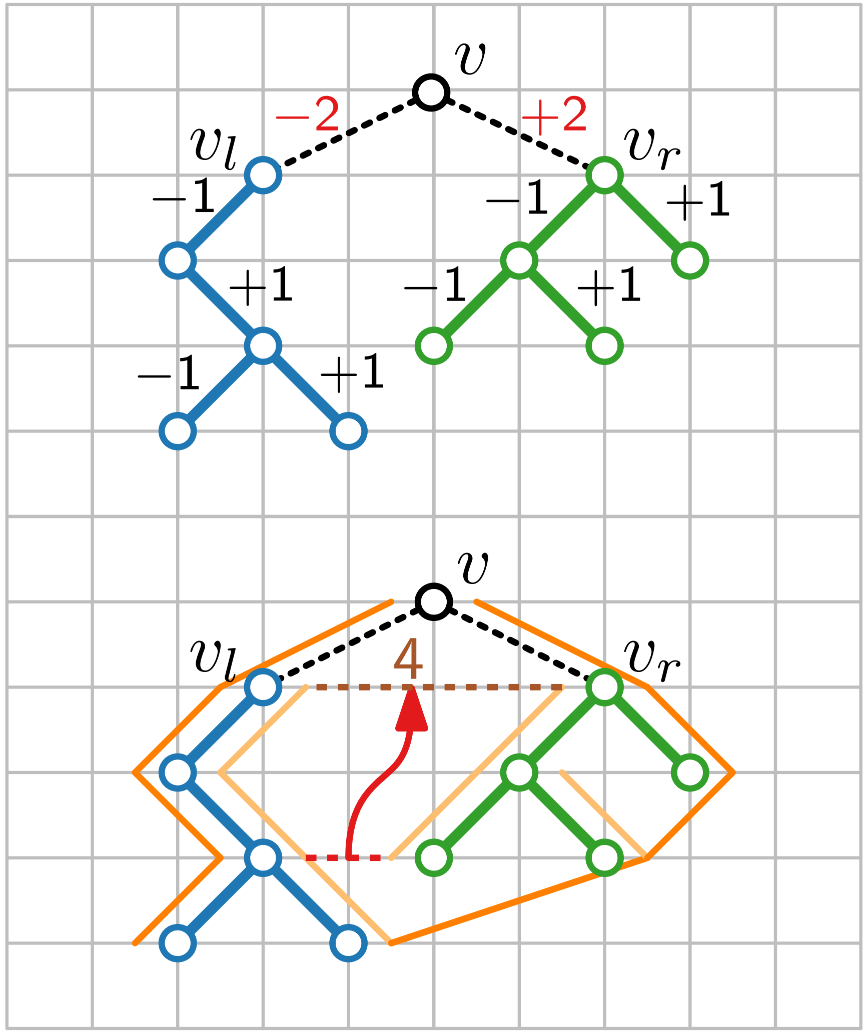
- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is linked list of vertex coordinates/offsets
- Find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

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

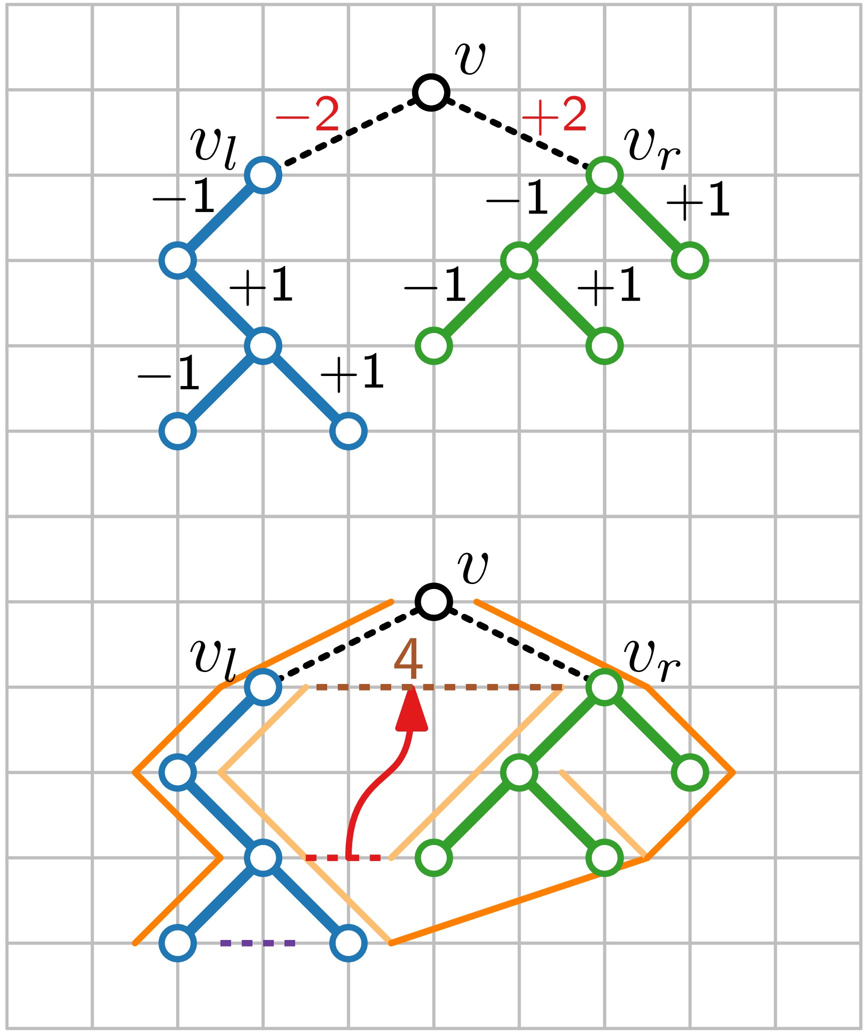
- How often do we have to walk along a contour?



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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

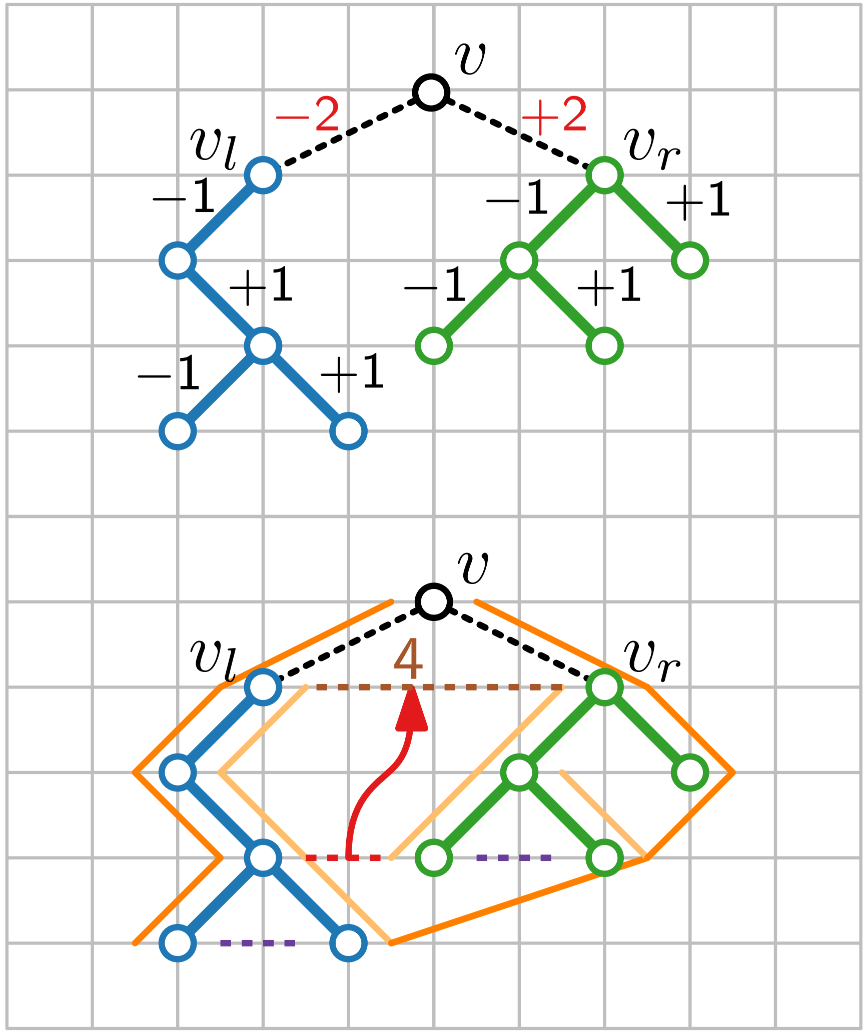
Runtime?

- How often do we have to walk along a contour?

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

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



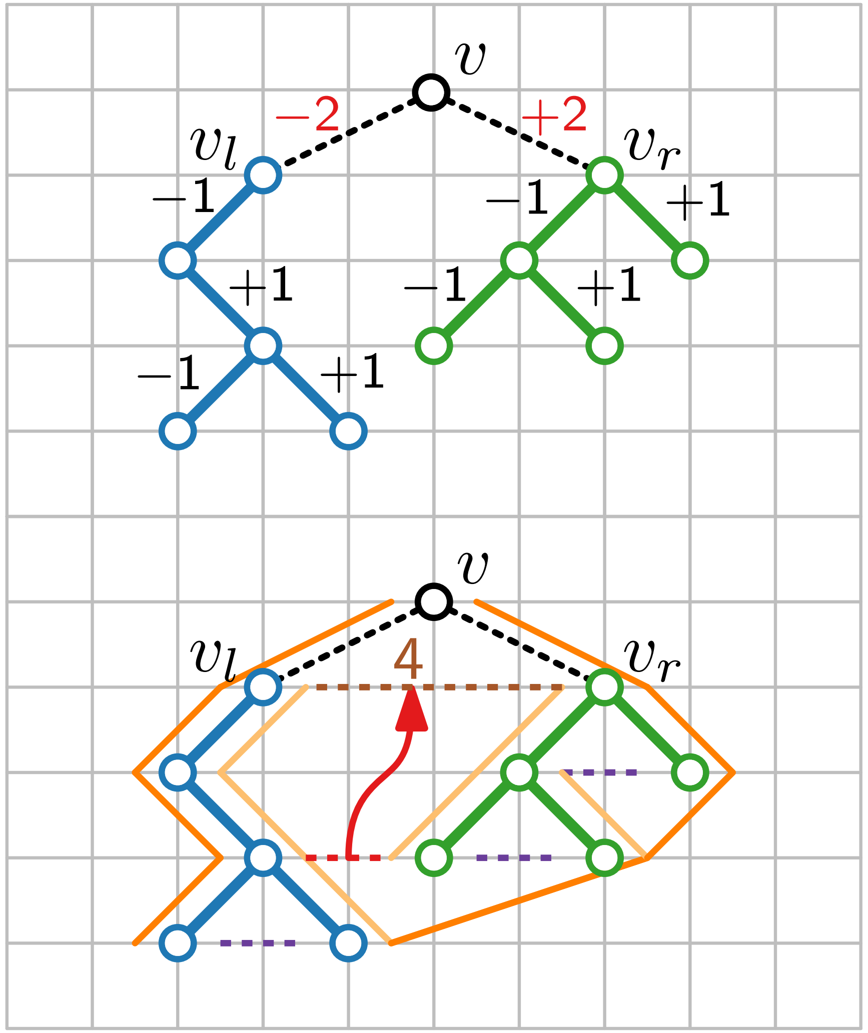
Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

- How often do we have to walk along a contour?

Layered Drawings – Algorithm Details



Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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 have to walk along a contour?

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

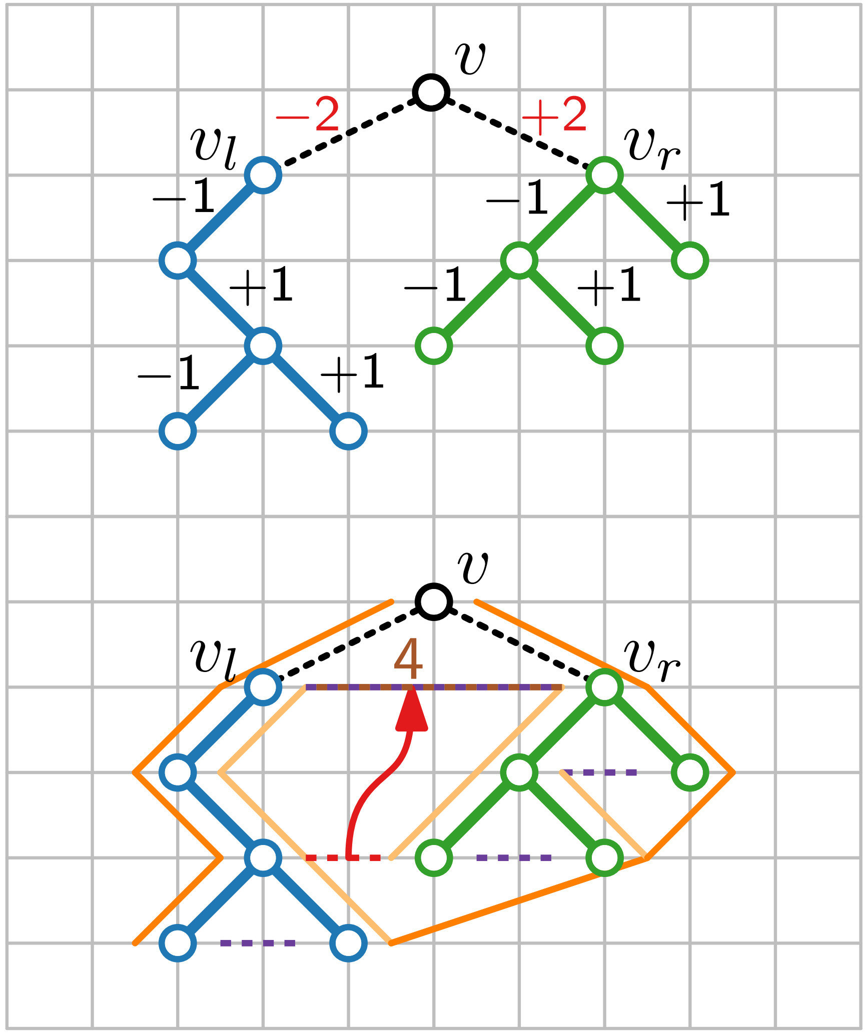
- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is linked list of vertex coordinates/offsets
- Find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

Phase 2 – preorder traversal:

- Compute x- and y-coordinates

Runtime?

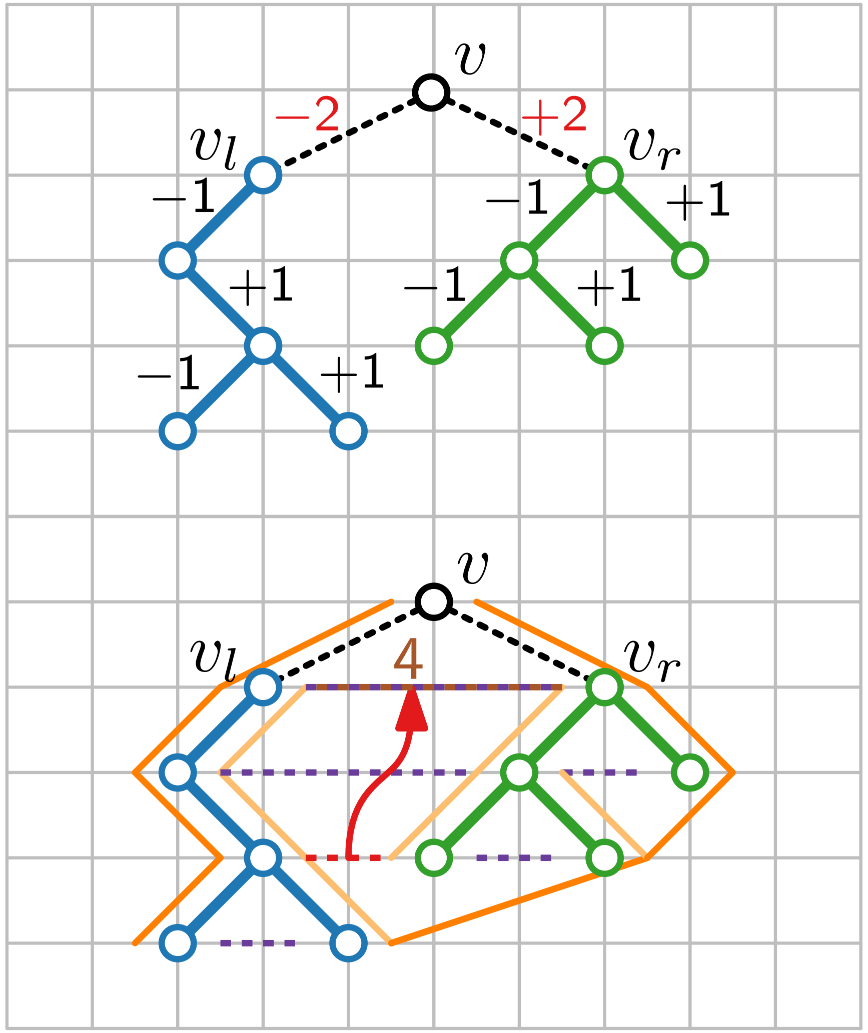
- How often do we have to walk along a contour?



Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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

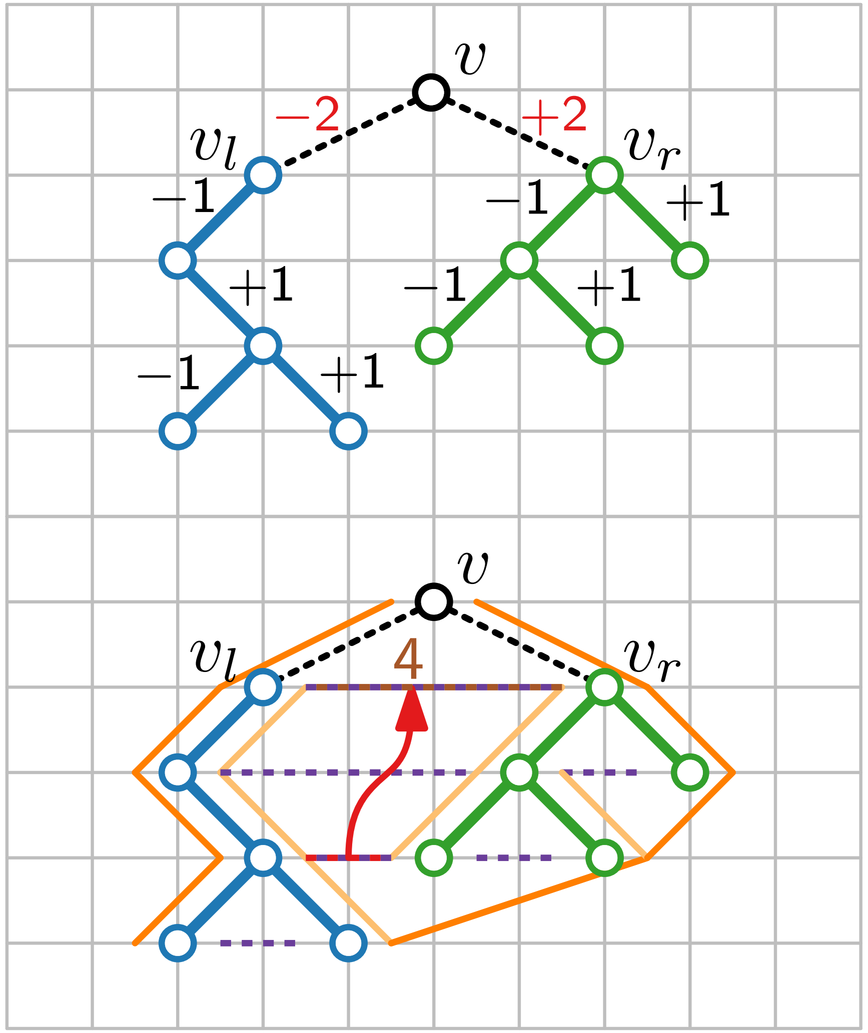
Runtime?

- How often do we have to walk along a contour?

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of subtree $T(u)$
- Contour is 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

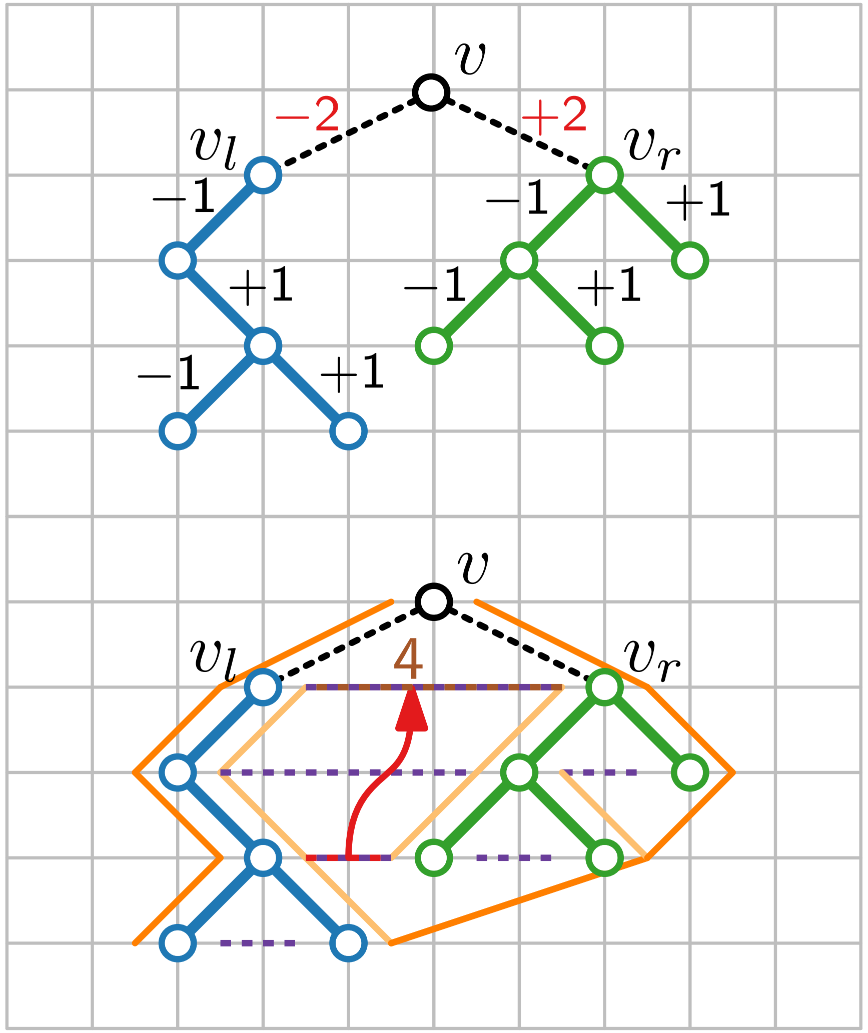
Runtime?

- How often do we have to walk along a contour?

Layered Drawings – Algorithm Details

Phase 1 – postorder traversal:

- For each vertex compute horizontal displacement of left and right child
- x-offset$(v_l) = -\lceil \frac{d_v}{2} \rceil$, x-offset$(v_r) = \lceil \frac{d_v}{2} \rceil$
- At vertex u (below v) store left and right contour of sub-tree $T(u)$
- Contour is 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 have to walk along a contour? – *Less than $n = \#$ vertices times!*

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 $-\mathrm{depth}(v)$
- 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 $-\text{depth}(v)$
- Horizontal and Vertical distances are at least 1
- Each vertex is centred 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 $-$depth(v)
- Horizontal and Vertical distances are at least 1
- Each vertex is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ -

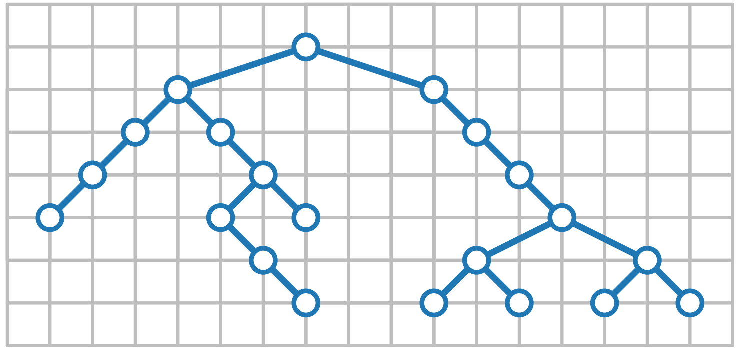
Layered Drawings – Result

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 $-$depth(v)
- Horizontal and Vertical distances are at least 1
- Each vertex is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

Layered Drawings – Result

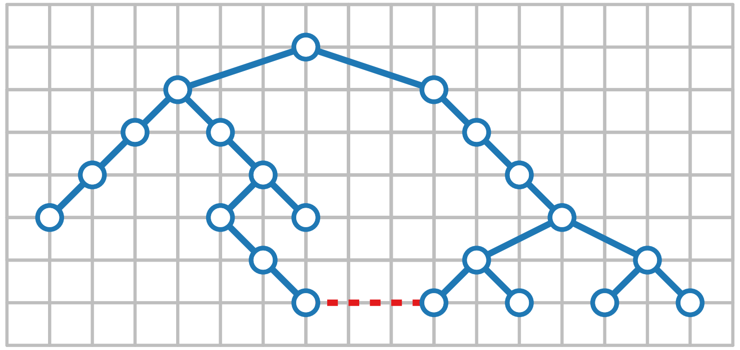


Theorem. [Reingold & Tilford '81]

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 is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

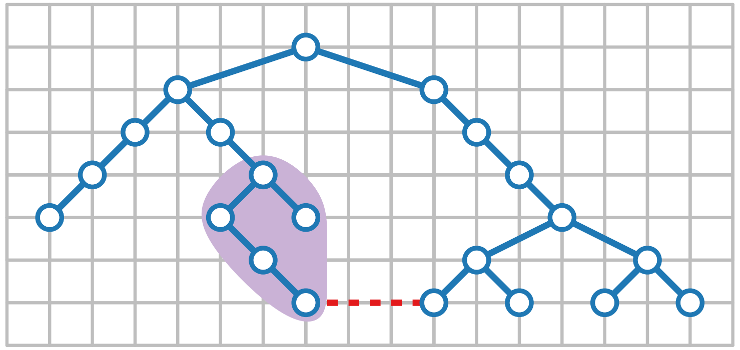
Layered Drawings – Result



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 is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

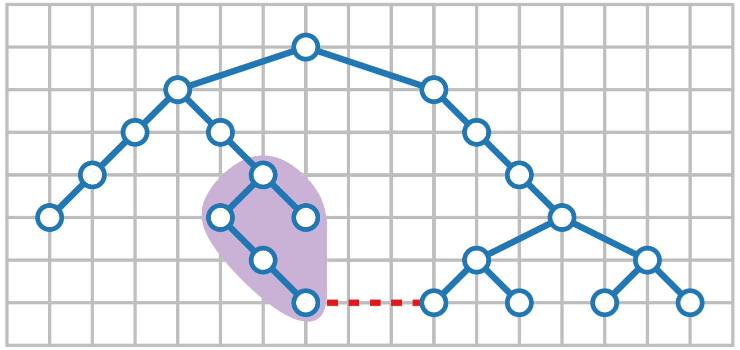
Layered Drawings – Result

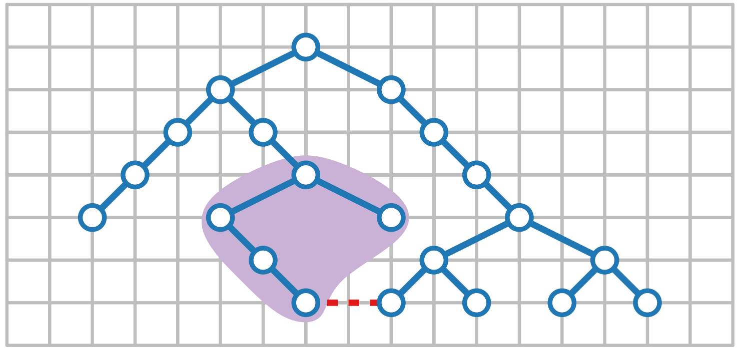


Theorem. [Reingold & Tilford '81]
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time, such that:

- 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 is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

Layered Drawings – Result

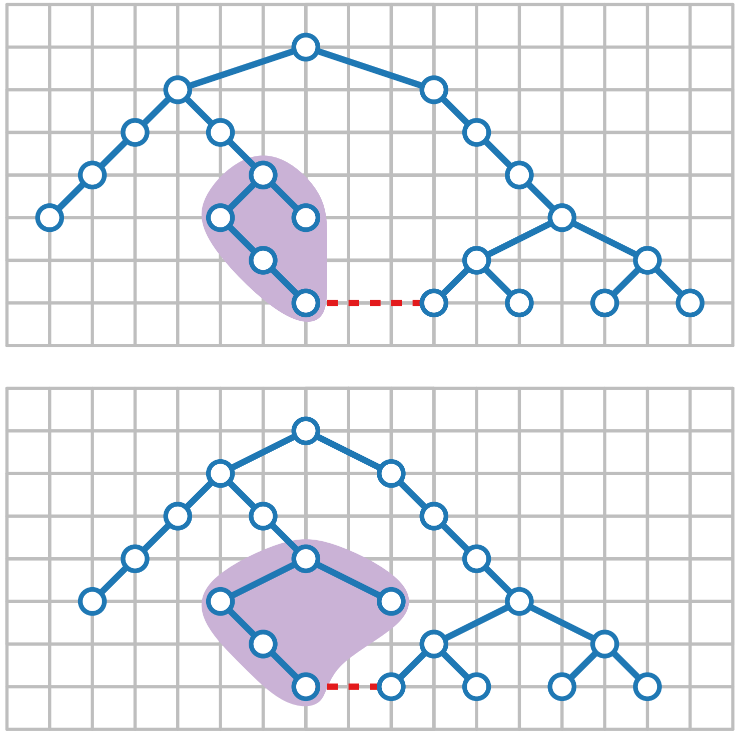




Theorem. [Reingold & Tilford '81]
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time, such that:

- 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 is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!

Layered Drawings – Result

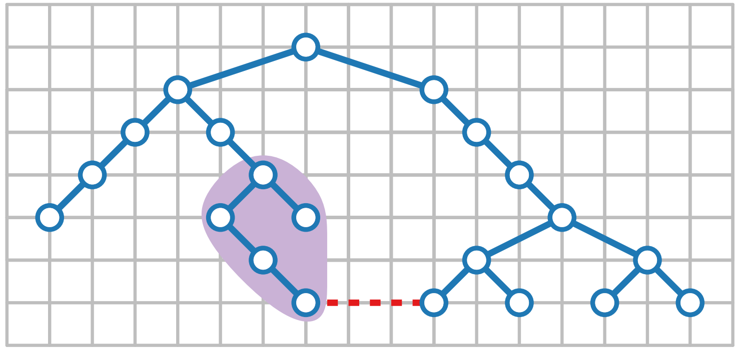


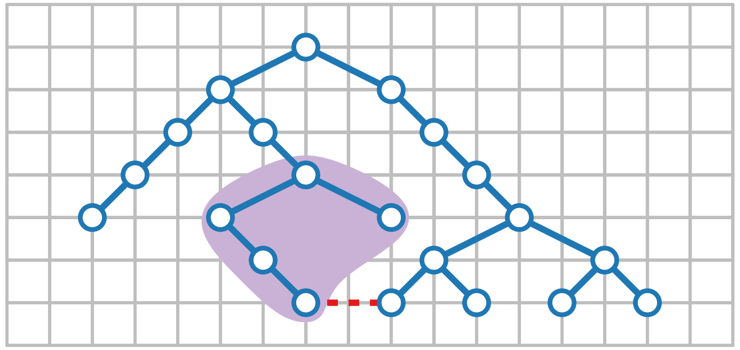
Theorem. [Reingold & Tilford '81]
Let T be a binary tree with n vertices. We can construct a drawing Γ of T in $\mathcal{O}(n)$ time, such that:

- 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 is centred 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 is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!
- Simply isomorphic subtrees have congruent drawings, up to translation

NP-hard

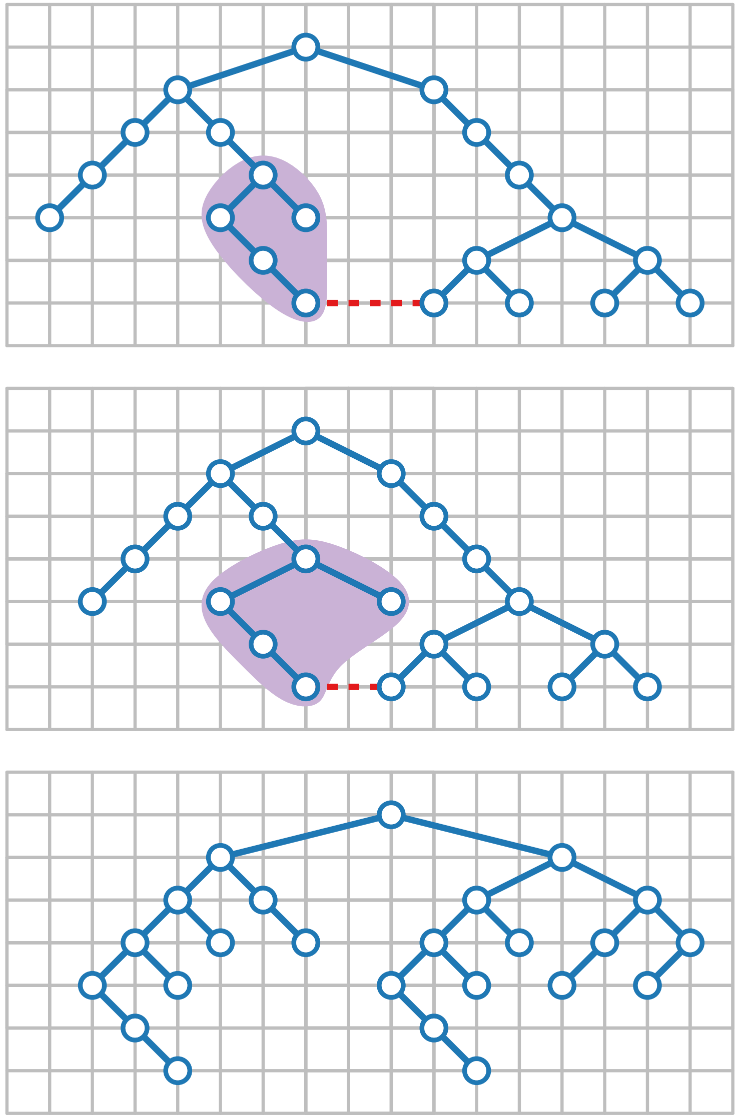
Layered Drawings – Result

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 is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!
- Simply isomorphic subtrees have congruent drawings, up to translation

NP-hard



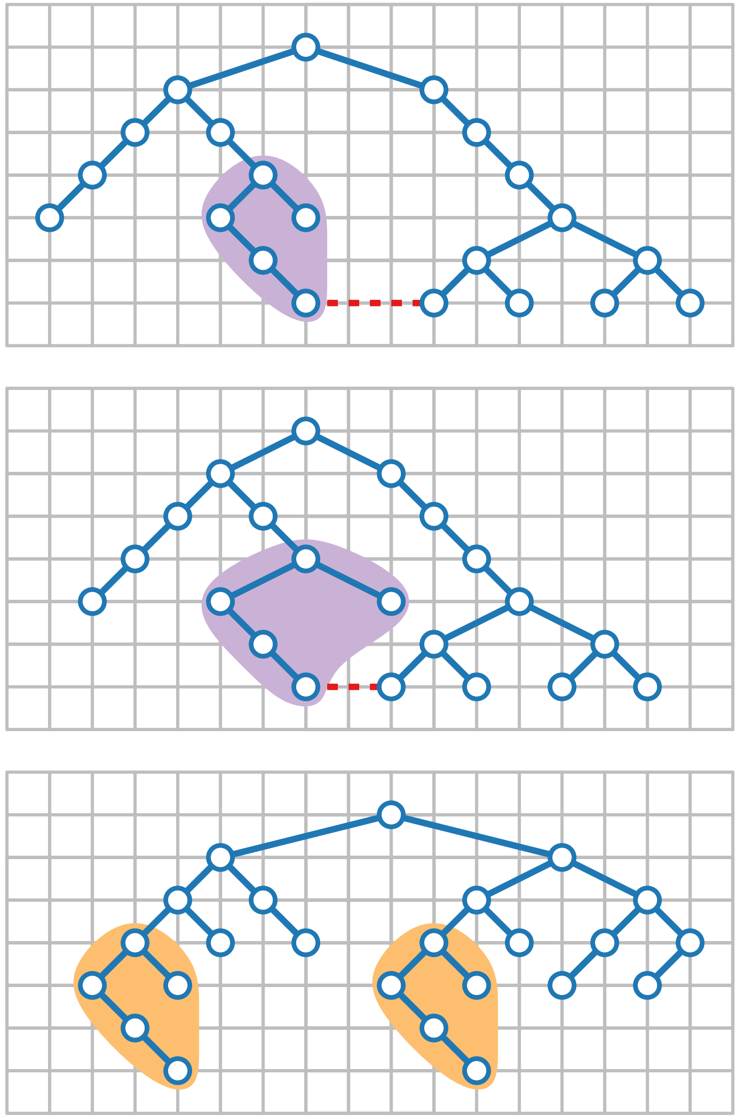
Layered Drawings – Result

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 is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!
- Simply isomorphic subtrees have congruent drawings, up to translation

NP-hard



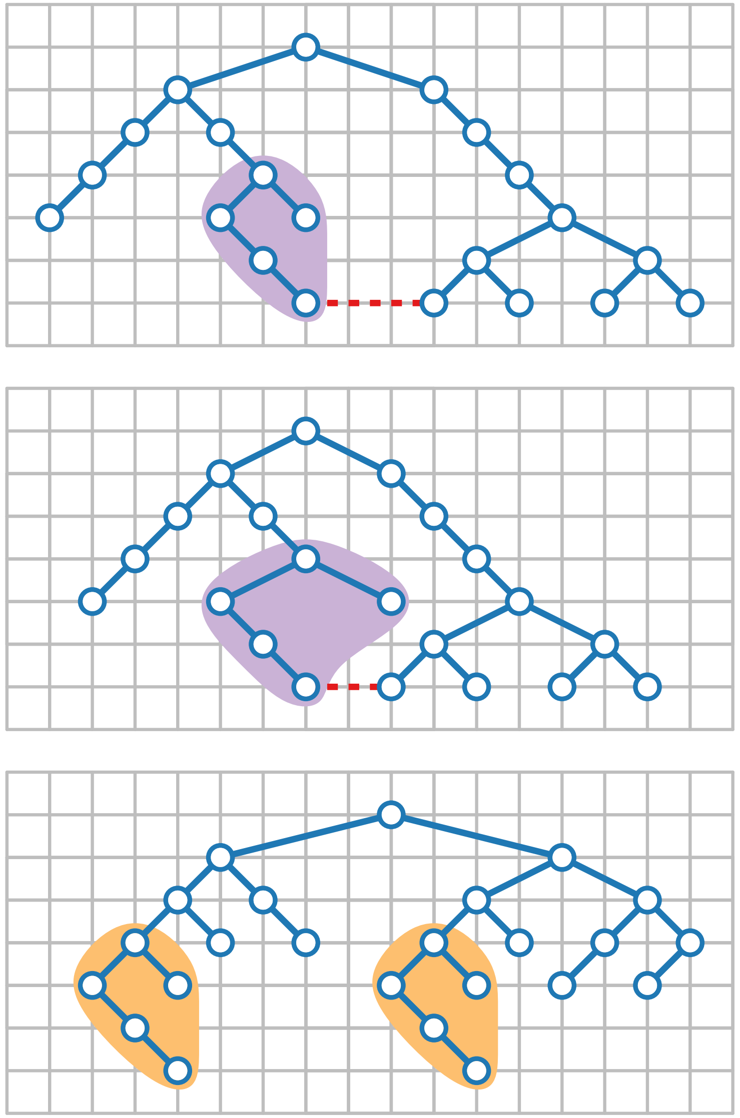
Layered Drawings – Result

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 $-$depth(v)
- Horizontal and Vertical distances are at least 1
- Each vertex is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection

NP-hard



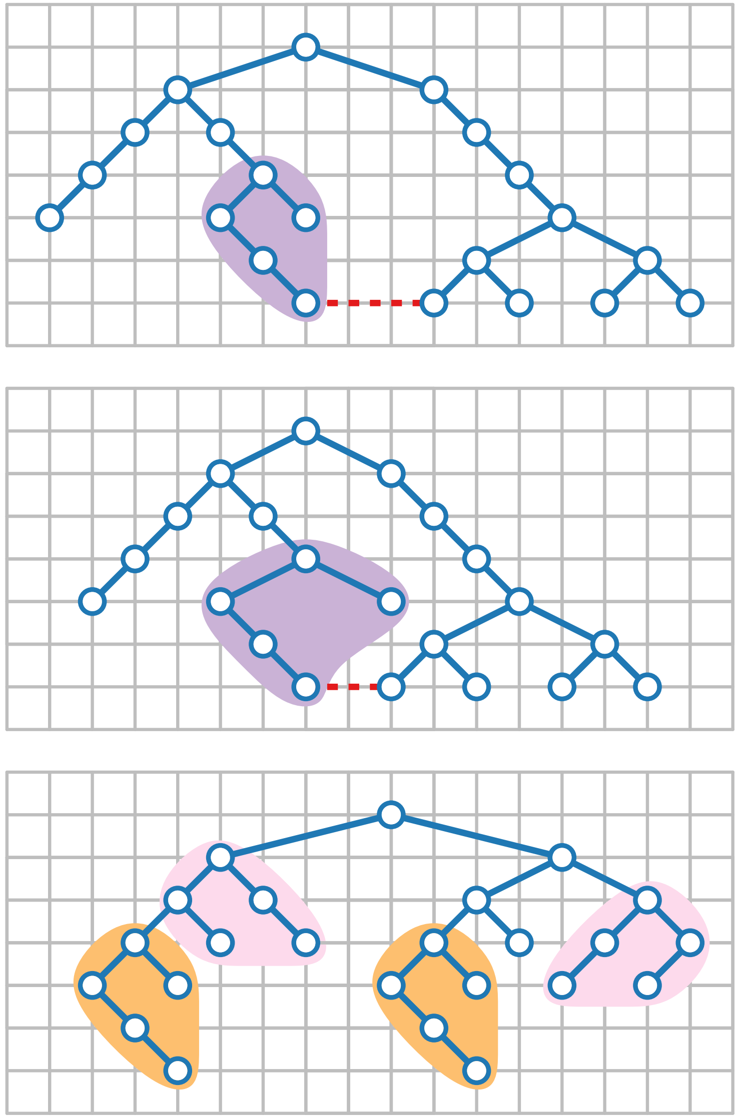
Layered Drawings – Result

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 $-$depth(v)
- Horizontal and Vertical distances are at least 1
- Each vertex is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection

NP-hard

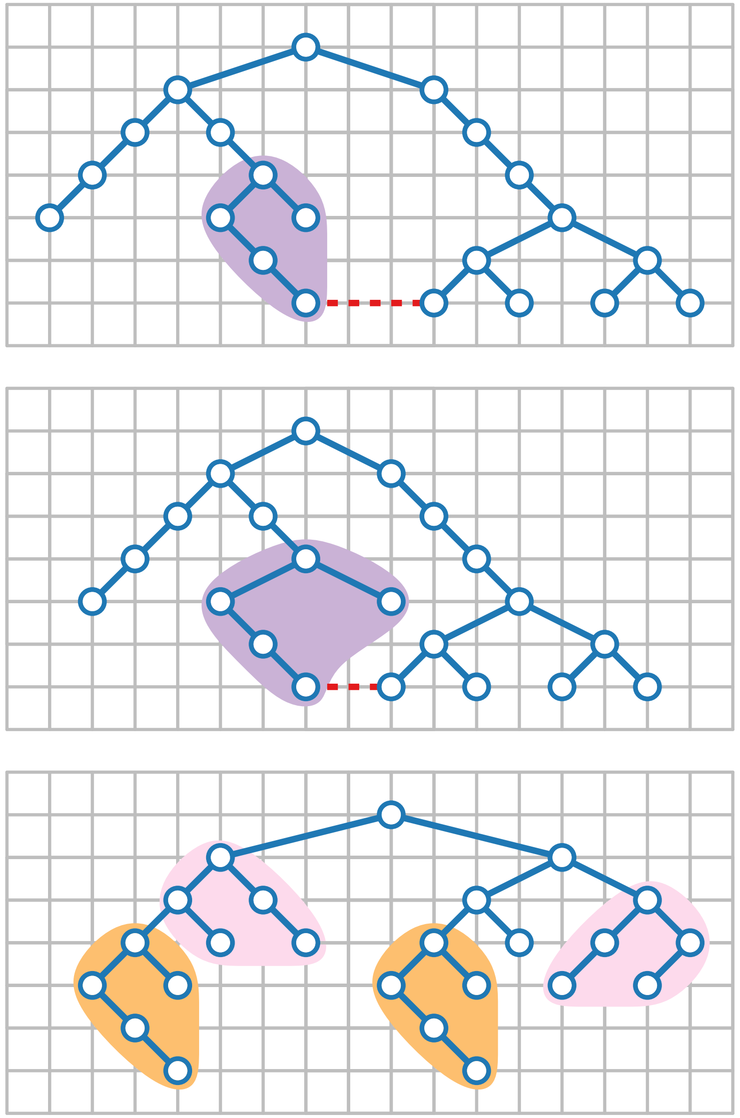


Layered Drawings – Result

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 $-\mathrm{depth}(v)$
- Horizontal and Vertical distances are at least 1
- Each vertex is centred 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

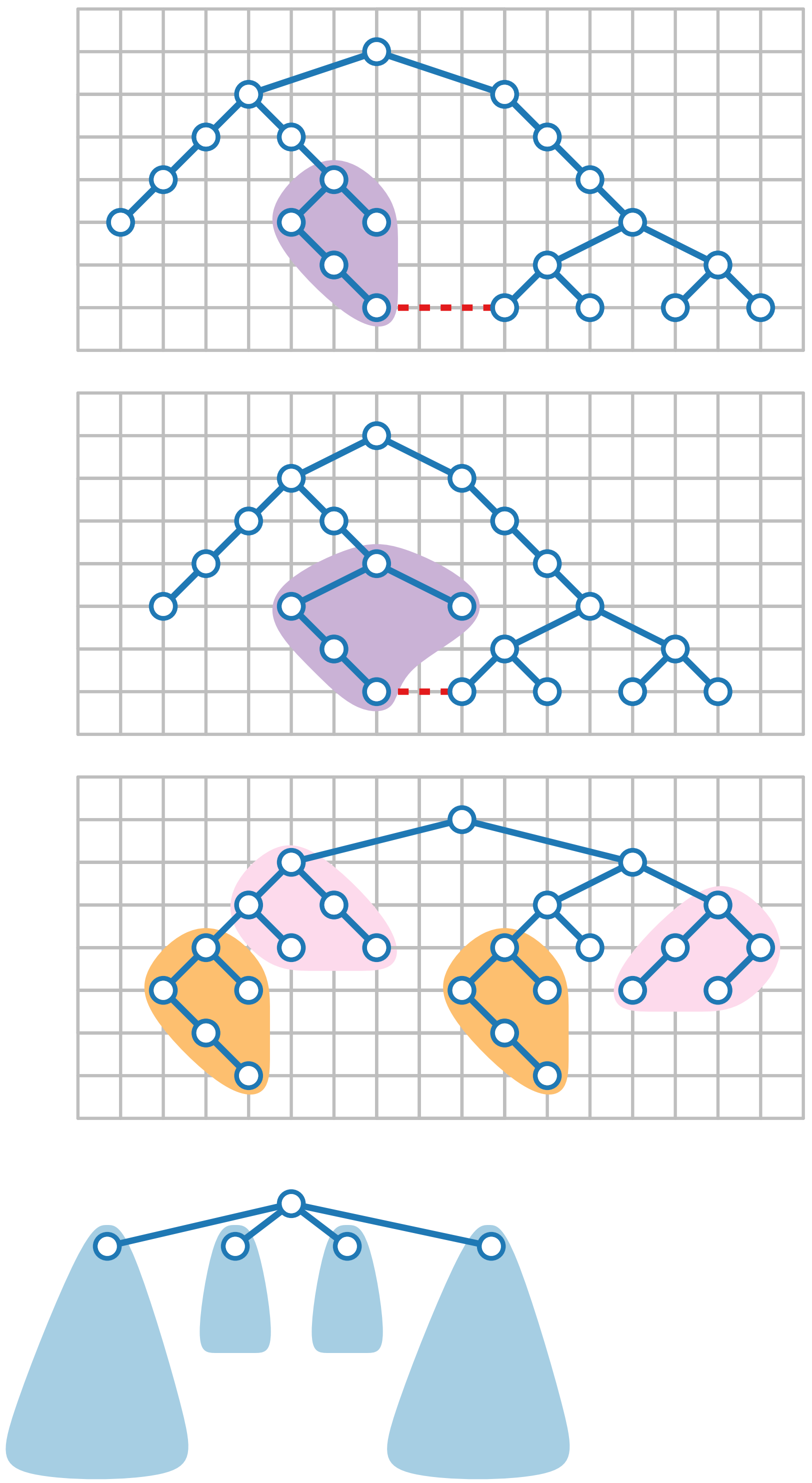


Layered Drawings – Result

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 is centred 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

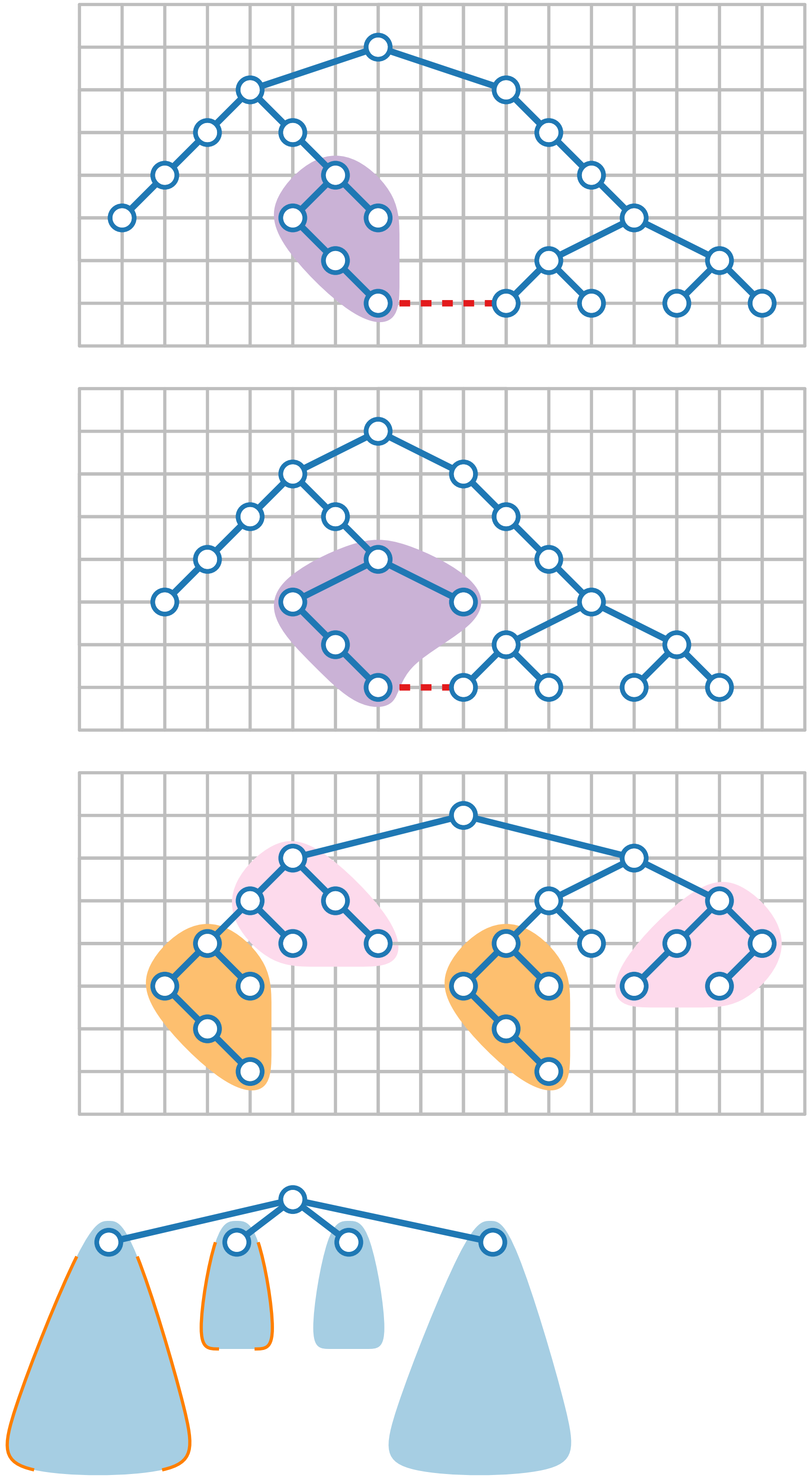


Layered Drawings – Result

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 is centred 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

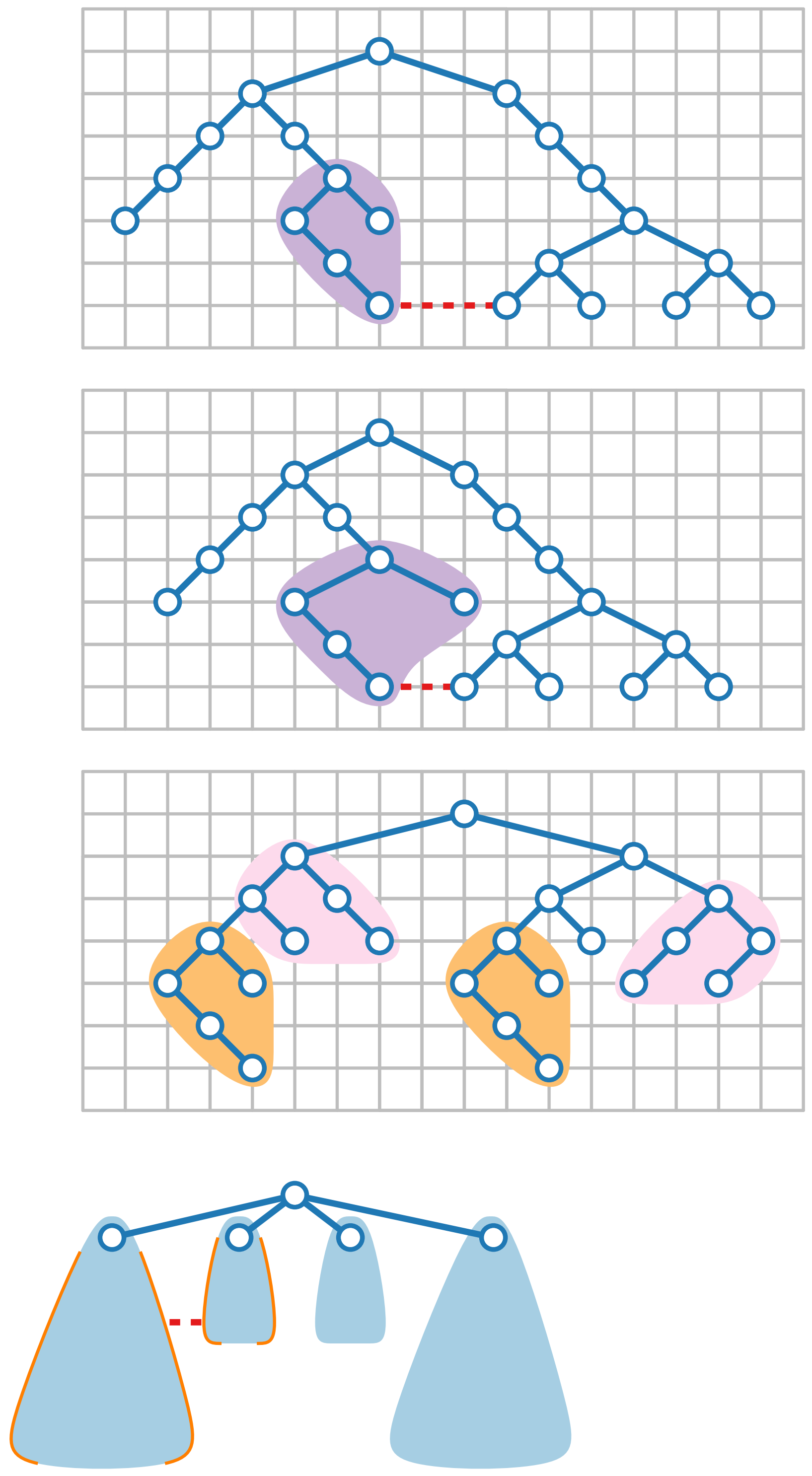


Layered Drawings – Result

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 is centred 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

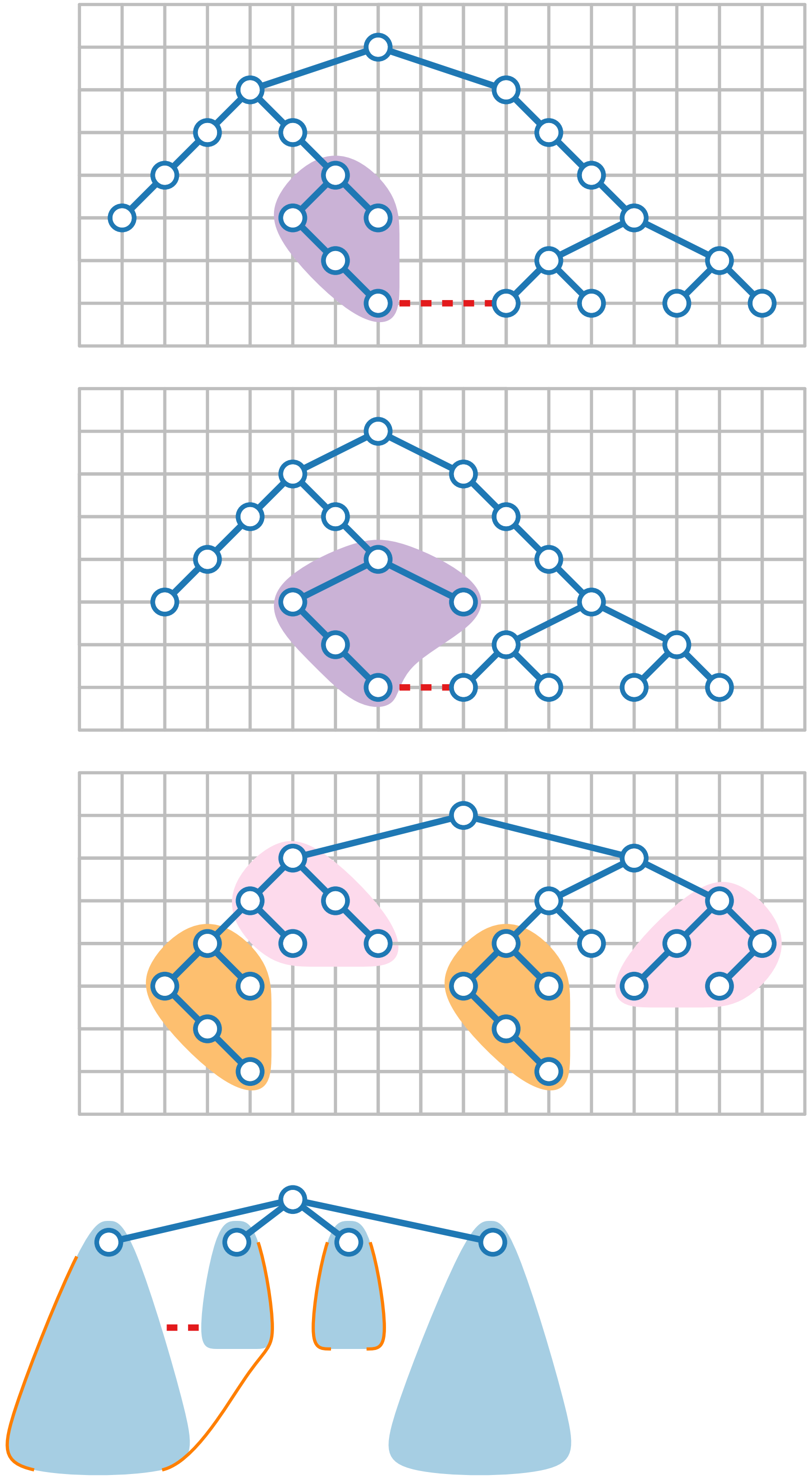


Layered Drawings – Result

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 is centred 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

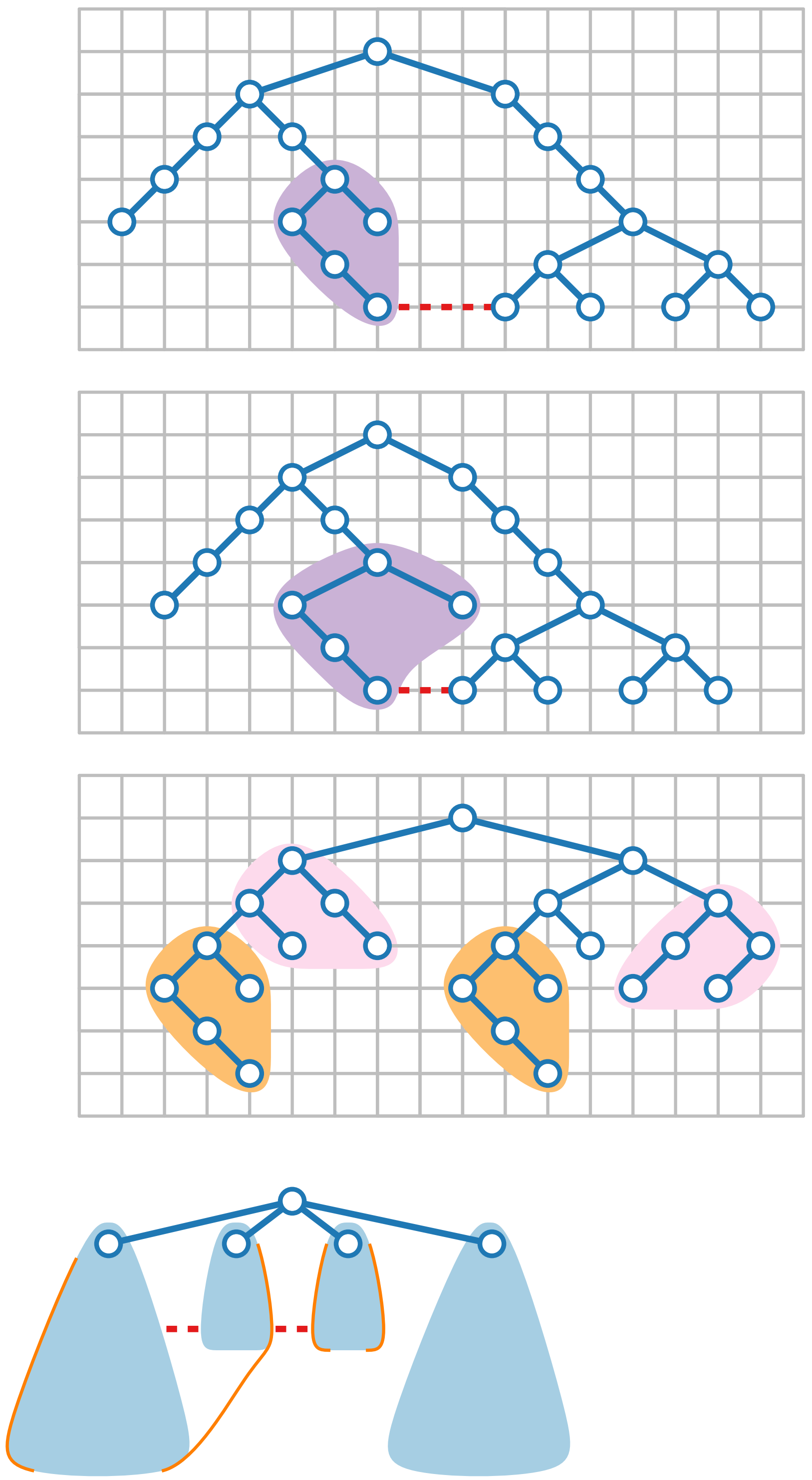


Layered Drawings – Result

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 $-$depth(v)
- Horizontal and Vertical distances are at least 1
- Each vertex is centred 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



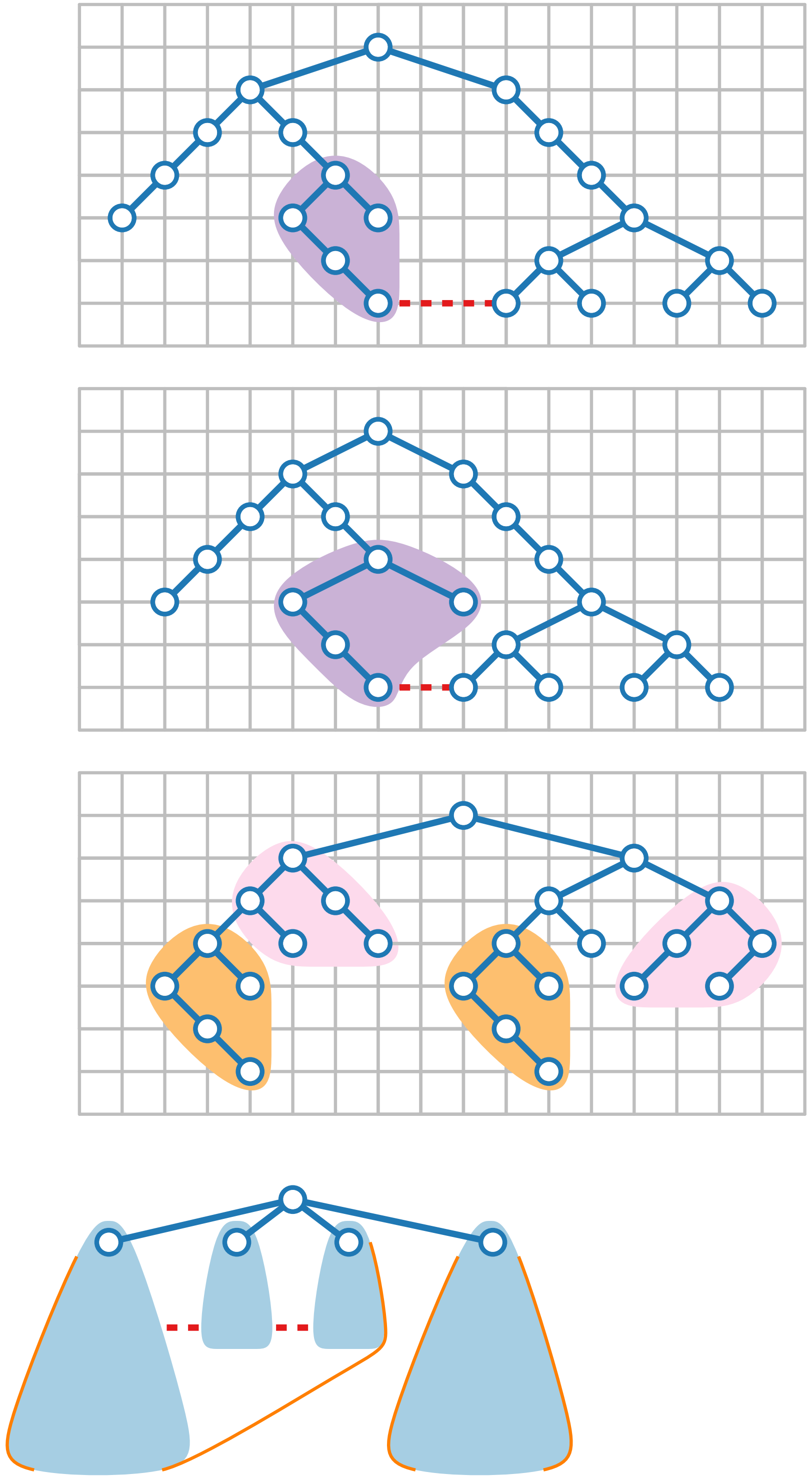
Layered Drawings – Result

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 $-$depth(v)
- Horizontal and Vertical distances are at least 1
- Each vertex is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal!
- Simply isomorphic subtrees have congruent drawings, up to translation
- Axially isomorphic subtrees have congruent drawings, up to translation and reflection

NP-hard



Layered Drawings – Result

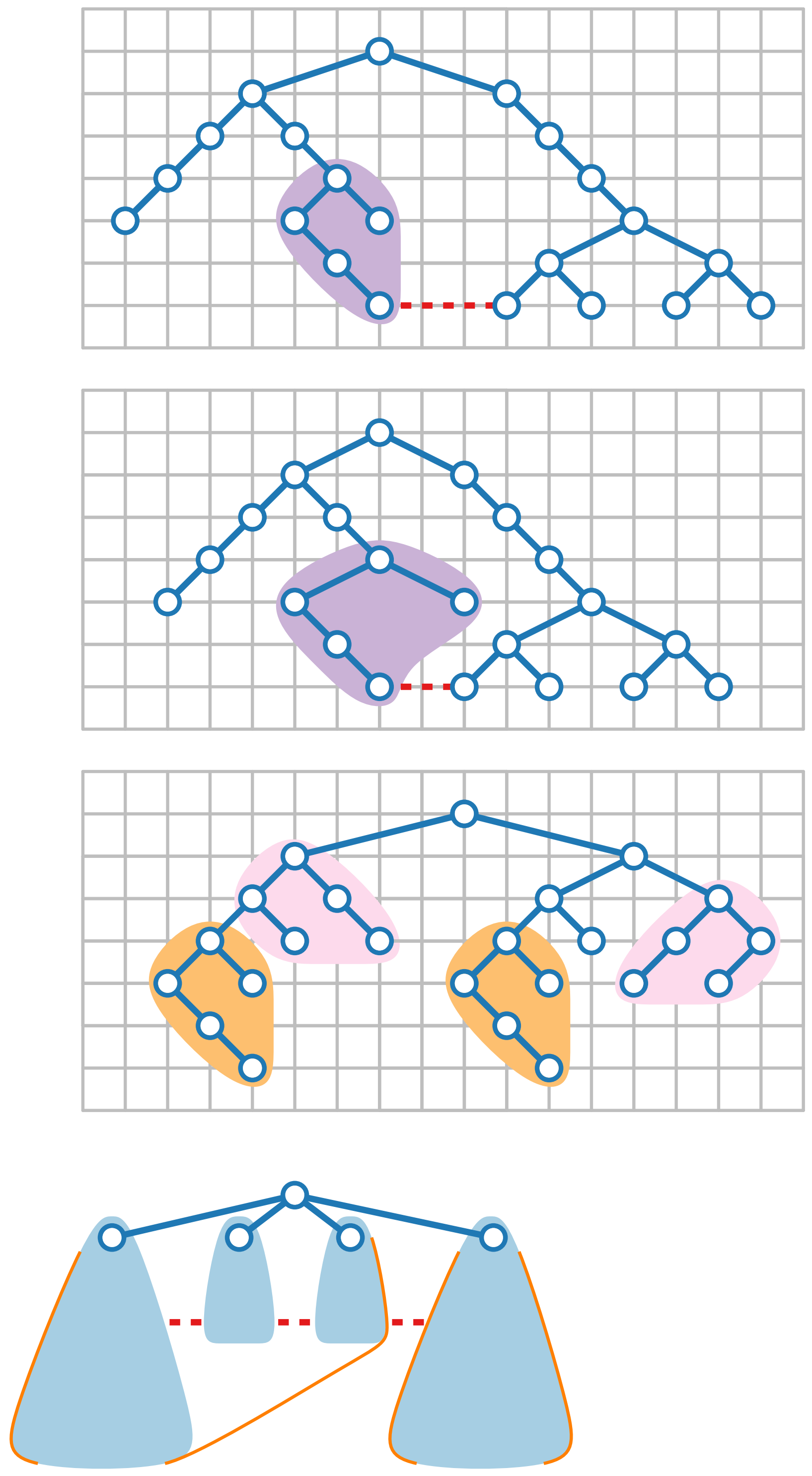
Theorem. [Reingold & Tilford '81]

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 is centred 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

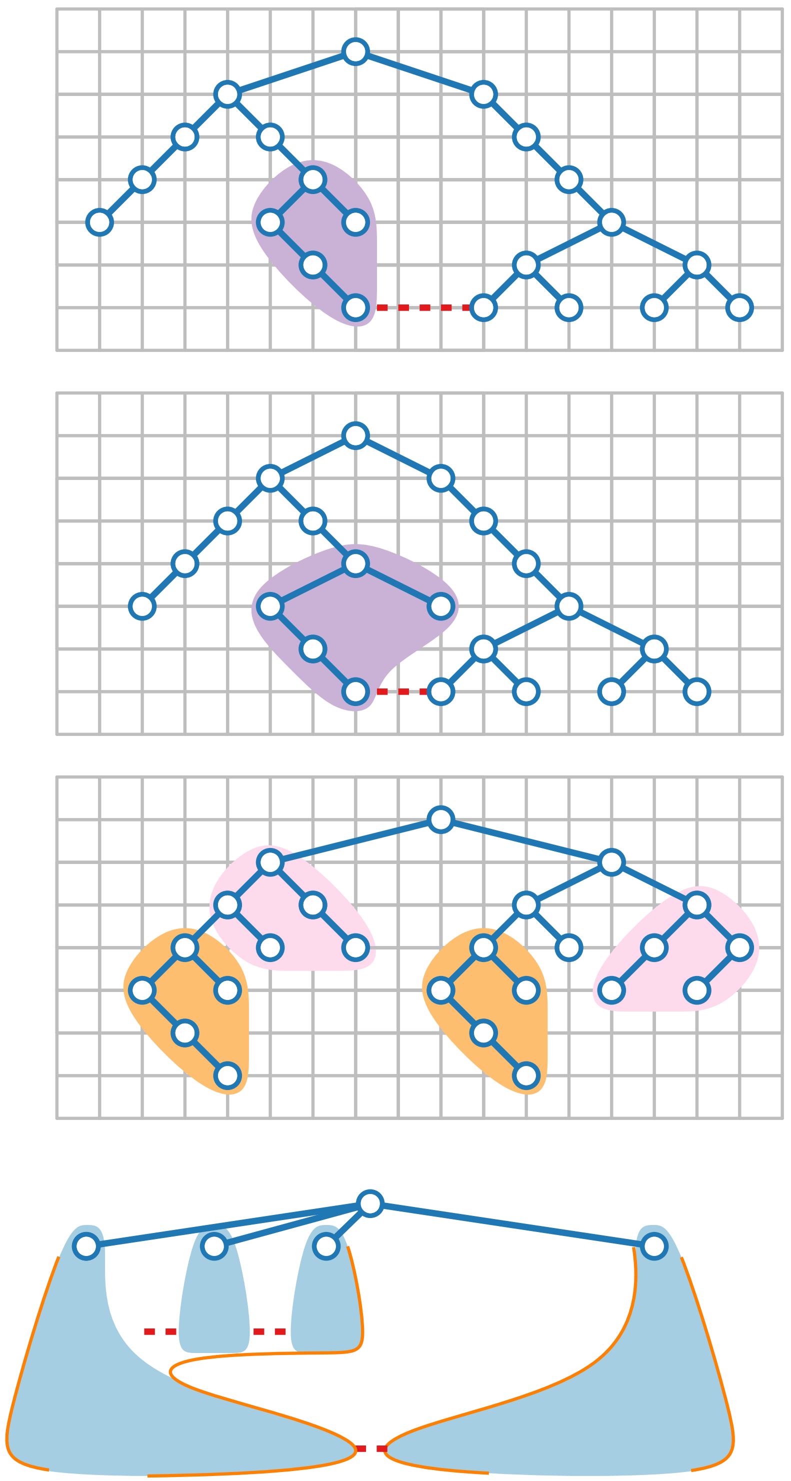


Layered Drawings – Result

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 is centred 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

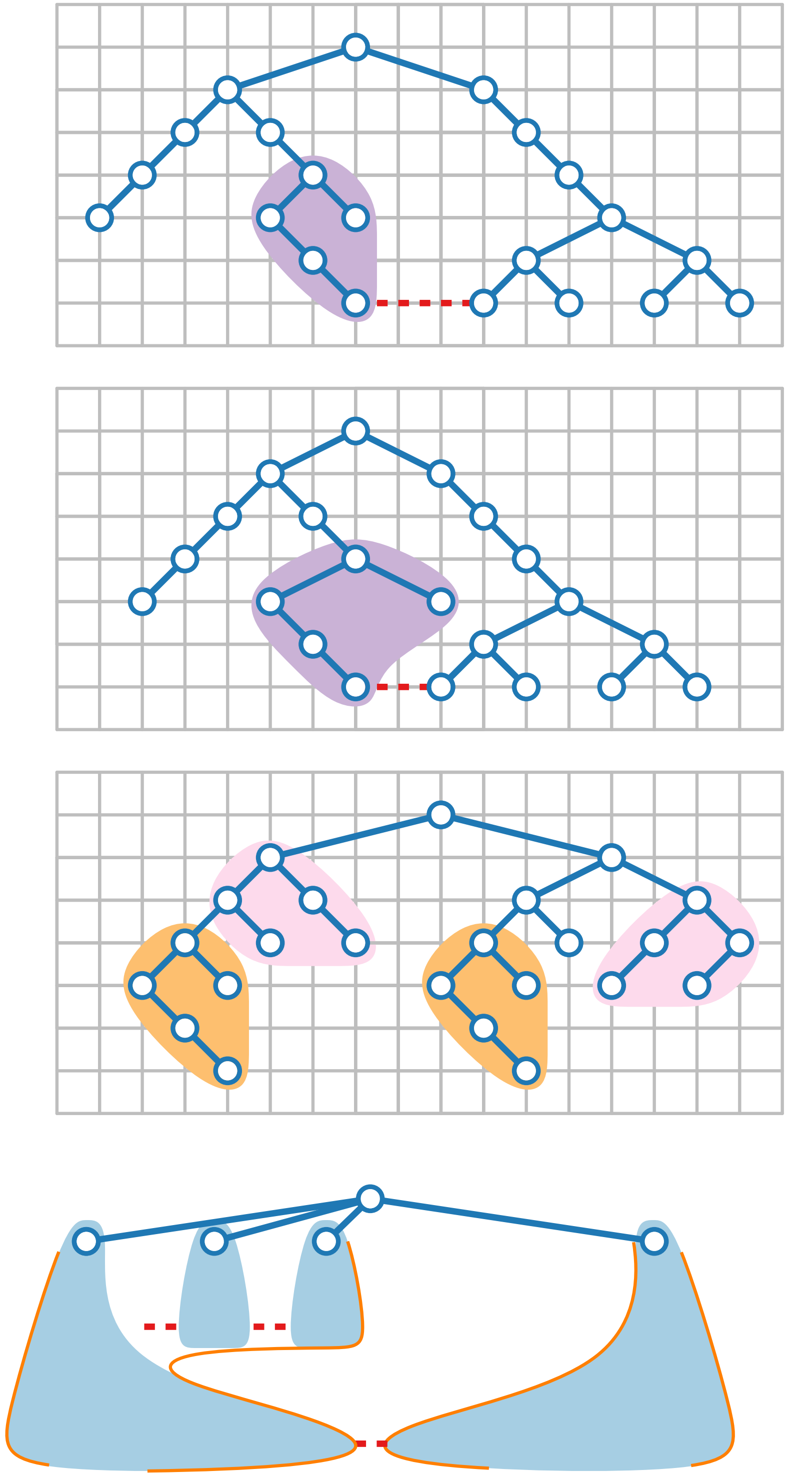


Layered Drawings – Result

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:

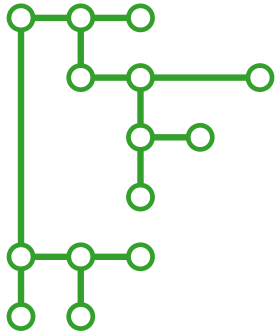
- Γ is planar, straight-line and strictly downward
- Γ is layered: y-coordinate of vertex v is $-\text{depth}(v)$
- Horizontal and Vertical distances are at least 1
- Each vertex is centred w.r.t. its children
- Area of Γ is in $\mathcal{O}(n^2)$ – but not optimal! NP-hard
- Simply isomorphic subtrees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

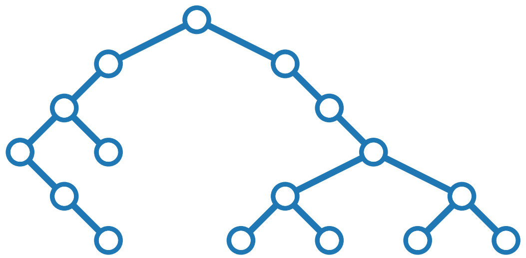


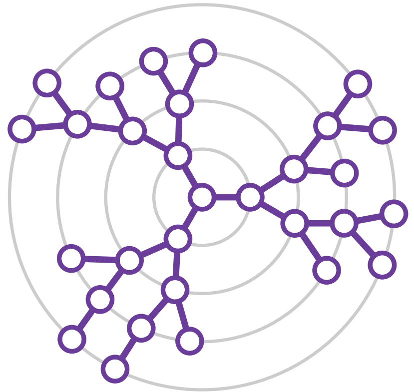
Visualization of Graphs

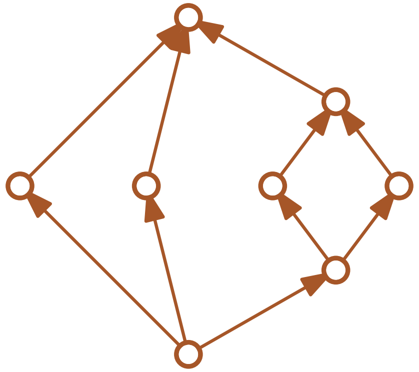
Lecture 1b:
Drawing Trees and Series-Parallel Graphs

Part II:
HV-Drawings









HV-Drawings – Drawing Style

Applications

- Cons cell diagram in LISP

HV-Drawings – Drawing Style

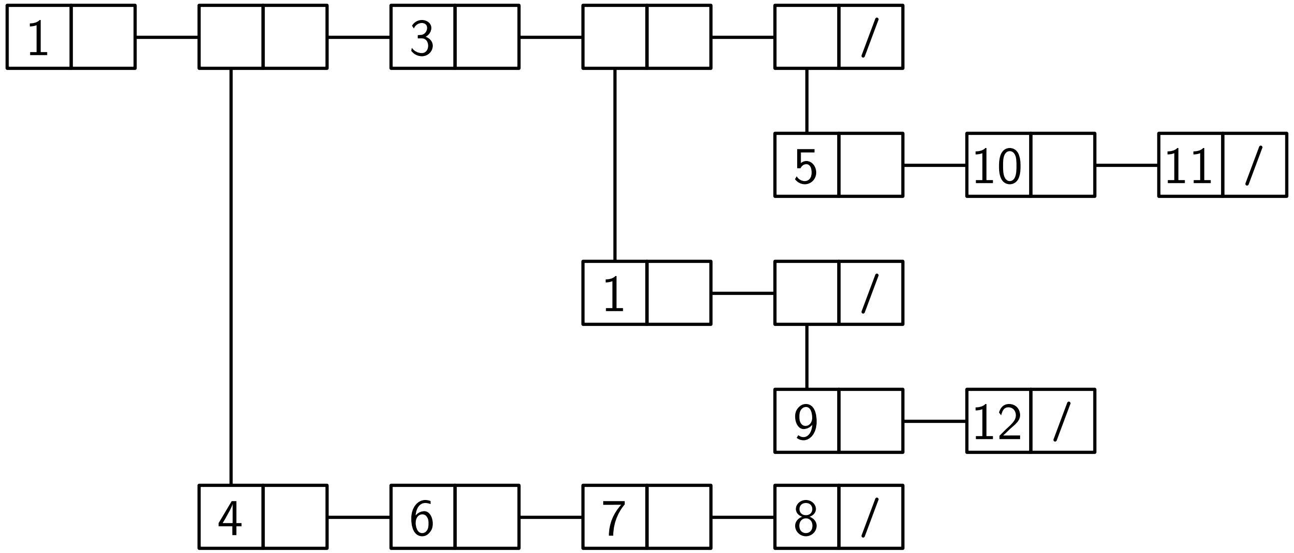
Applications

- 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

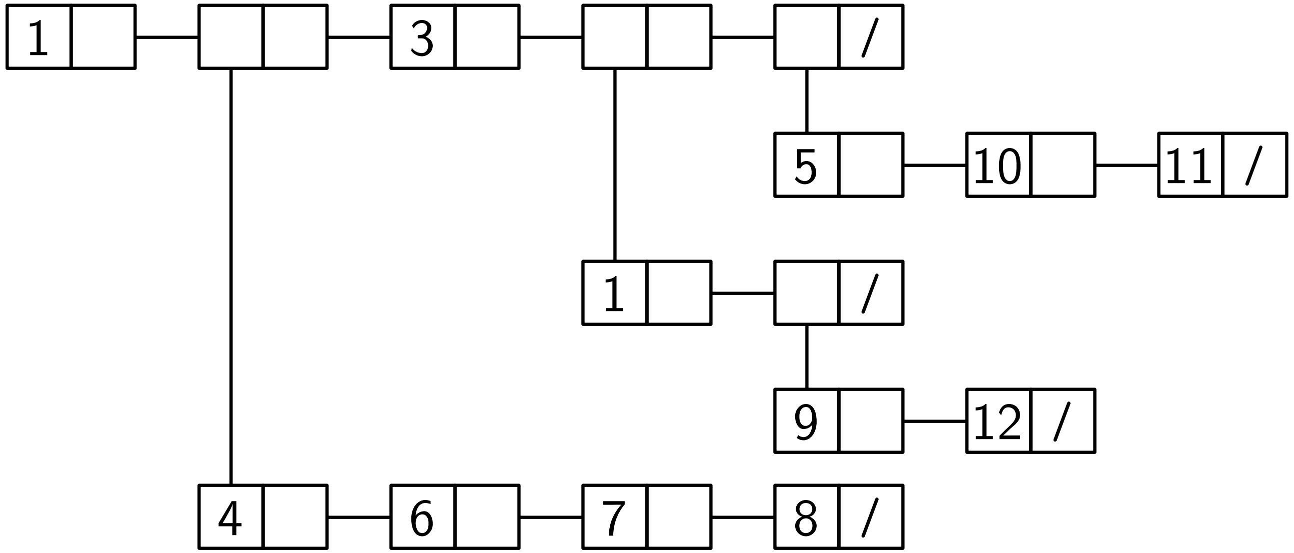


Source: after `gajon.org/trees-linked-lists-common-lisp/`

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/

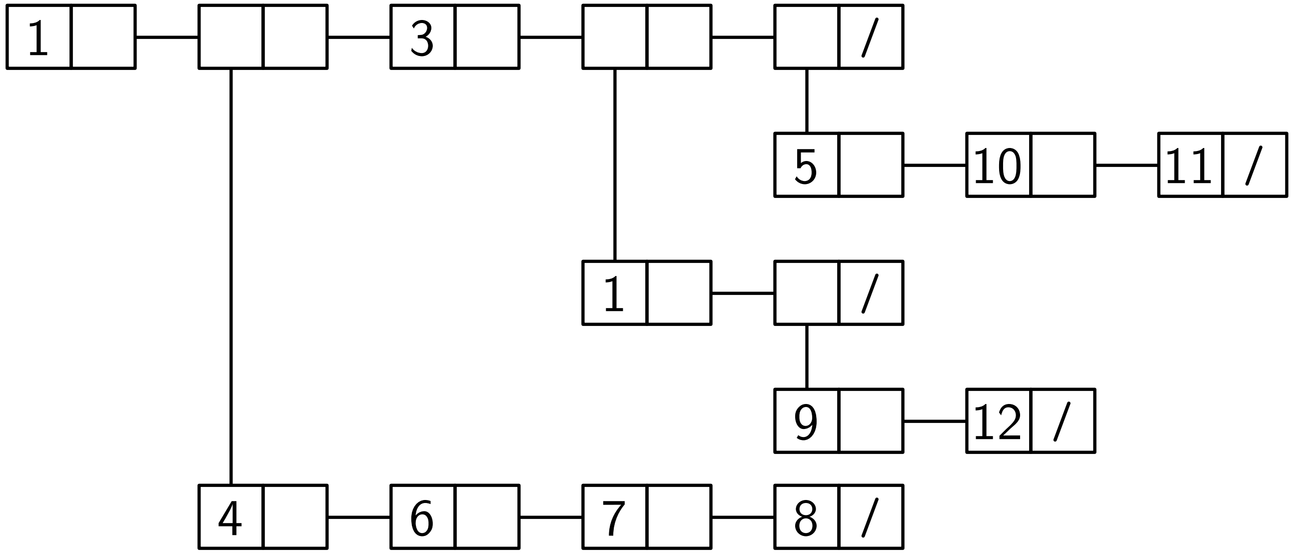
Drawing conventions

Drawing aesthetics

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

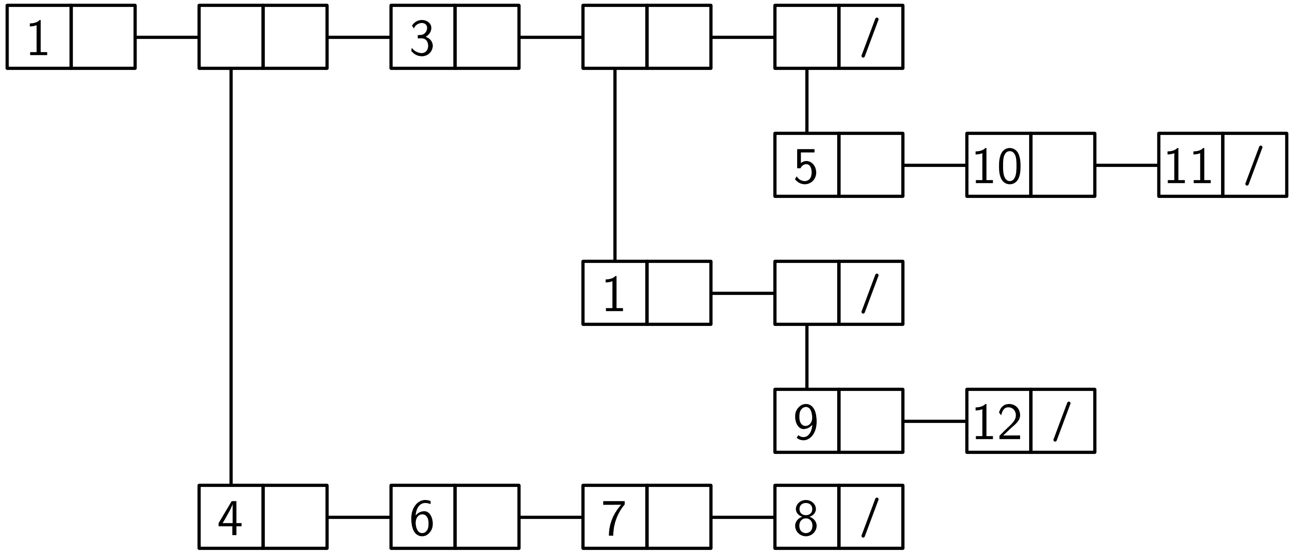
- Children are vertically or horizontally aligned with their parent

Drawing aesthetics

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

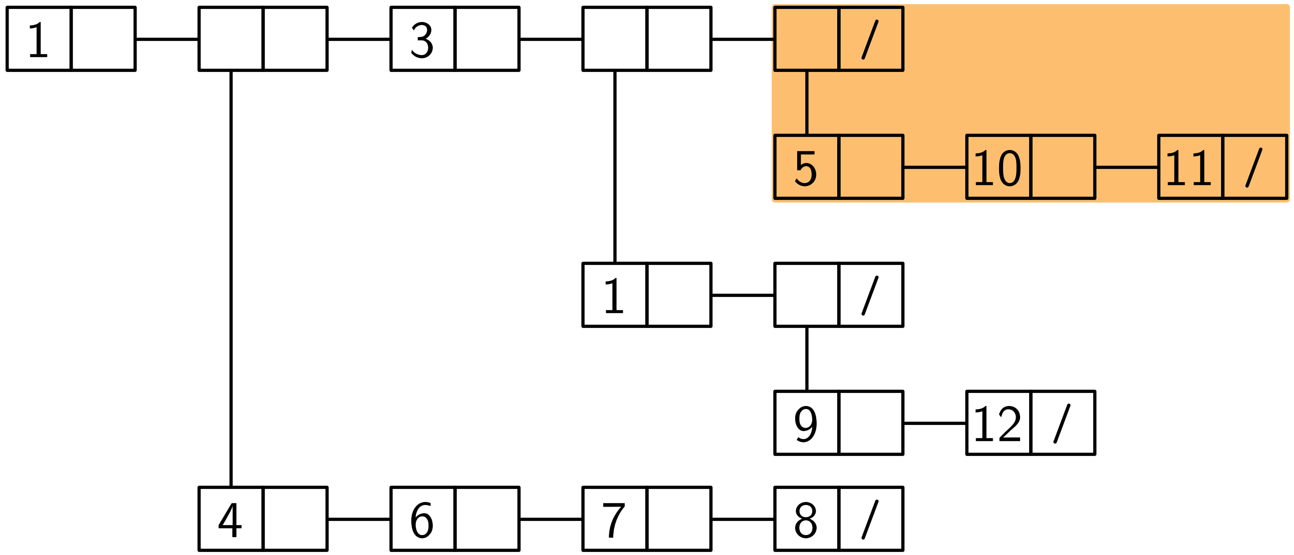
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics

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

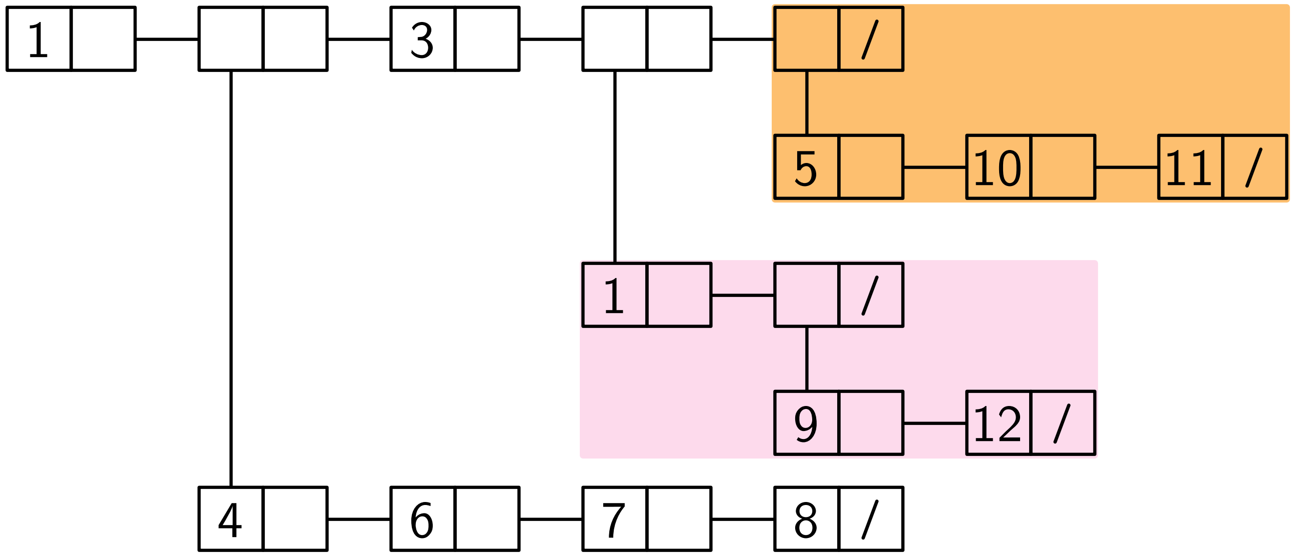
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics

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

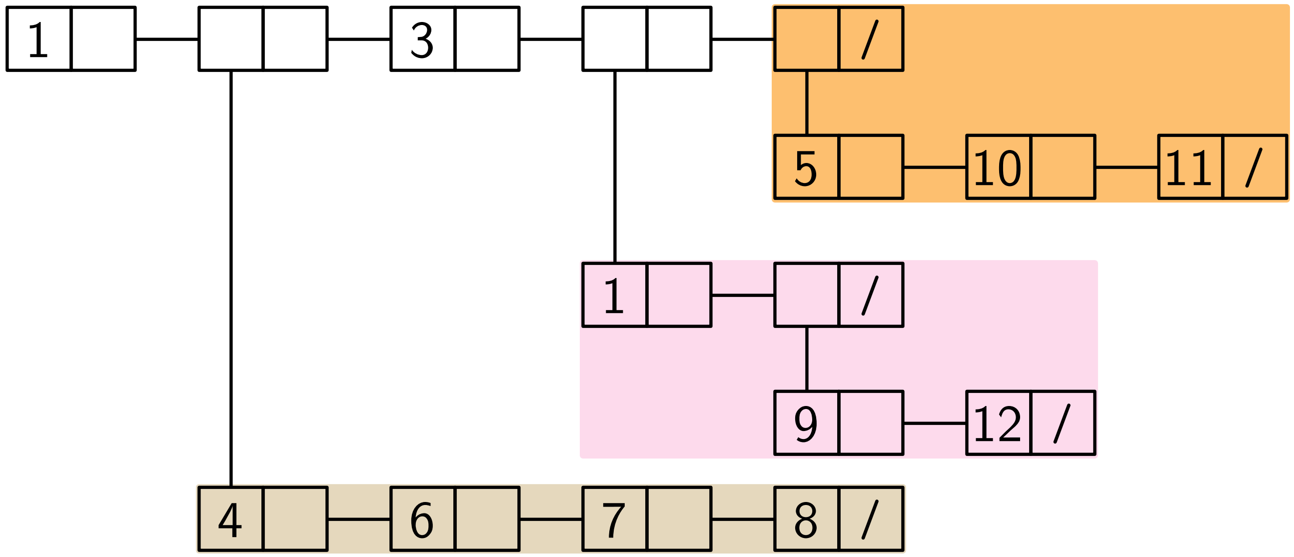
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics

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

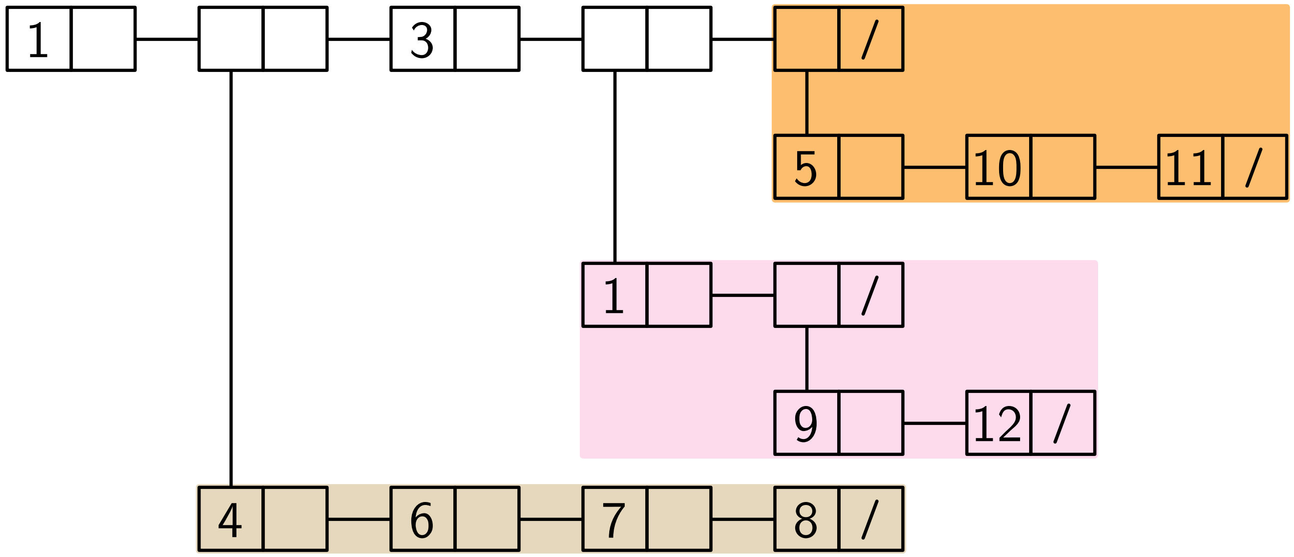
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint

Drawing aesthetics

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

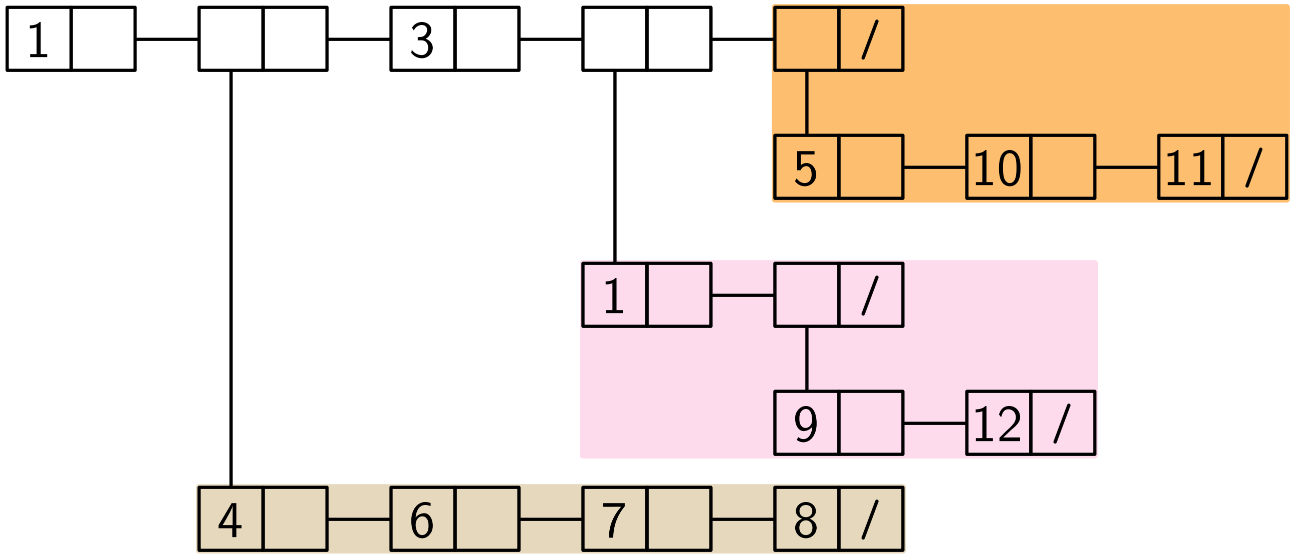
- Children are vertically or horizontally aligned with their parent
- The bounding boxes of the subtrees of the children are disjoint
- Edges are strictly down- or rightwards

Drawing aesthetics

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

- 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

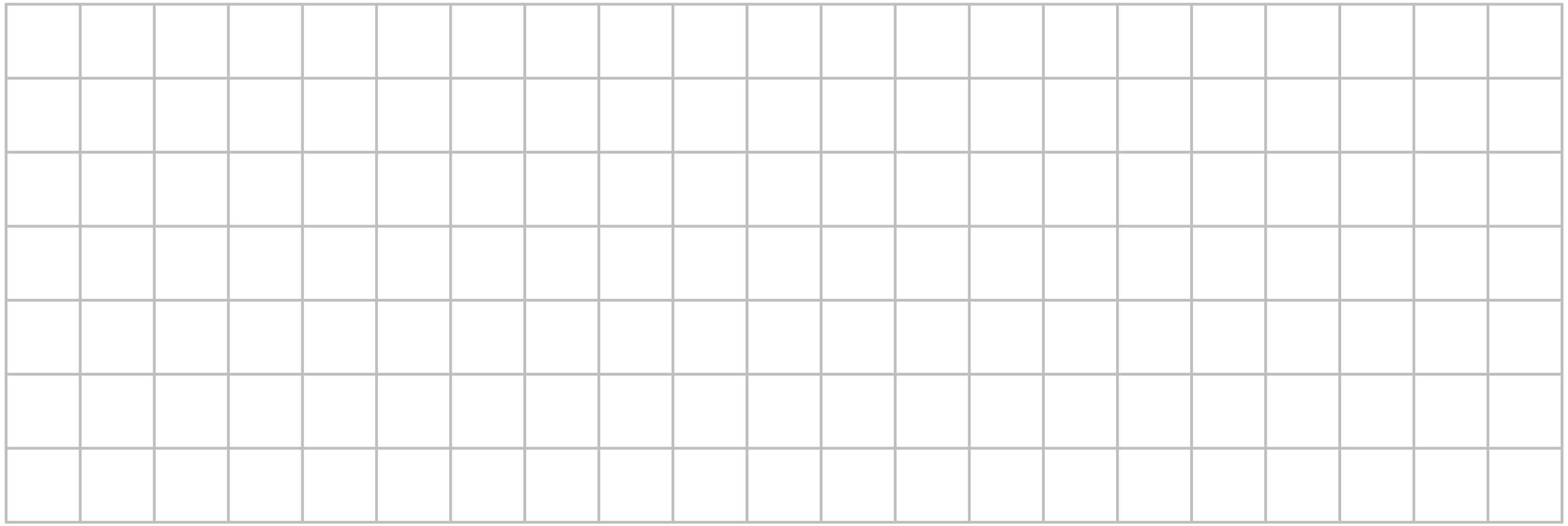
Base case: 
Divide: Recursively apply the algorithm to draw the left and right subtrees

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:

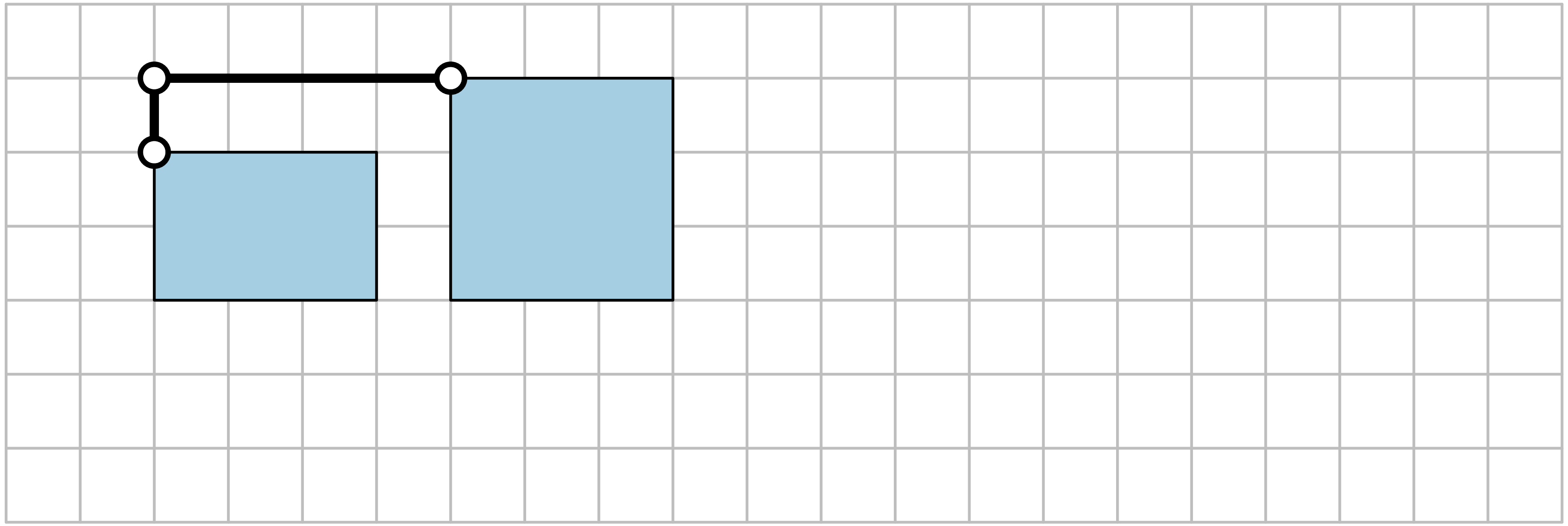


HV-Drawings – Algorithm

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

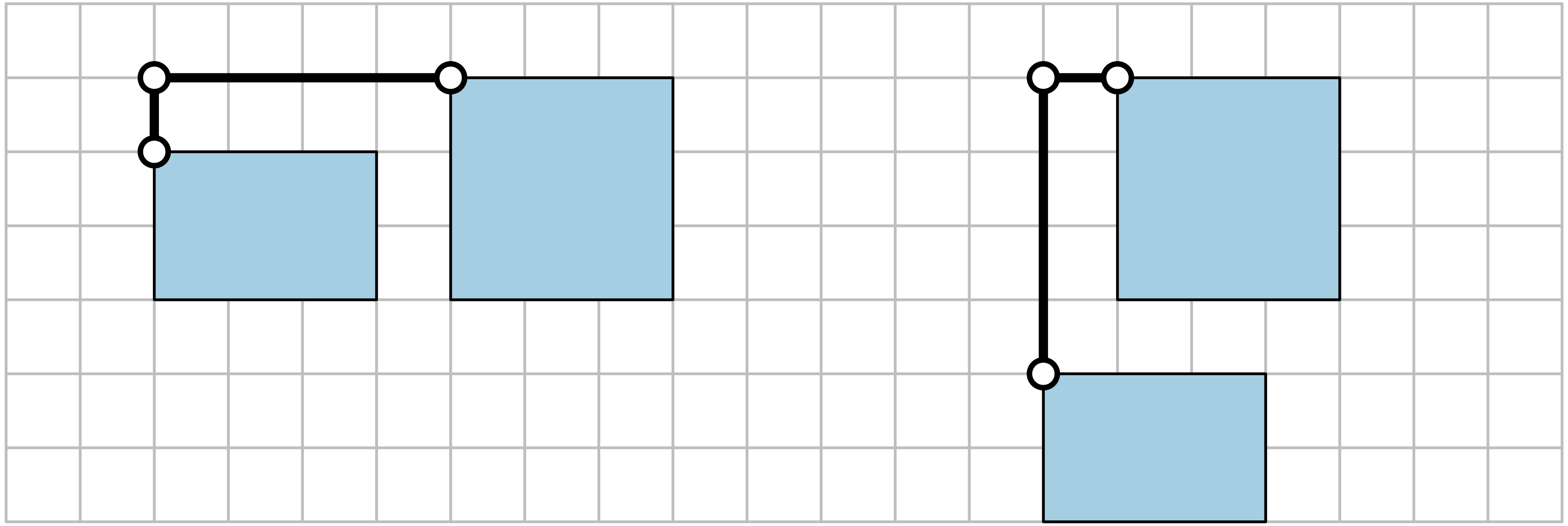


HV-Drawings – Algorithm

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: horizontal combination vertical combination



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!*

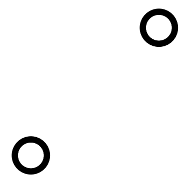
  Size of subtree $:=$ number of vertices

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

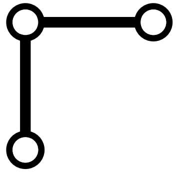


HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

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

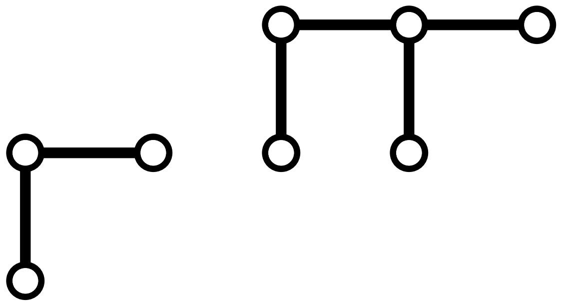
  Size of subtree := number of vertices



HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

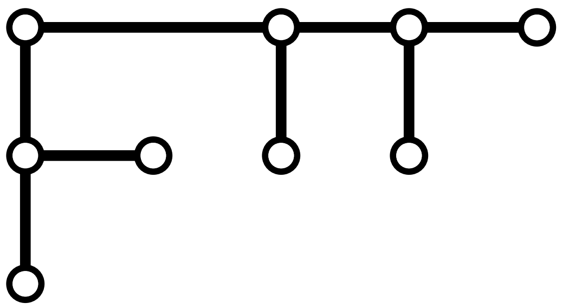
- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*
  Size of subtree := number of vertices



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

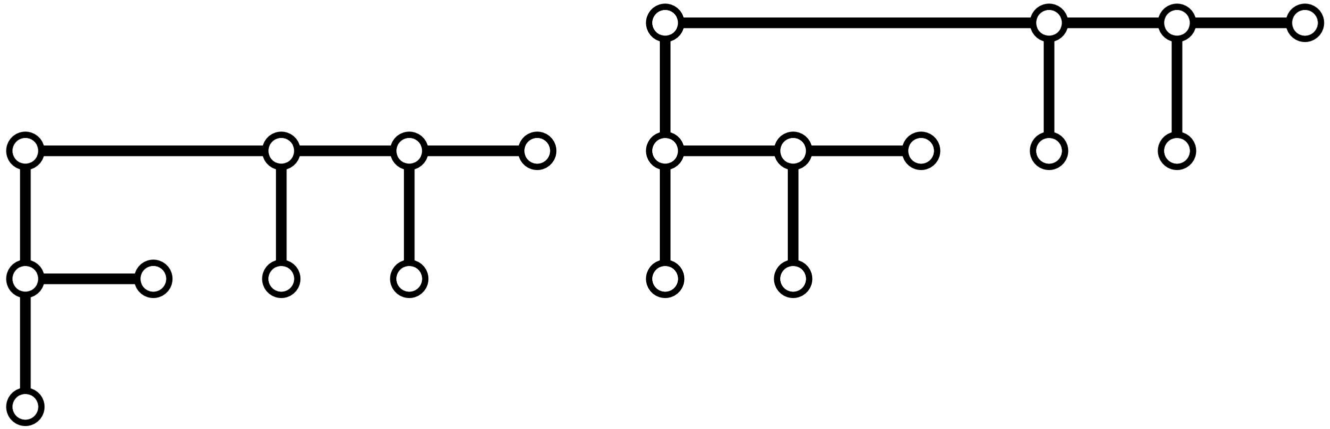


HV-Drawings – Right-Heavy HV-Layout

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

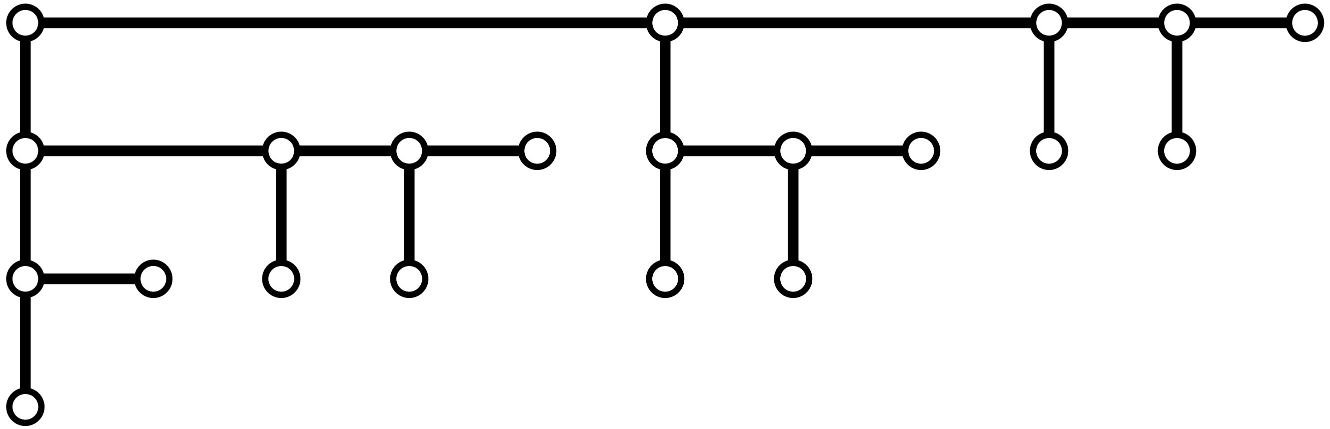


HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

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

  Size of subtree := number of vertices

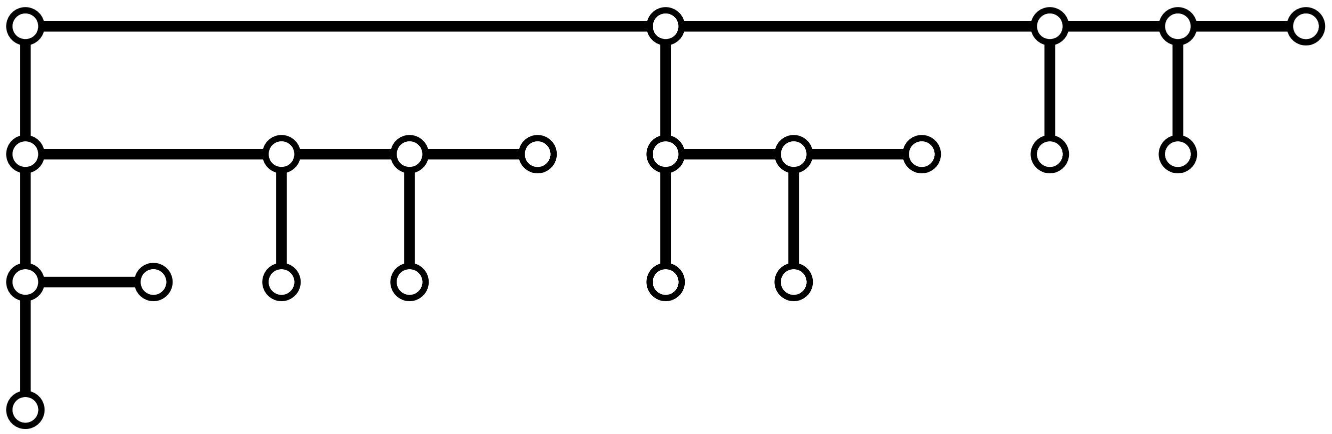


HV-Drawings – Right-Heavy HV-Layout

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

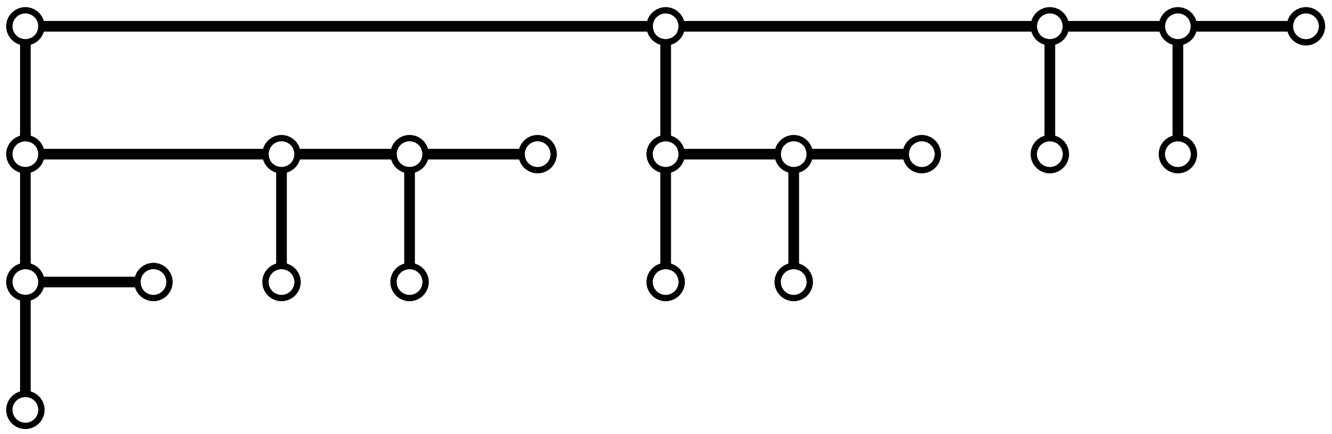
- width at most and
- height at most

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

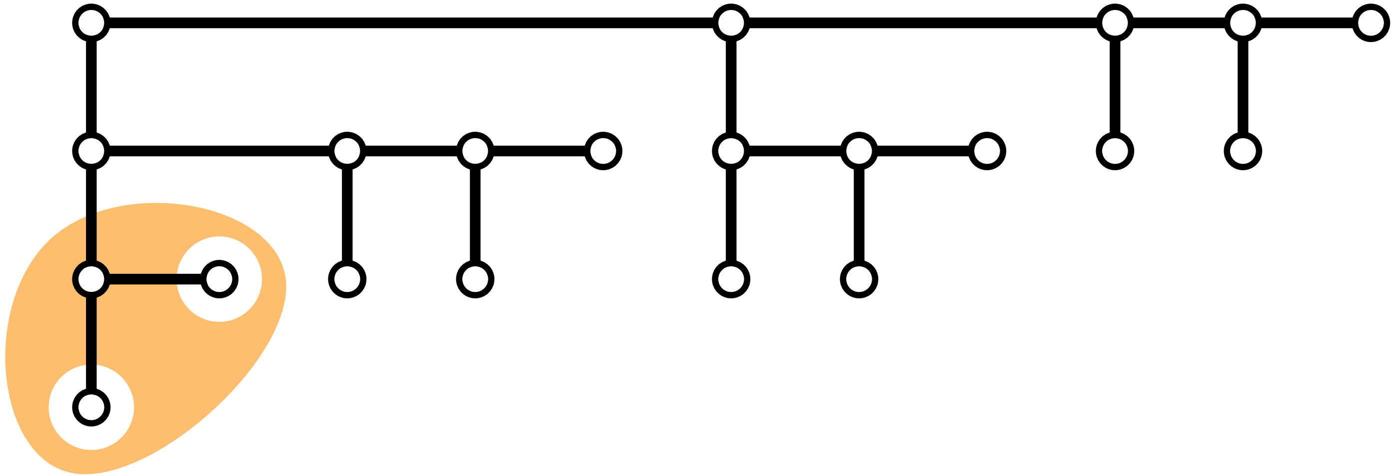
- width at most $n - 1$ and
- height at most

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



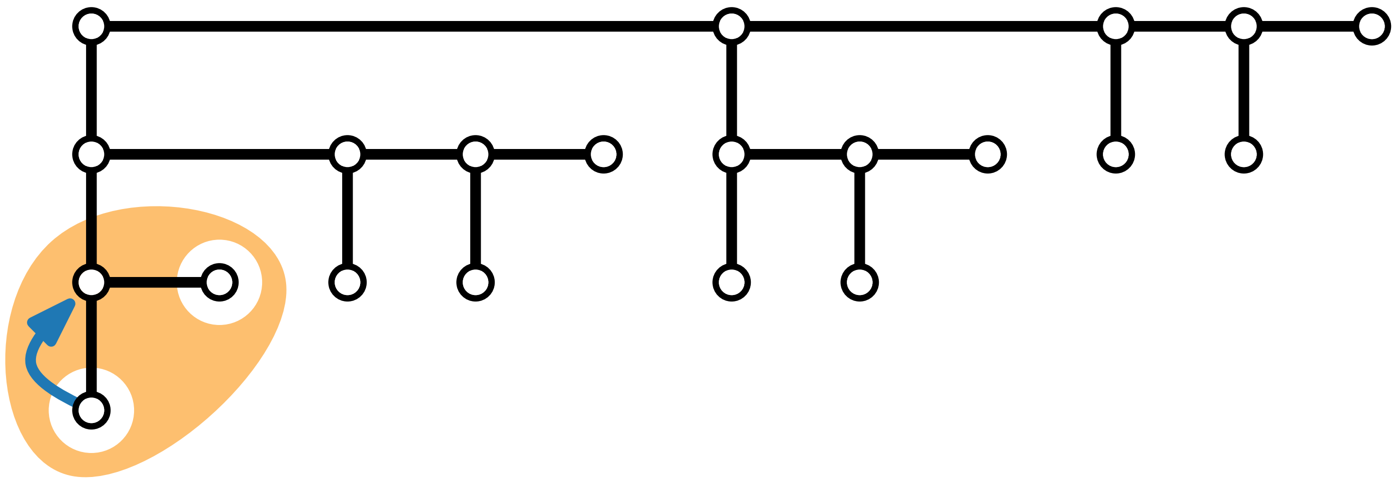
Lemma. Let T be a binary tree. The drawing constructed by the right-heavy approach has

- 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

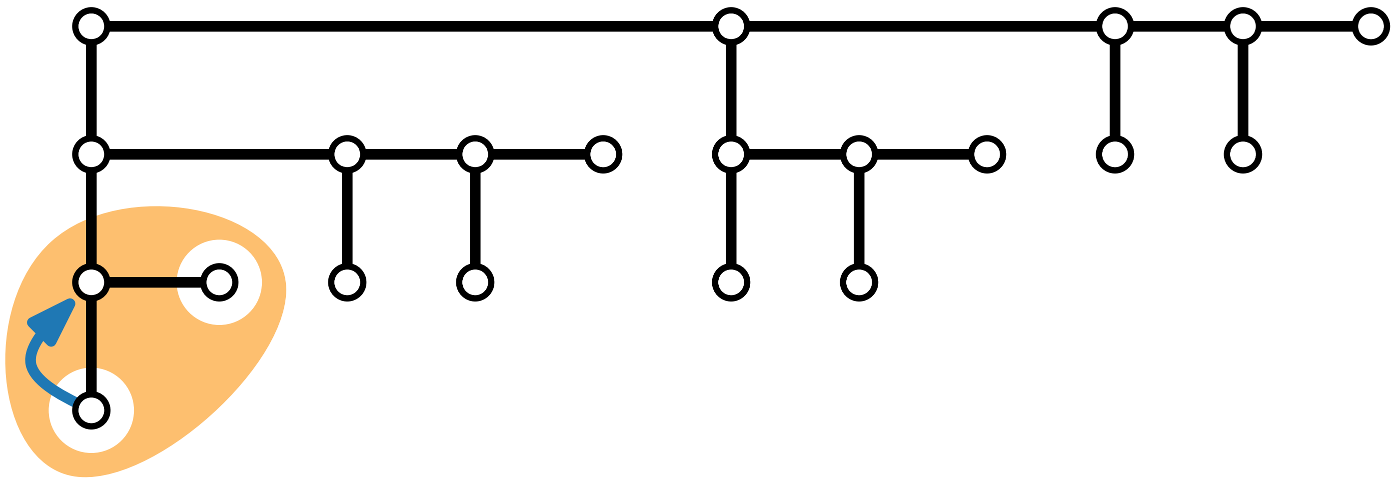
- width at most $n-1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

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

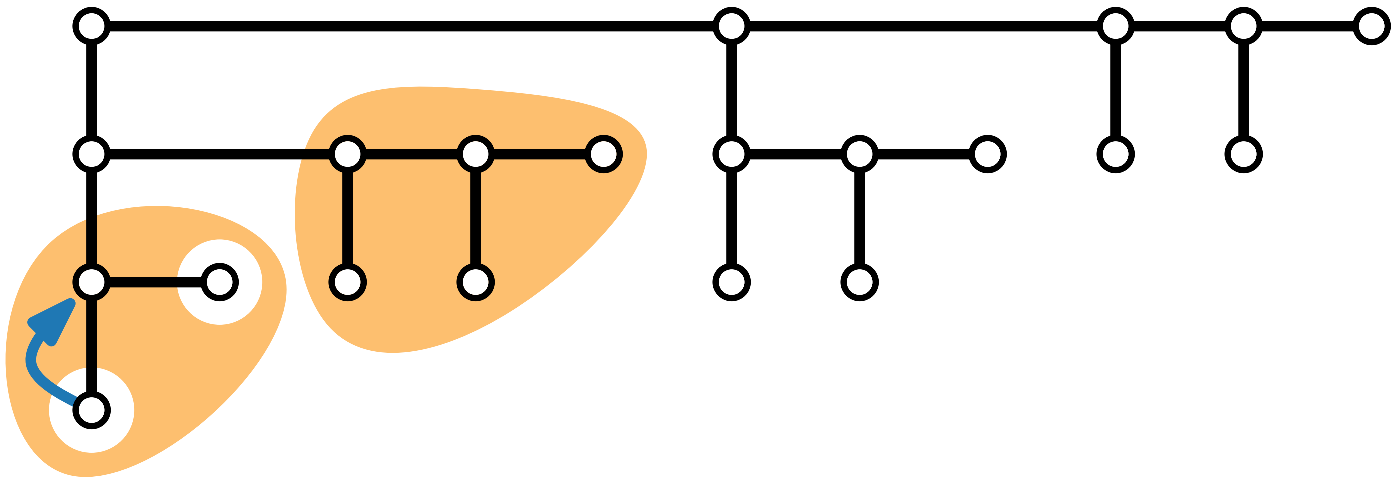
- width at most $n - 1$ and
- height at most

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

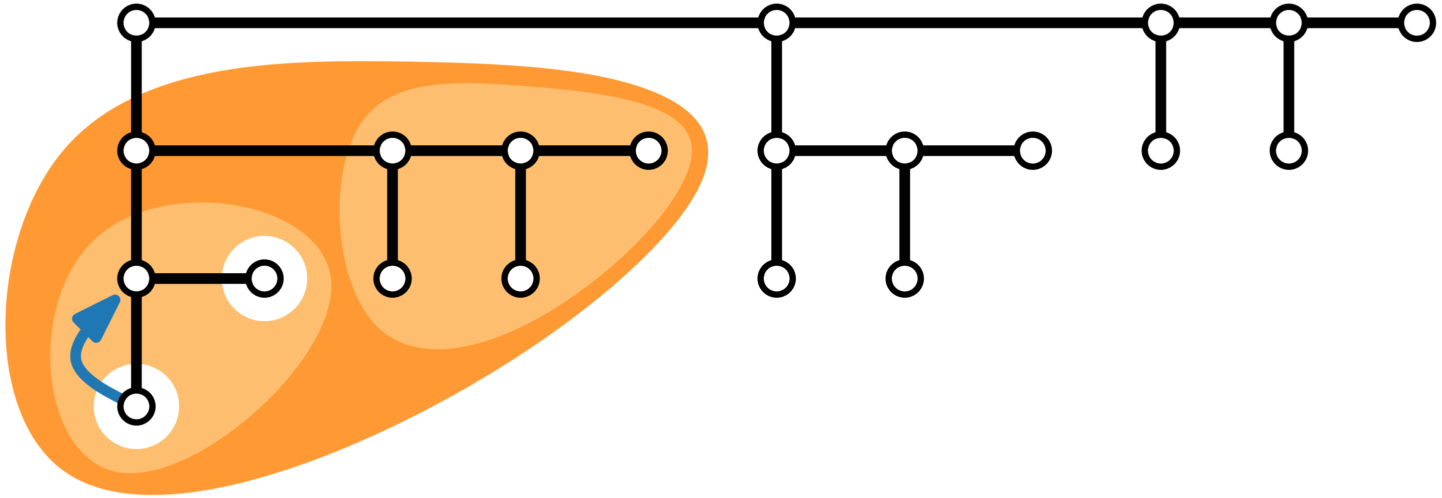
- width at most $n - 1$ and
- height at most

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

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

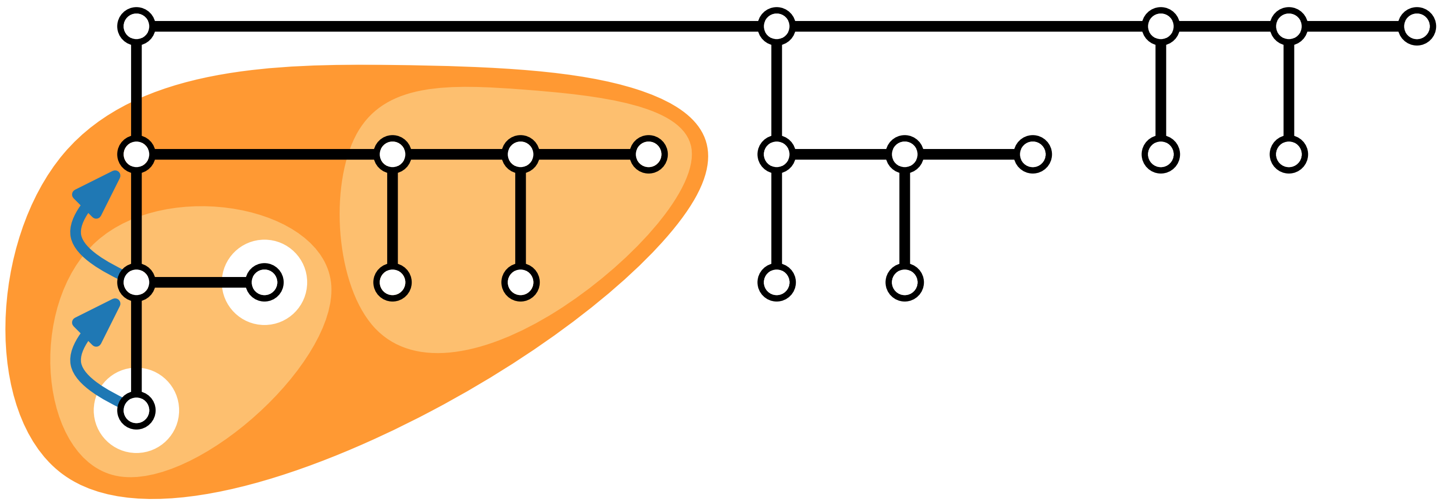
Right-heavy approach

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

  Size of subtree $:=$ number of vertices

at least $\cdot 2$

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

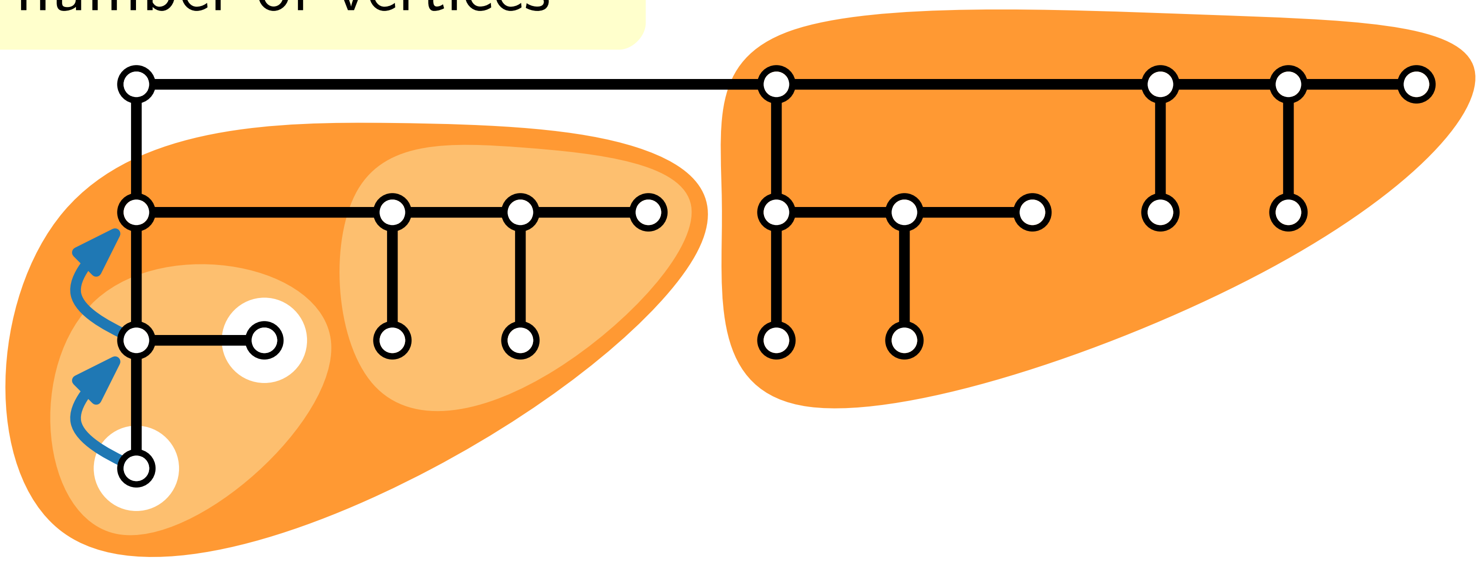
Right-heavy approach

- 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 ·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

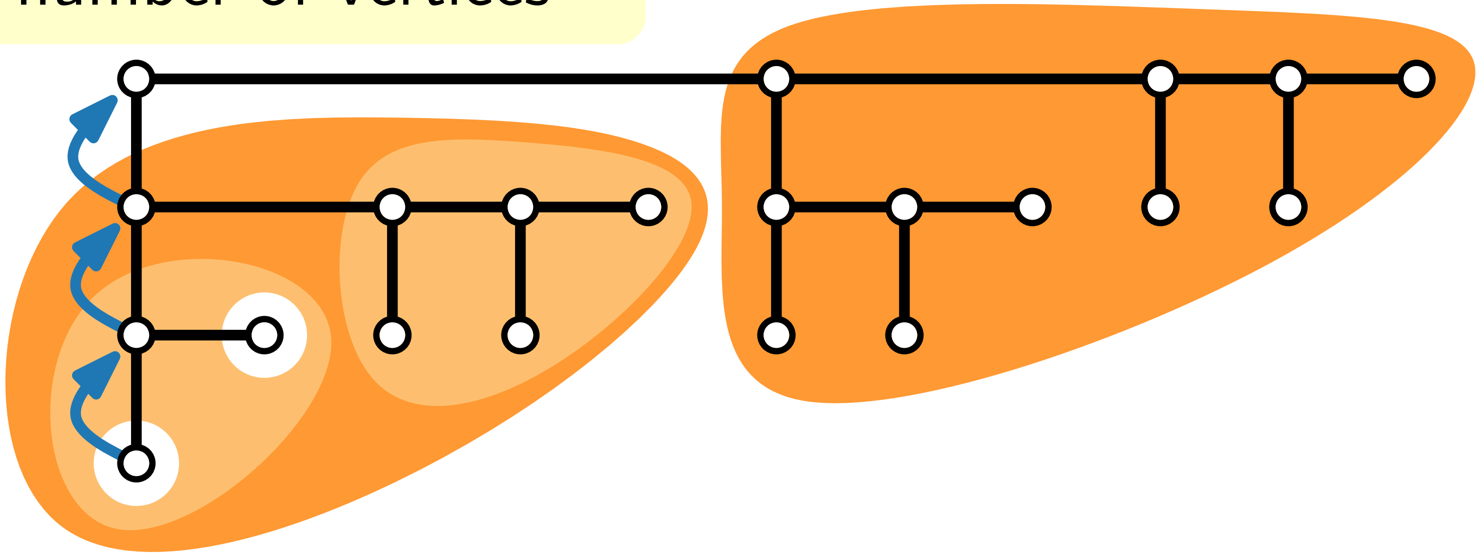
- Always apply horizontal combination
- Place the larger subtree to the right $\leftarrow$ *This can change the embedding!*

  Size of subtree := number of vertices

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

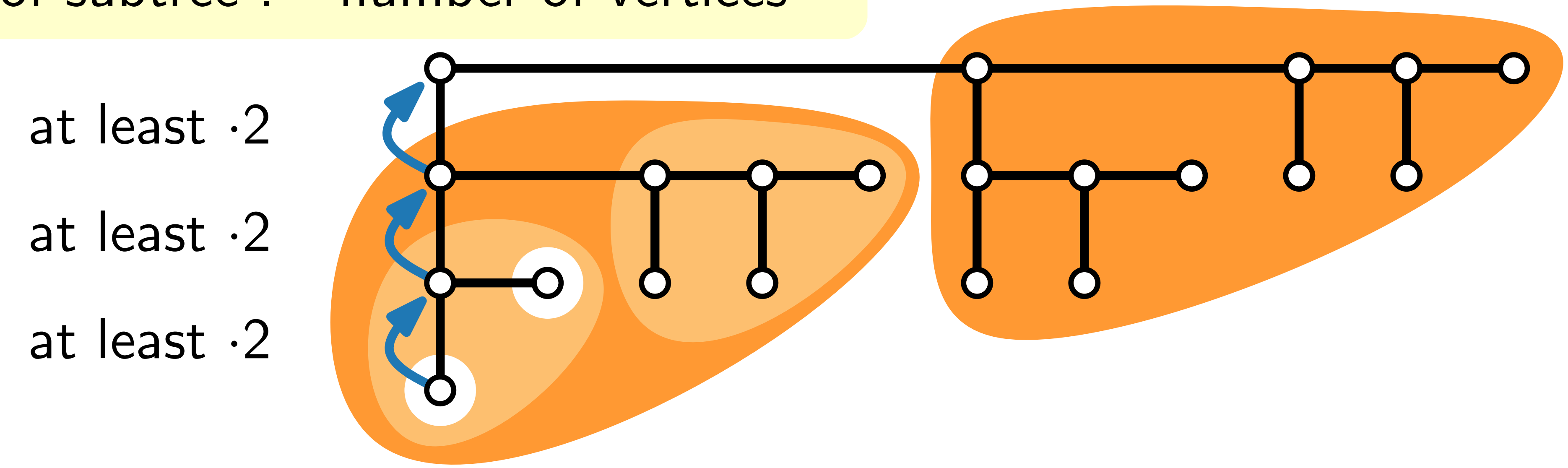
- width at most $n-1$ and
- height at most

HV-Drawings – Right-Heavy HV-Layout

Right-heavy approach

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

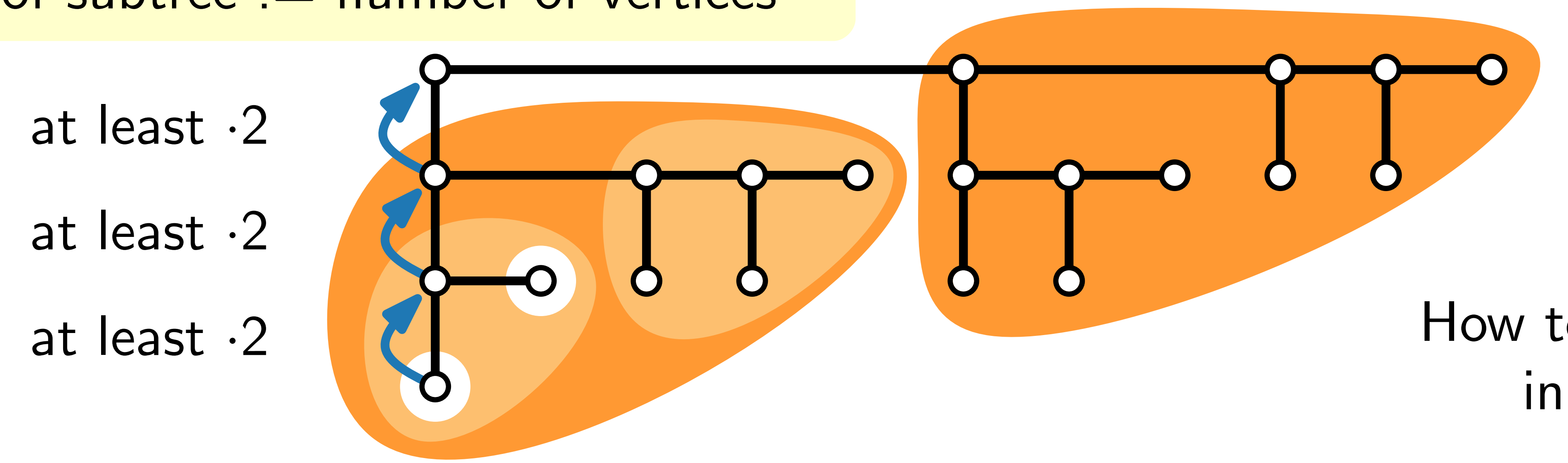
- width at most $n - 1$ and
- height at most $\log n$.

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 $\log n$.

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 $\log n$

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 $\log n$
- 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 $\log n$
- Area is in $\mathcal{O}(n \log n)$
- 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 $\log n$
- Area is in $\mathcal{O}(n \log n)$
- 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 $\log n$
- Area is in $\mathcal{O}(n \log n)$
- 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 $\log n$
- Area is in $\mathcal{O}(n \log n)$
- Simply and axially isomorphic subtrees have congruent drawings up to translation

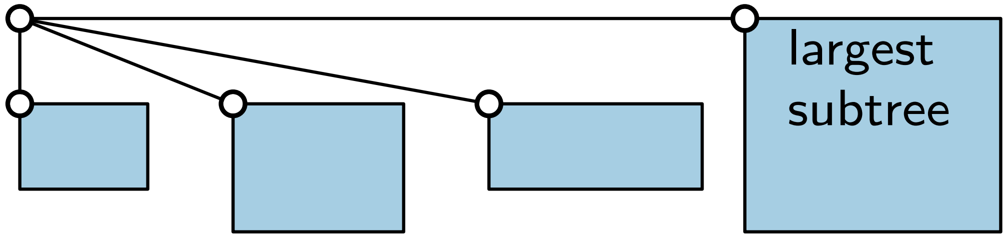
General rooted tree

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 $\log n$
- Area is in $\mathcal{O}(n \log n)$
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree

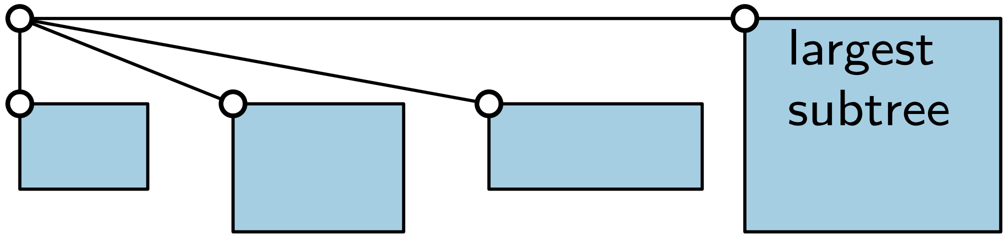


HV-Drawings – Result

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 $\log n$
- Area is in $\mathcal{O}(n \log n)$
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree

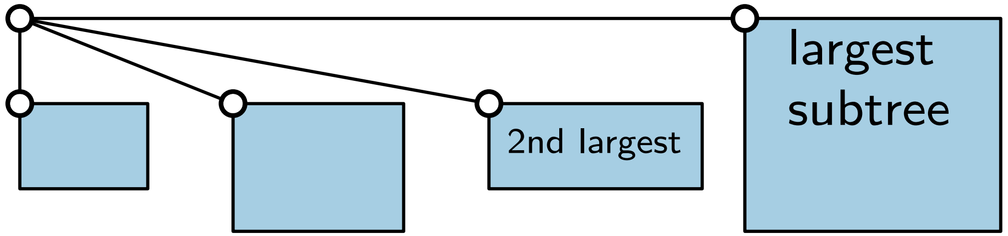


HV-Drawings – Result

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 $\log n$
- Area is in $\mathcal{O}(n \log n)$
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree

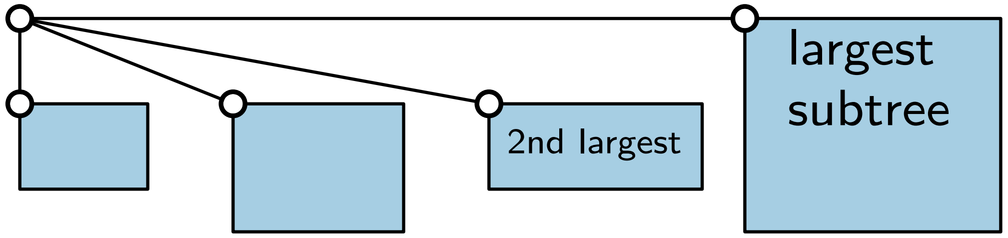


HV-Drawings – Result

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 $\log n$
- Area is in $\mathcal{O}(n \log n)$
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree



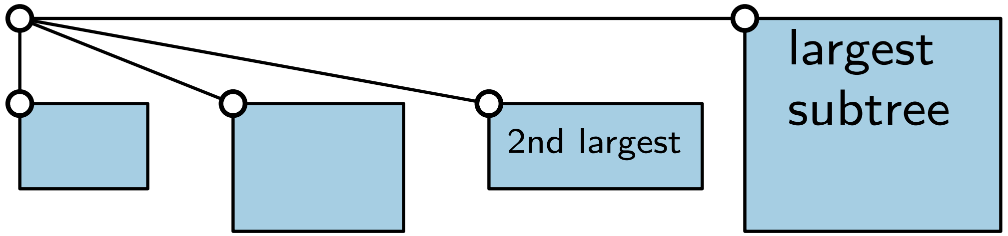
Optimal area?

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 $\log n$
- Area is in $\mathcal{O}(n \log n)$
- Simply and axially isomorphic subtrees have congruent drawings up to translation

General rooted tree

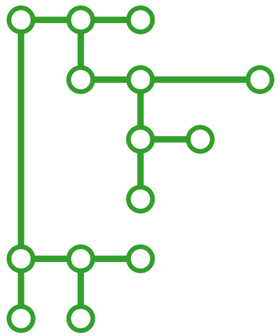


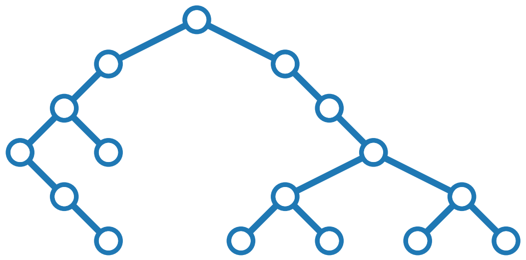
Optimal area?

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

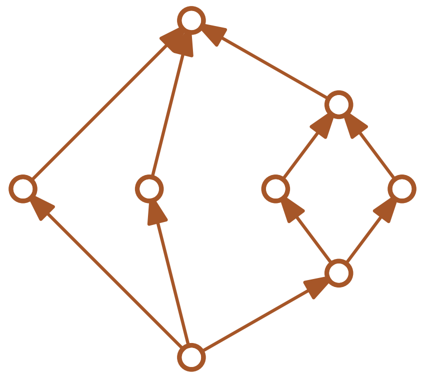
Visualization of Graphs

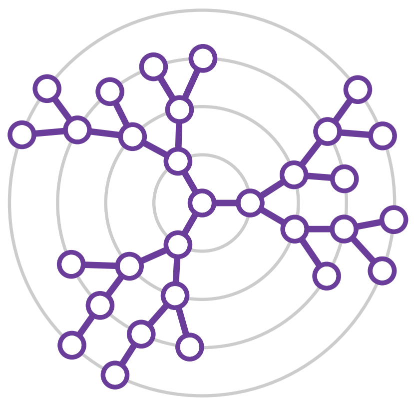
Lecture 1b:
Drawing Trees and Series-Parallel Graphs



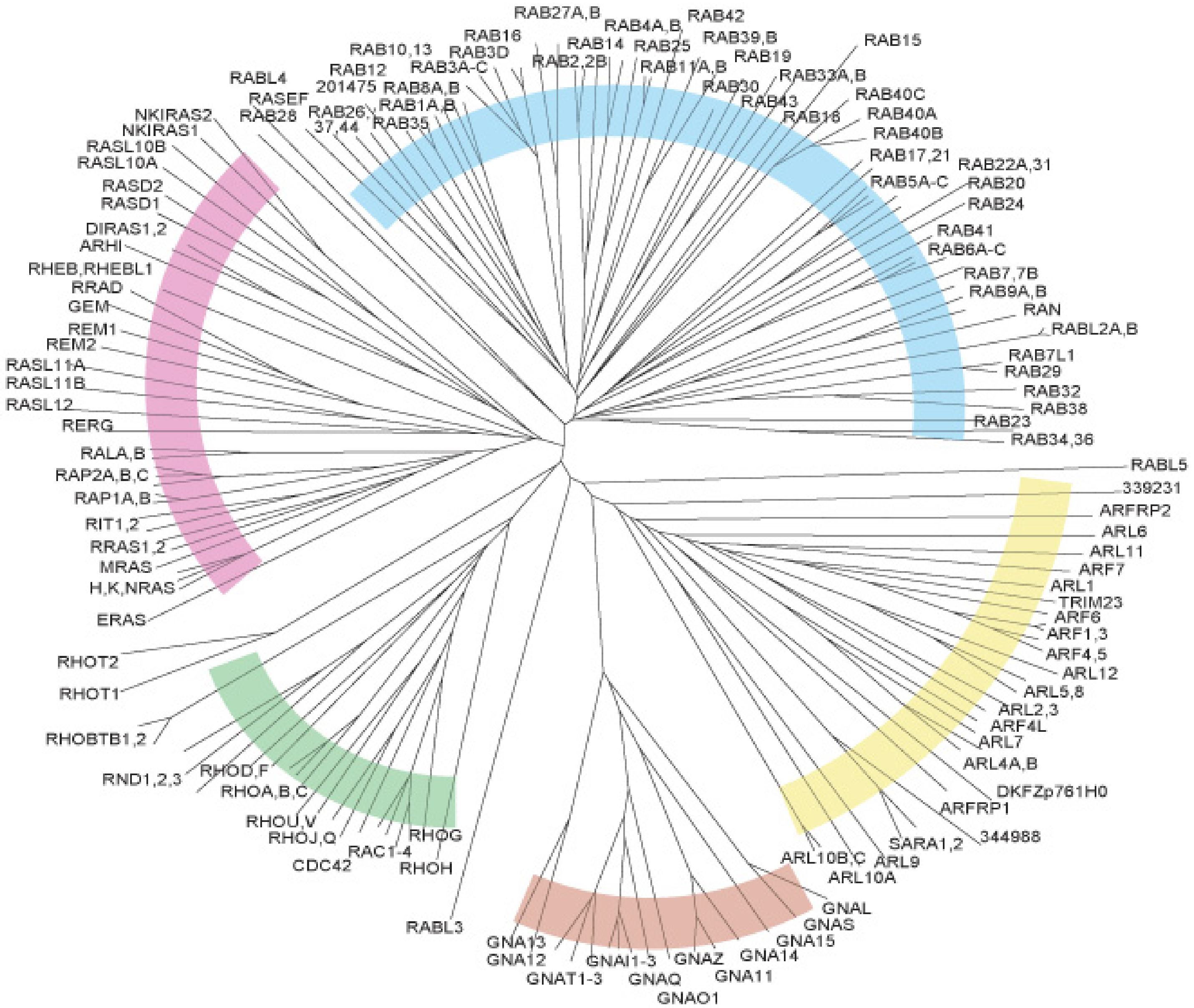


Part III:
Radial Layouts



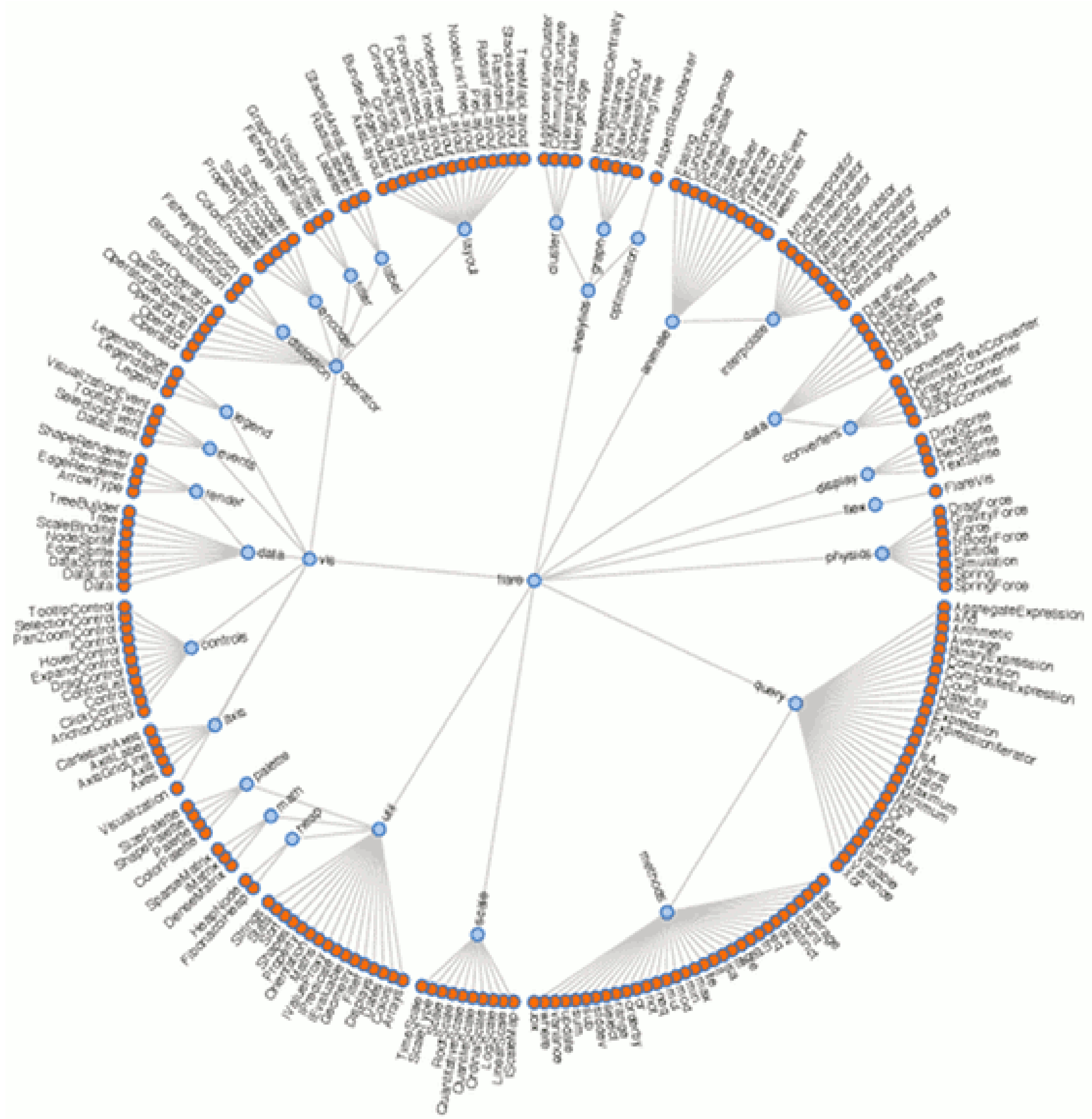


Radial Layouts – Applications

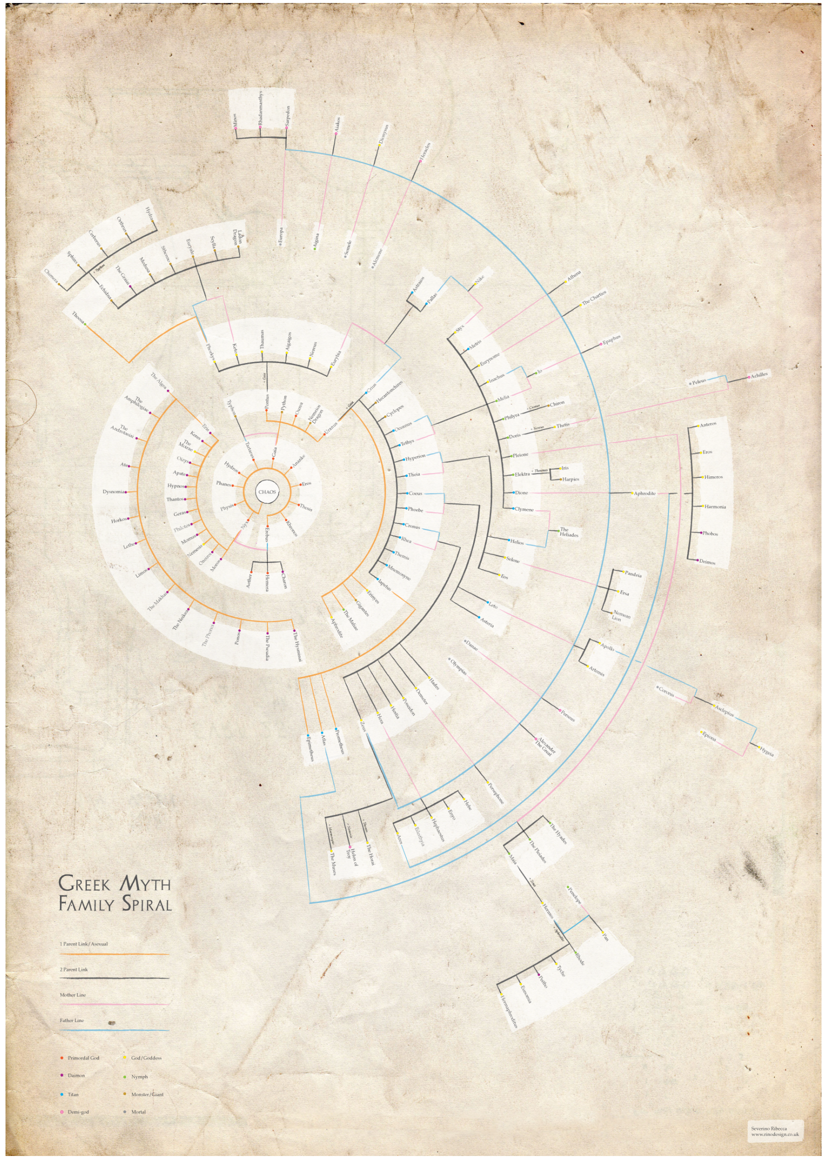


Phylogenetic tree
by Colicelli, ScienceSignaling, 2004

Radial Layouts – Applications

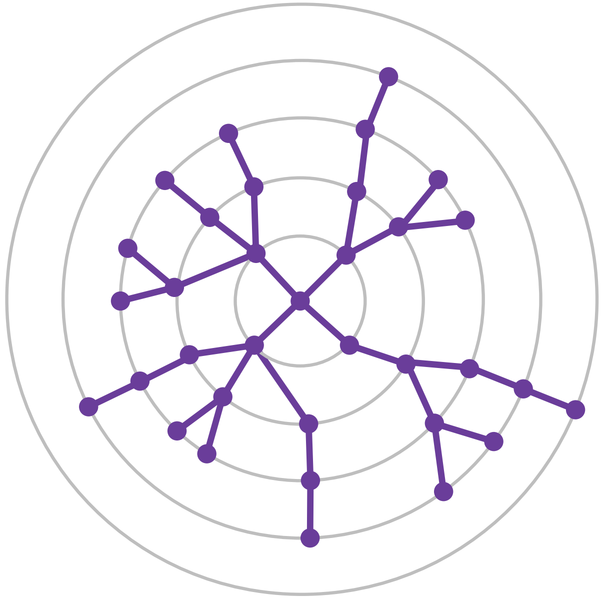


Flare Visualization Toolkit code structure
by Heer, Bostock and Ogievetsky, 2010



Greek Myth Family
by Ribecca, 2011

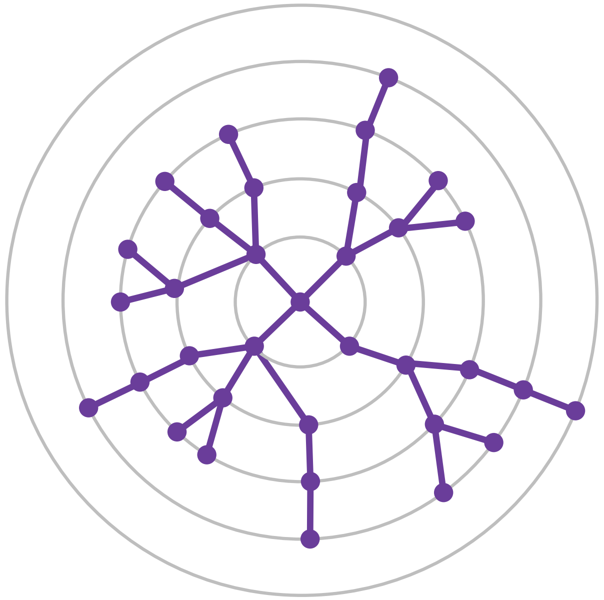
Radial Layouts – Drawing Style



Drawing conventions

Drawing aesthetics

Radial Layouts – Drawing Style

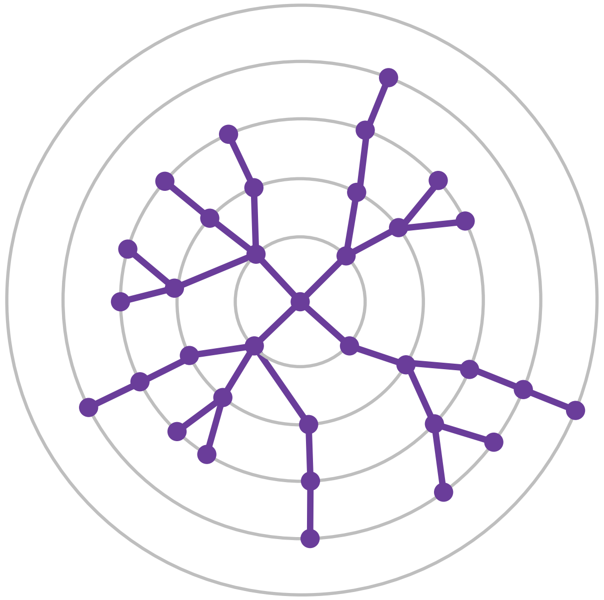


Drawing conventions

- Vertices lie on circular layers according to their depth

Drawing aesthetics

Radial Layouts – Drawing Style

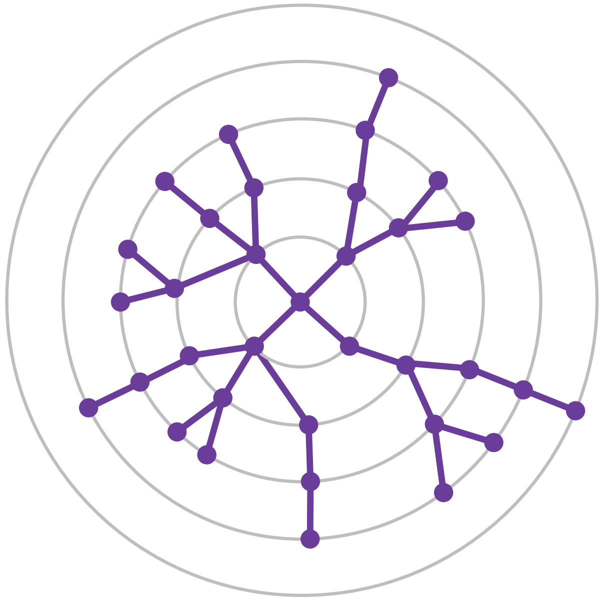


Drawing conventions

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

Drawing aesthetics

Radial Layouts – Drawing Style



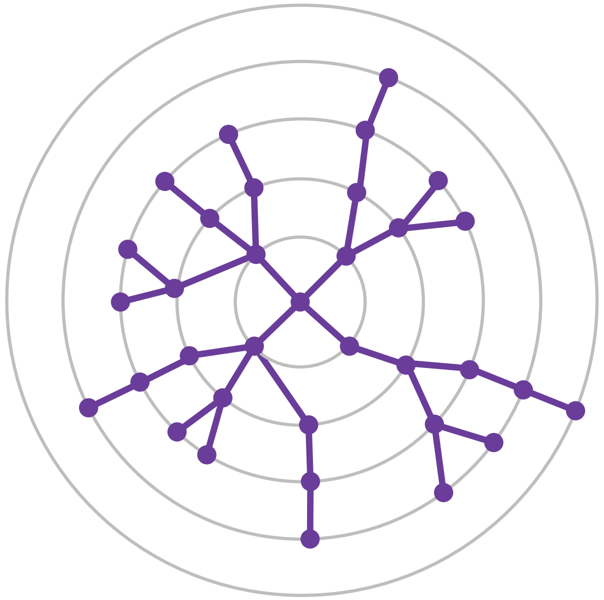
Drawing conventions

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

Drawing aesthetics

- Distribution of the vertices

Radial Layouts – Drawing Style



Drawing conventions

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

Drawing aesthetics

- Distribution of the vertices

How can an algorithm optimize the distribution of the vertices?

Radial Layouts – Algorithm Attempt

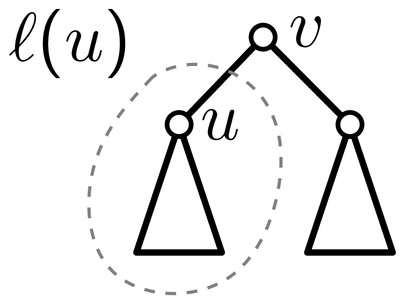
Idea

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

Radial Layouts – Algorithm Attempt

Idea

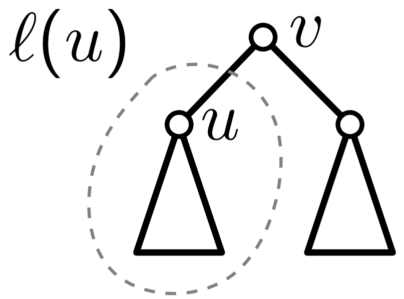
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

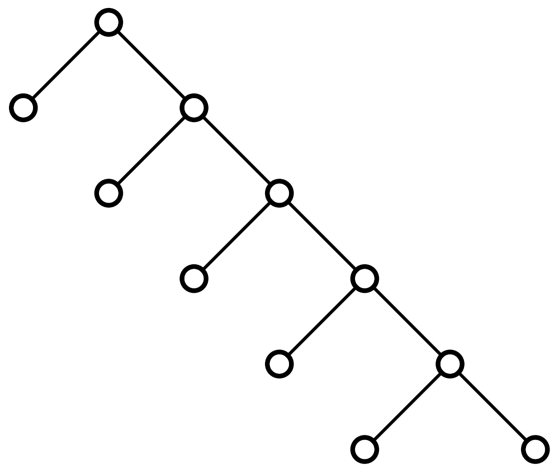


Radial Layouts – Algorithm Attempt

Idea

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

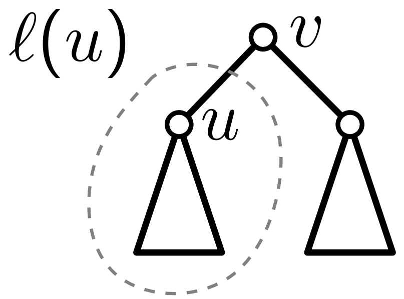


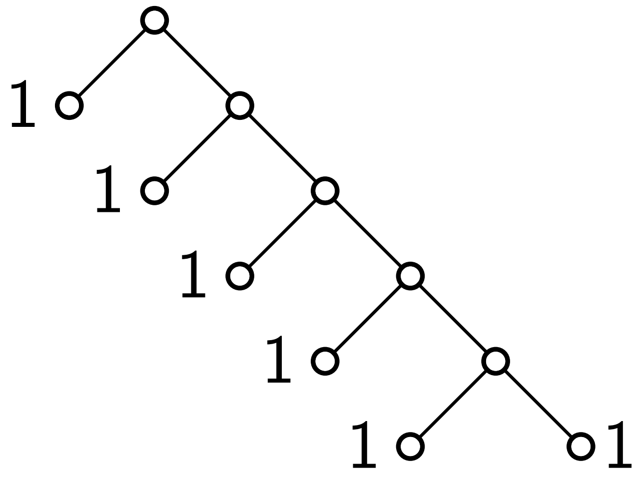


Radial Layouts – Algorithm Attempt

Idea

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

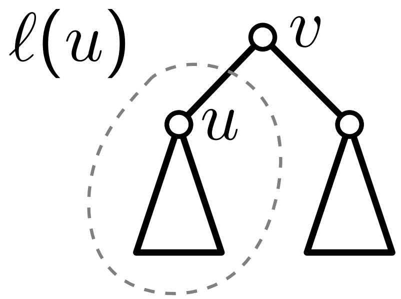


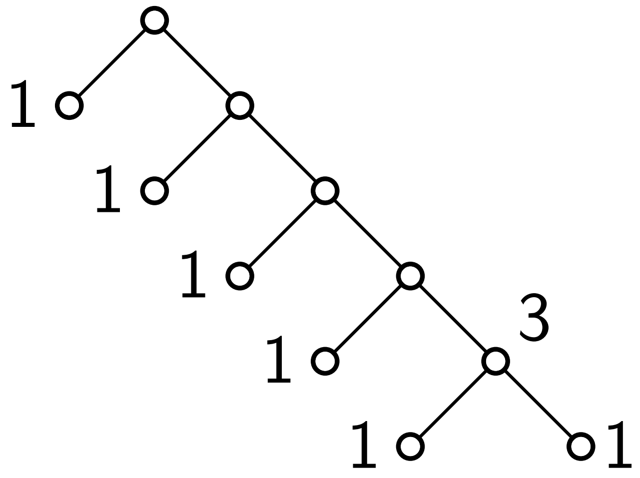


Radial Layouts – Algorithm Attempt

Idea

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

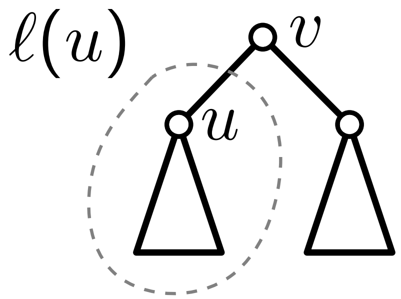


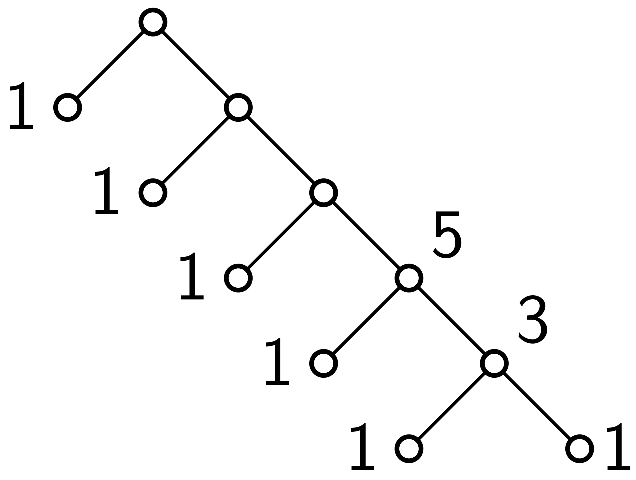


Radial Layouts – Algorithm Attempt

Idea

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

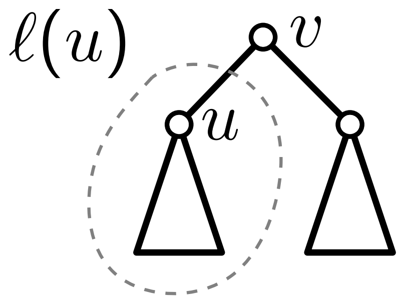


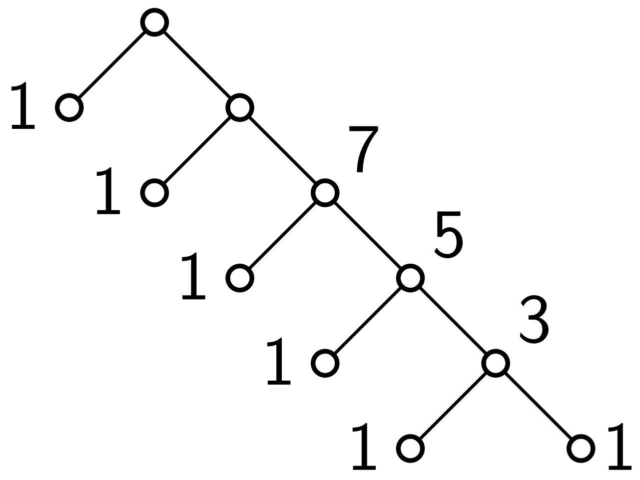


Radial Layouts – Algorithm Attempt

Idea

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

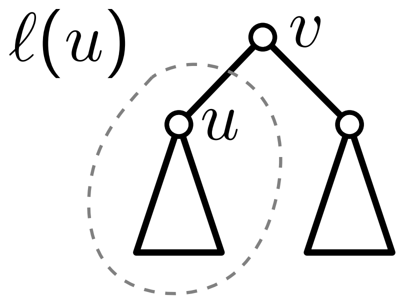


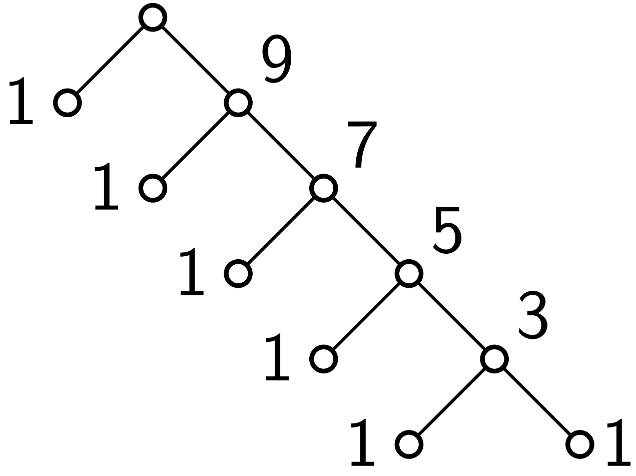


Radial Layouts – Algorithm Attempt

Idea

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

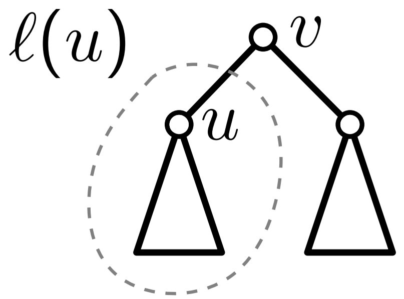


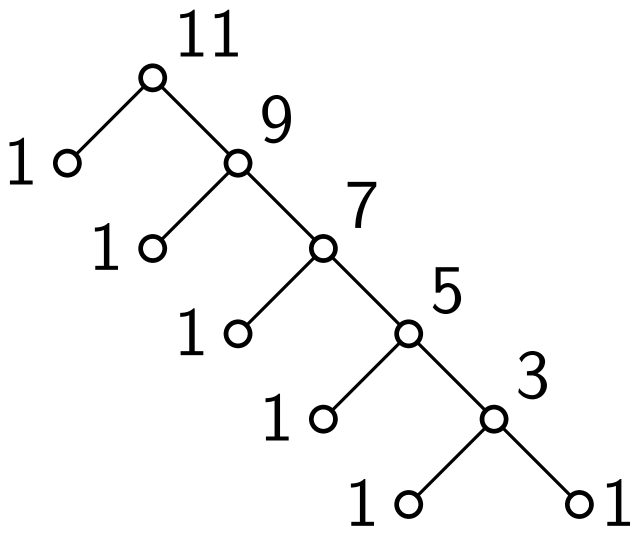


Radial Layouts – Algorithm Attempt

Idea

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



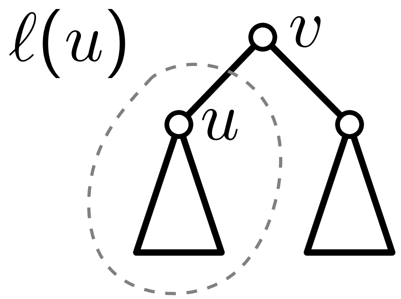


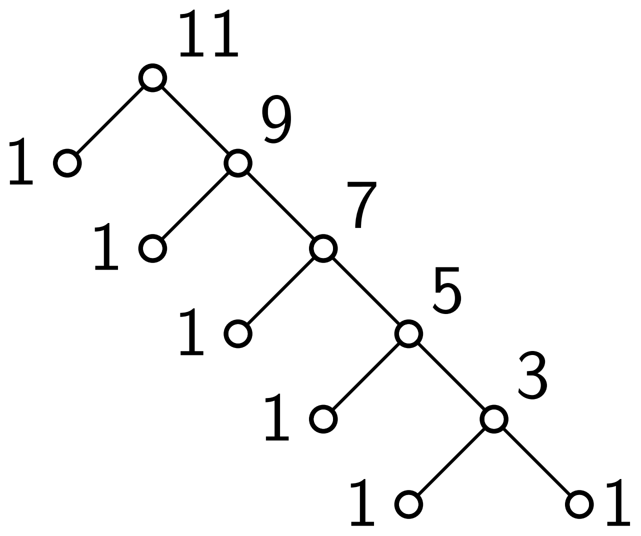
Radial Layouts – Algorithm Attempt

Idea

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

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



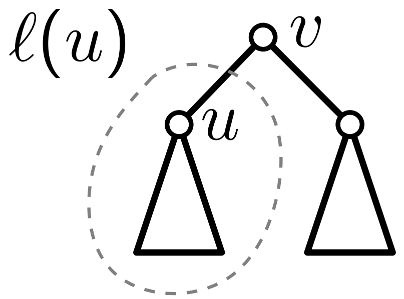


Radial Layouts – Algorithm Attempt

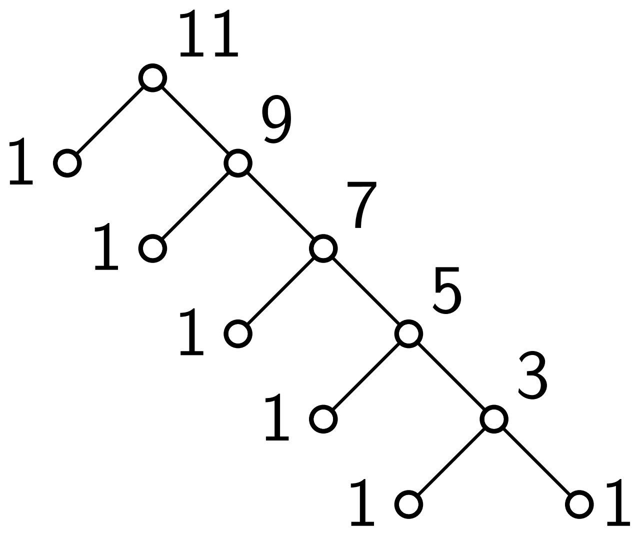
Idea

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

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



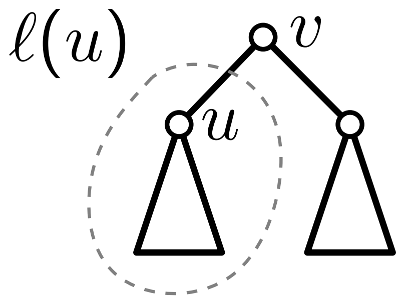
- Place u in middle of area



Radial Layouts – Algorithm Attempt

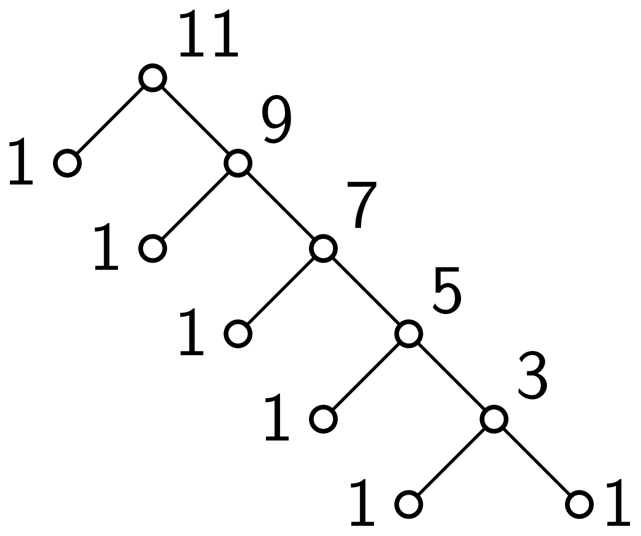
Idea

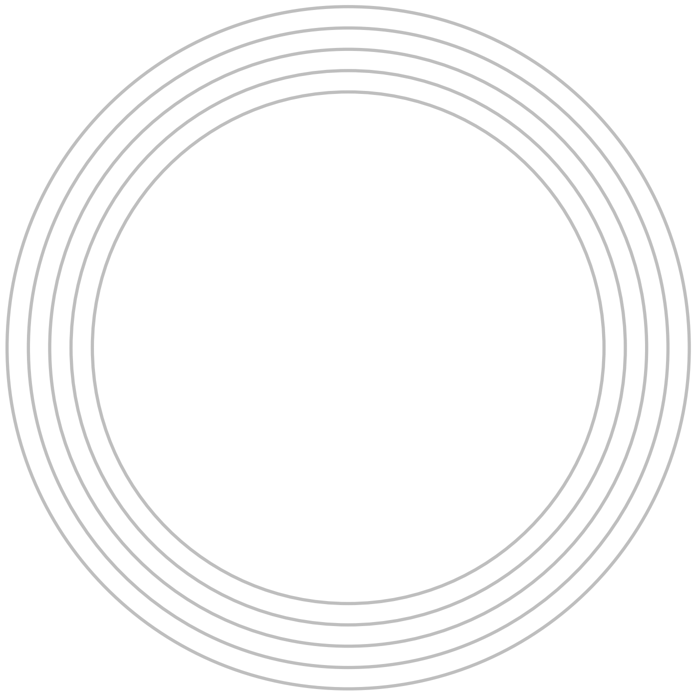
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



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

- Place u in middle of area



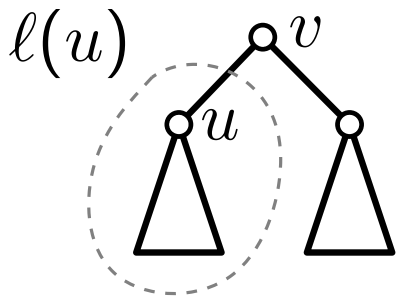


Radial Layouts – Algorithm Attempt

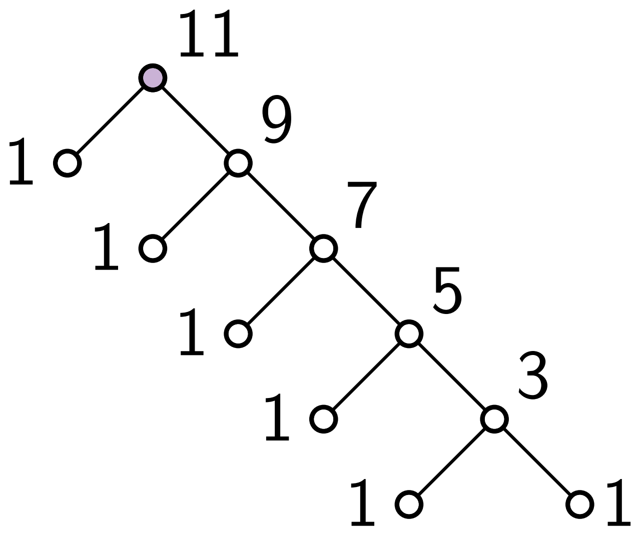
Idea

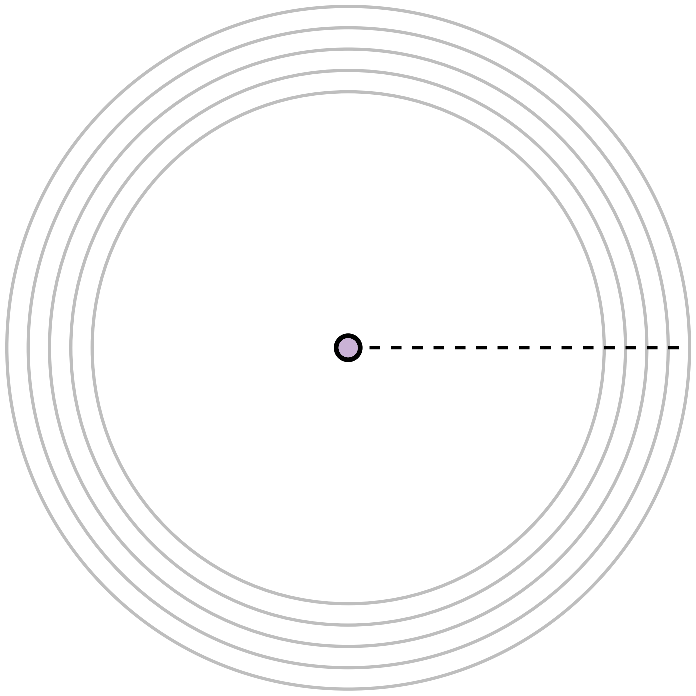
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

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



- Place u in middle of area

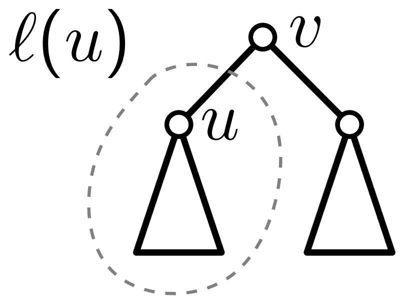




Radial Layouts – Algorithm Attempt

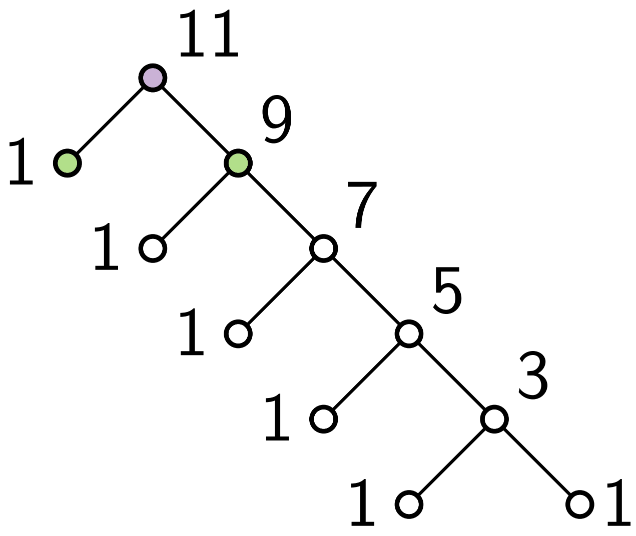
Idea

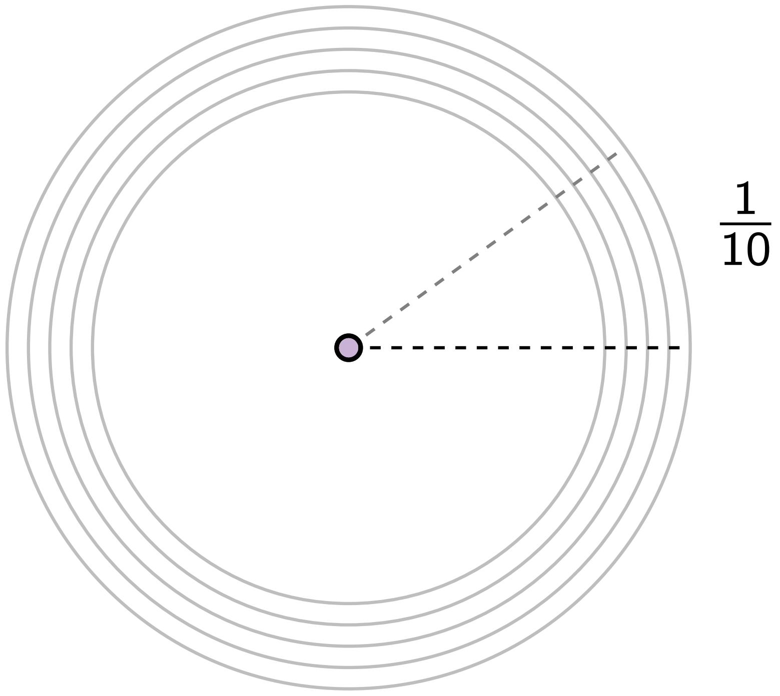
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



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

- Place u in middle of area

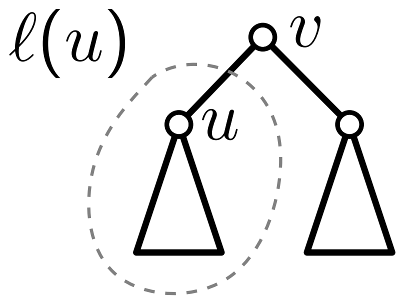




Radial Layouts – Algorithm Attempt

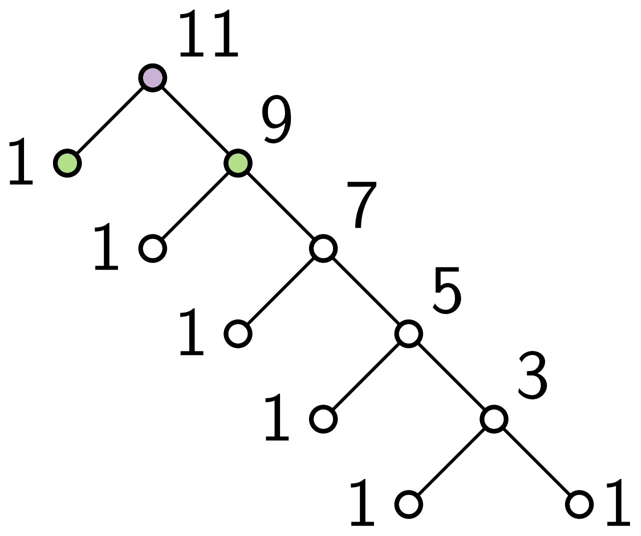
Idea

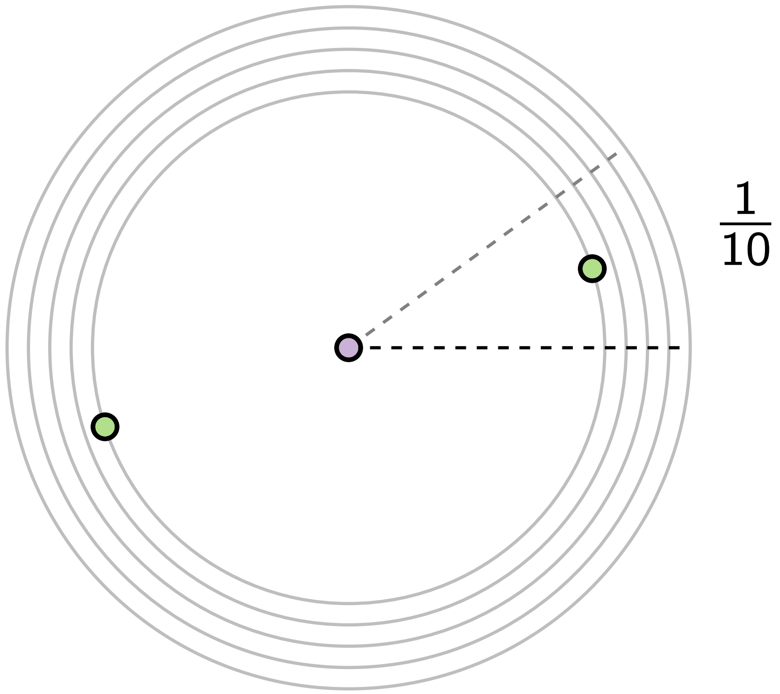
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



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

- Place u in middle of area

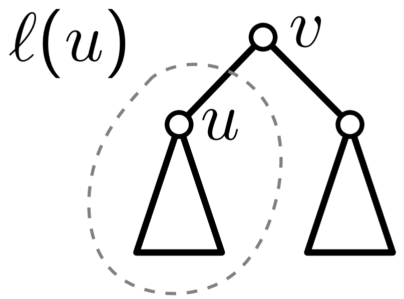




Radial Layouts – Algorithm Attempt

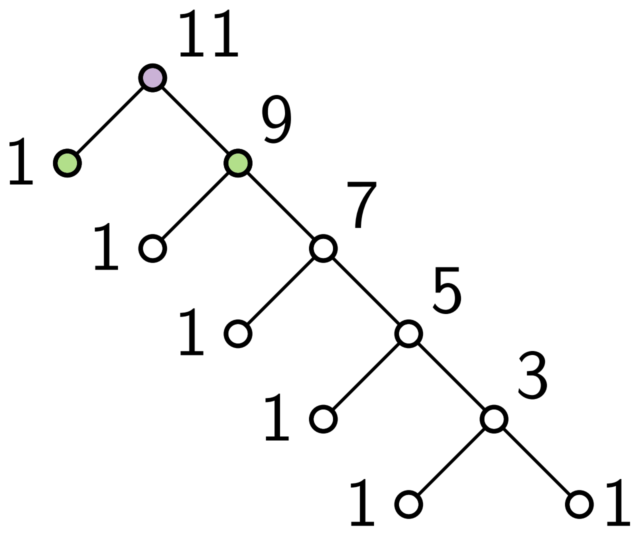
Idea

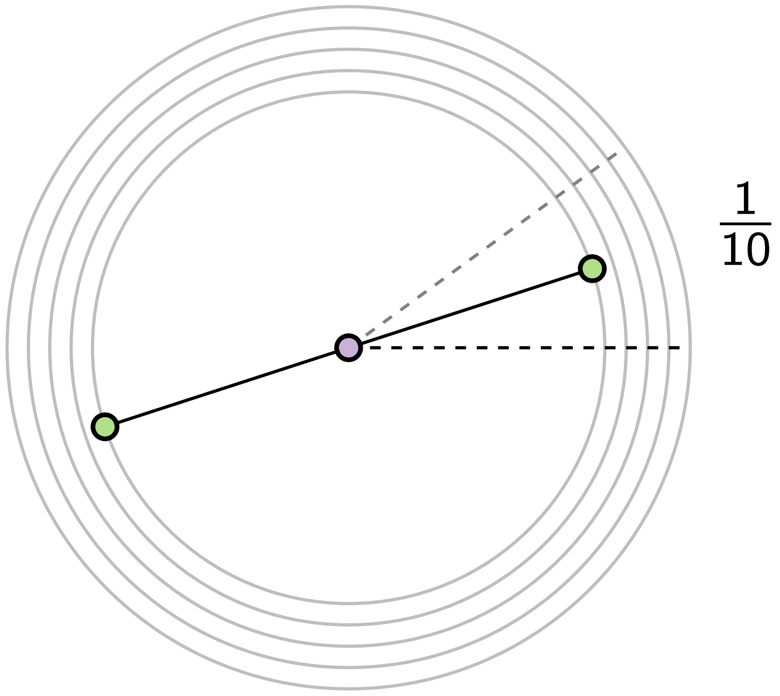
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



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

- Place u in middle of area

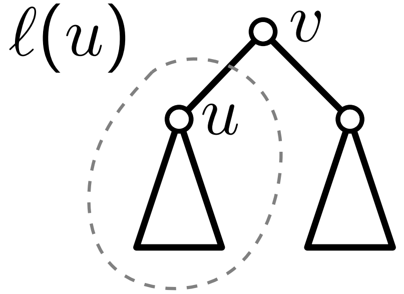




Radial Layouts – Algorithm Attempt

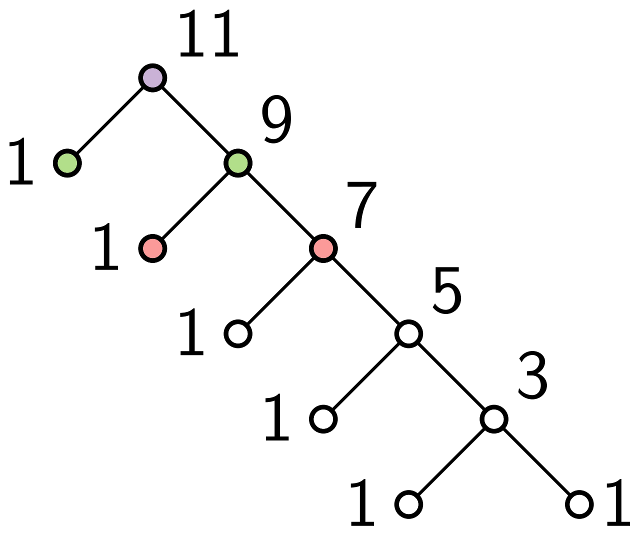
Idea

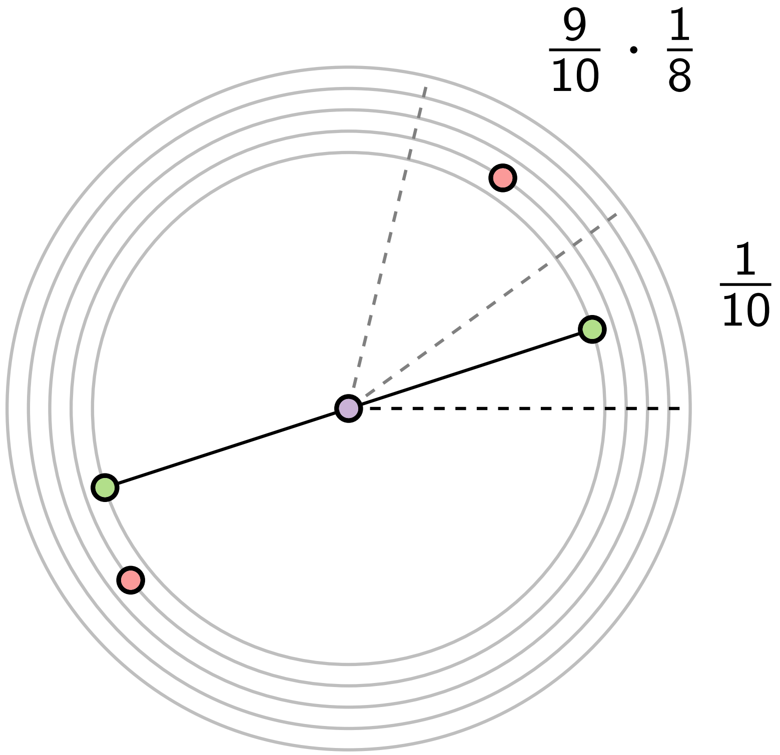
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



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

- Place u in middle of area



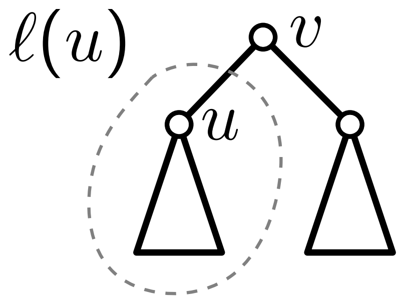


Radial Layouts – Algorithm Attempt

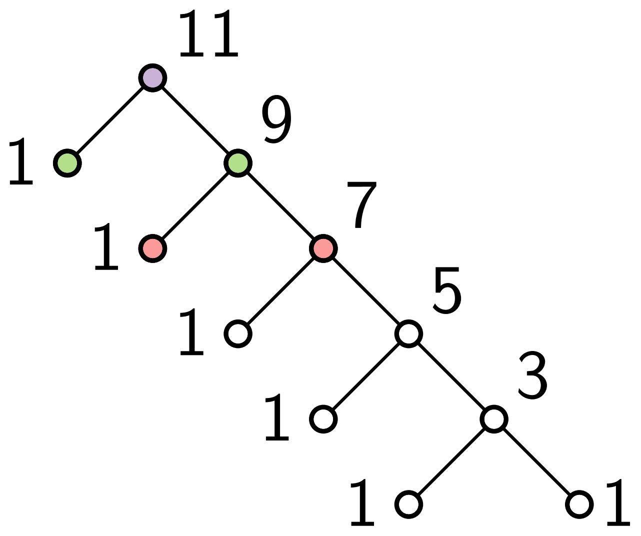
Idea

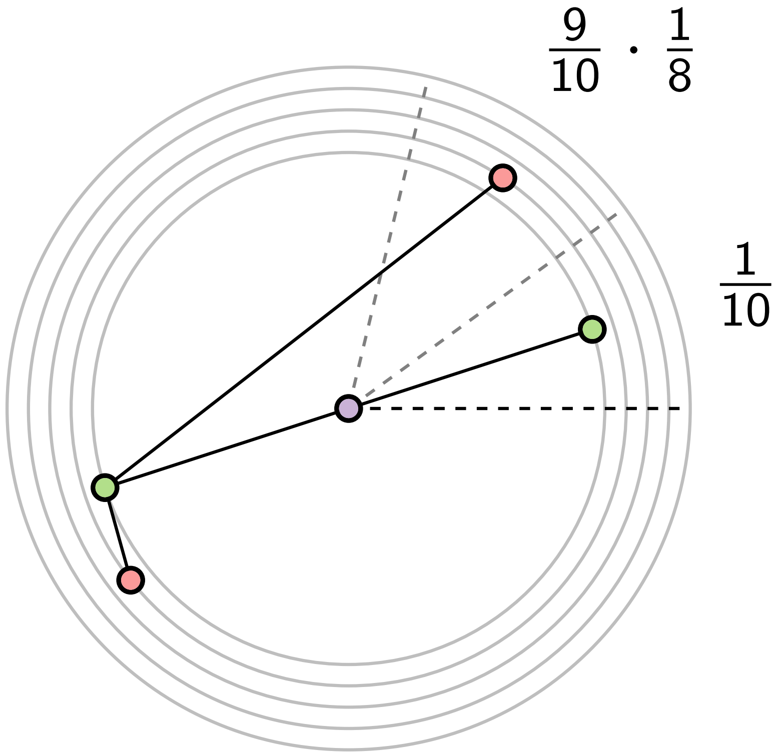
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:

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



- Place u in middle of area

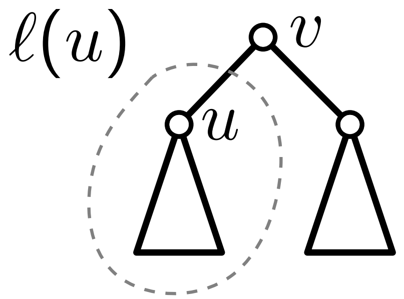




Radial Layouts – Algorithm Attempt

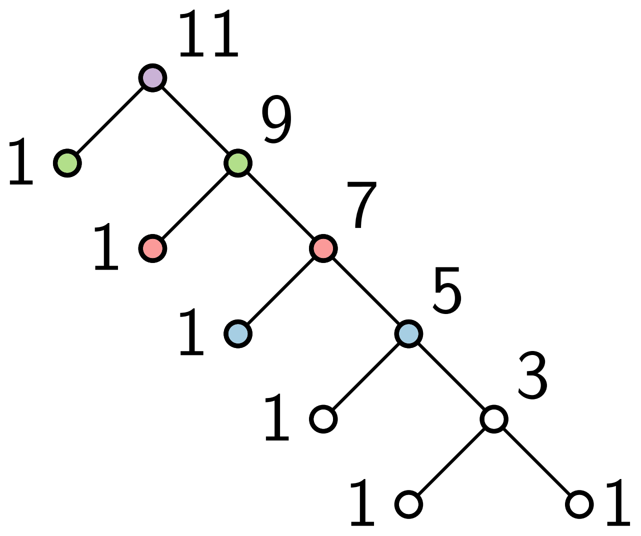
Idea

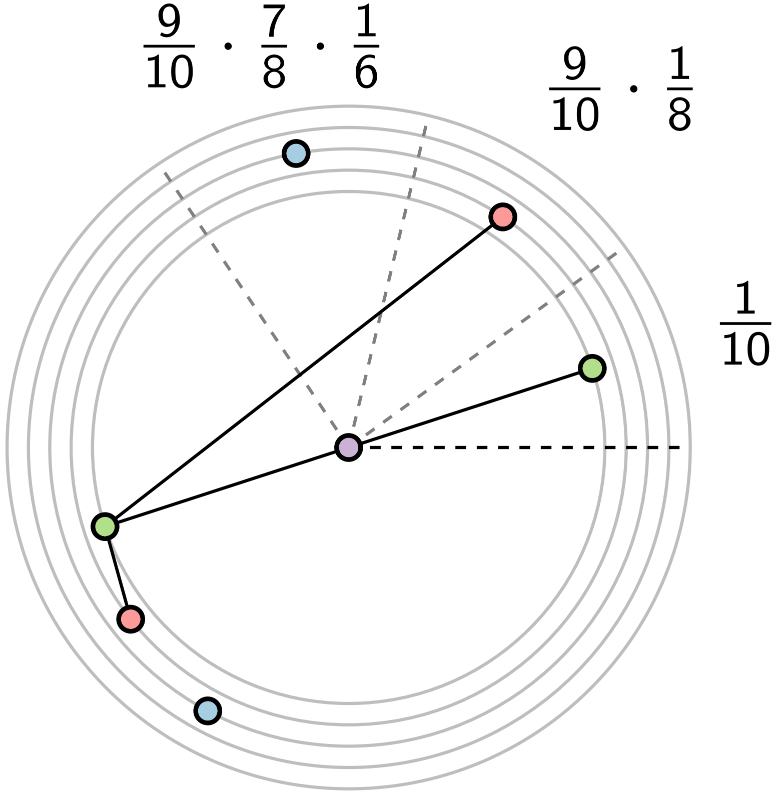
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



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

- Place u in middle of area

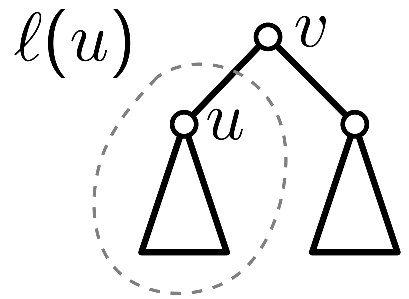




Radial Layouts – Algorithm Attempt

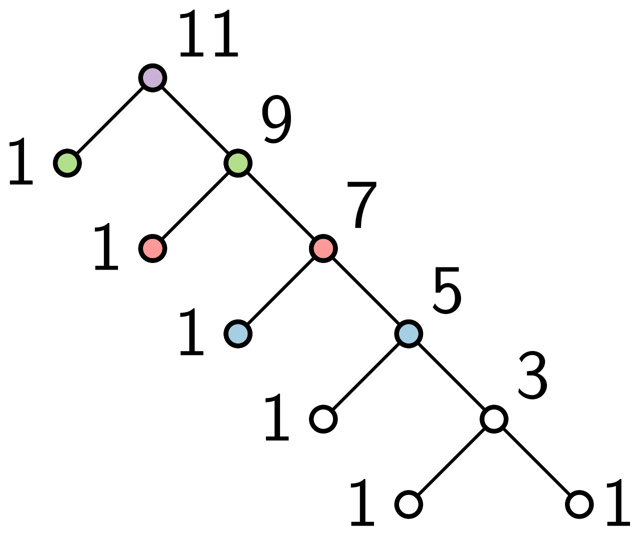
Idea

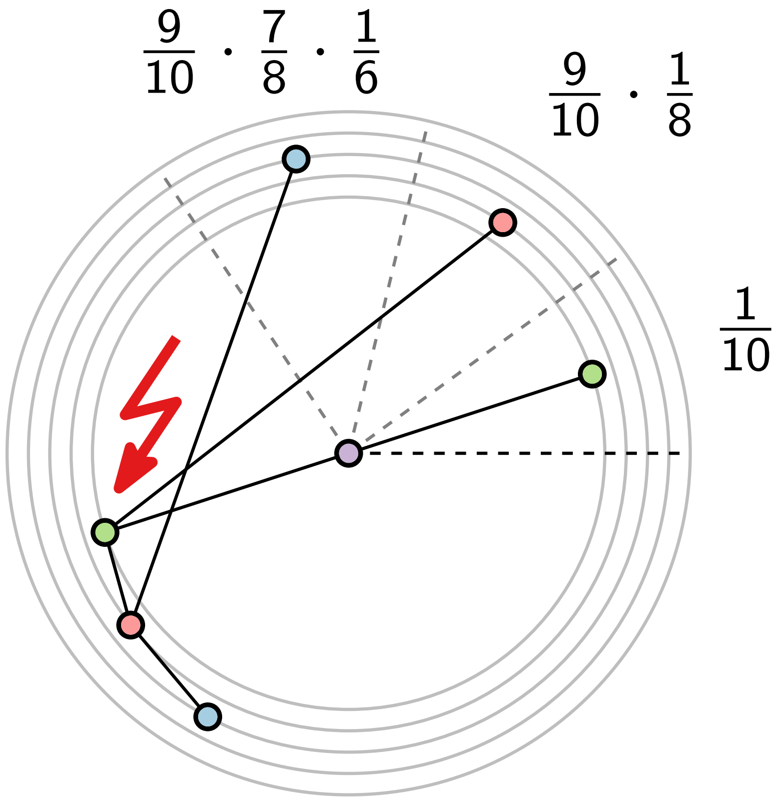
- Reserve area corresponding to size $\ell(u)$ of $T(u)$:



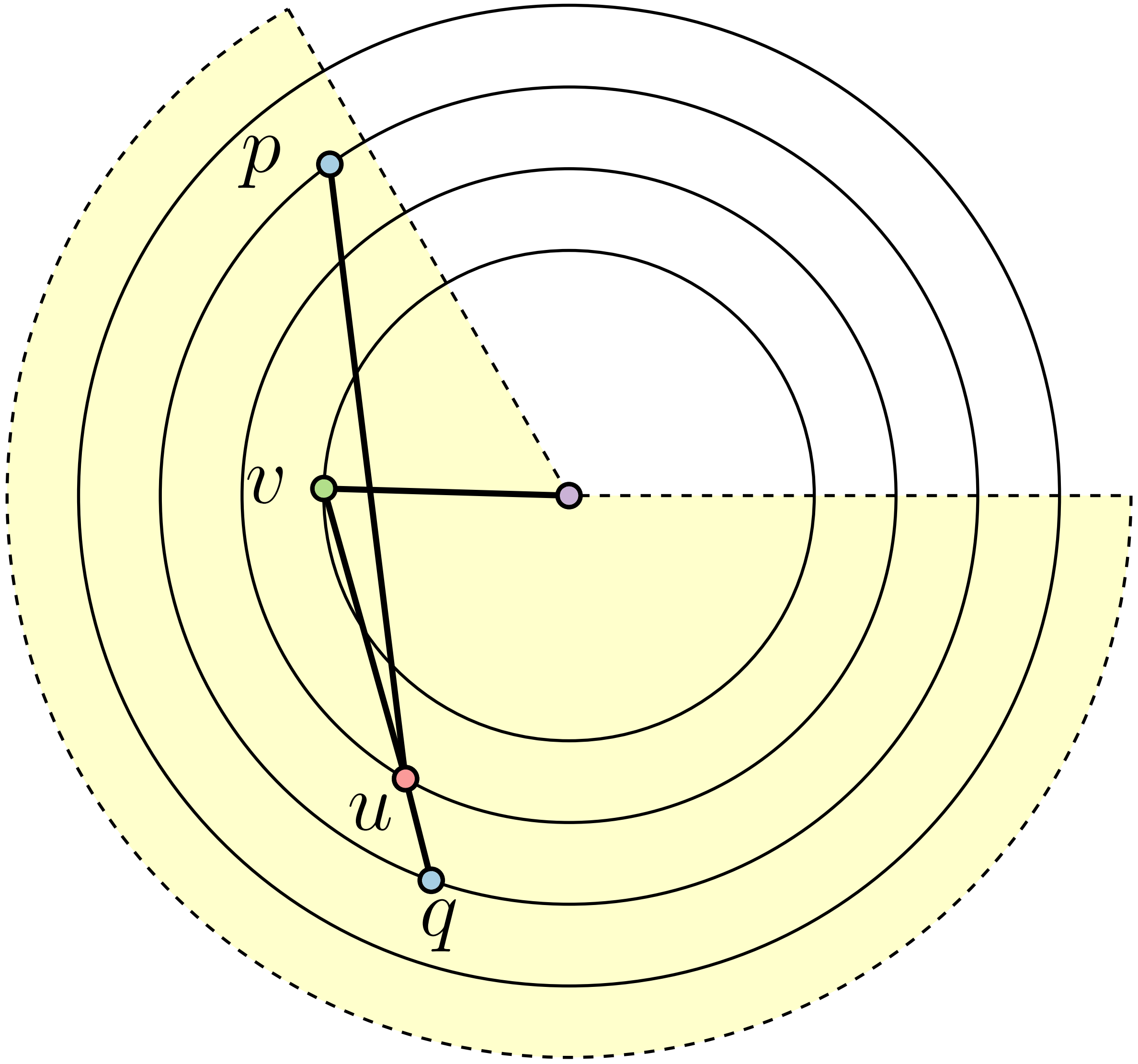
$$\tau_u = \frac{\ell(u)}{\ell(v) - 1}$$

- Place u in middle of area

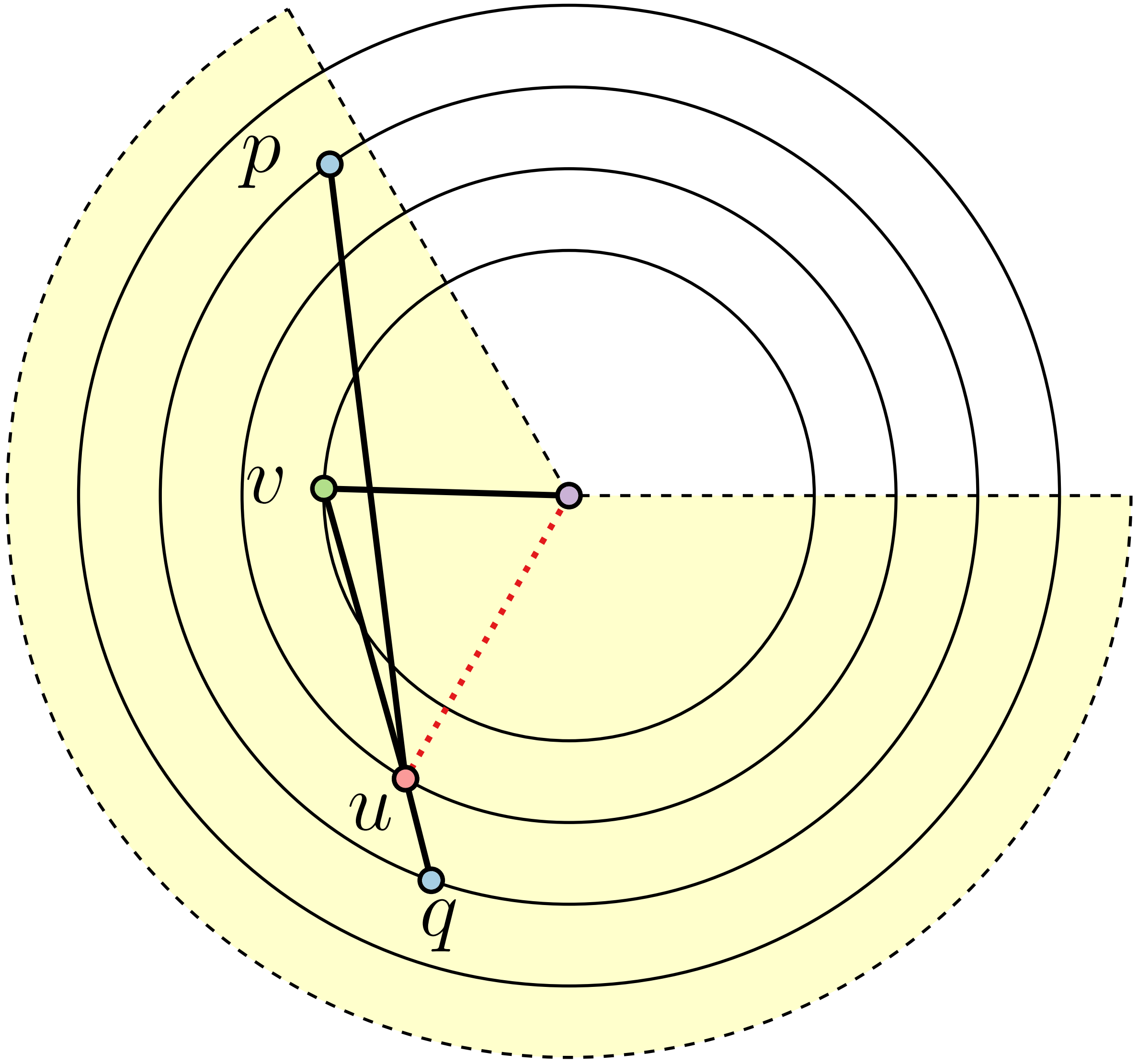




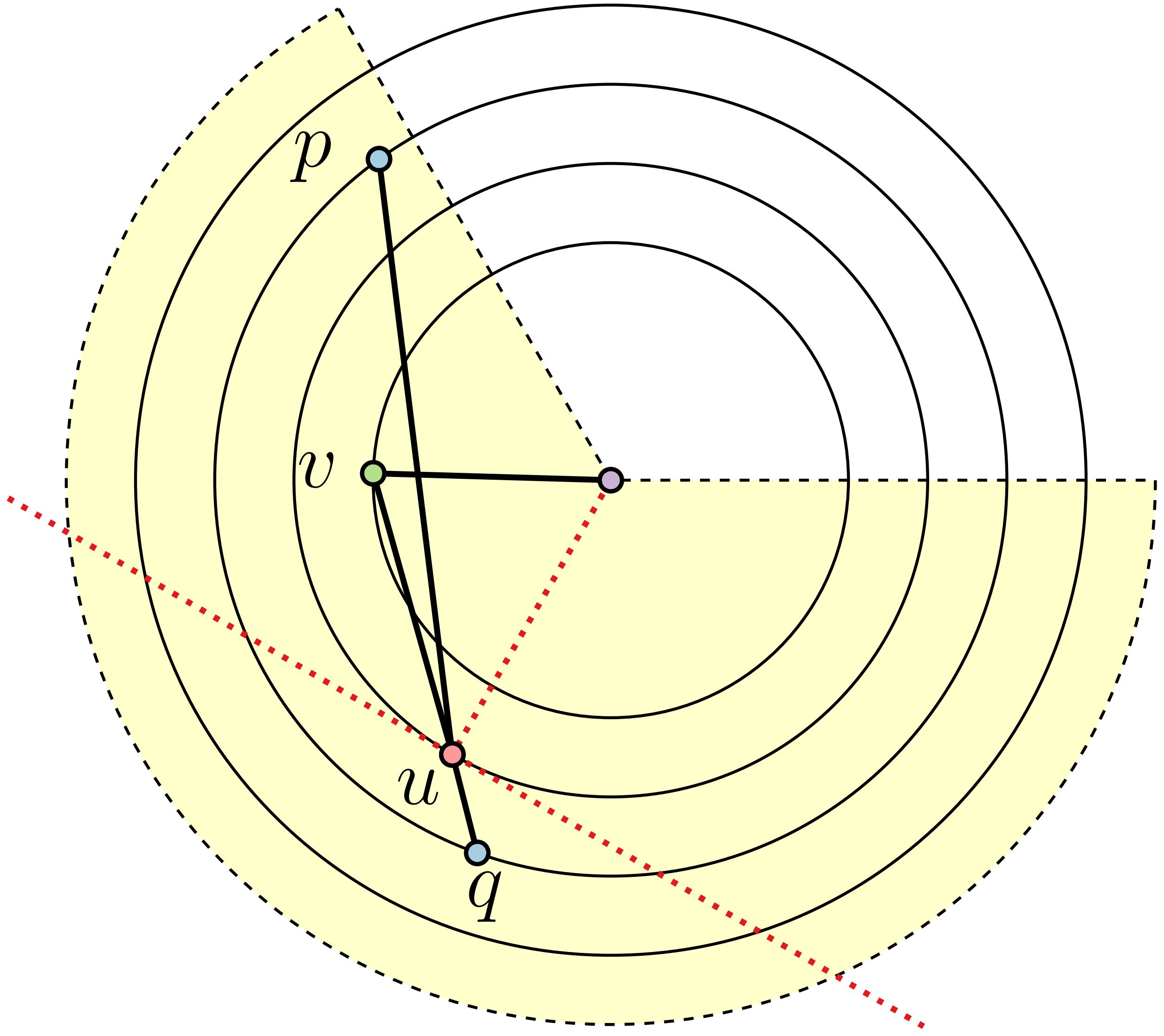
Radial Layouts – How To Avoid Crossings



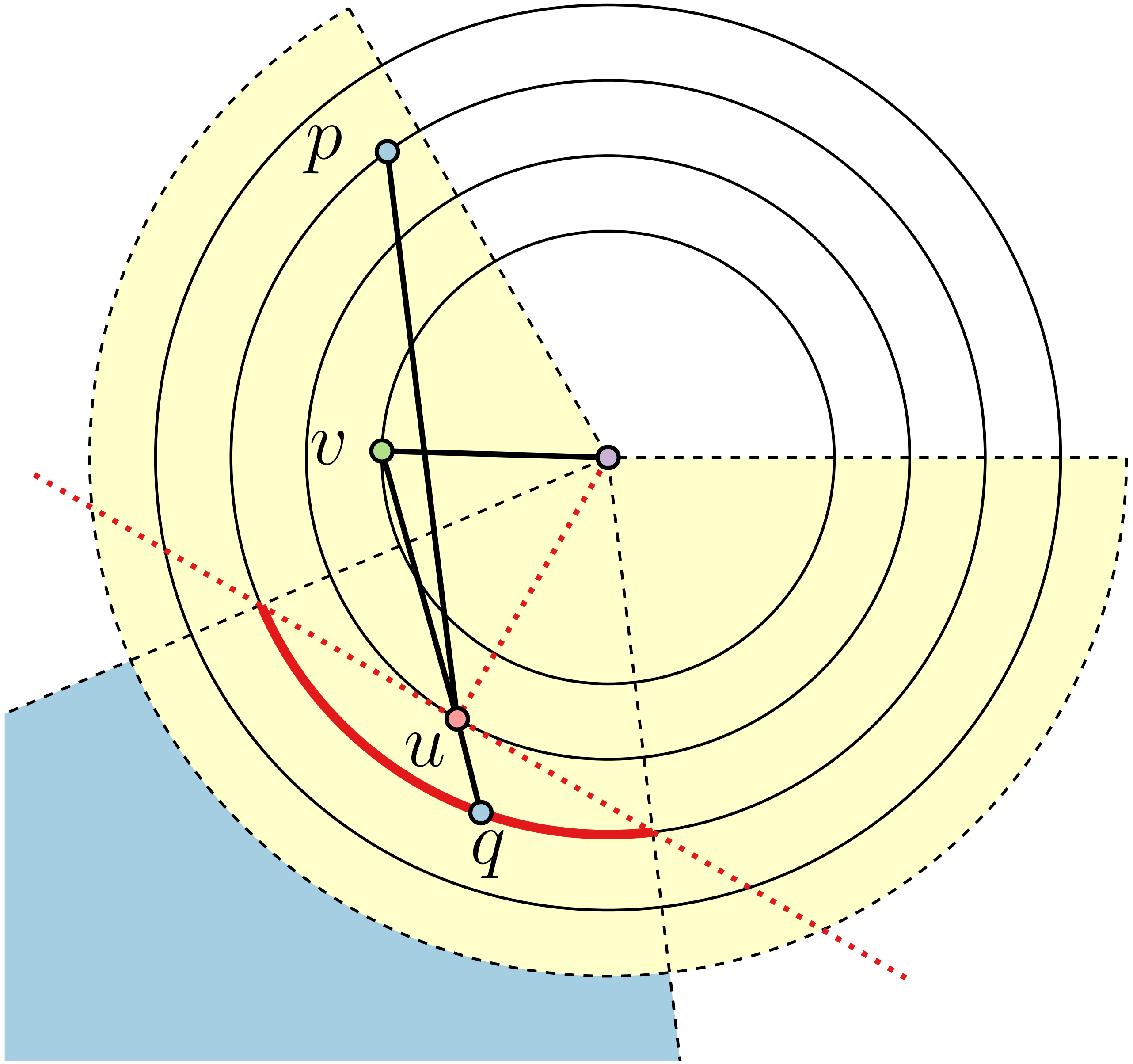
Radial Layouts – How To Avoid Crossings



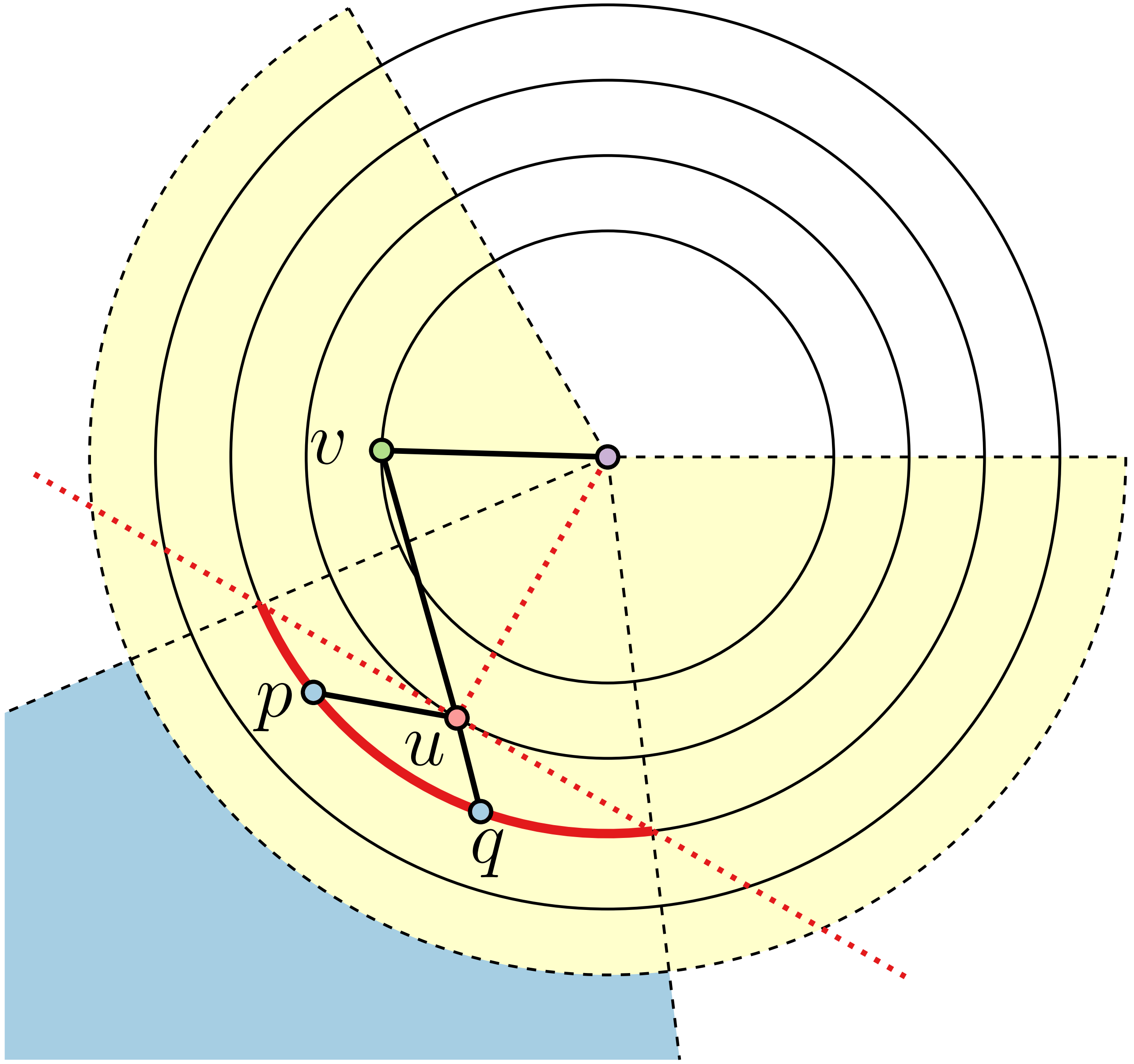
Radial Layouts – How To Avoid Crossings



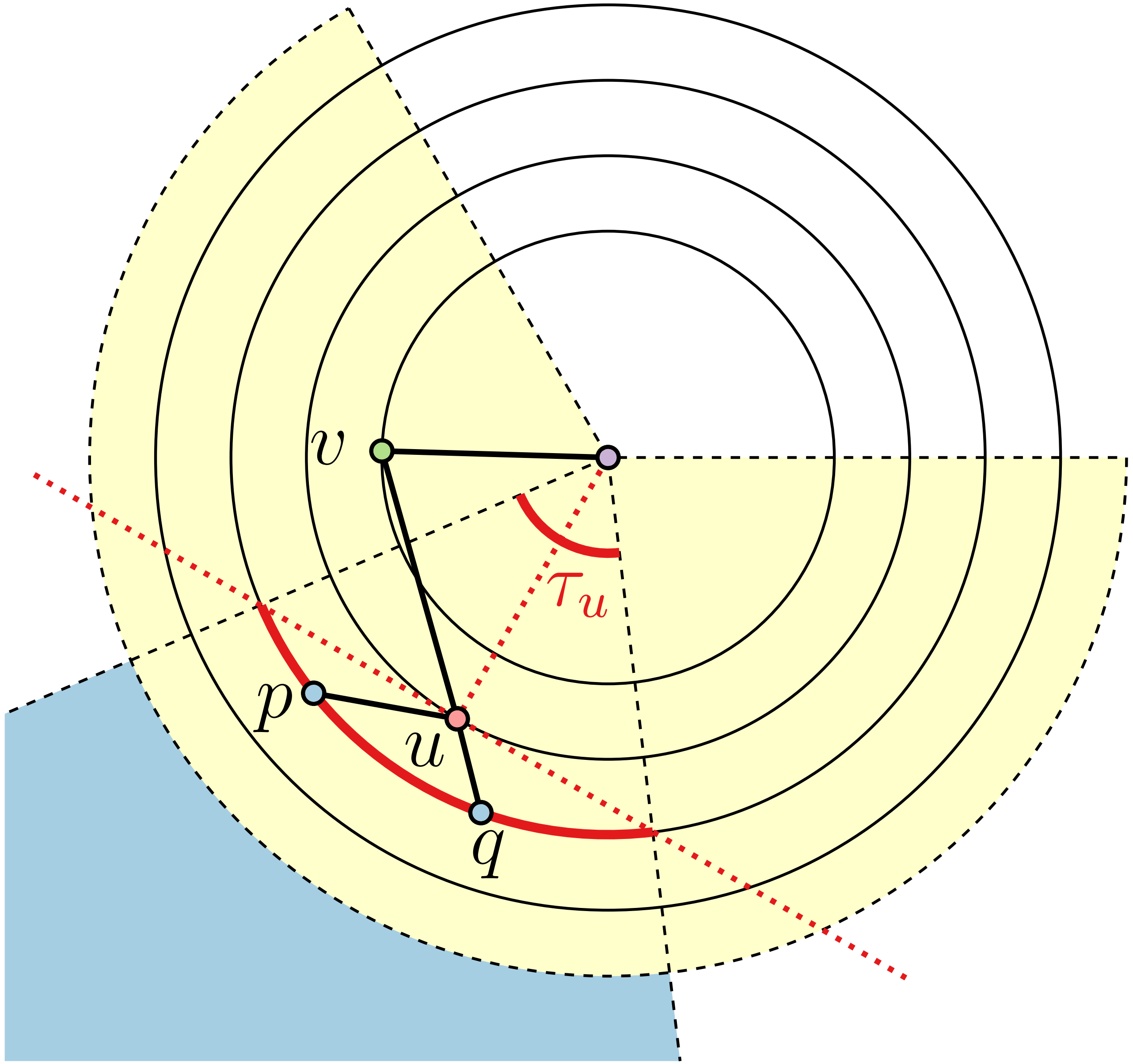
Radial Layouts – How To Avoid Crossings



Radial Layouts – How To Avoid Crossings

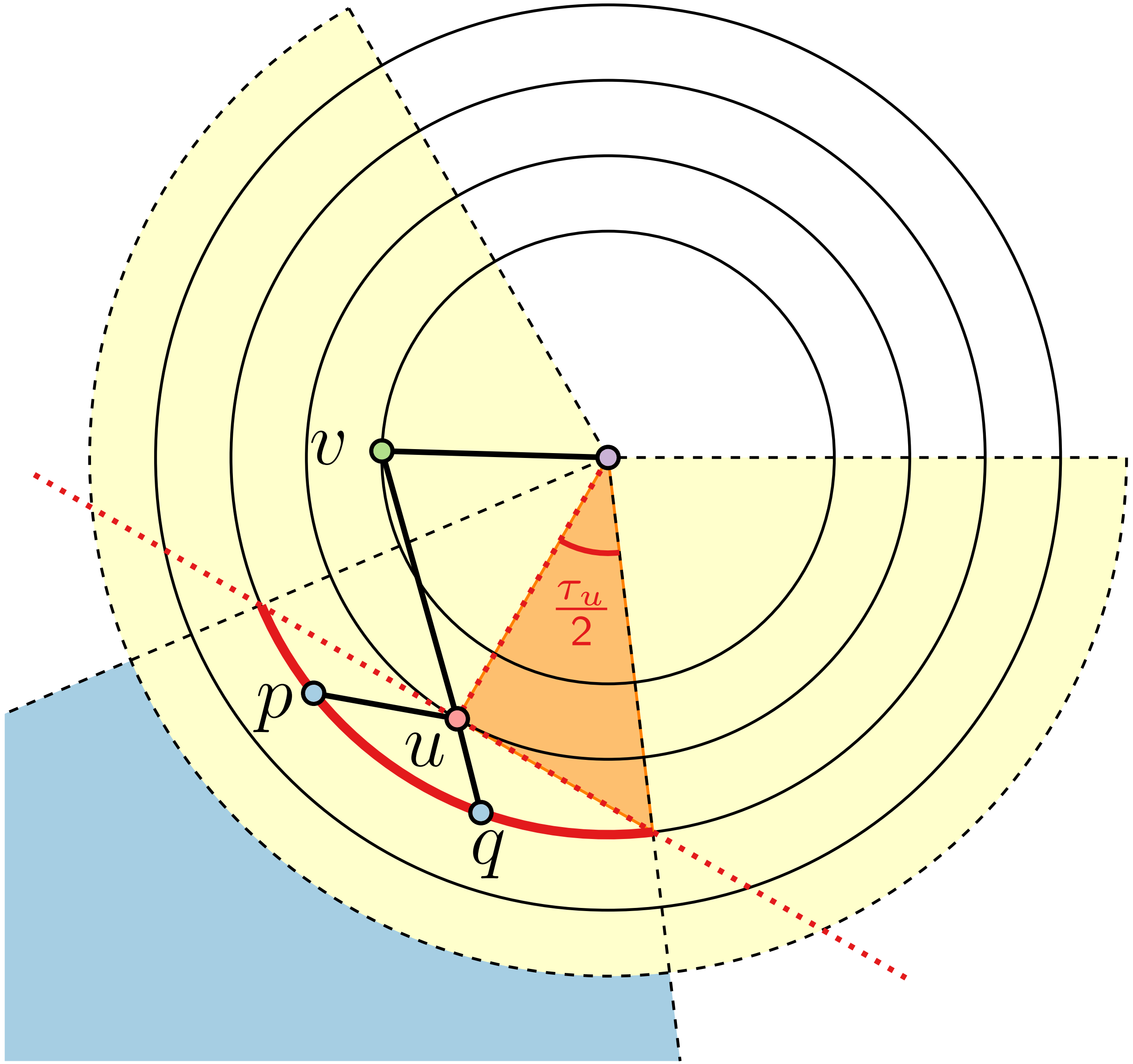


Radial Layouts – How To Avoid Crossings



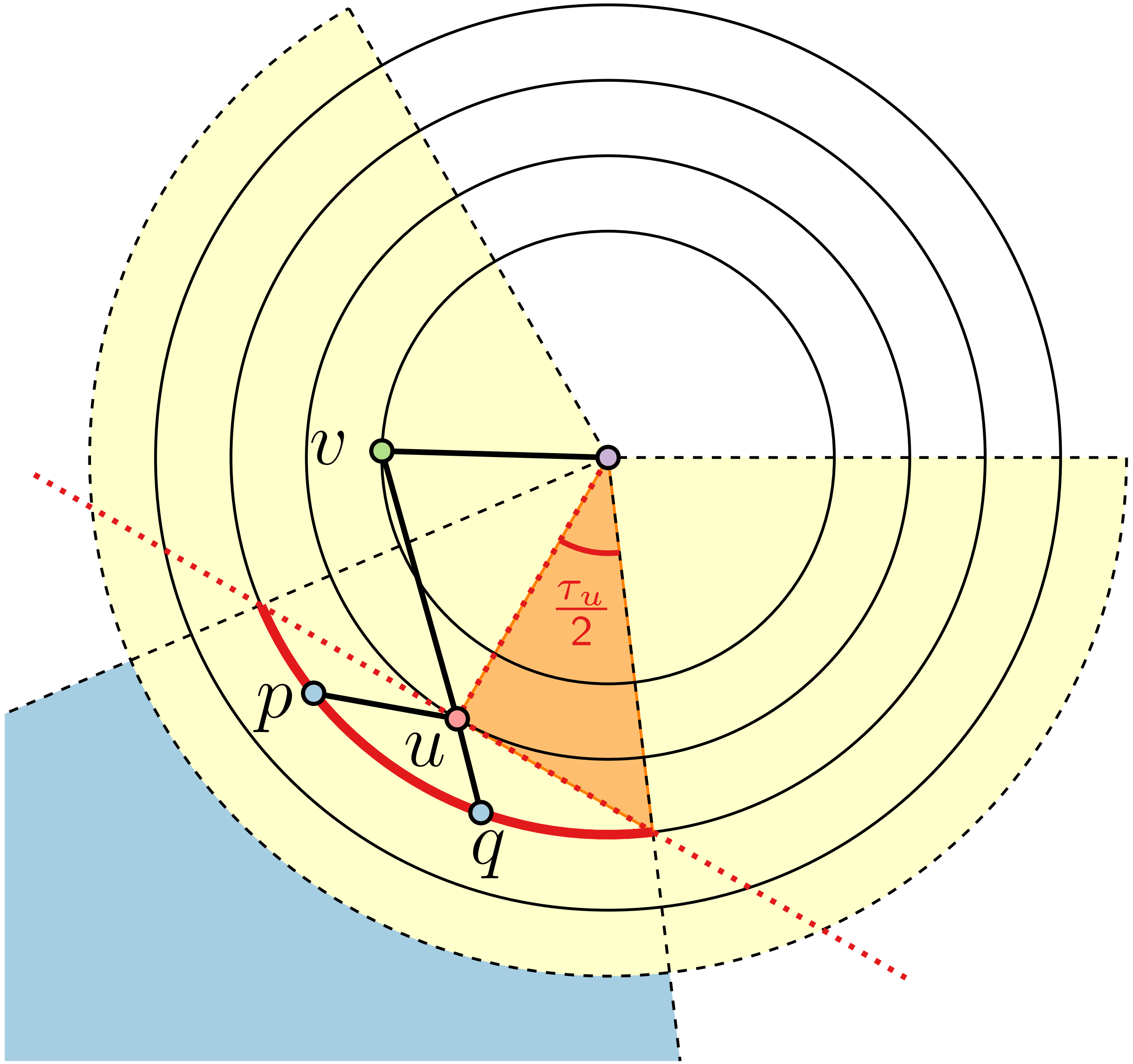
- τ_u – angle of the wedge corresponding to vertex u

Radial Layouts – How To Avoid Crossings



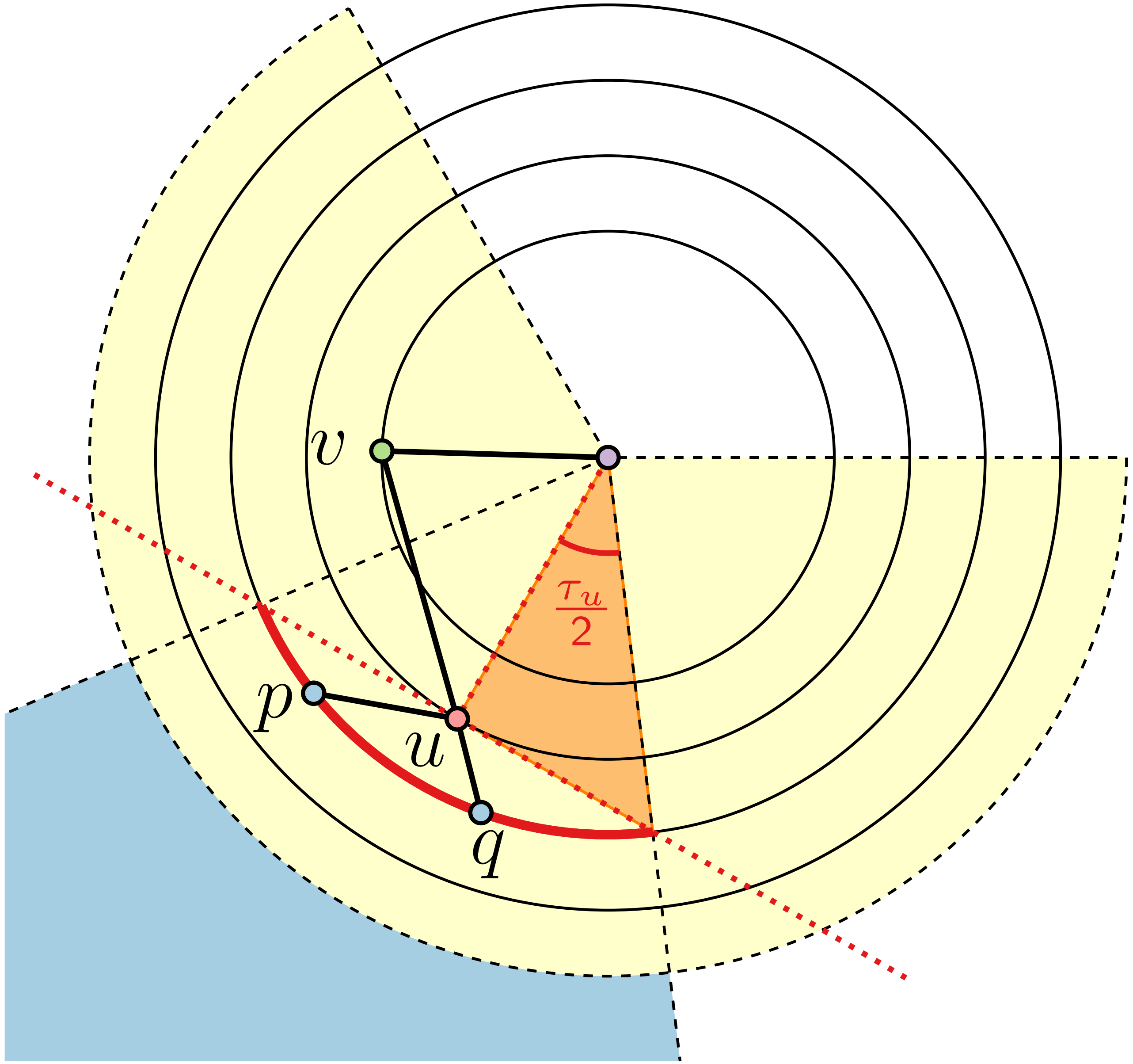
- τ_u – angle of the wedge corresponding to vertex u

Radial Layouts – How To Avoid Crossings



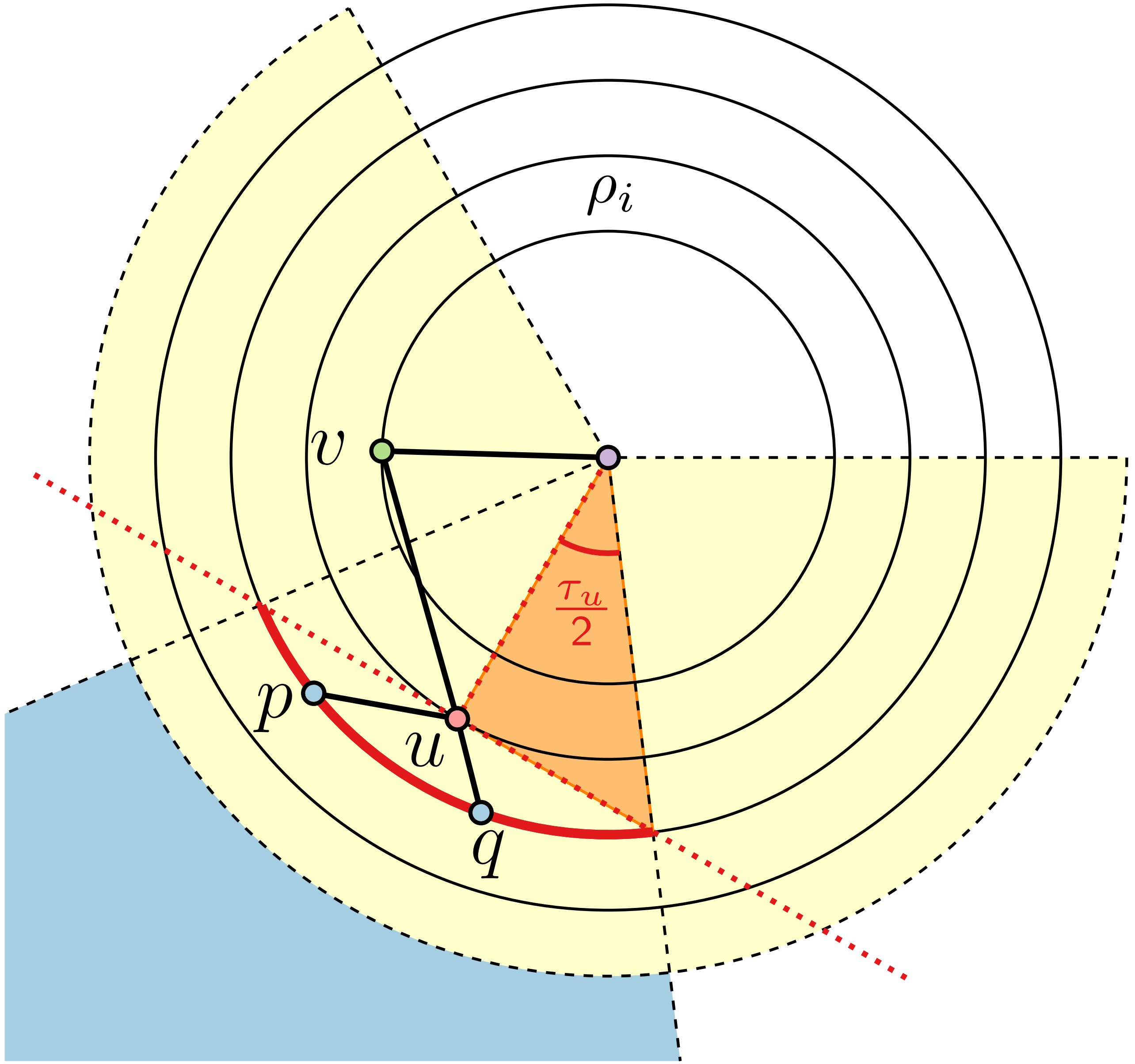
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u

Radial Layouts – How To Avoid Crossings



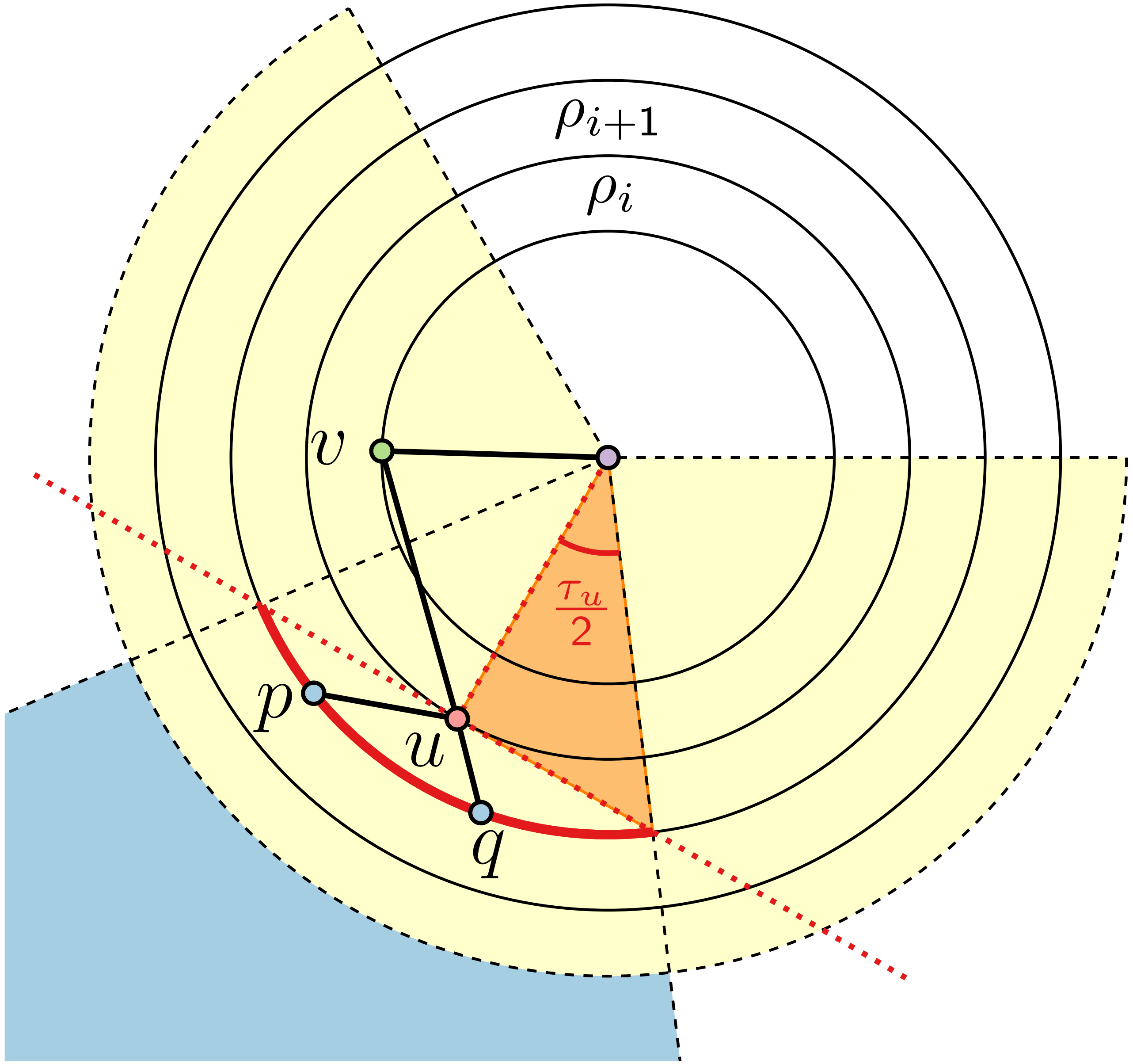
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i

Radial Layouts – How To Avoid Crossings



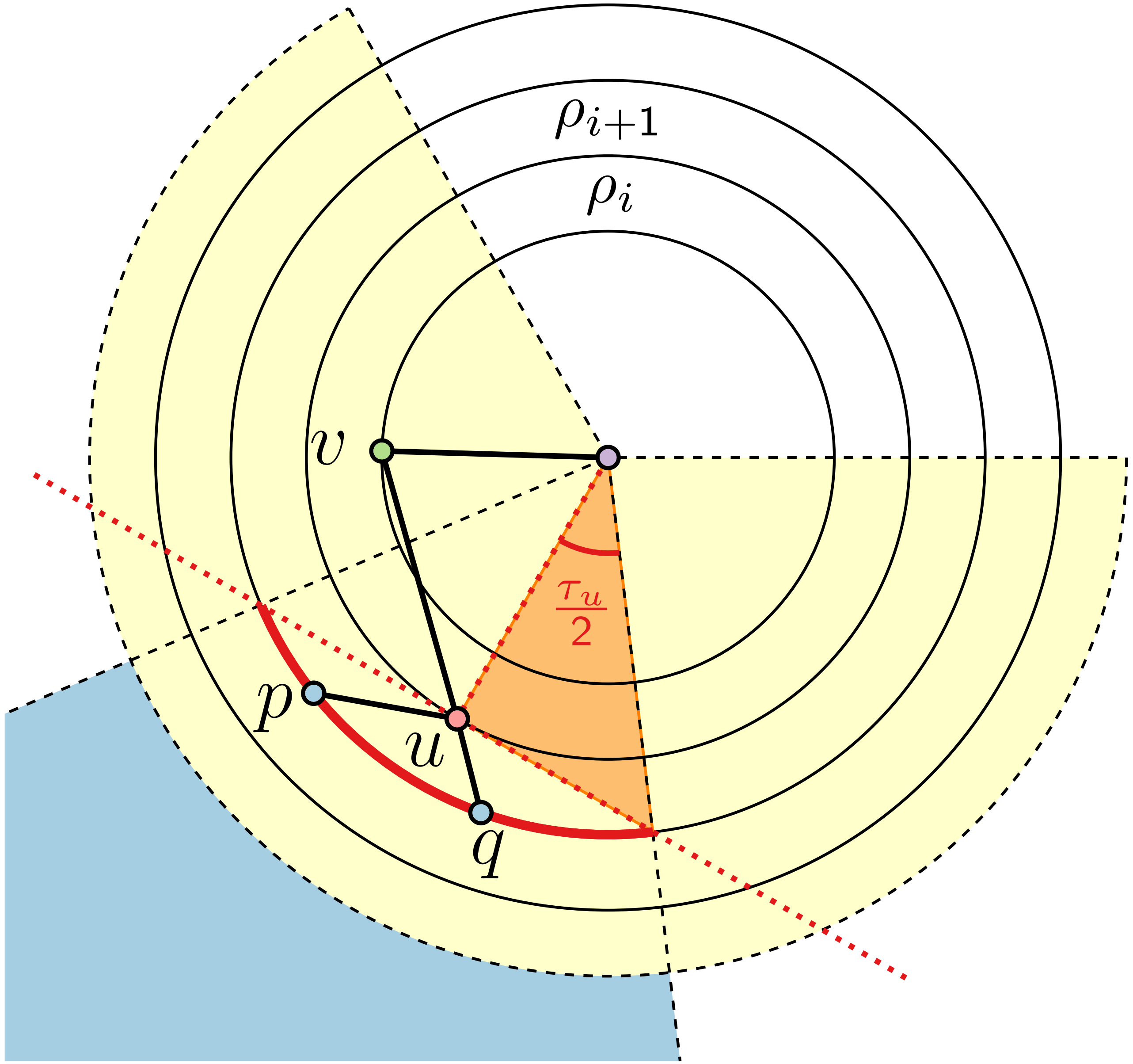
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i

Radial Layouts – How To Avoid Crossings



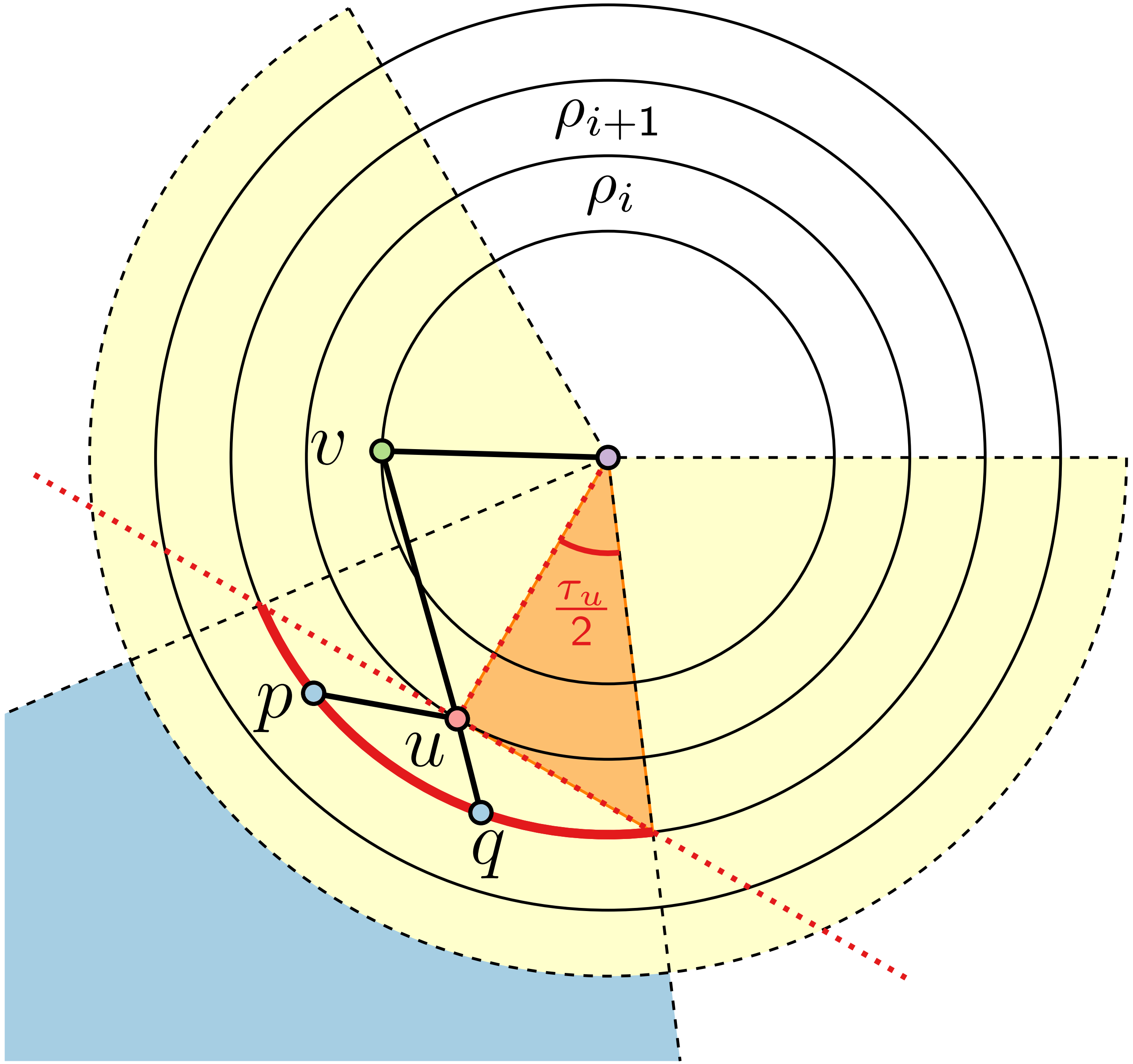
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i

Radial Layouts – How To Avoid Crossings



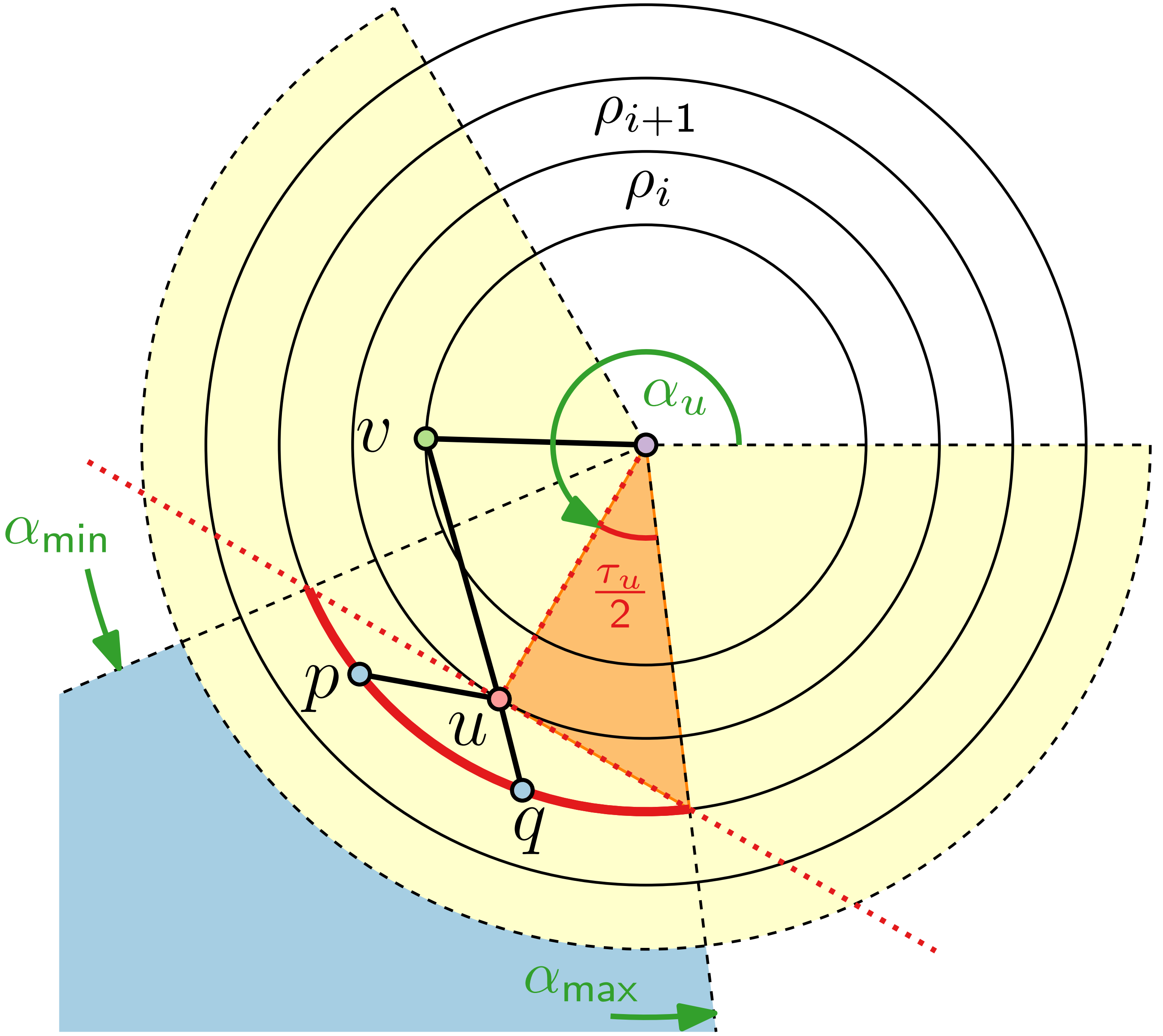
- τ_u – angle of the wedge corresponding to vertex u
- $\ell(u)$ – number of nodes in the subtree rooted at u
- ρ_i – radius of layer i
- $\cos \frac{\tau_u}{2} = \frac{\rho_i}{\rho_{i+1}}$

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
- $\cos \frac{\tau_u}{2} = \frac{\rho_i}{\rho_{i+1}}$
- $\tau_u = \min \left\{ \frac{\ell(u)}{\ell(v)-1},\ 2 \arccos \frac{\rho_i}{\rho_{i+1}} \right\}$

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
- $\cos \frac{\tau_u}{2} = \frac{\rho_i}{\rho_{i+1}}$
- $\tau_u = \min \left\{ \frac{\ell(u)}{\ell(v)-1},\ 2 \arccos \frac{\rho_i}{\rho_{i+1}} \right\}$
- Alternative:

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

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

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

return $(d_v, \alpha_v)_{v \in V(T)}$

// vertex positions in polar coordinates

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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**

calculate the size of the subtree recursively

Radial Layouts – Pseudocode

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

begin

  $postorder(r)$

  $preorder(r, 0, 0, 2\pi)$

  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**

    $postorder(w)$

    $\ell(v) \leftarrow \ell(v) + \ell(w)$

Radial Layouts – Pseudocode

RadialTreeLayout(tree T, root $r \in T$, radii $\rho_1 < \cdots < \rho_k$)
begin
  $postorder(r)$
  $preorder(r, 0, 0, 2\pi)$
  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, $t, \alpha_{\mathsf{min}}, \alpha_{\mathsf{max}}$)

Radial Layouts – Pseudocode

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

begin

  $postorder(r)$
  $preorder(r, 0, 0, 2\pi)$
  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, $t, \alpha_{\mathsf{min}}, \alpha_{\mathsf{max}}$)

  $d_v \leftarrow \rho_t$

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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, $t, \alpha_{\mathsf{min}}, \alpha_{\mathsf{max}}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\mathsf{min}} + \alpha_{\mathsf{max}})/2$

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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, $t, \alpha_{\text{min}}, \alpha_{\text{max}}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2$

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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, $t, \alpha_{\text{min}}, \alpha_{\text{max}}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2$

$//output$

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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, $t, \alpha_{\mathsf{min}}, \alpha_{\mathsf{max}}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\mathsf{min}} + \alpha_{\mathsf{max}})/2$

$//output$

if $t > 0$ **then**

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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, $t, \alpha_{\mathsf{min}}, \alpha_{\mathsf{max}}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\mathsf{min}} + \alpha_{\mathsf{max}})/2$

//output

if $t > 0$ **then**

$\alpha_{\mathsf{min}} \leftarrow \max\{\alpha_{\mathsf{min}}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$

$\alpha_{\mathsf{max}} \leftarrow \min\{\alpha_{\mathsf{max}}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$

Radial Layouts – Pseudocode

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

begin

$\quad$ *postorder*(r)

$\quad$ *preorder*$(r, 0, 0, 2\pi)$

$\quad$ **return** $(d_v, \alpha_v)_{v \in V(T)}$

$\quad$ // vertex positions in polar coordinates

postorder(vertex v)

$\quad$ $\ell(v) \leftarrow 1$

$\quad$ **foreach** child w of v **do**

$\quad\quad$ *postorder*(w)

$\quad\quad$ $\ell(v) \leftarrow \ell(v) + \ell(w)$

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

$\quad$ $d_v \leftarrow \rho_t$

$\quad$ $\alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2$ $\quad$ *//output*

$\quad$ **if** $t > 0$ **then**

$\quad\quad$ $\alpha_{\text{min}} \leftarrow \max\{\alpha_{\text{min}}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$

$\quad\quad$ $\alpha_{\text{max}} \leftarrow \min\{\alpha_{\text{max}}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$

$\quad$ *left* $\leftarrow \alpha_{\text{min}}$

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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, $t, \alpha_{\min}, \alpha_{\max}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$

//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 $\rho_1 < \cdots < \rho_k$)

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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, $t, \alpha_{\min}, \alpha_{\max}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$

//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**

$right \leftarrow left + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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, $t, \alpha_{\min}, \alpha_{\max}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$

//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**

$right \leftarrow left + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$

$preorder(w, t+1, left, right)$

Radial Layouts – Pseudocode

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

begin

  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, $t, \alpha_{\min}, \alpha_{\max}$)

  $d_v \leftarrow \rho_t$
  $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ //*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**
    right $\leftarrow$ *left* $+ \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$
    preorder$(w, t+1, \mathit{left}, \mathit{right})$
    left $\leftarrow$ *right*

Radial Layouts – Pseudocode

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

begin

- $postorder(r)$
- $preorder(r, 0, 0, 2\pi)$
- **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, $t, \alpha_{\text{min}}, \alpha_{\text{max}}$)

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

Runtime?

Radial Layouts – Pseudocode

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

begin

  $postorder(r)$

  $preorder(r, 0, 0, 2\pi)$

  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, $t, \alpha_{\text{min}}, \alpha_{\text{max}}$)

  $d_v \leftarrow \rho_t$

  $\alpha_v \leftarrow (\alpha_{\text{min}} + \alpha_{\text{max}})/2$ *//output*

  if $t > 0$ **then**

    $\alpha_{\text{min}} \leftarrow \max\{\alpha_{\text{min}}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$

    $\alpha_{\text{max}} \leftarrow \min\{\alpha_{\text{max}}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$

  $left \leftarrow \alpha_{\text{min}}$

  foreach child w of v **do**

    $right \leftarrow left + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\text{max}} - \alpha_{\text{min}})$

    $preorder(w, t+1, left, right)$

    $left \leftarrow right$

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

Radial Layouts – Pseudocode

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

begin

$postorder(r)$

$preorder(r, 0, 0, 2\pi)$

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, $t, \alpha_{\min}, \alpha_{\max}$)

$d_v \leftarrow \rho_t$

$\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ //*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**

$right \leftarrow left + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$

$preorder(w, t+1, left, right)$

$left \leftarrow right$

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

Correctness?

Radial Layouts – Pseudocode

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

begin

$\quad$ *postorder*(r)

$\quad$ *preorder*$(r, 0, 0, 2\pi)$

$\quad$ **return** $(d_v, \alpha_v)_{v \in V(T)}$

$\quad$ // vertex positions in polar coordinates

postorder(vertex v)

$\quad$ $\ell(v) \leftarrow 1$

$\quad$ **foreach** child w of v **do**

$\quad\quad$ *postorder*(w)

$\quad\quad$ $\ell(v) \leftarrow \ell(v) + \ell(w)$

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

$\quad$ $d_v \leftarrow \rho_t$

$\quad$ $\alpha_v \leftarrow (\alpha_{\min} + \alpha_{\max})/2$ $\quad$ *//output*

$\quad$ **if** $t > 0$ **then**

$\quad\quad$ $\alpha_{\min} \leftarrow \max\{\alpha_{\min}, \alpha_v - \arccos \frac{\rho_t}{\rho_{t+1}}\}$

$\quad\quad$ $\alpha_{\max} \leftarrow \min\{\alpha_{\max}, \alpha_v + \arccos \frac{\rho_t}{\rho_{t+1}}\}$

$\quad$ $\mathit{left} \leftarrow \alpha_{\min}$

$\quad$ **foreach** child w of v **do**

$\quad\quad$ $\mathit{right} \leftarrow \mathit{left} + \frac{\ell(w)}{\ell(v)-1} \cdot (\alpha_{\max} - \alpha_{\min})$

$\quad\quad$ *preorder*$(w, t+1, \mathit{left}, \mathit{right})$

$\quad\quad$ $\mathit{left} \leftarrow \mathit{right}$

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

Correctness? ✓

Radial Layouts – Result

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

Radial Layouts – Result

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

- Γ is radial drawing

Radial Layouts – Result

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

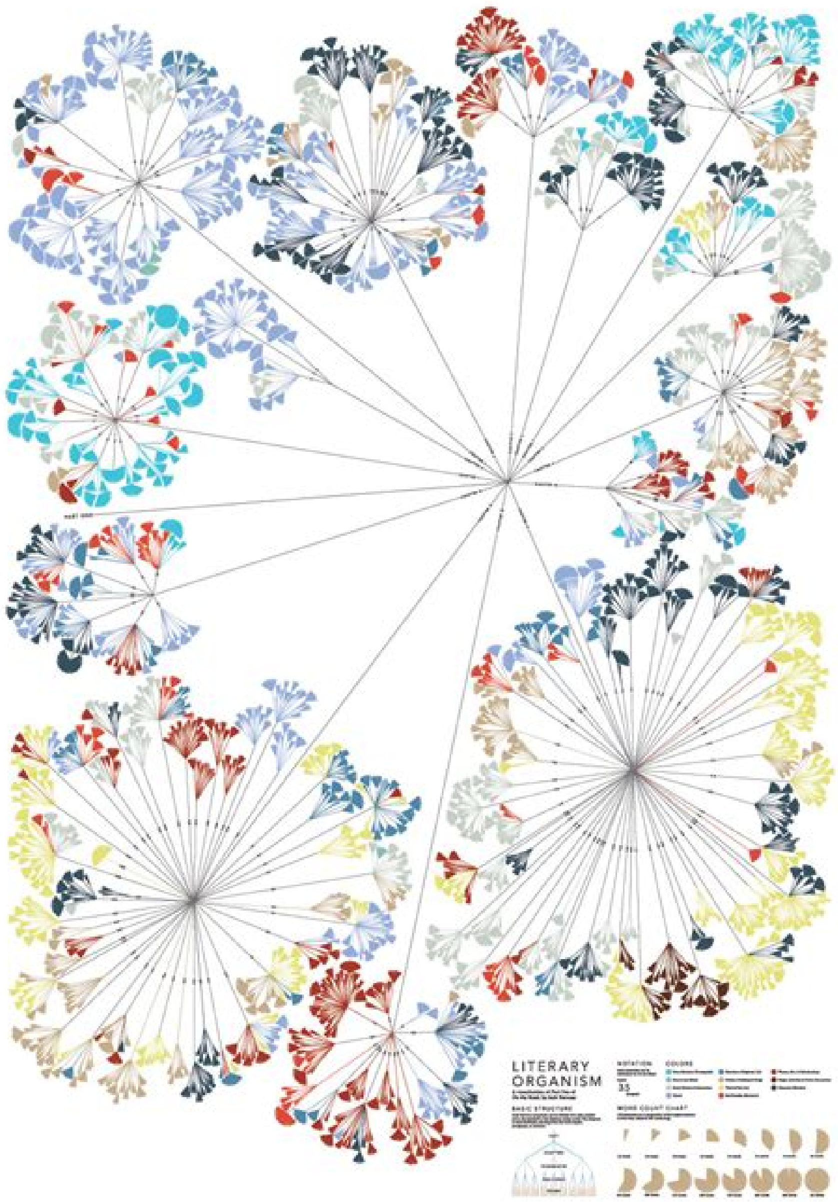
- Γ is radial drawing
- Vertices lie on circle according to their depth

Radial Layouts – Result

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

- Γ is radial drawing
- Vertices lie on circle according to their depth
- Area quadratic in max-degree(T) $\times$ height(T)
  (see [GD Ch. 3.1.3] if interested)

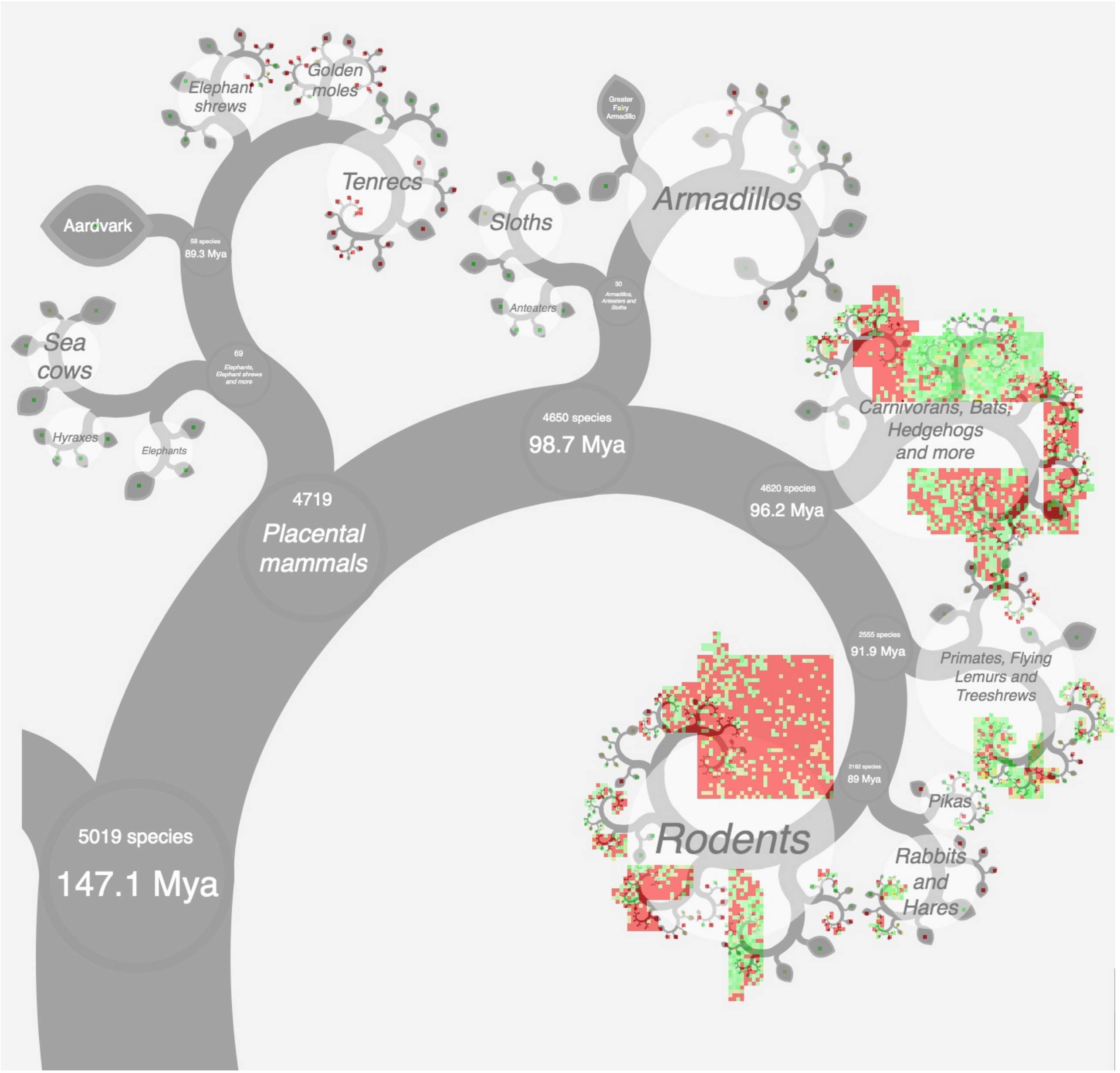
Other tree visualisation styles



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

Similar to ballon layout

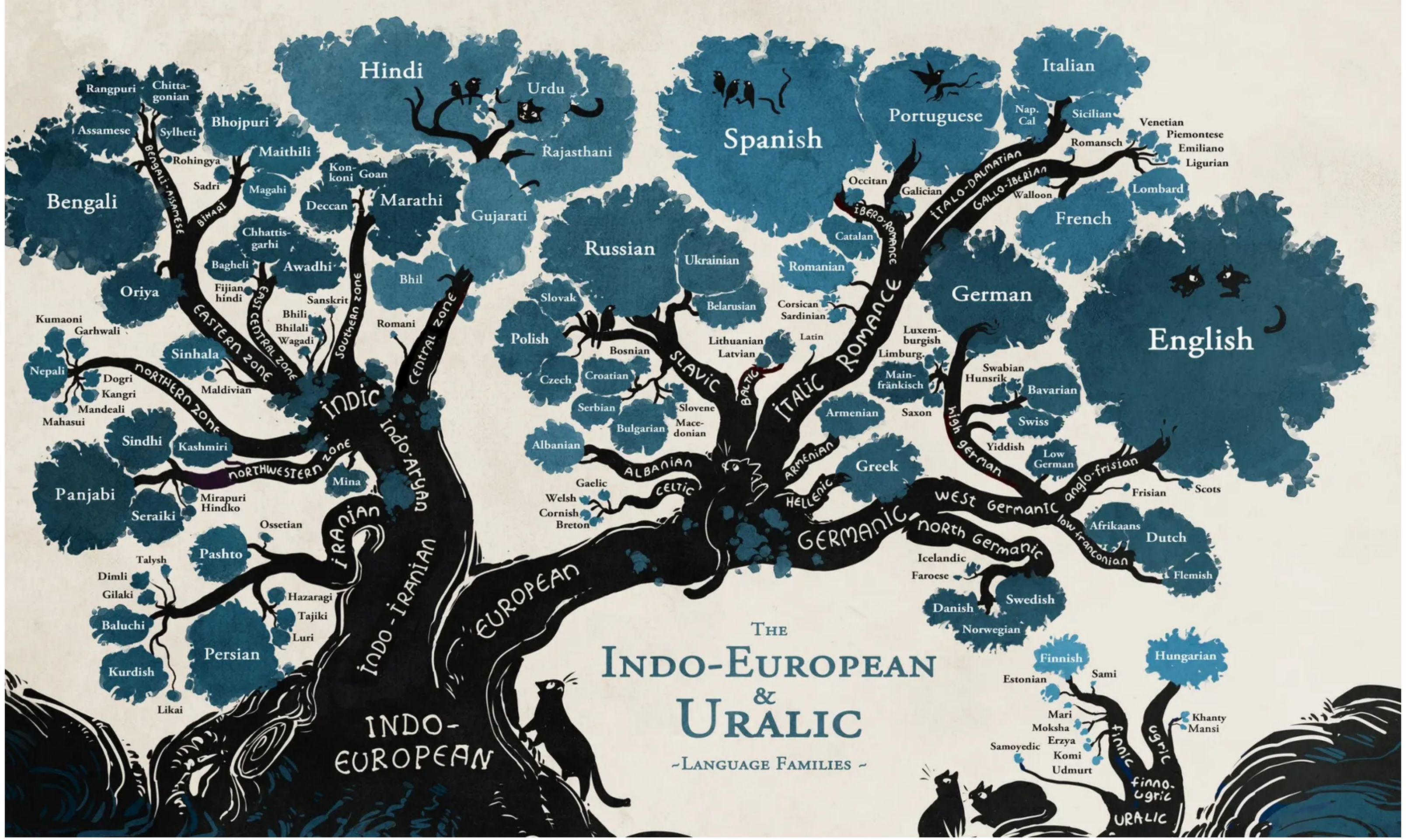
Other tree visualisation styles



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

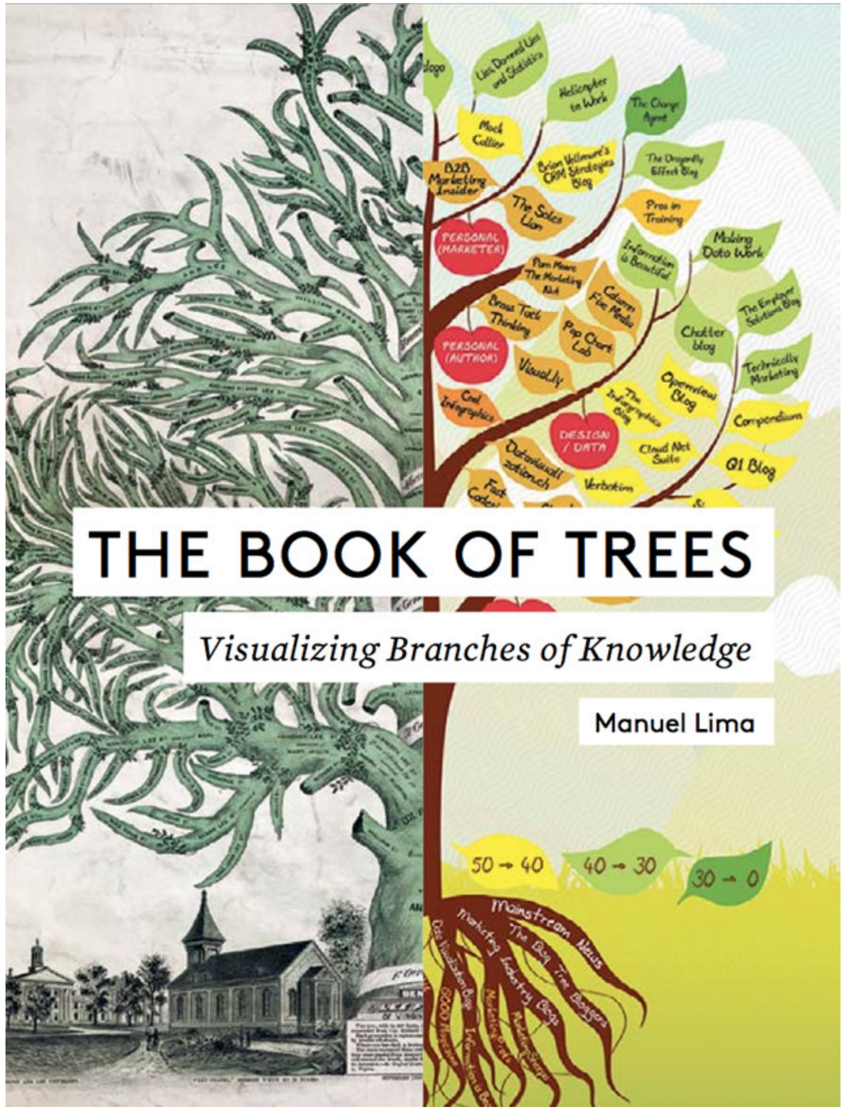
Fractal layout

Other tree visualisation styles

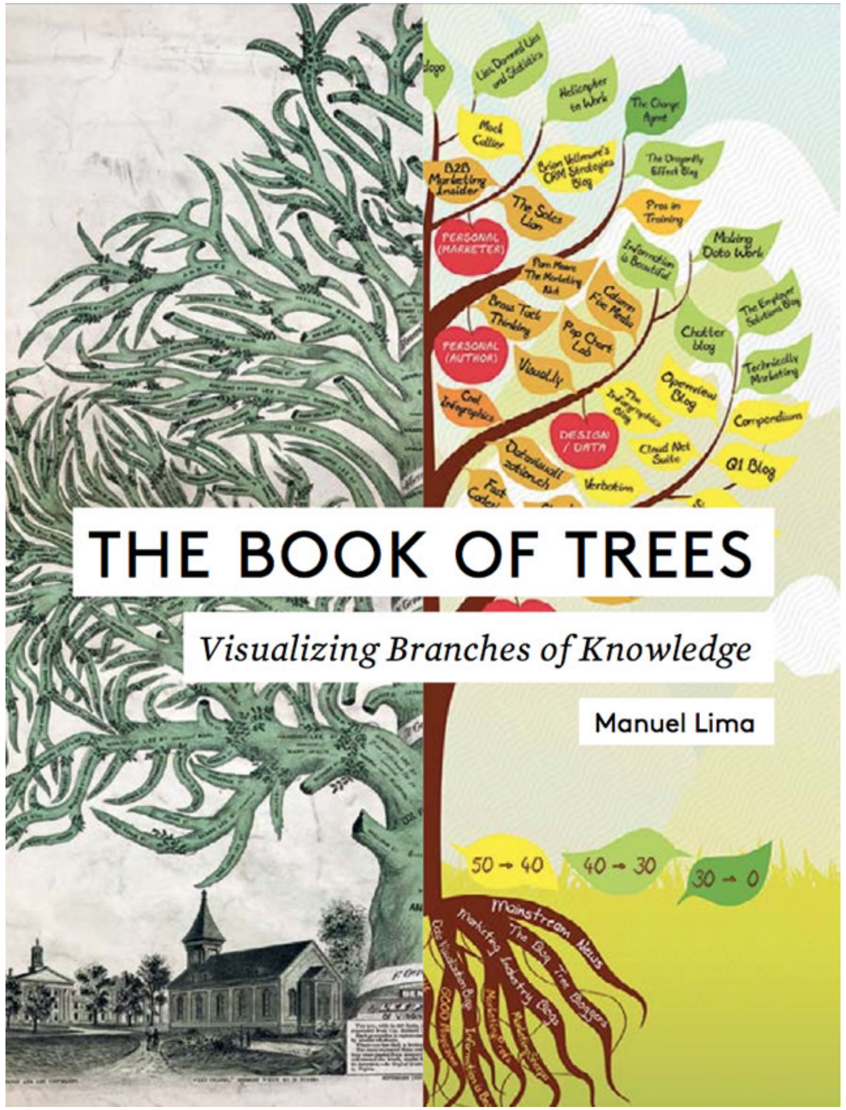


A language family tree – in pictures

Other tree visualisation styles



Other tree visualisation styles

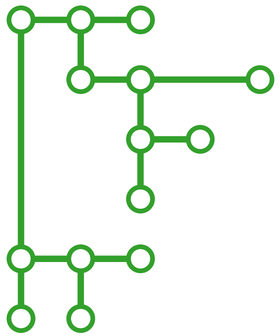


treevis.net

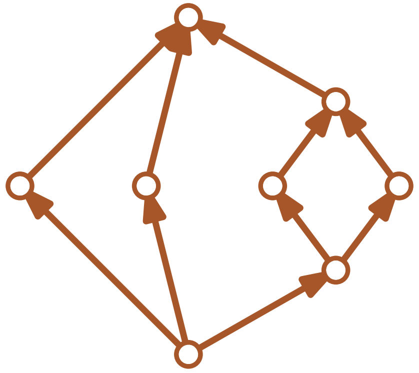
Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

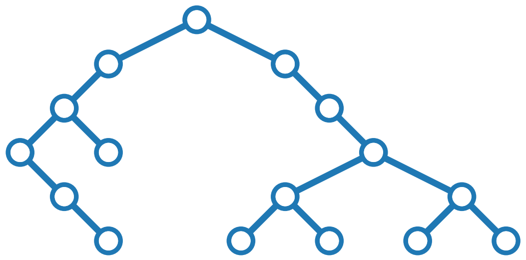
Visualization of Graphs

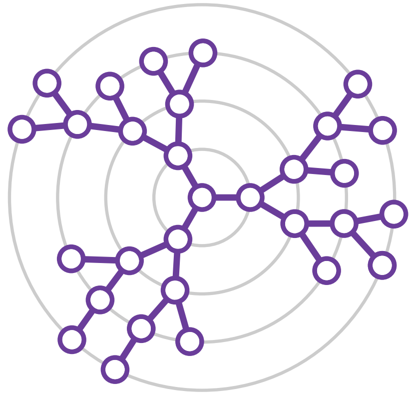
Lecture 1b:
Drawing Trees and Series-Parallel Graphs



Part IV:
Series-Parallel Graphs







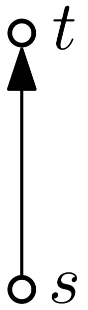
Series-Parallel Graphs

A graph G is **series-parallel**, if

Series-Parallel Graphs

A graph G is **series-parallel**, if

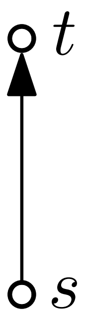
- it contains a single (directed) edge (s, t), or

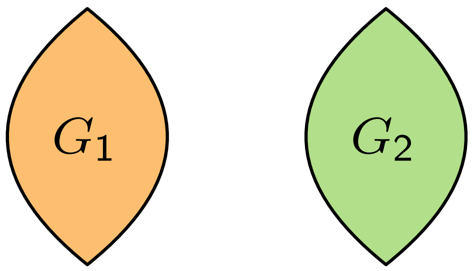


Series-Parallel Graphs

A graph G is **series-parallel**, if

- it contains a single (directed) edge (s, t), or
- it consists of two series-parallel graphs G_1, G_2

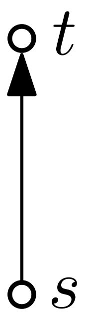


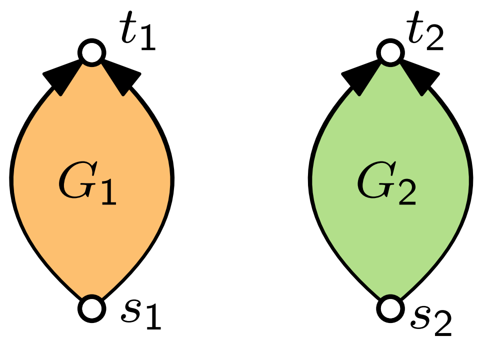


Series-Parallel Graphs

A graph G is **series-parallel**, if

- it contains a single (directed) edge (s, t), or
- it consists of two series-parallel graphs G_1, G_2 with sources s_1, s_2 and sinks t_1, t_2

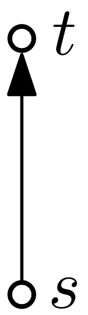


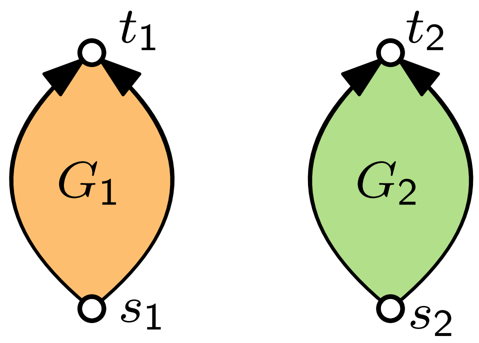


Series-Parallel Graphs

A graph G is **series-parallel**, if

- it contains a single (directed) edge (s, t), or
- it consists of two series-parallel graphs G_1, G_2 with sources s_1, s_2 and sinks t_1, t_2 that are combined using one of the following rules:

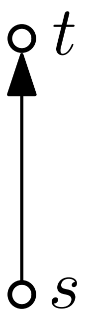




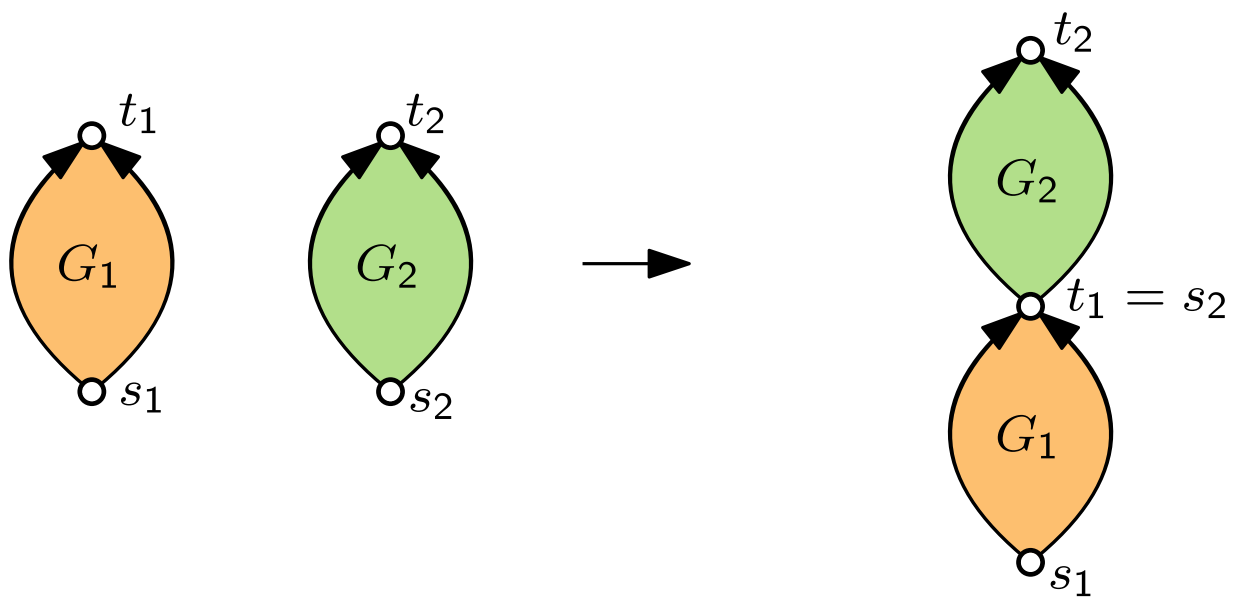
Series-Parallel Graphs

A graph G is **series-parallel**, if

- it contains a single (directed) edge (s, t), or
- it consists of two series-parallel graphs G_1, G_2 with sources s_1, s_2 and sinks t_1, t_2 that are combined using one of the following rules:



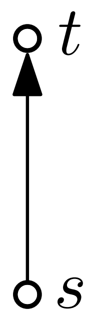
Series composition



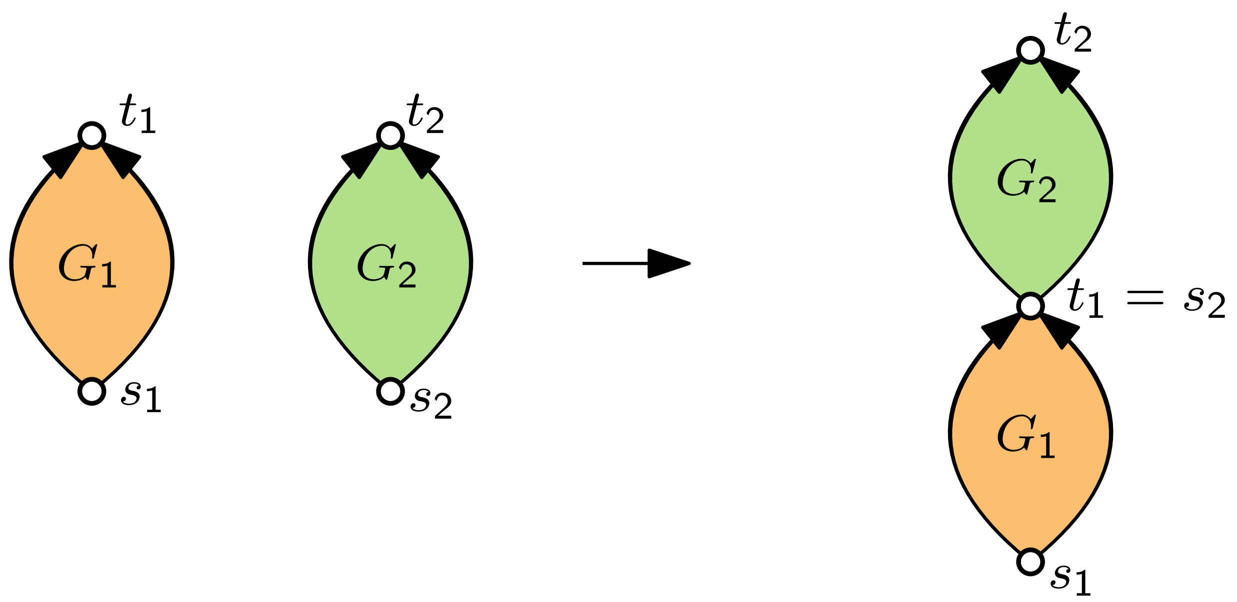
Series-Parallel Graphs

A graph G is **series-parallel**, if

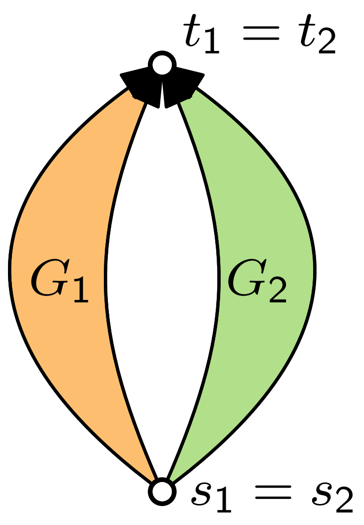
- it contains a single (directed) edge (s, t), or
- it consists of two series-parallel graphs G_1, G_2 with sources s_1, s_2 and sinks t_1, t_2 that are combined using one of the following rules:



Series composition



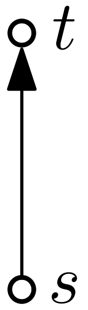
Parallel composition



Series-Parallel Graphs

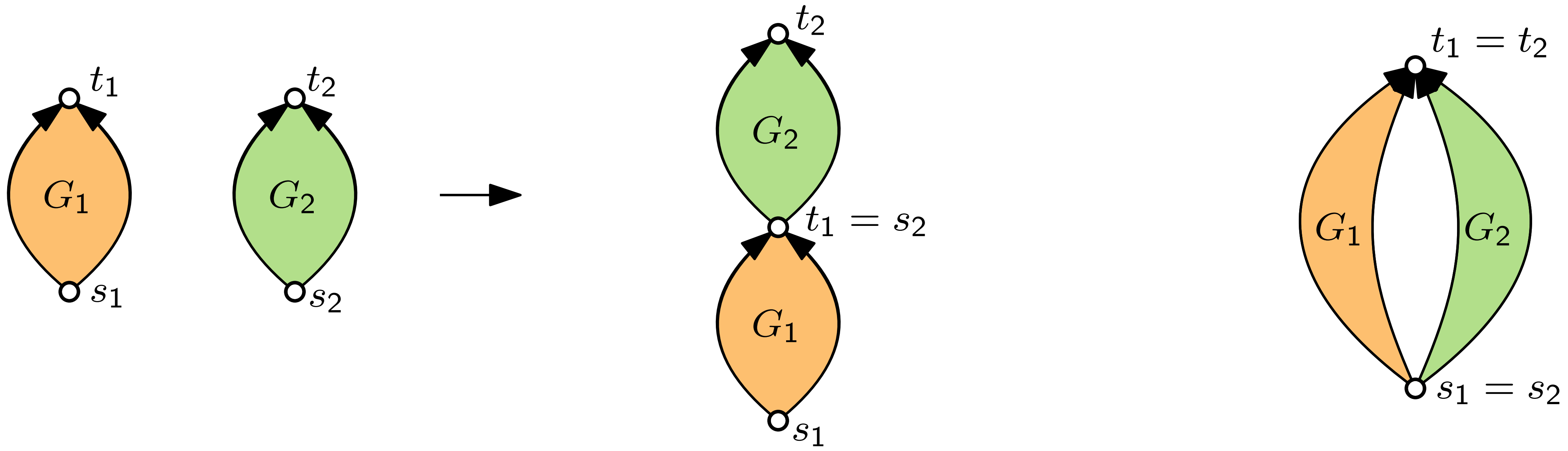
A graph G is **series-parallel**, if

- it contains a single (directed) edge (s, t), or
- it consists of two series-parallel graphs G_1, G_2 with sources s_1, s_2 and sinks t_1, t_2 that are combined using one of the following rules:



Convince yourself that series-parallel graphs are planar!

Series composition **Parallel composition**



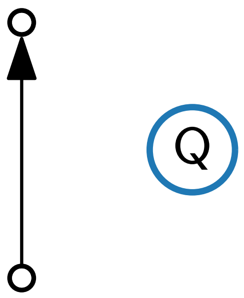
Series-Parallel Graphs – Decomposition Tree

A **decomposition tree** of G is a binary tree T with nodes of three types: **S**, **P** and **Q**:

Series-Parallel Graphs – Decomposition Tree

A **decomposition tree** of G is a binary tree T with nodes of three types: **S**, **P** and **Q**:

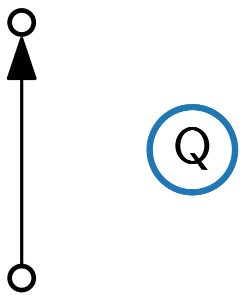
- A **Q**-node represents a single edge

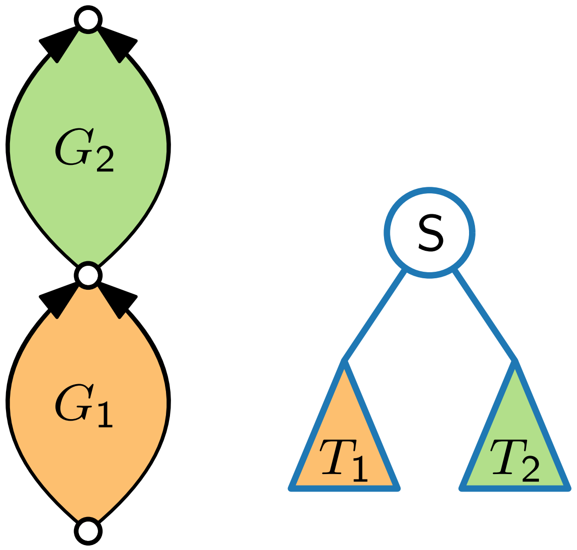


Series-Parallel Graphs – Decomposition Tree

A **decomposition tree** of G is a binary tree T with nodes of three types: **S**, **P** and **Q**:

- A **Q**-node represents a single edge
- An **S**-node represents a series composition;
  its children T_1 and T_2 represent G_1 and G_2

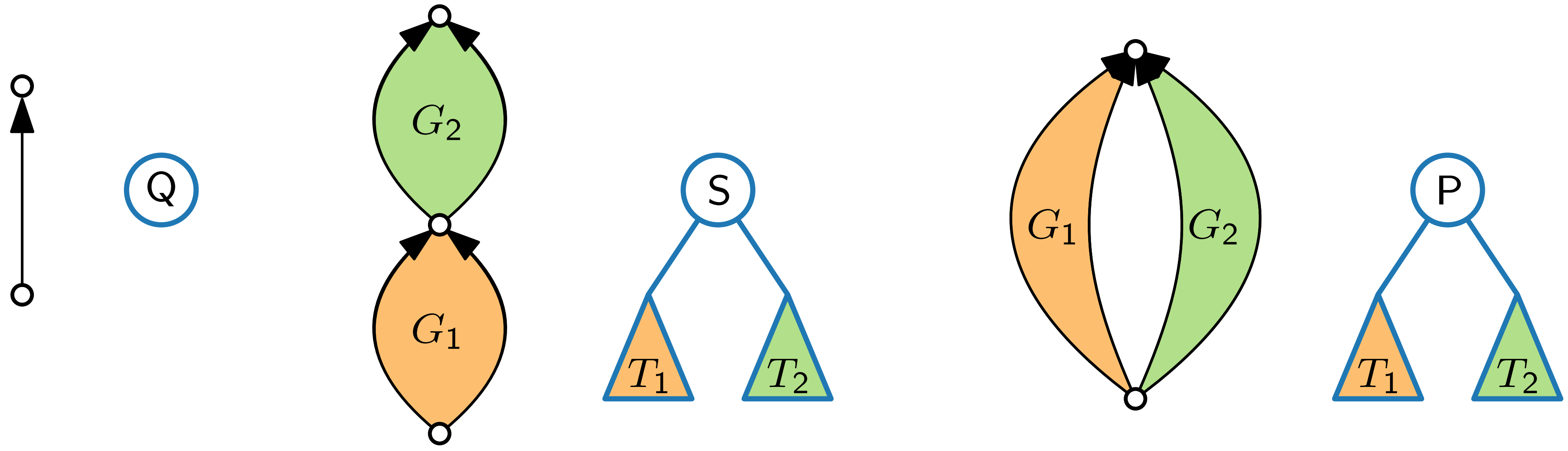




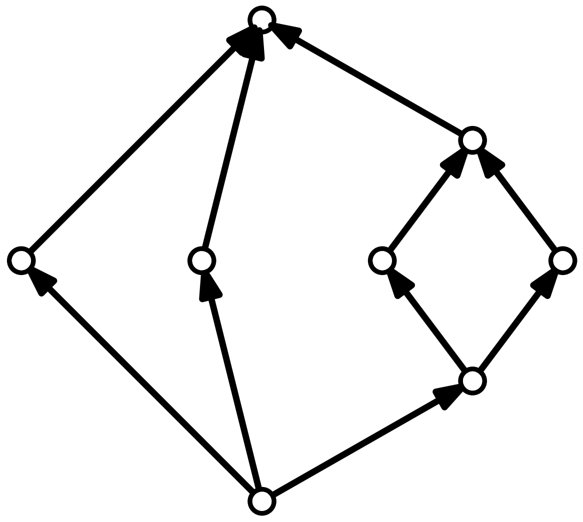
Series-Parallel Graphs – Decomposition Tree

A **decomposition tree** of G is a binary tree T with nodes of three types: **S**, **P** and **Q**:

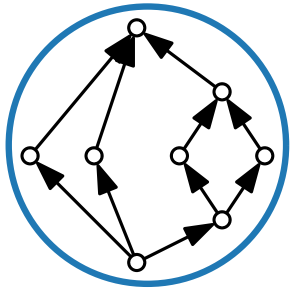
- A **Q**-node represents a single edge
- An **S**-node represents a series composition; its children T_1 and T_2 represent G_1 and G_2
- A **P**-node represents a parallel composition; its children T_1 and T_2 represent G_1 and G_2



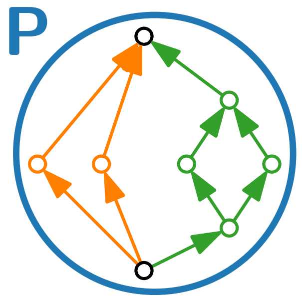
Series-Parallel Graphs – Decomposition Example



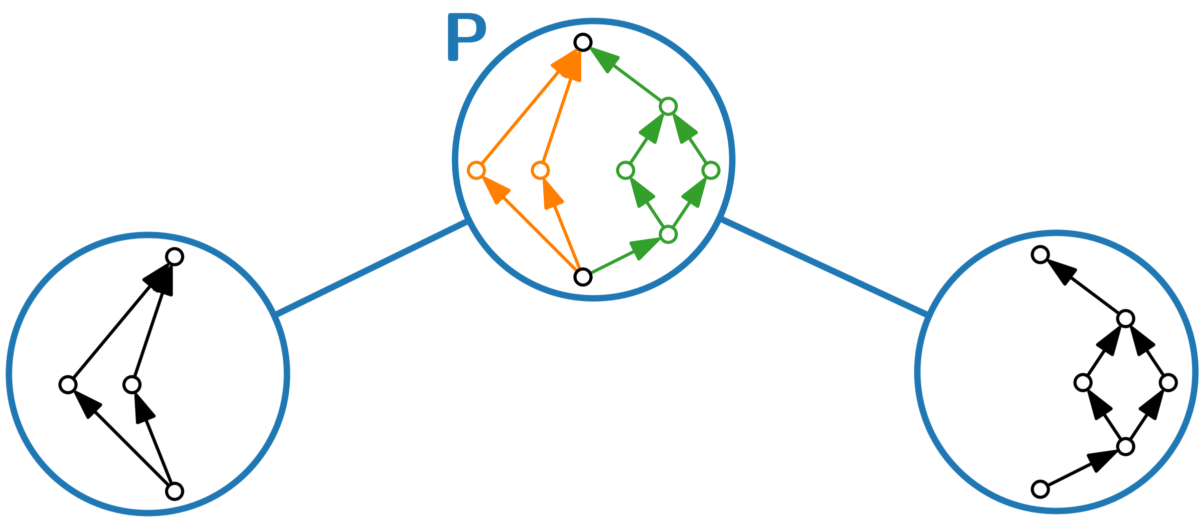
Series-Parallel Graphs – Decomposition Example



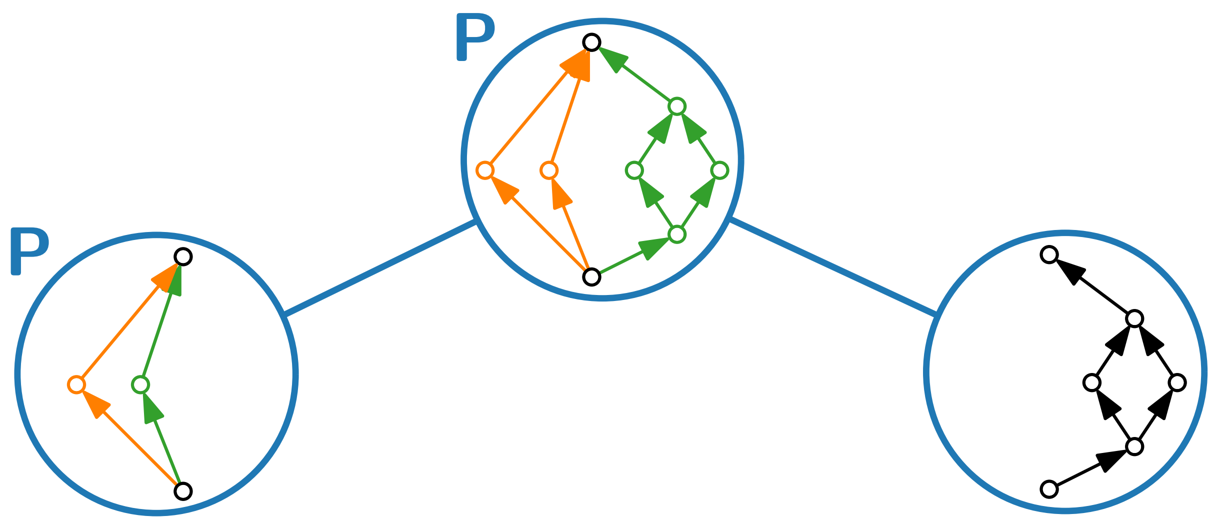
Series-Parallel Graphs – Decomposition Example



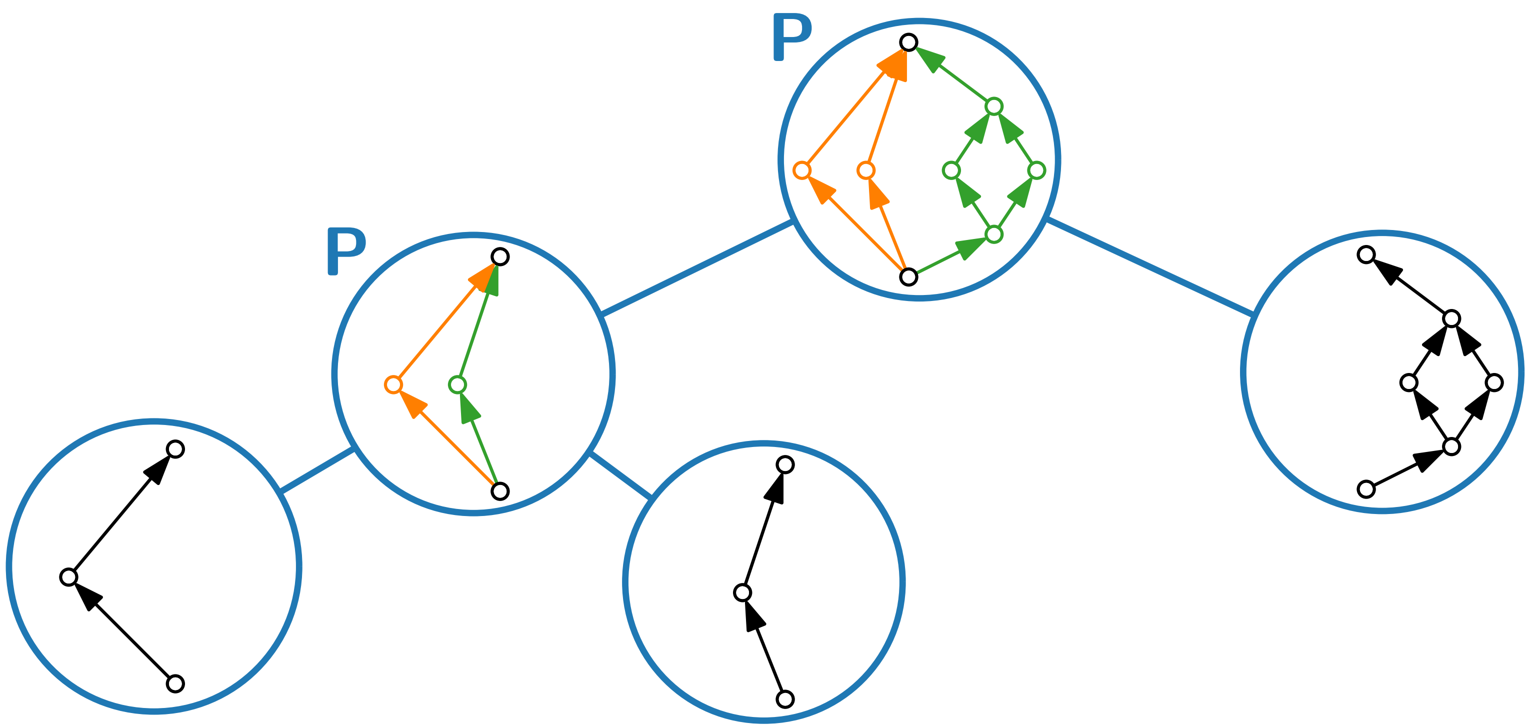
Series-Parallel Graphs – Decomposition Example



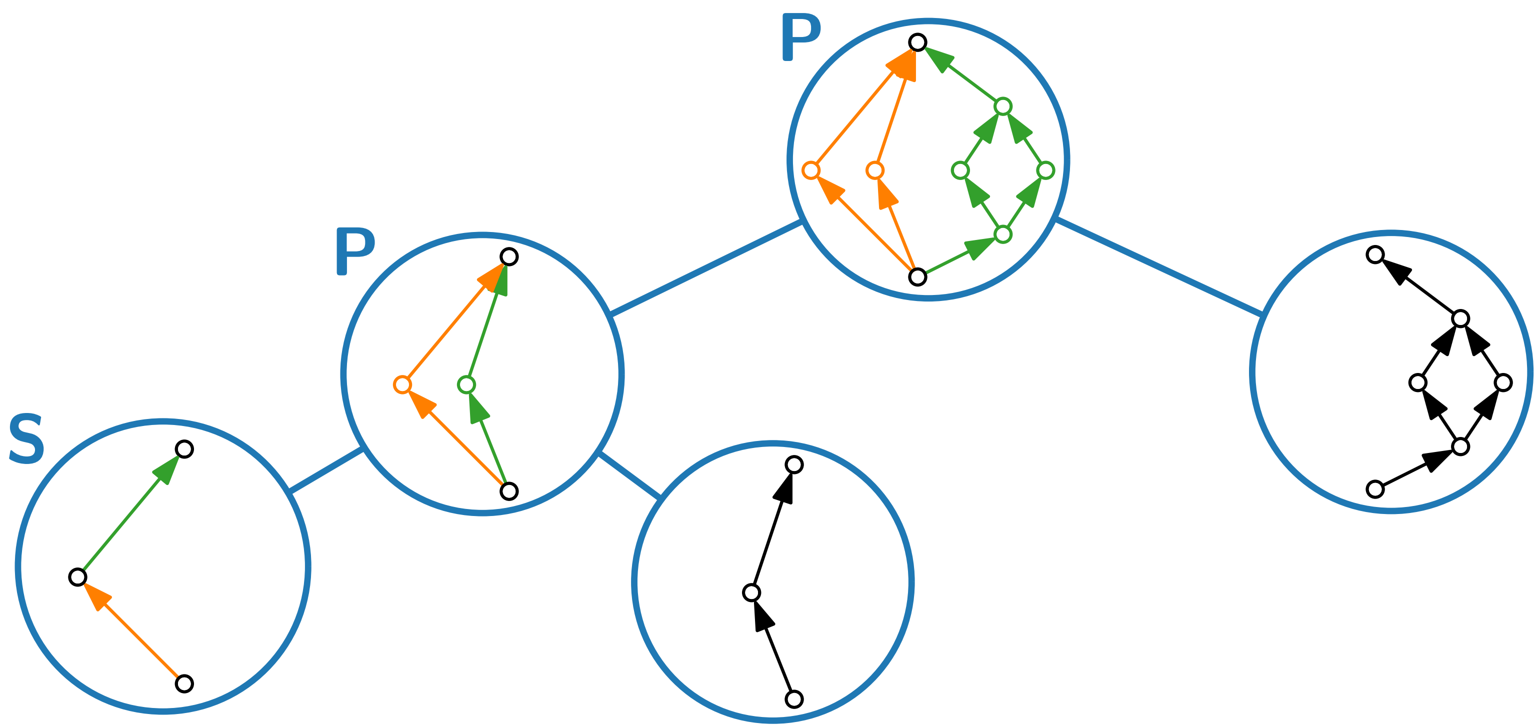
Series-Parallel Graphs – Decomposition Example



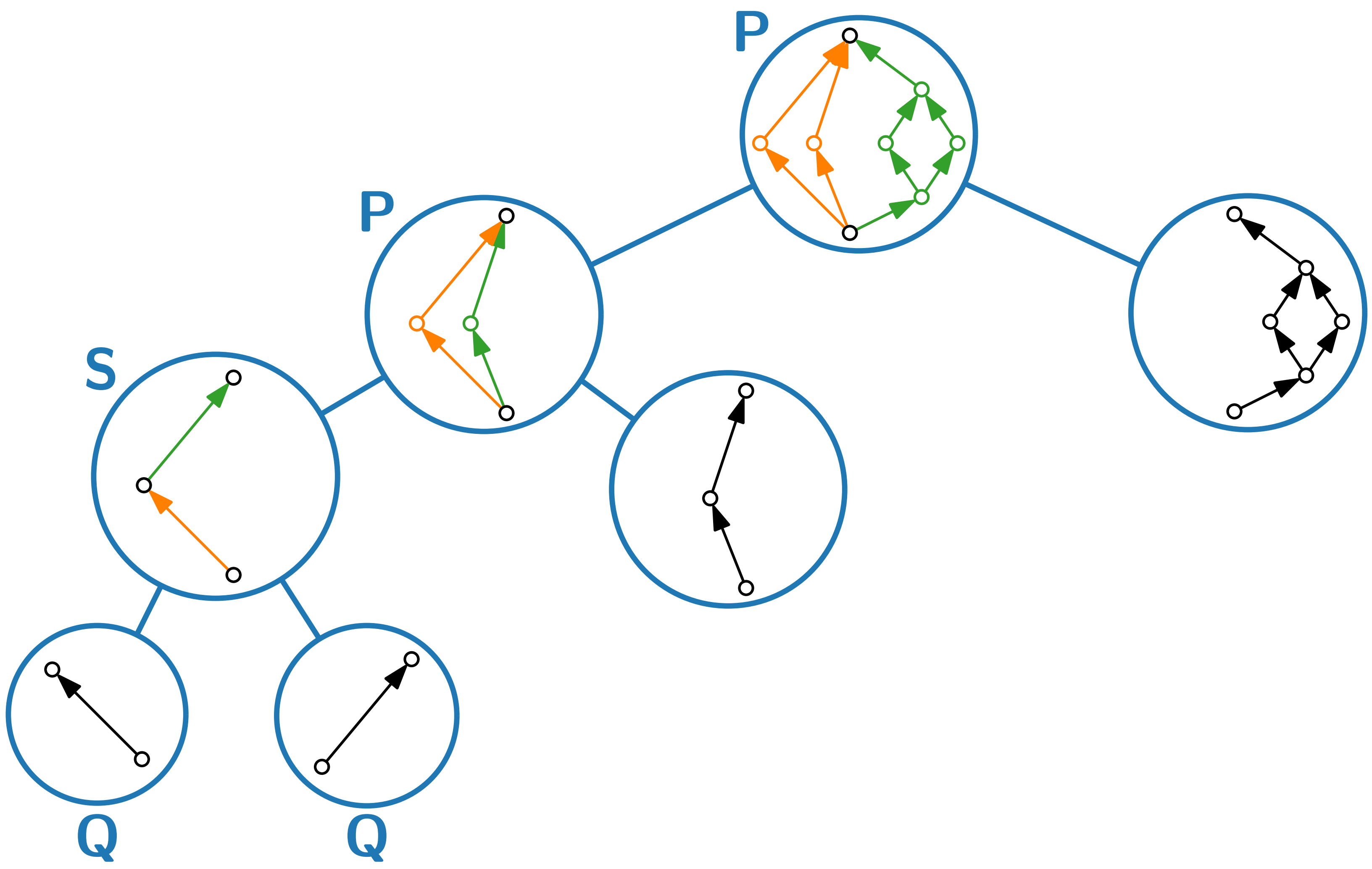
Series-Parallel Graphs – Decomposition Example



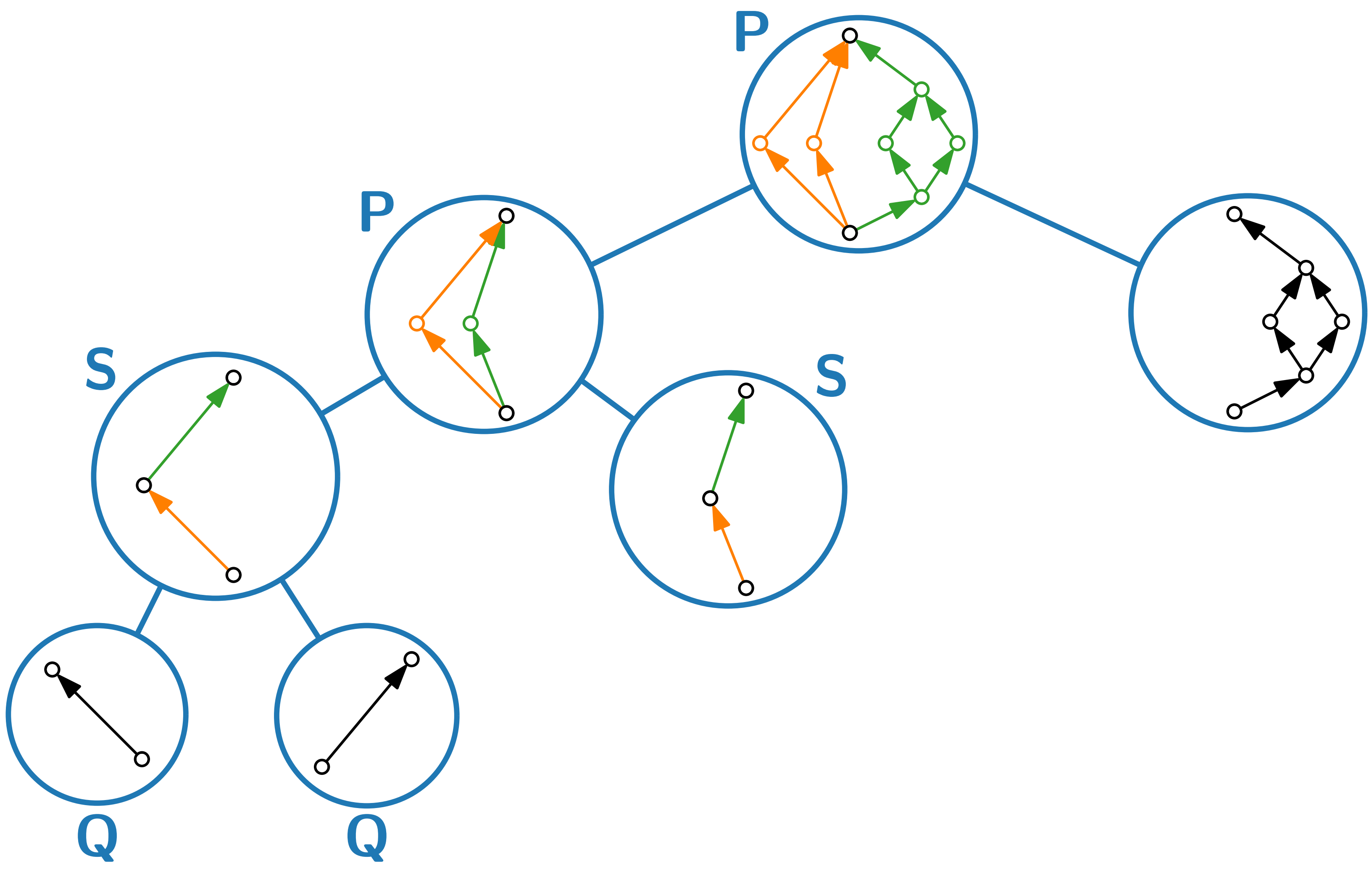
Series-Parallel Graphs – Decomposition Example



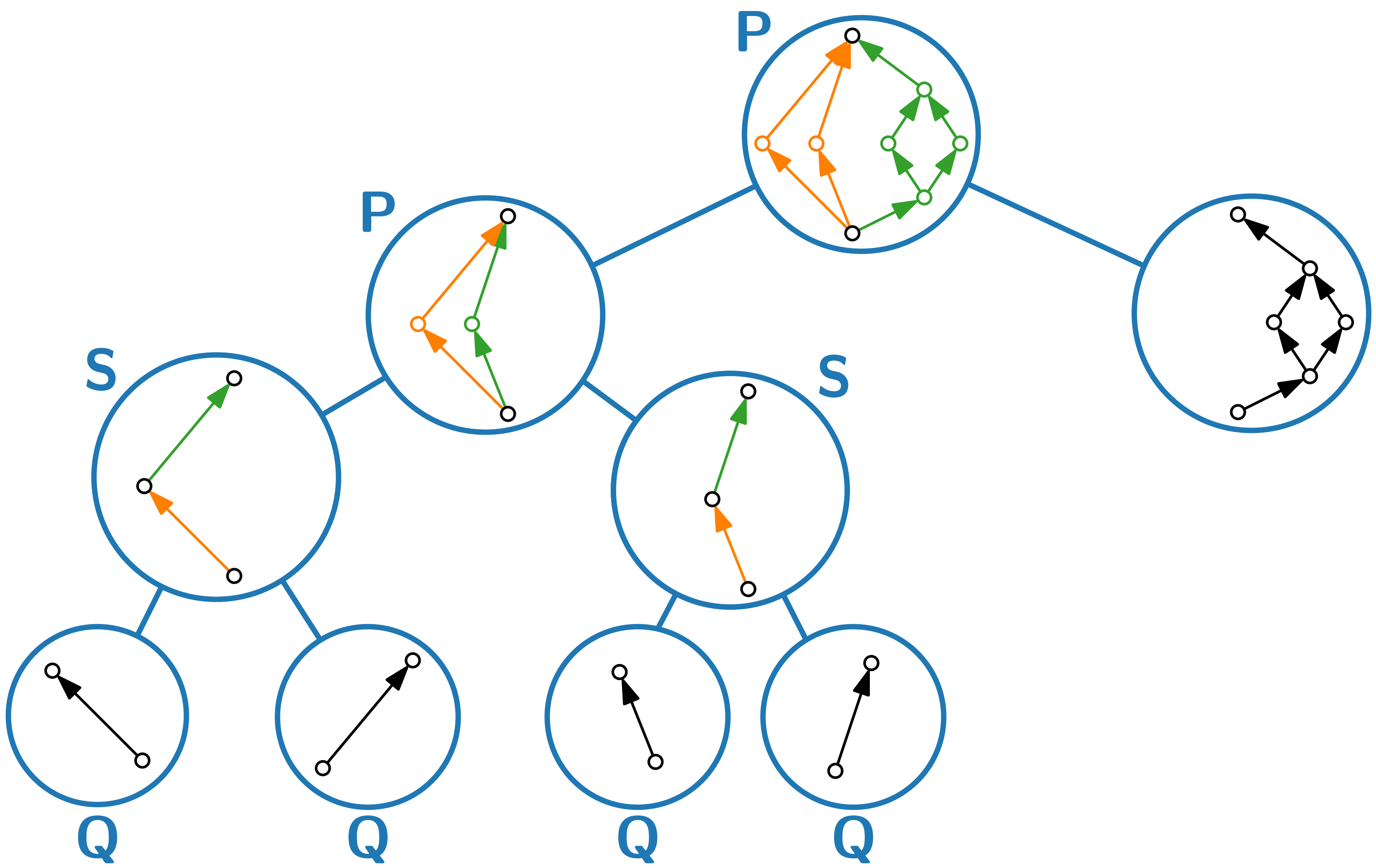
Series-Parallel Graphs – Decomposition Example



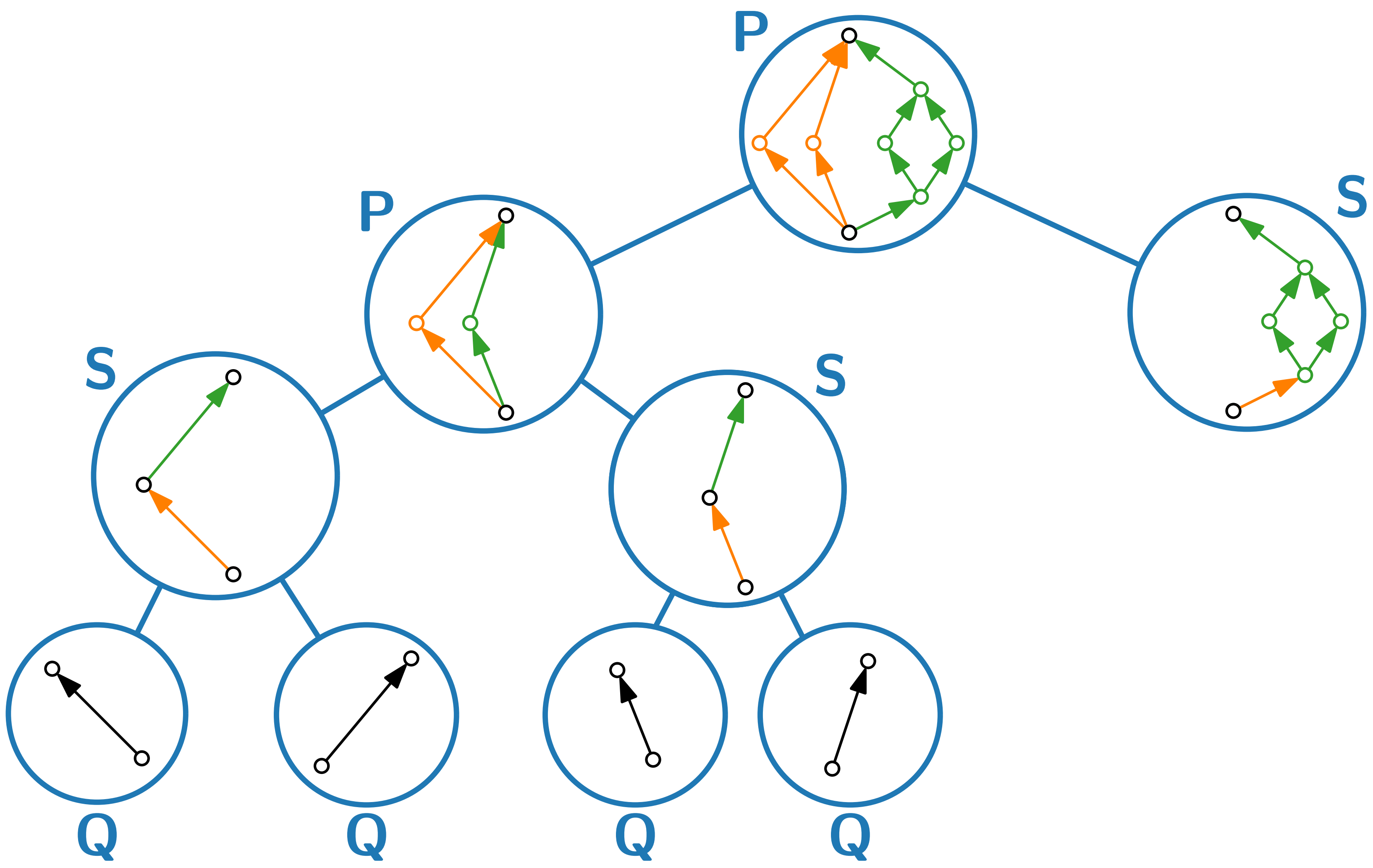
Series-Parallel Graphs – Decomposition Example



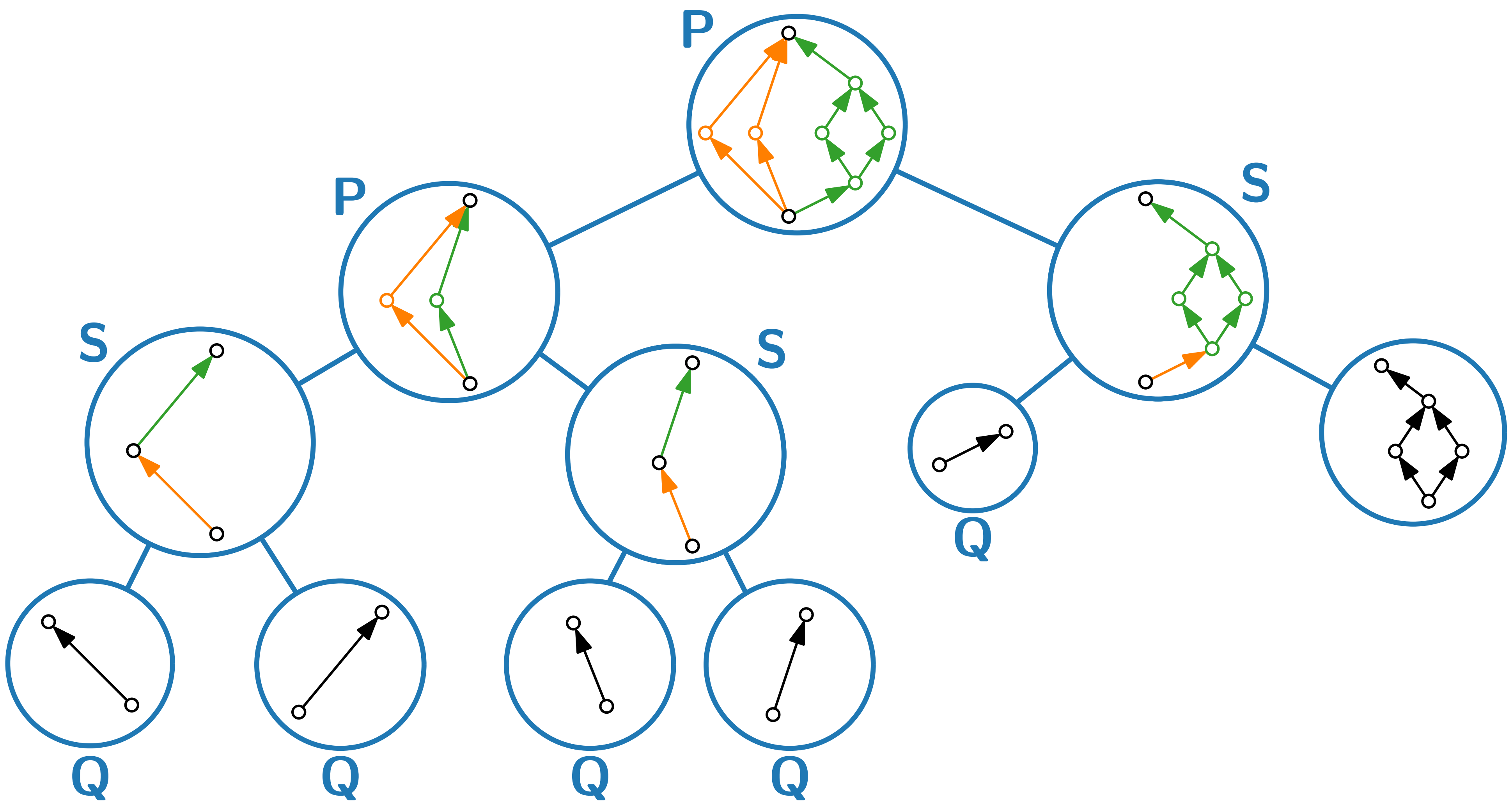
Series-Parallel Graphs – Decomposition Example



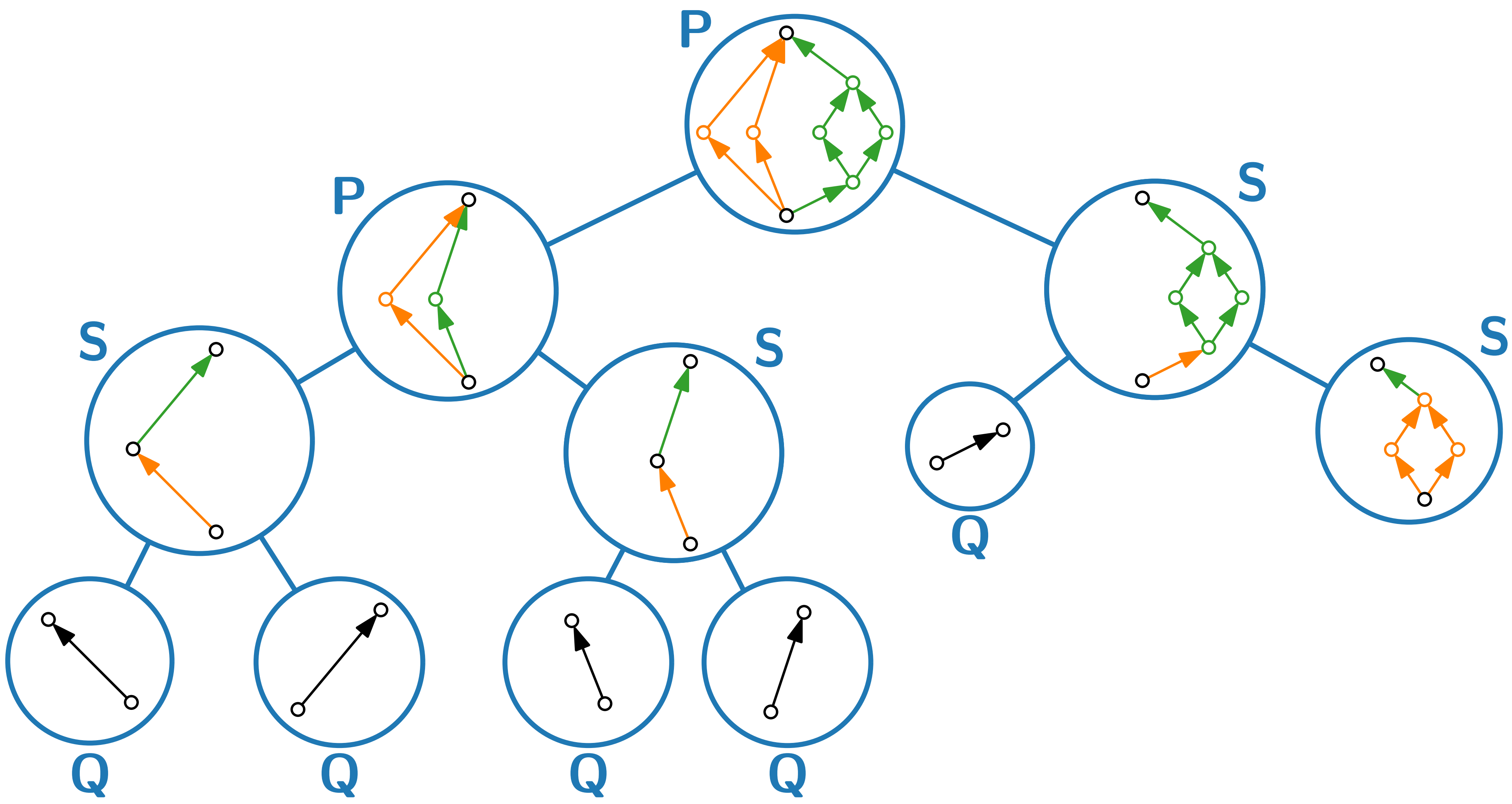
Series-Parallel Graphs – Decomposition Example



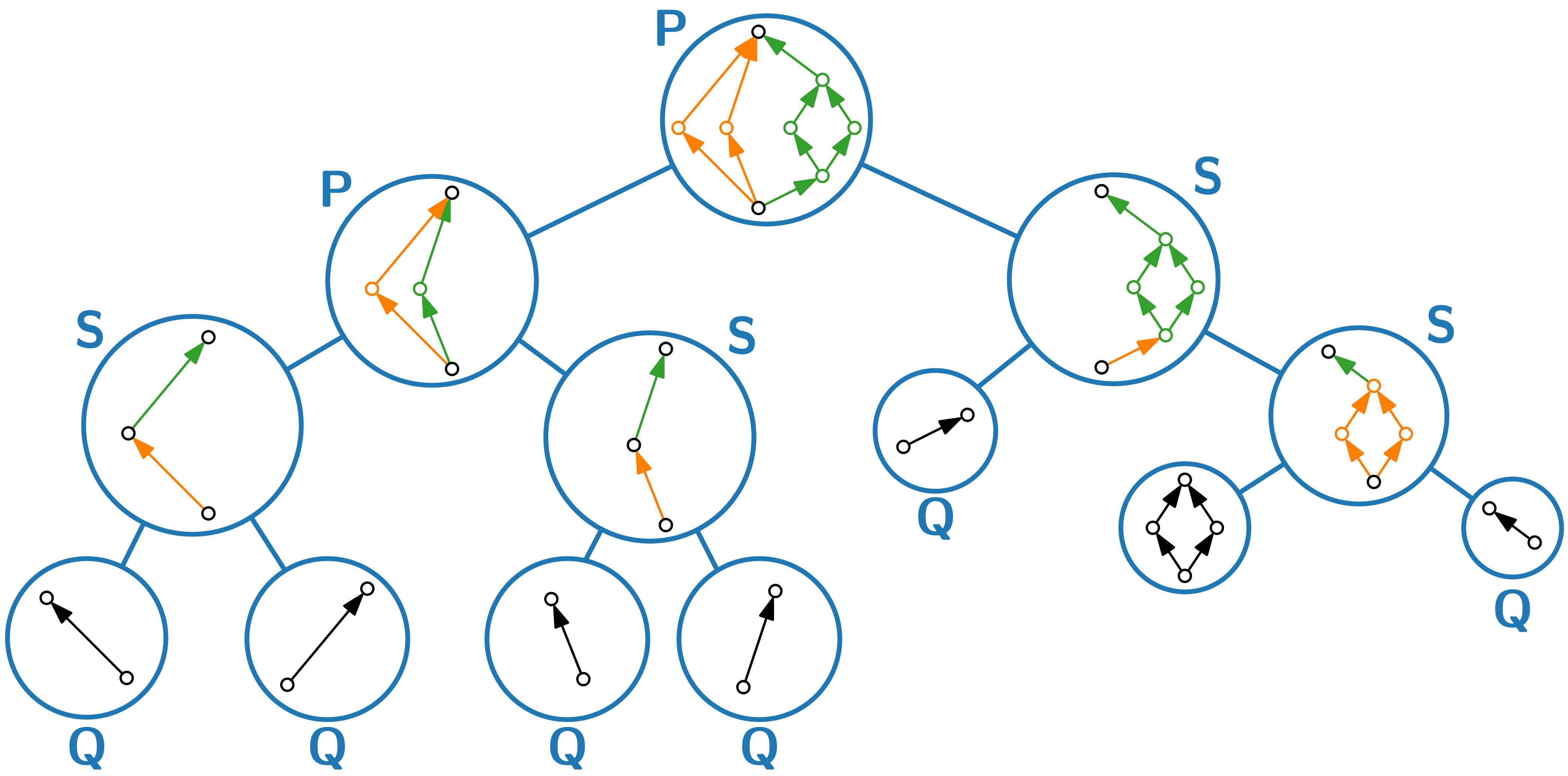
Series-Parallel Graphs – Decomposition Example



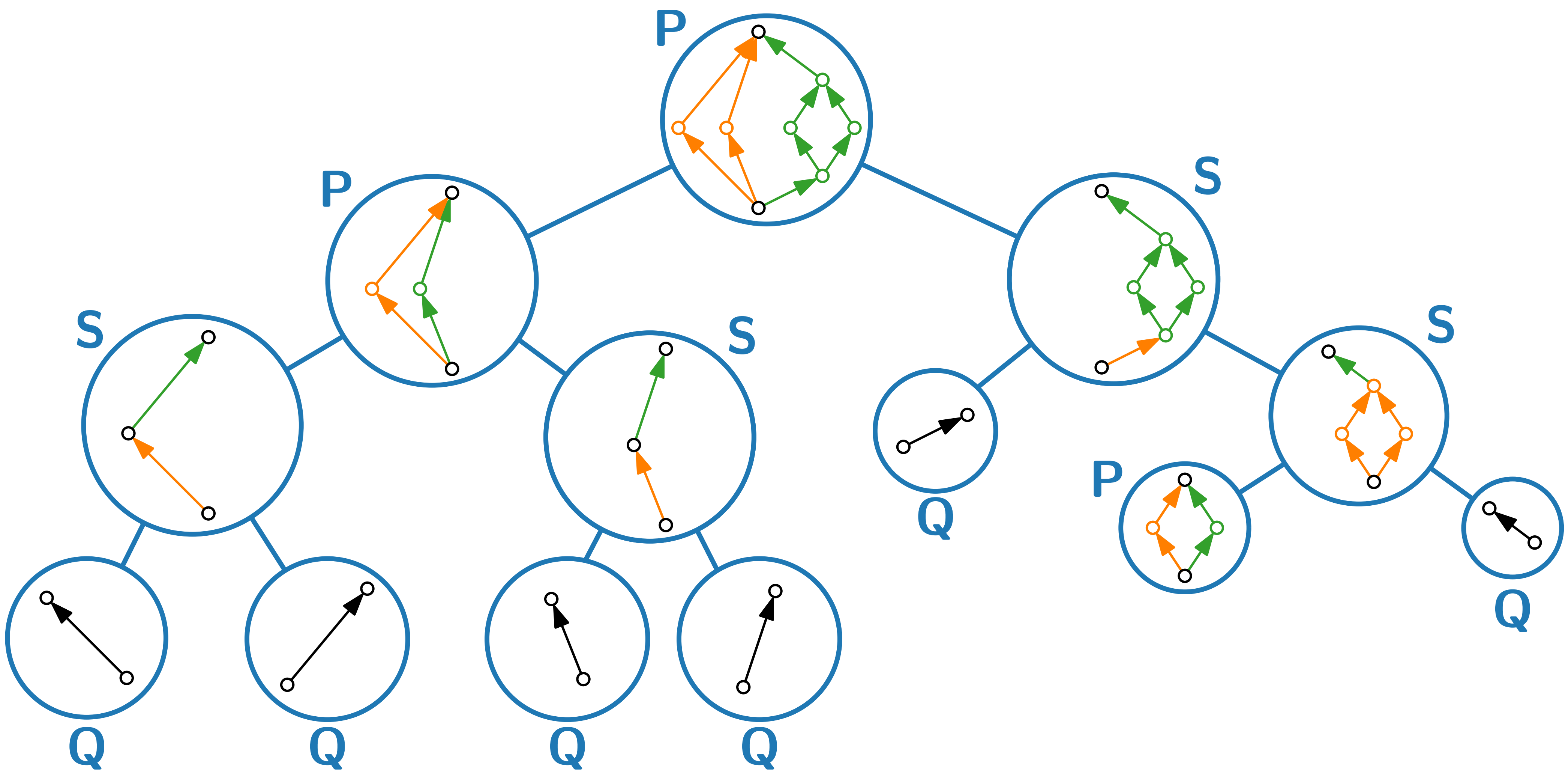
Series-Parallel Graphs – Decomposition Example



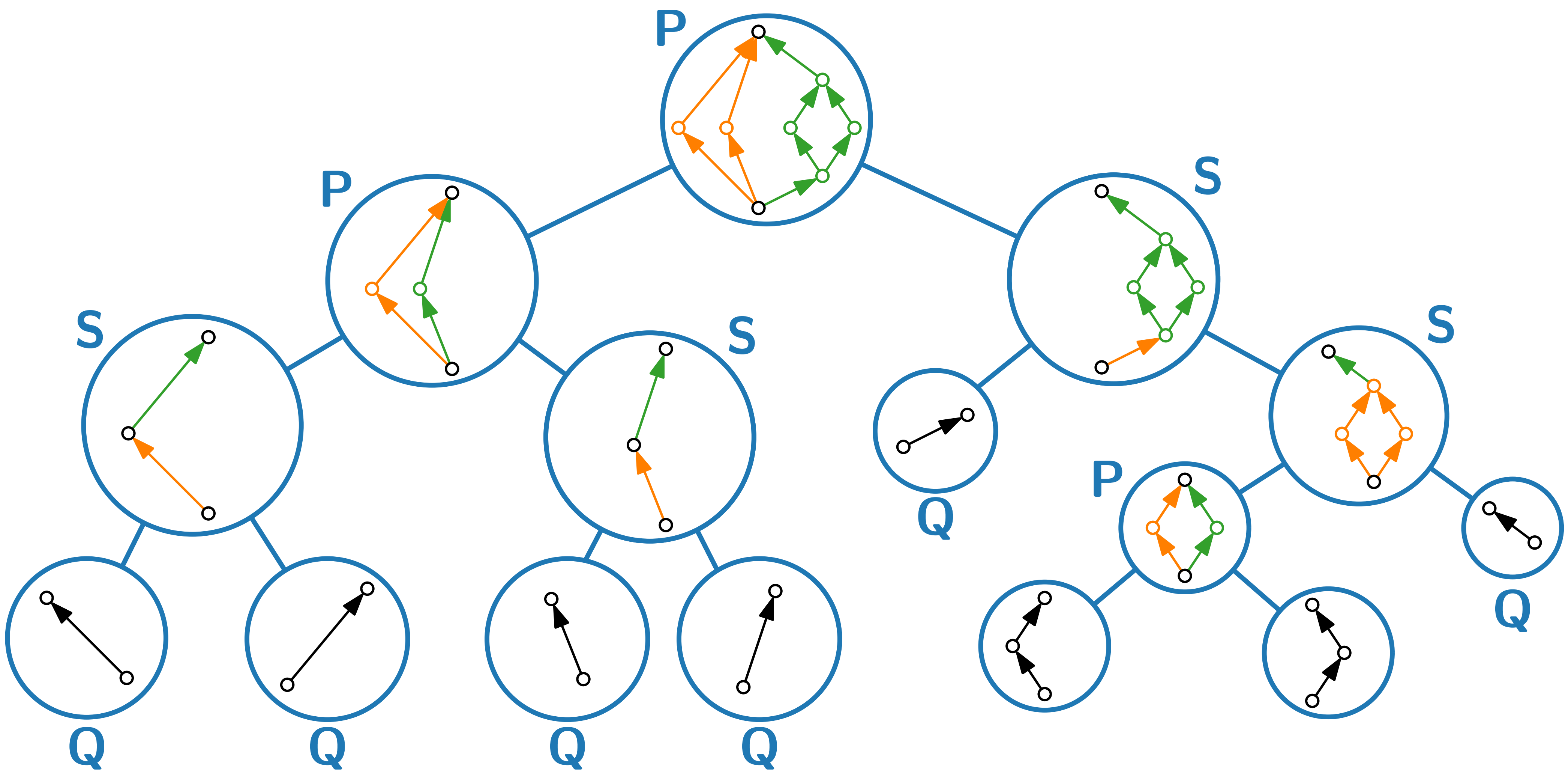
Series-Parallel Graphs – Decomposition Example



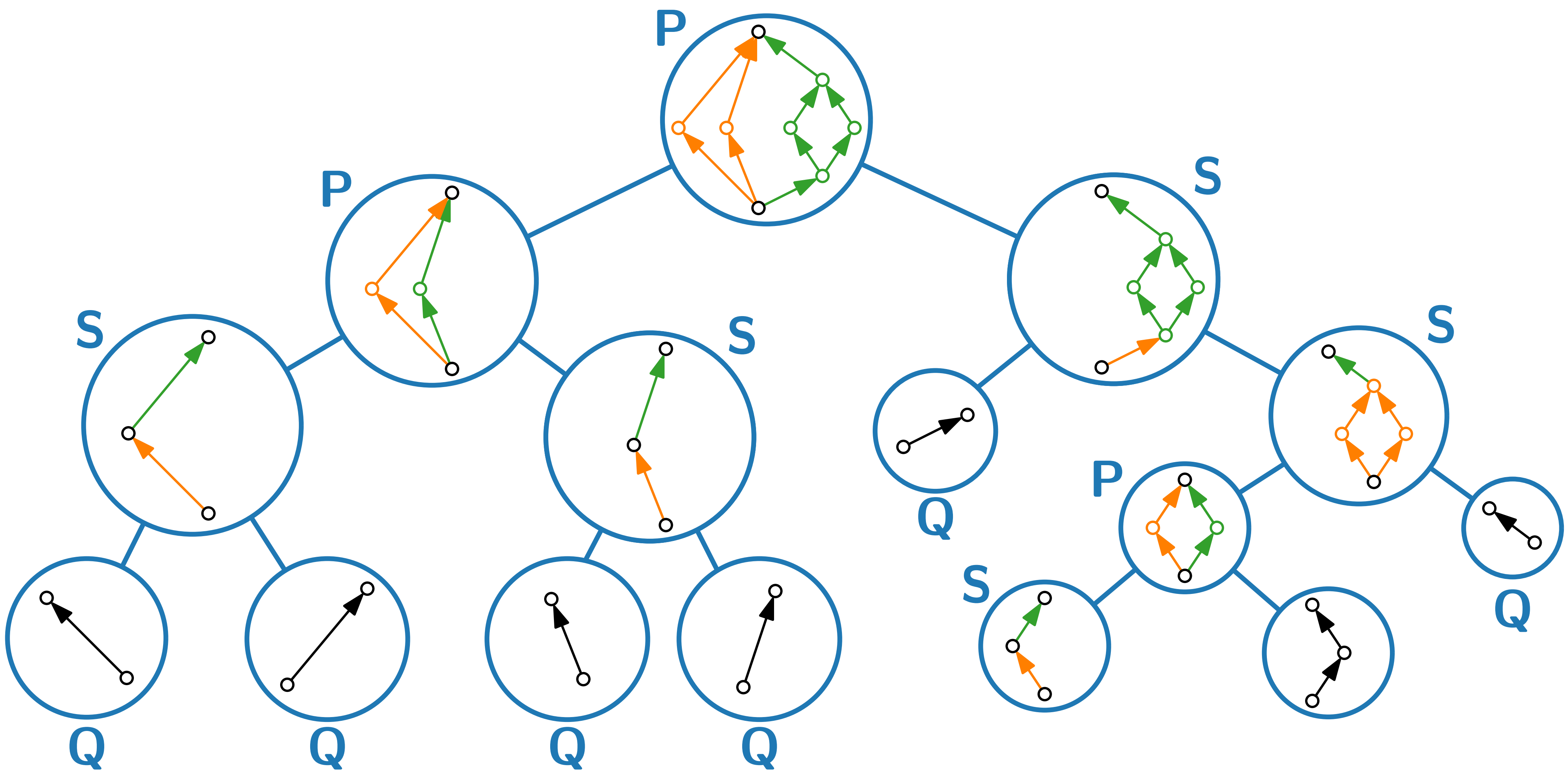
Series-Parallel Graphs – Decomposition Example



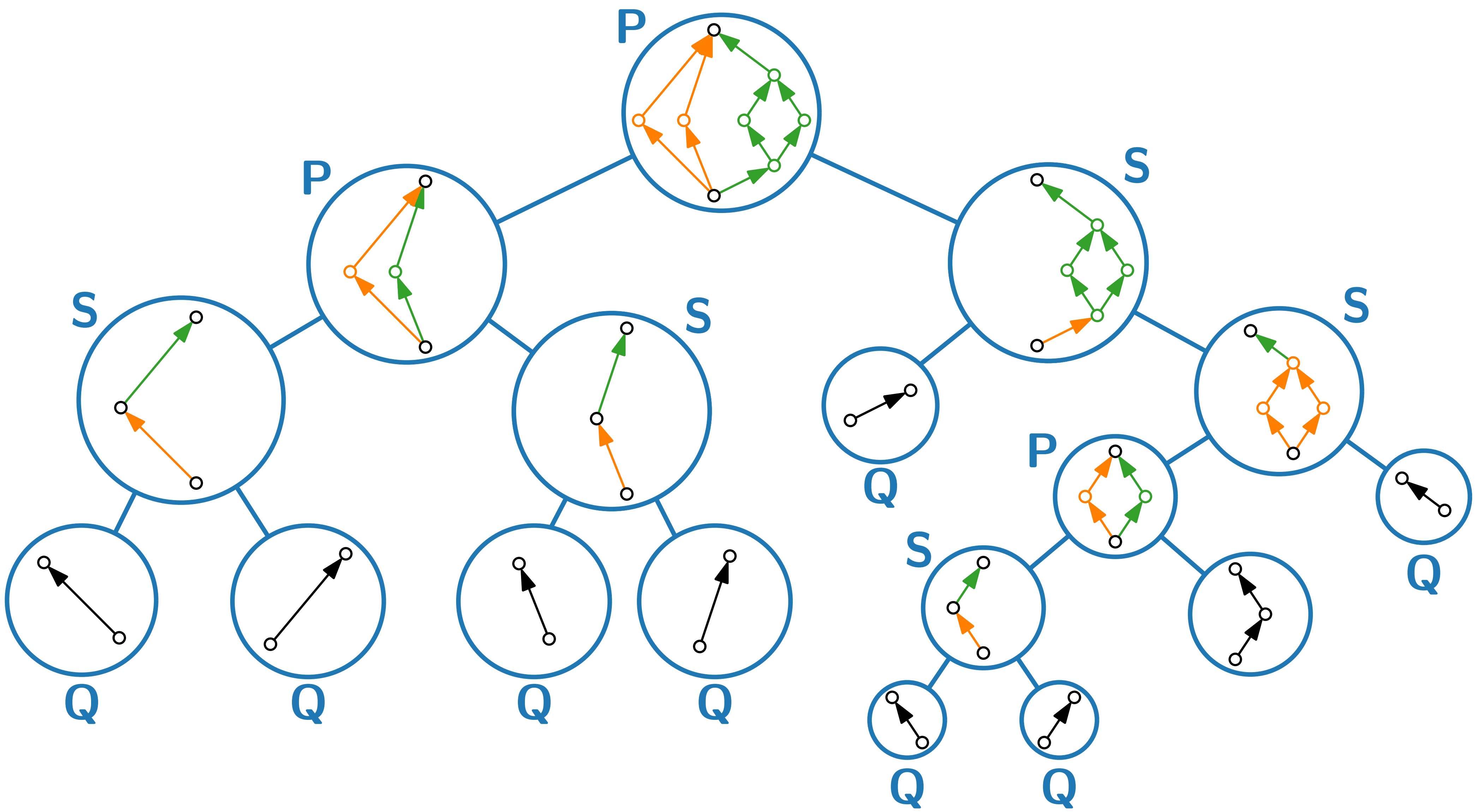
Series-Parallel Graphs – Decomposition Example



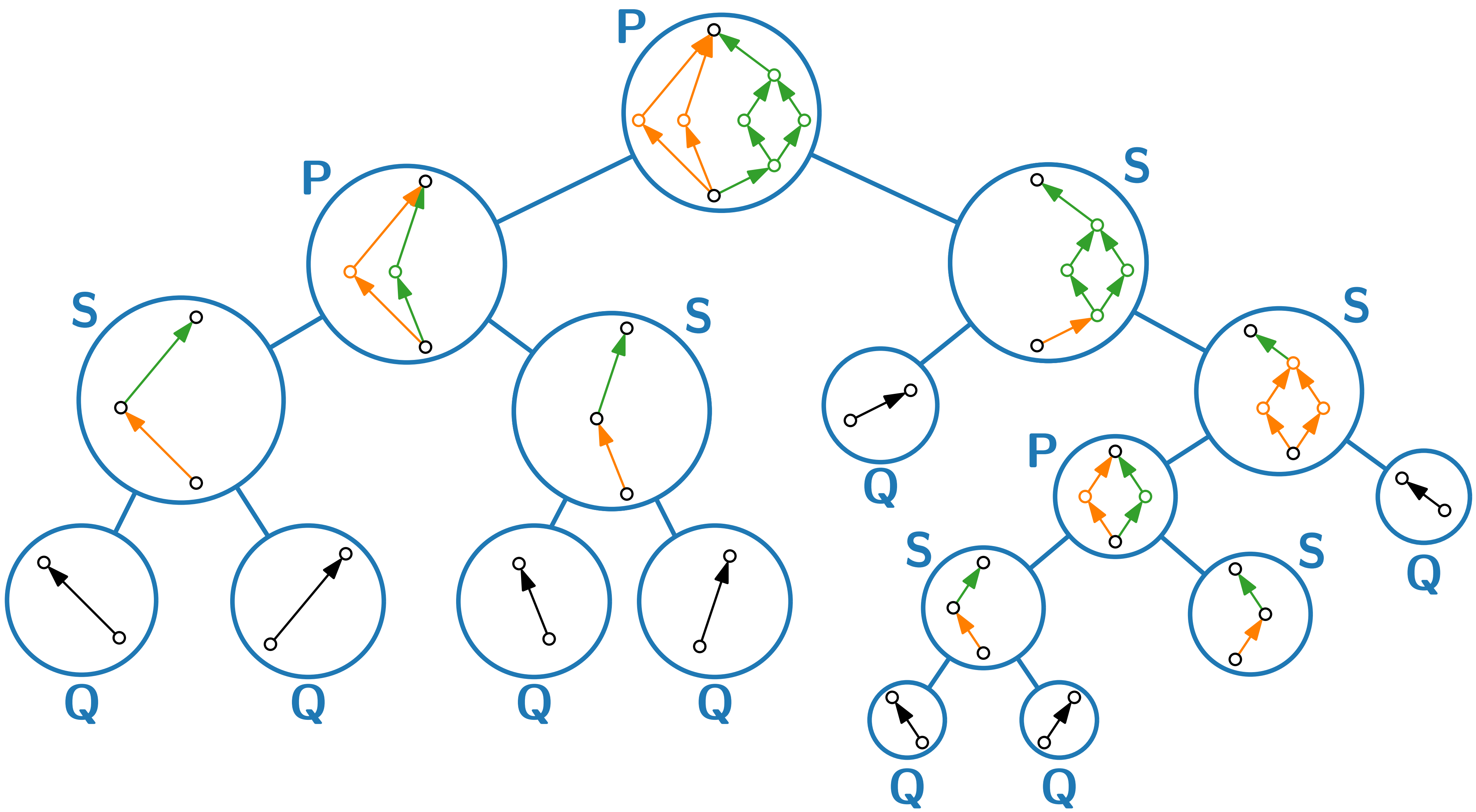
Series-Parallel Graphs – Decomposition Example



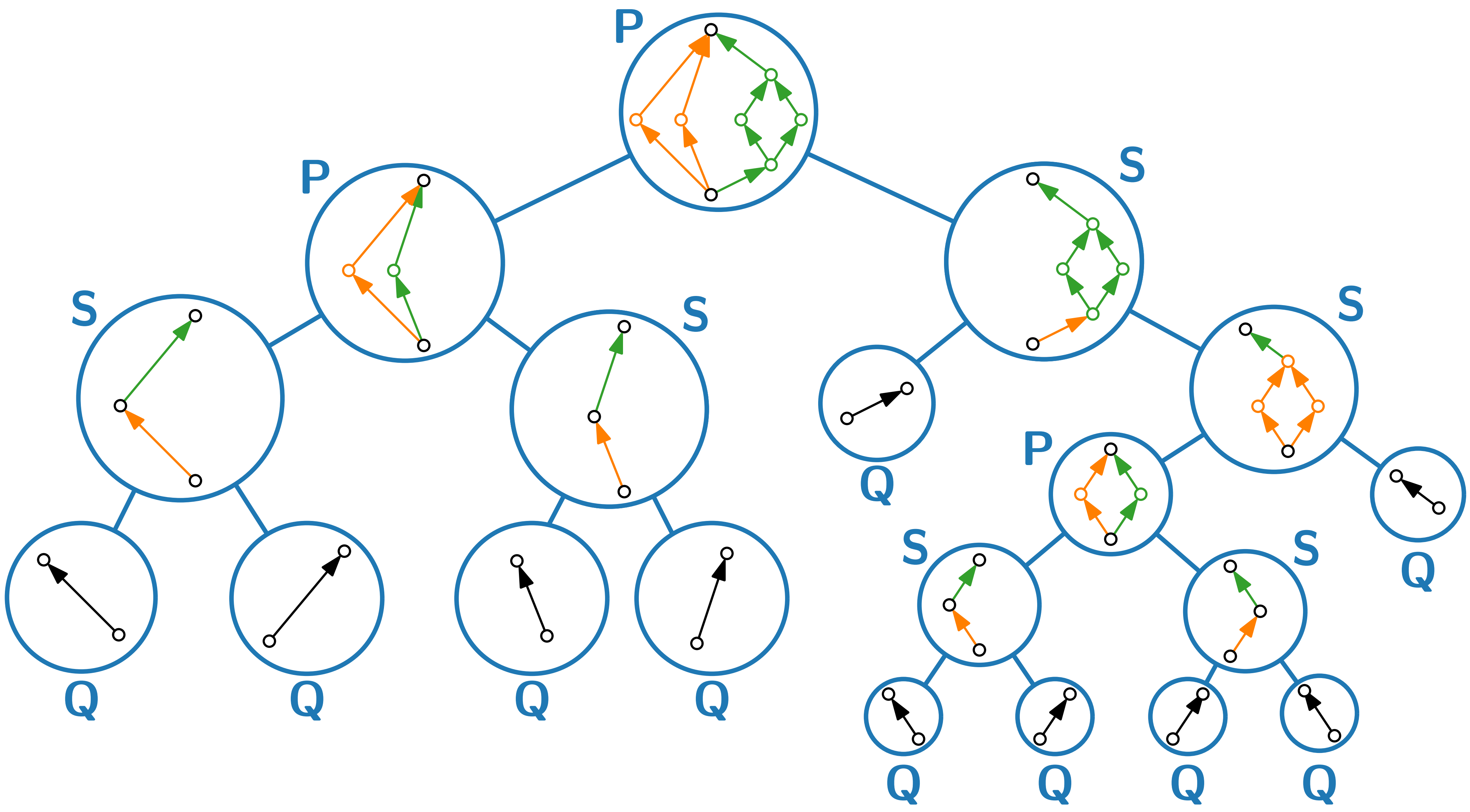
Series-Parallel Graphs – Decomposition Example



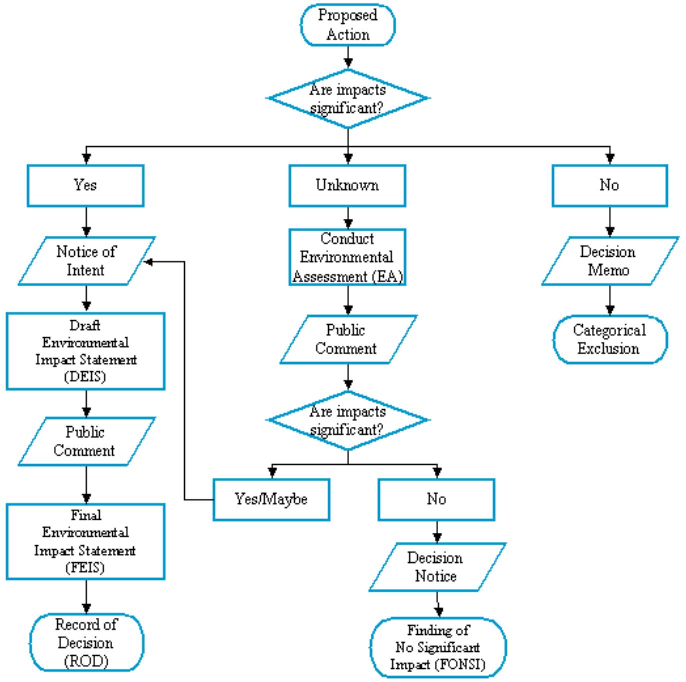
Series-Parallel Graphs – Decomposition Example



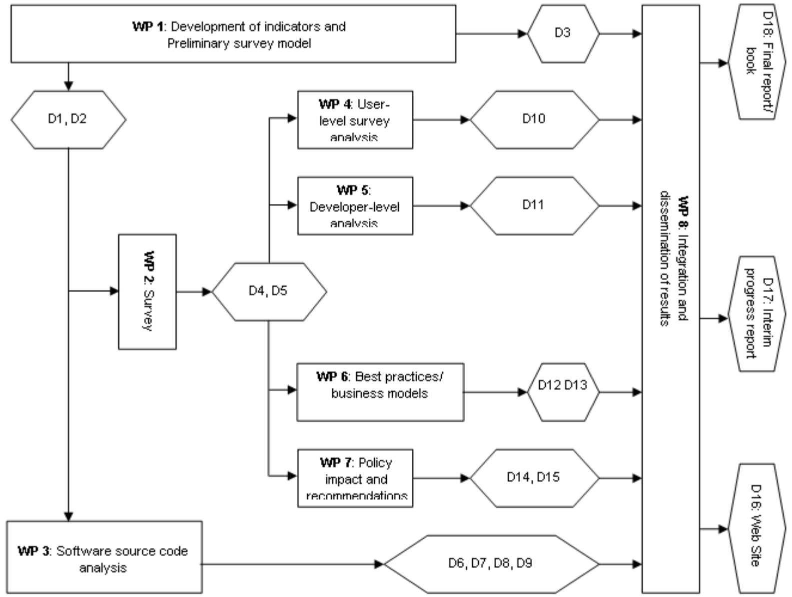
Series-Parallel Graphs – Decomposition Example



Series-Parallel Graphs – Applications



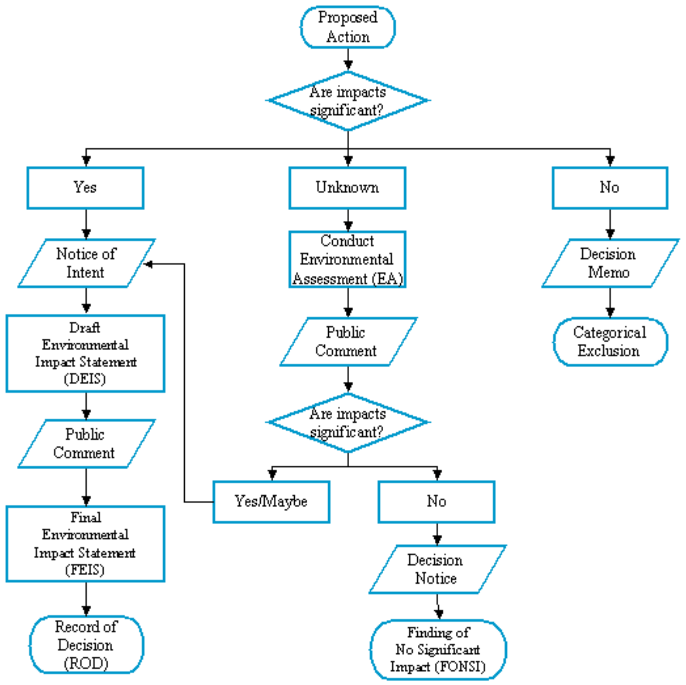
Flowcharts



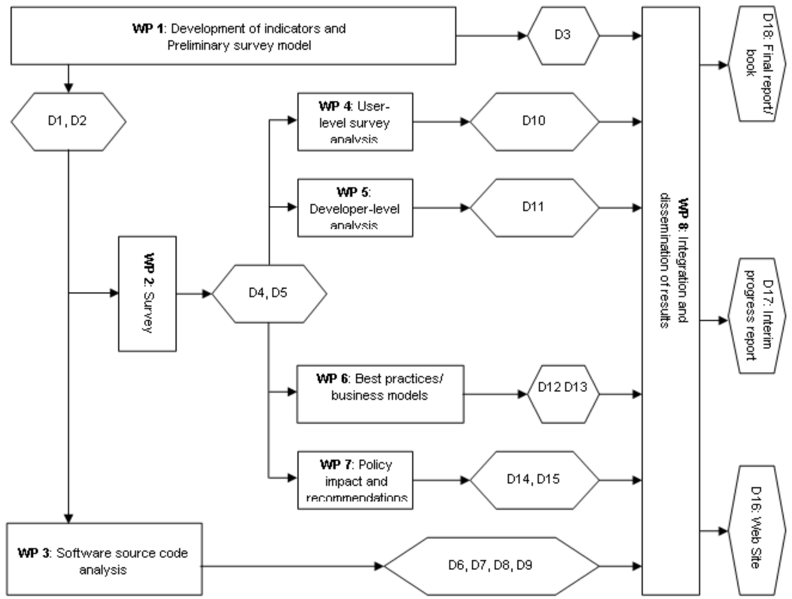
PERT-Diagrams

(Program Evaluation and Review Technique)

Series-Parallel Graphs – Applications



Flowcharts



PERT-Diagrams

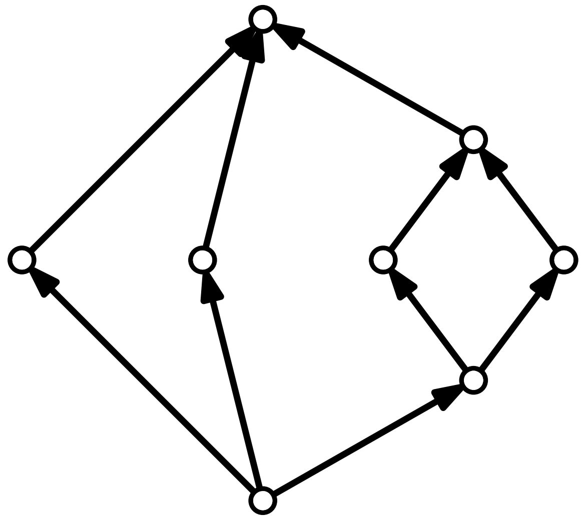
(Program Evaluation and Review Technique)

Computational complexity:

Series-parallel graphs often admit linear-time algorithms for $\mathcal{NP}$-hard problems, e.g., minimum maximal matching, MIS, Hamiltonian completion

Series-Parallel Graphs – Drawing Style

Drawing conventions

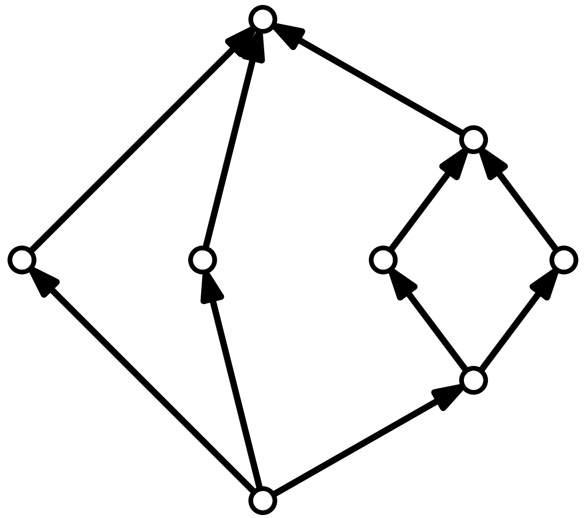


Drawing aesthetics

Series-Parallel Graphs – Drawing Style

Drawing conventions

- Planarity

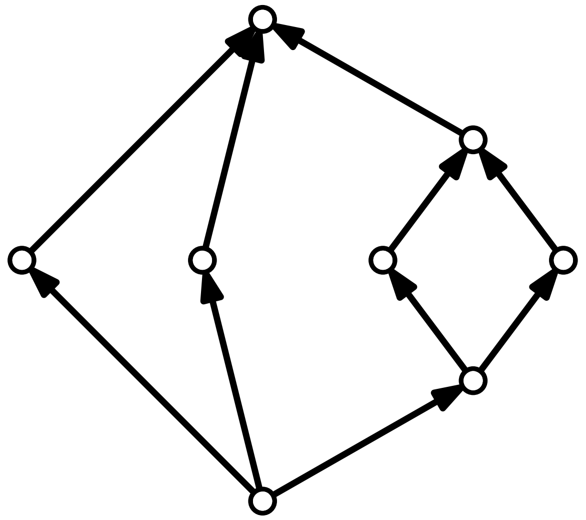


Drawing aesthetics

Series-Parallel Graphs – Drawing Style

Drawing conventions

- Planarity
- Straight-line edges



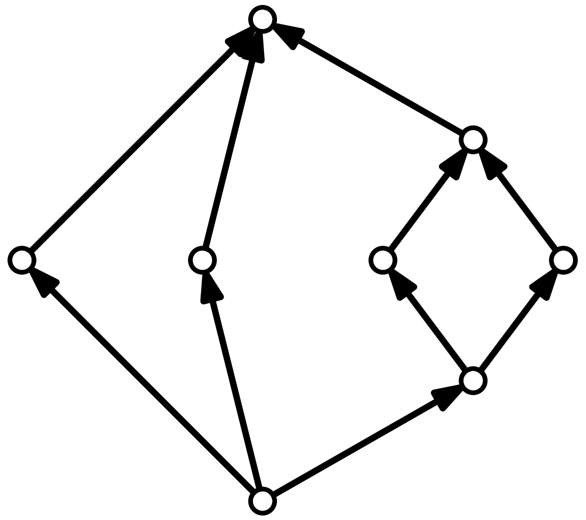
Drawing aesthetics

Series-Parallel Graphs – Drawing Style

Drawing conventions

- Planarity
- Straight-line edges
- Upward

Drawing aesthetics



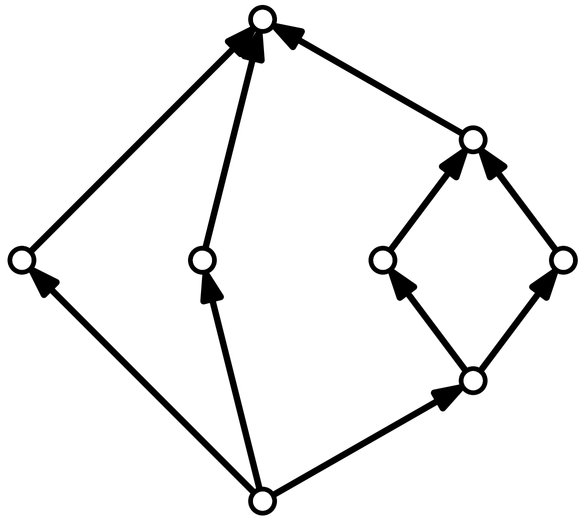
Series-Parallel Graphs – Drawing Style

Drawing conventions

- Planarity
- Straight-line edges
- Upward

Drawing aesthetics

- Area



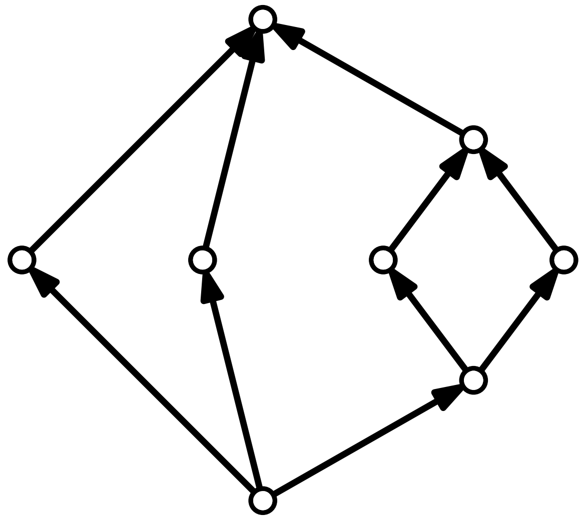
Series-Parallel Graphs – Drawing Style

Drawing conventions

- Planarity
- Straight-line edges
- Upward

Drawing aesthetics

- Area
- Symmetry



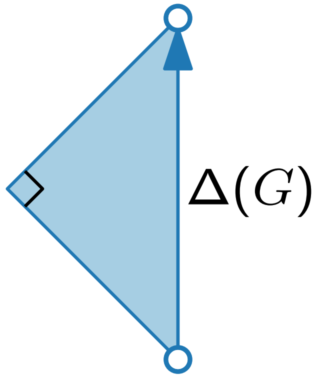
Series-Parallel Graphs – Straight-Line Drawings

Divide & conquer algorithm using the decomposition tree

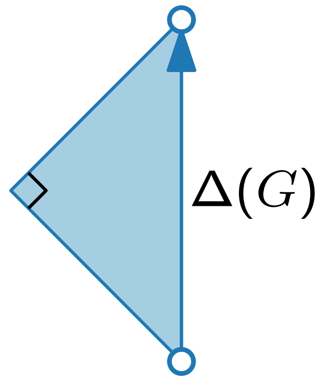
Series-Parallel Graphs – Straight-Line Drawings

Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



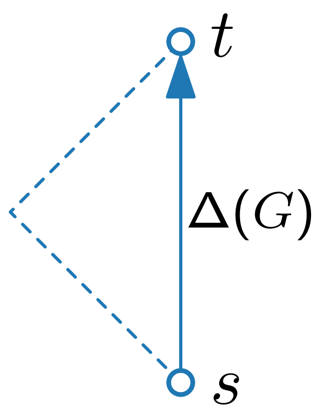
Series-Parallel Graphs – Straight-Line Drawings



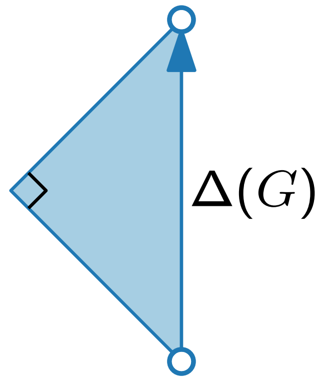
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$

Base case: Q-nodes



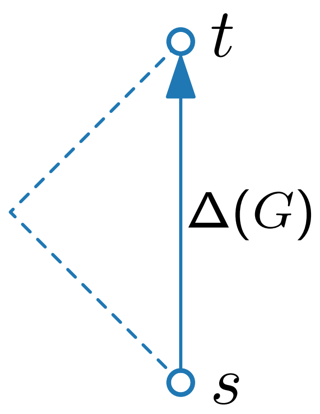
Series-Parallel Graphs – Straight-Line Drawings



Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$

Base case: Q-nodes

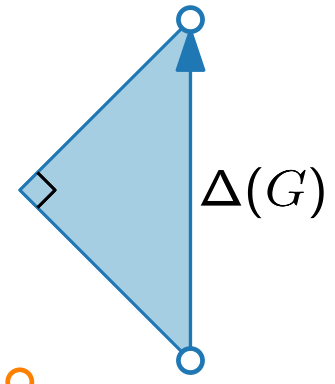


Divide: Draw G_1 and G_2 first

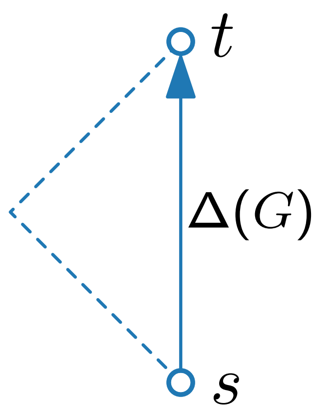
Series-Parallel Graphs – Straight-Line Drawings

Divide & conquer algorithm using the decomposition tree

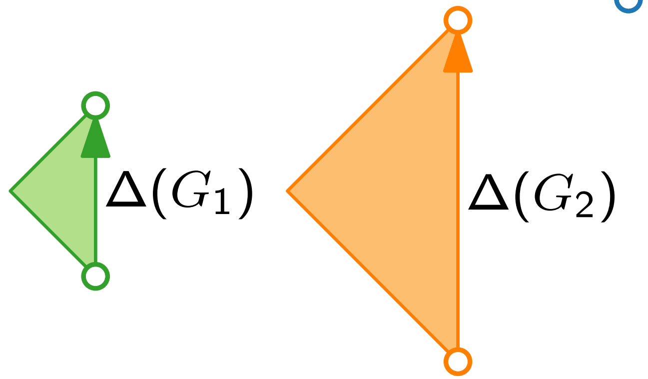
- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



Base case: Q-nodes



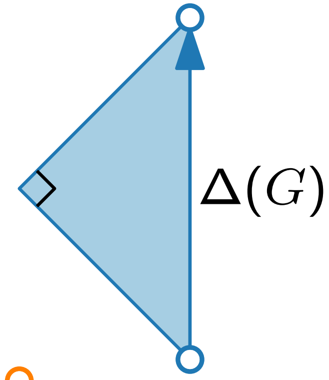
Divide: Draw G_1 and G_2 first



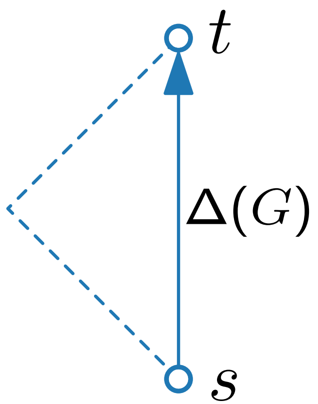
Series-Parallel Graphs – Straight-Line Drawings

Divide & conquer algorithm using the decomposition tree

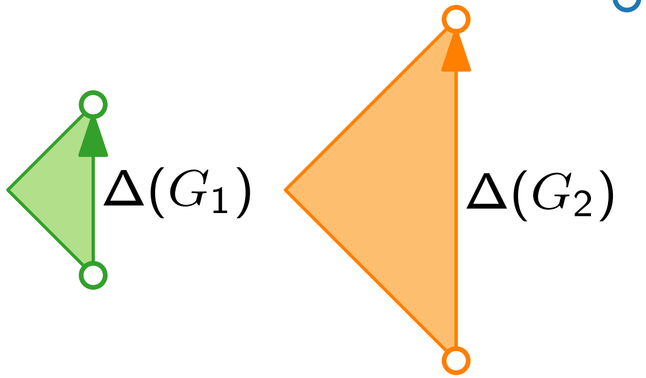
- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



Base case: Q-nodes



Divide: Draw G_1 and G_2 first

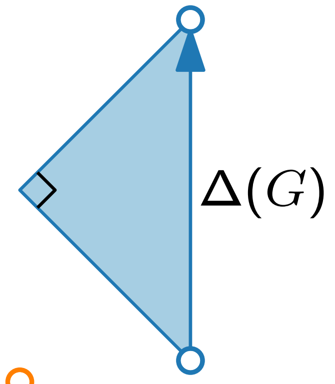


Conquer:

Series-Parallel Graphs – Straight-Line Drawings

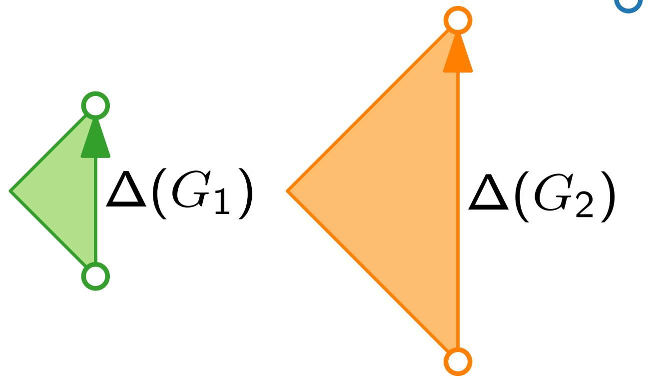
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



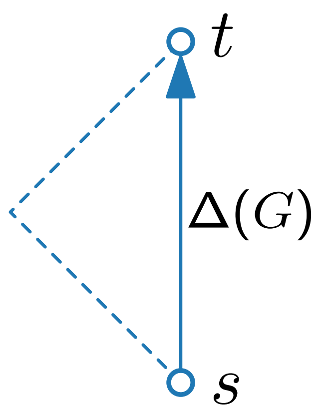
Base case: Q-nodes

Divide: Draw G_1 and G_2 first



Conquer:

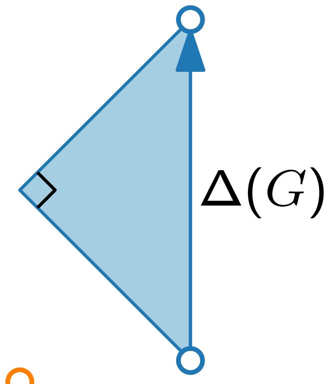
- S-nodes / series composition



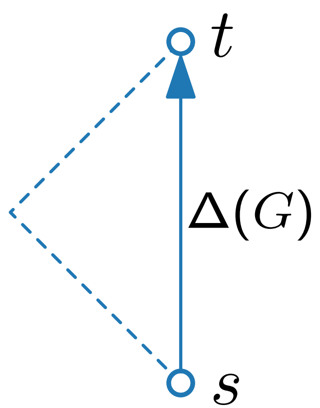
Series-Parallel Graphs – Straight-Line Drawings

Divide & conquer algorithm using the decomposition tree

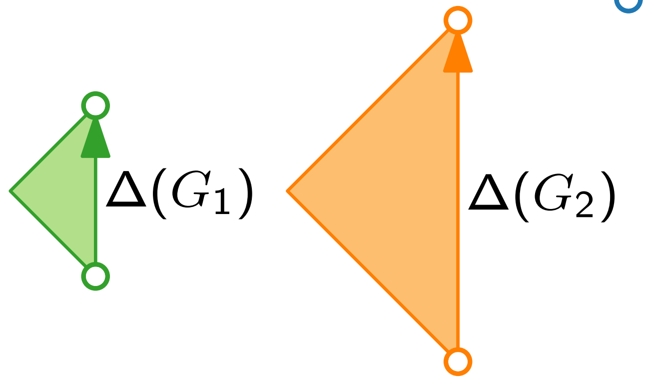
- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



Base case: Q-nodes

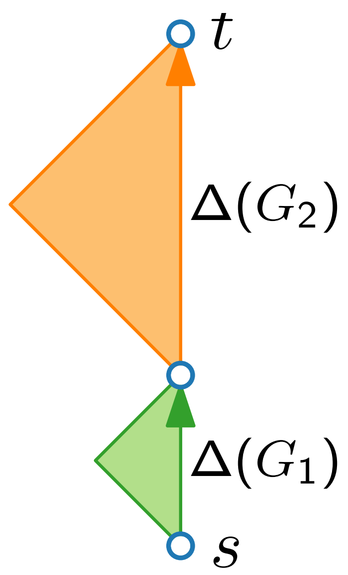


Divide: Draw G_1 and G_2 first

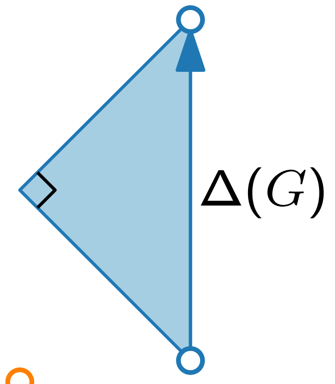


Conquer:

- S-nodes / series composition



Series-Parallel Graphs – Straight-Line Drawings

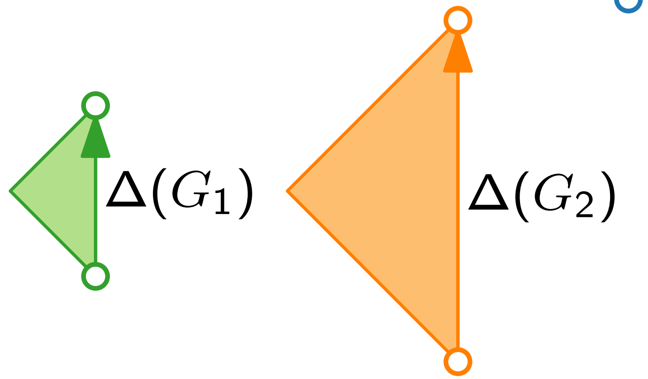


Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$

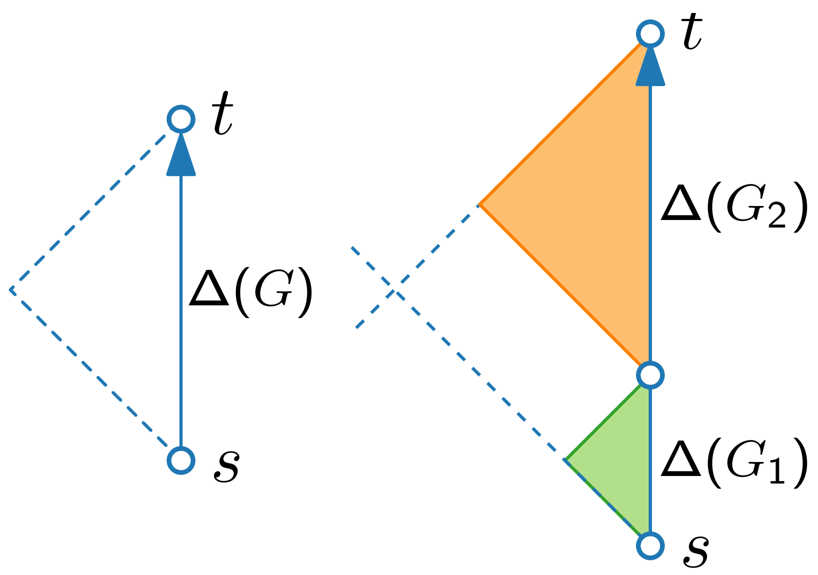
Base case: Q-nodes

Divide: Draw G_1 and G_2 first



Conquer:

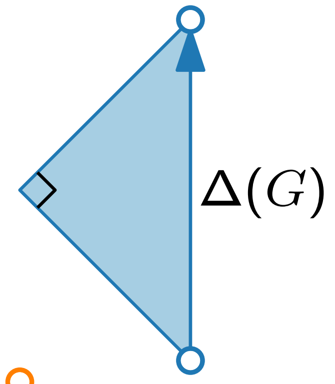
- S-nodes / series composition



Series-Parallel Graphs – Straight-Line Drawings

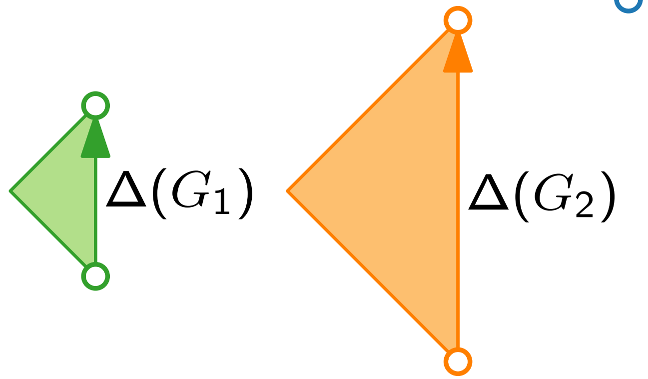
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



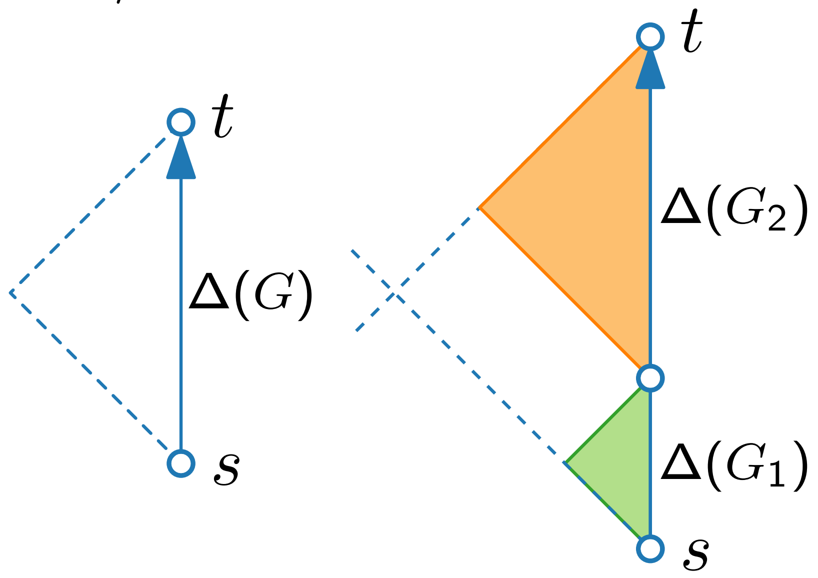
Base case: Q-nodes

Divide: Draw G_1 and G_2 first



Conquer:

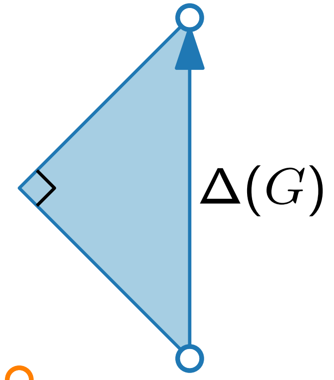
- S-nodes / series composition
- P-nodes / parallel composition



Series-Parallel Graphs – Straight-Line Drawings

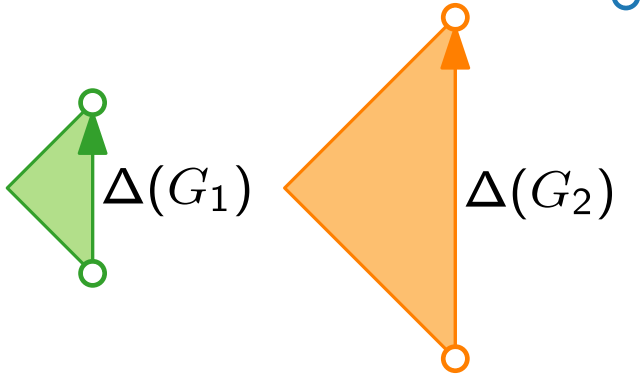
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



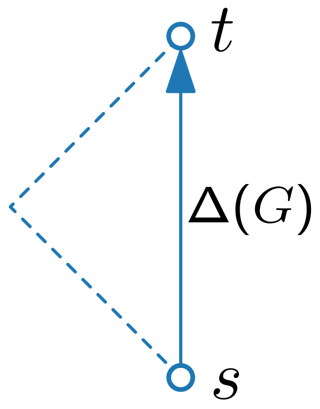
Base case: Q-nodes

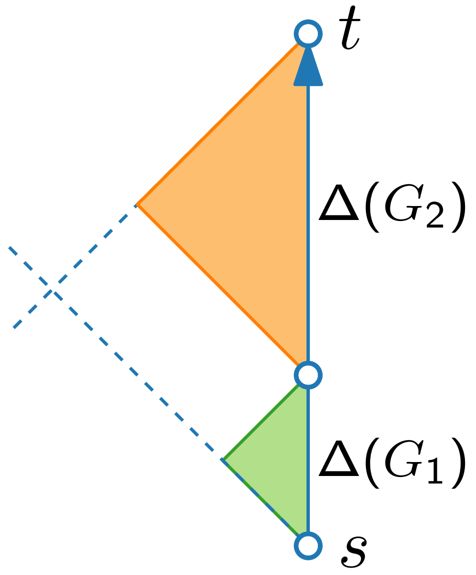
Divide: Draw G_1 and G_2 first

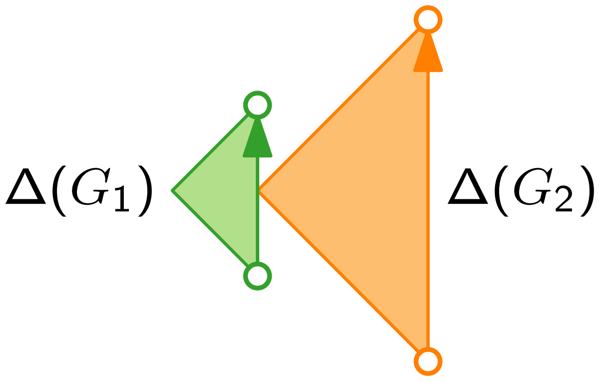


Conquer:

- S-nodes / series composition
- P-nodes / parallel composition



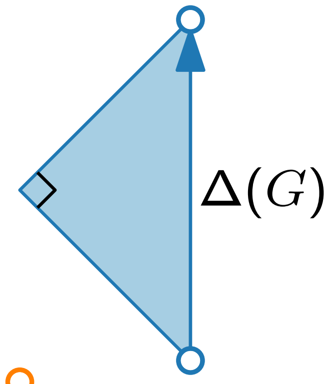




Series-Parallel Graphs – Straight-Line Drawings

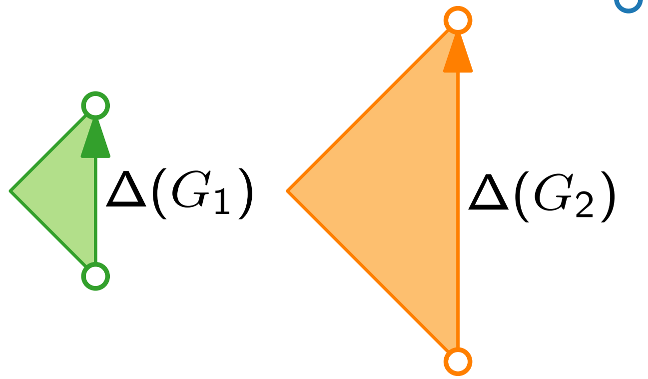
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



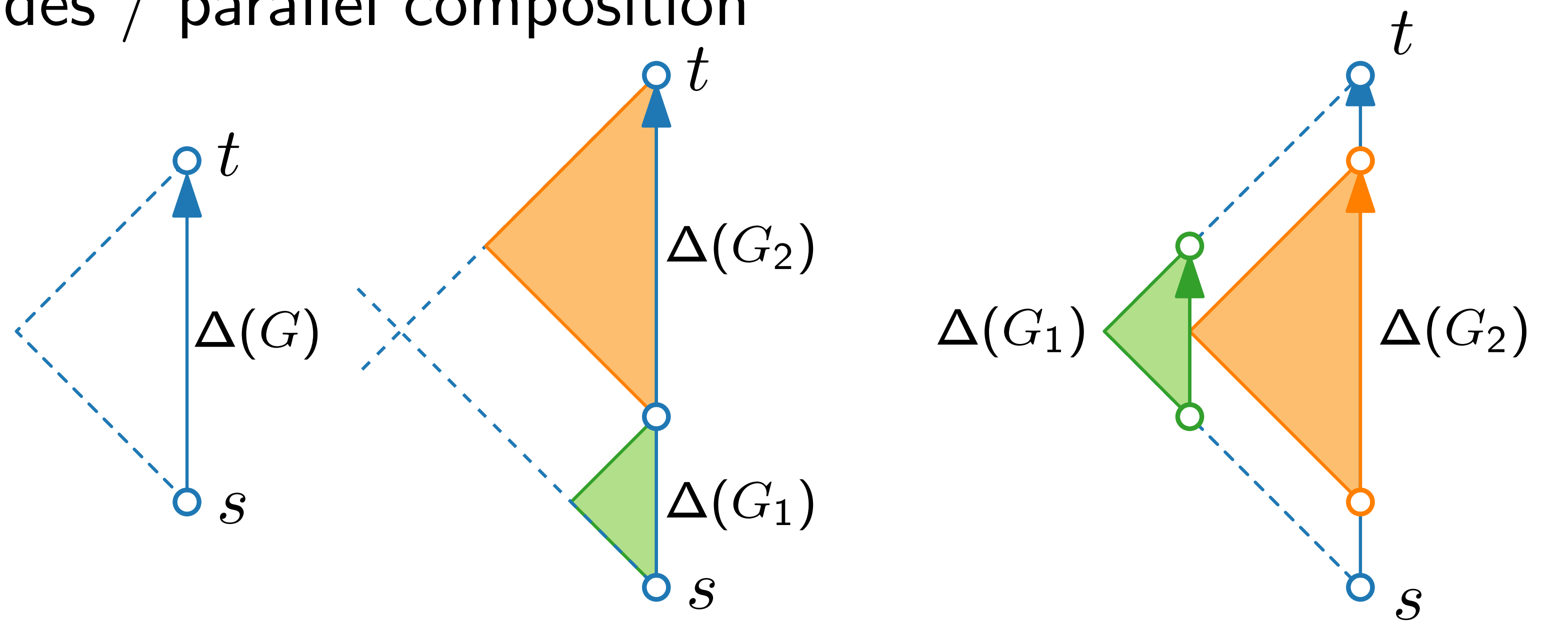
Base case: Q-nodes

Divide: Draw G_1 and G_2 first



Conquer:

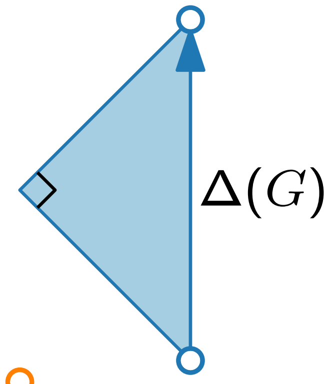
- S-nodes / series composition
- P-nodes / parallel composition



Series-Parallel Graphs – Straight-Line Drawings

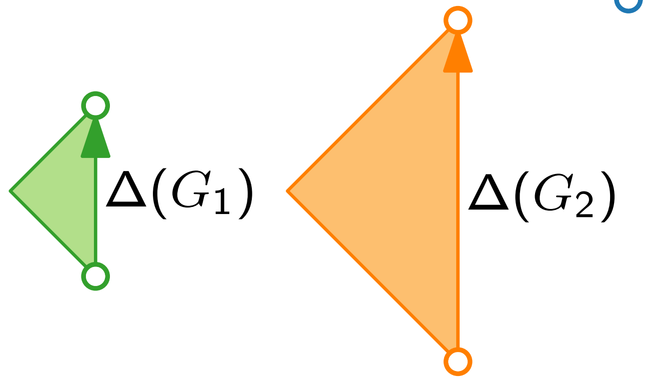
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



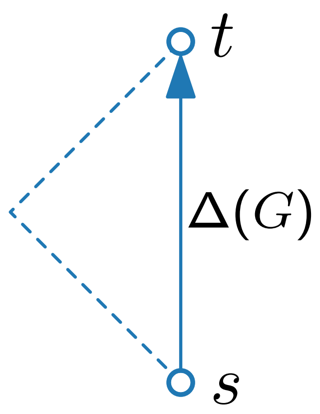
Base case: Q-nodes

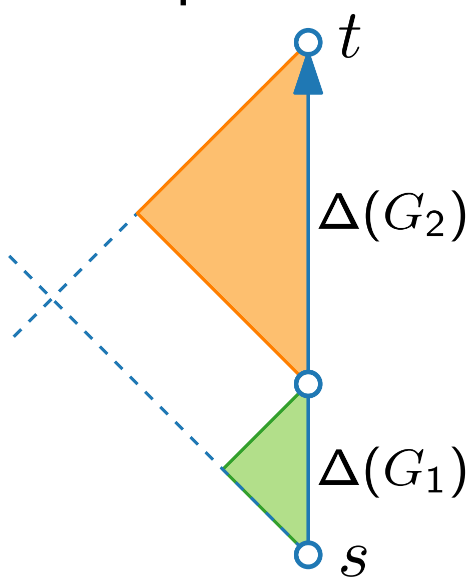
Divide: Draw G_1 and G_2 first

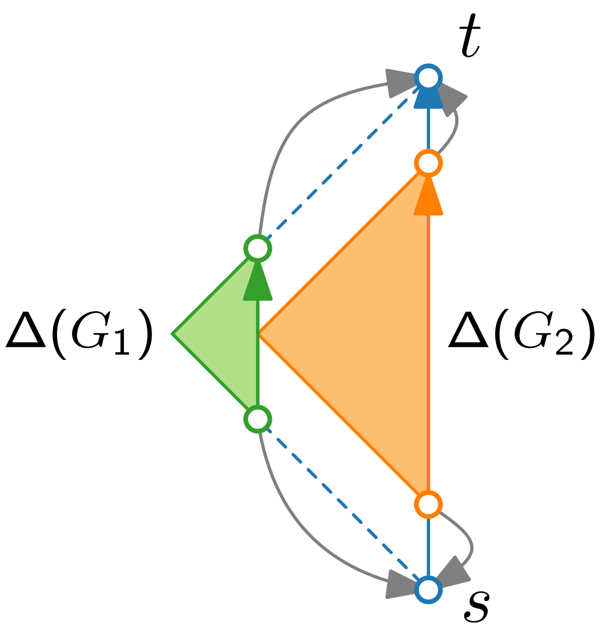


Conquer:

- S-nodes / series composition
- P-nodes / parallel composition



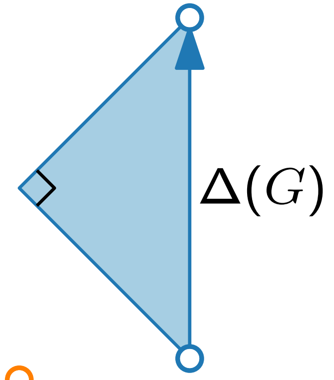




Series-Parallel Graphs – Straight-Line Drawings

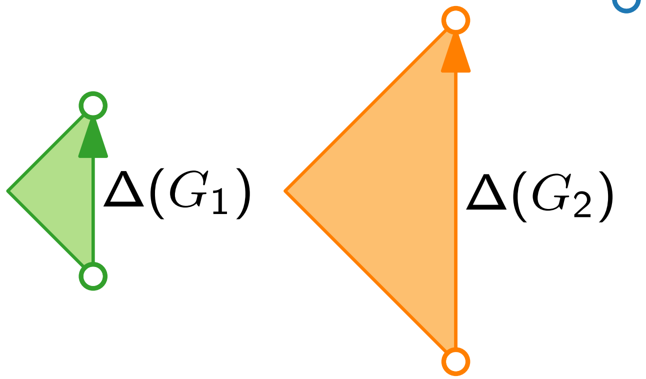
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



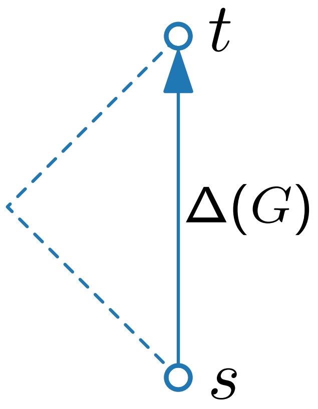
Base case: Q-nodes

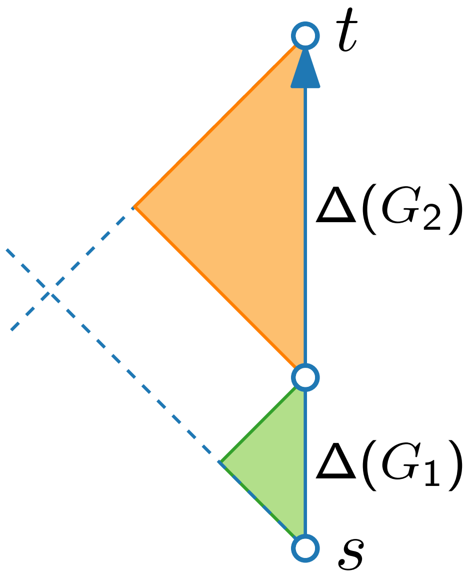
Divide: Draw G_1 and G_2 first

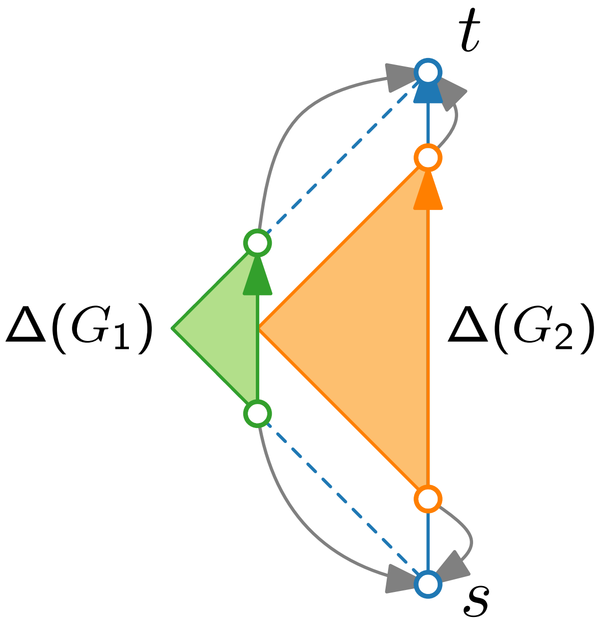


Conquer:

- S-nodes / series composition
- P-nodes / parallel composition





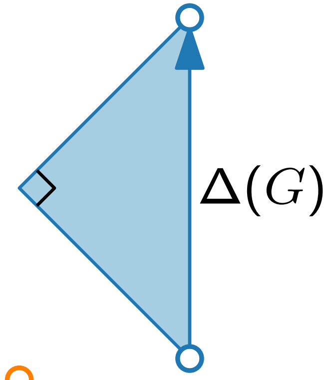


Do you see any problem?

Series-Parallel Graphs – Straight-Line Drawings

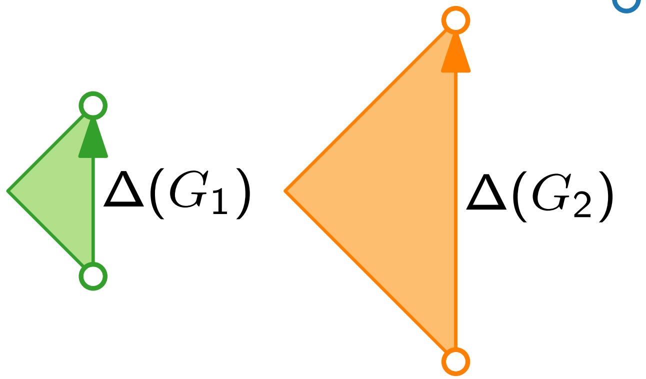
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



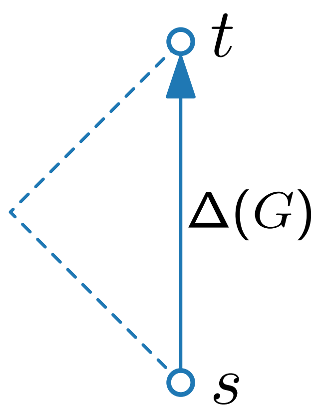
Base case: Q-nodes

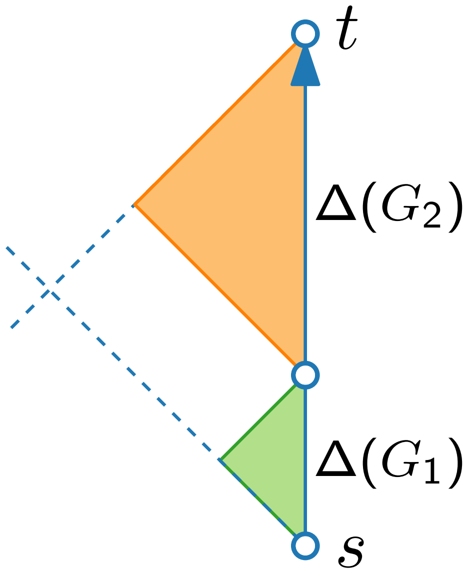
Divide: Draw G_1 and G_2 first

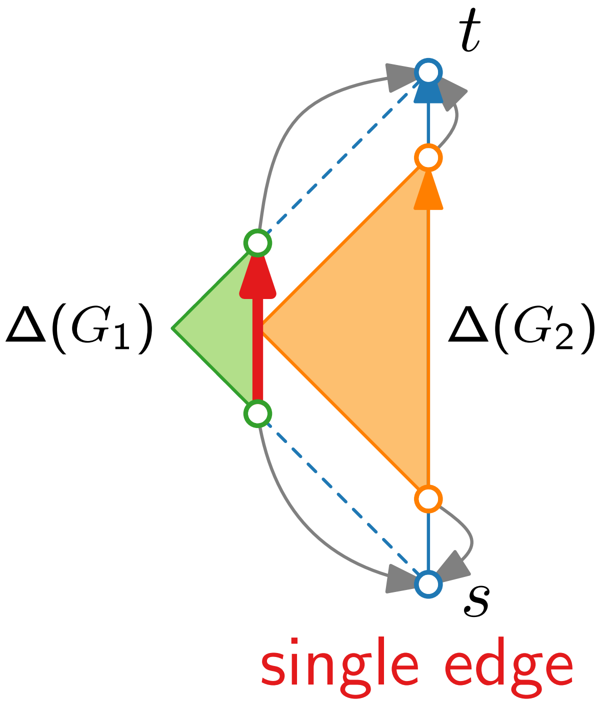


Conquer:

- S-nodes / series composition
- P-nodes / parallel composition



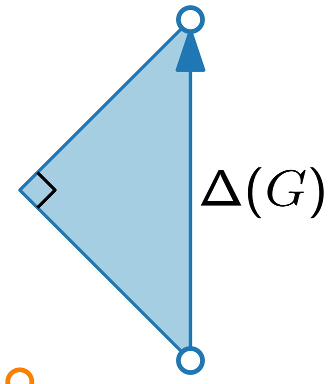




Series-Parallel Graphs – Straight-Line Drawings

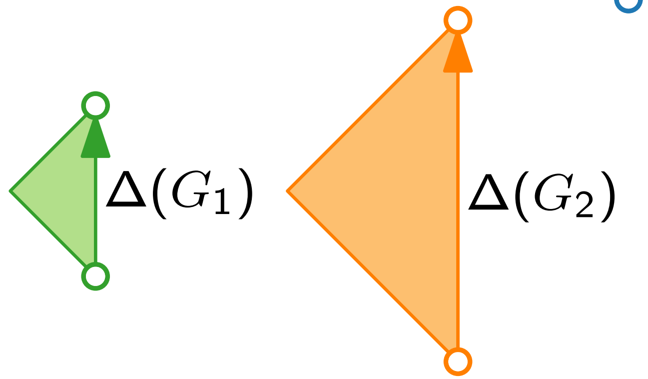
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



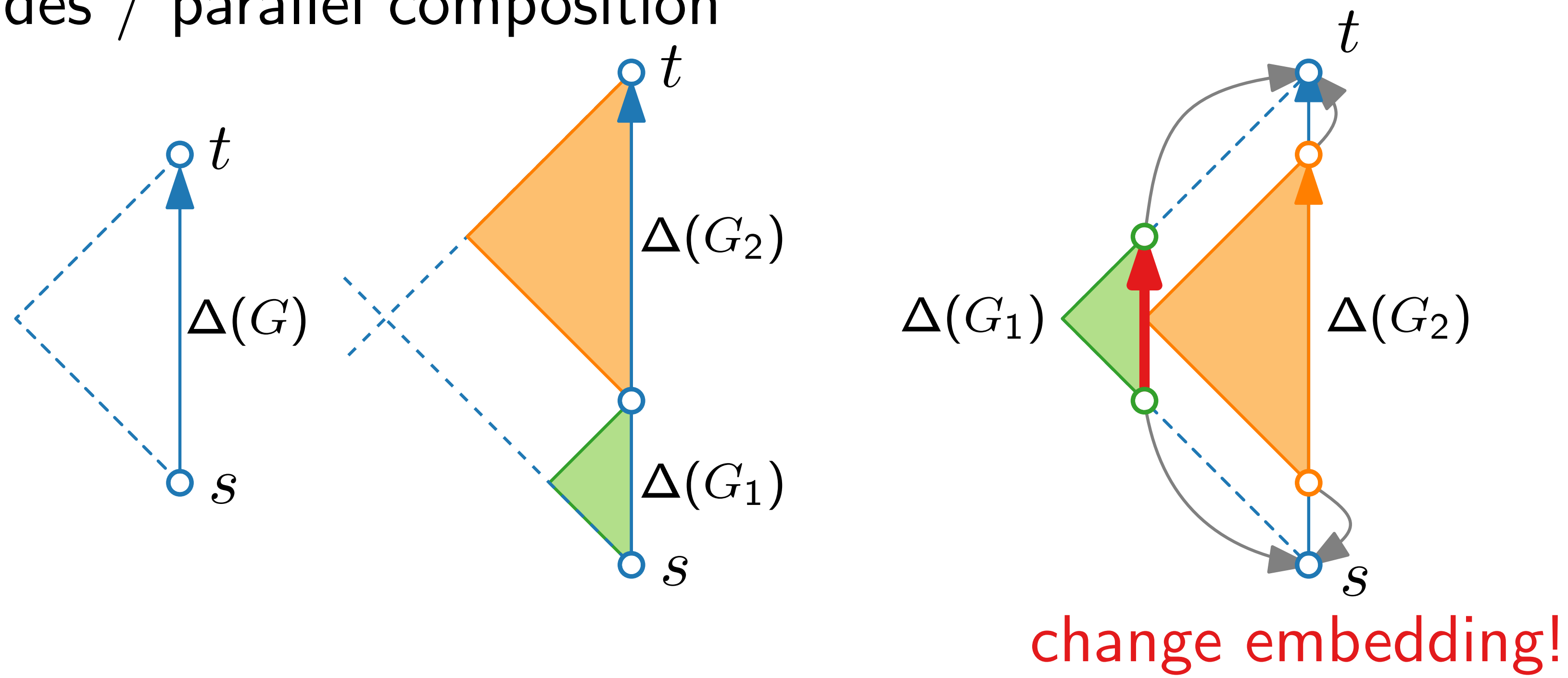
Base case: Q-nodes

Divide: Draw G_1 and G_2 first



Conquer:

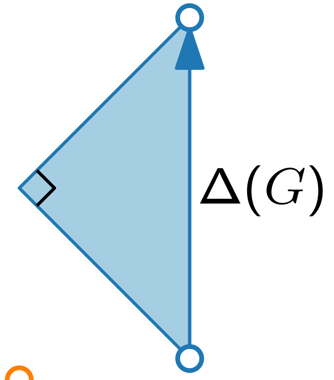
- S-nodes / series composition
- P-nodes / parallel composition



Series-Parallel Graphs – Straight-Line Drawings

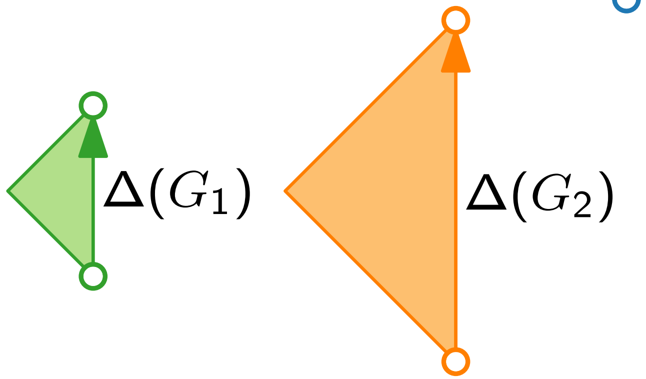
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



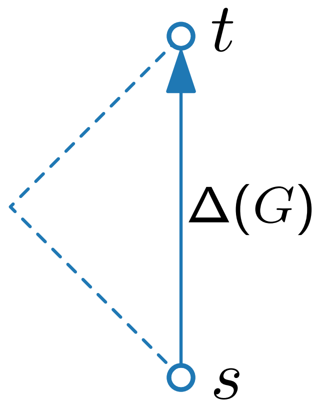
Base case: Q-nodes

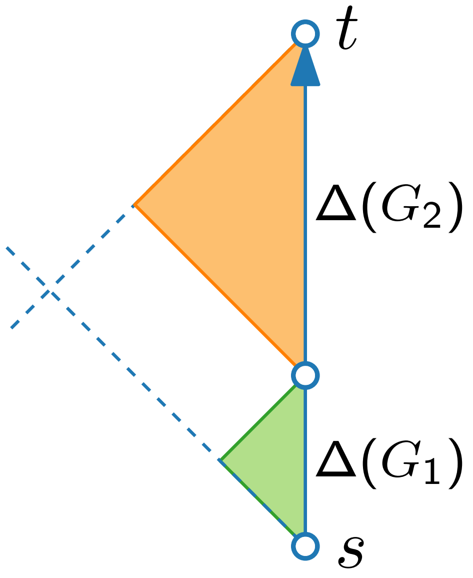
Divide: Draw G_1 and G_2 first

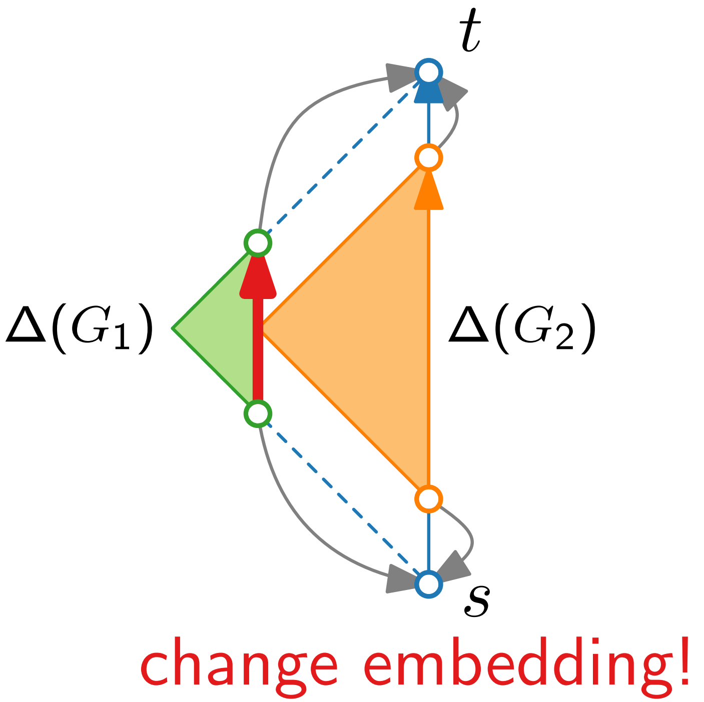


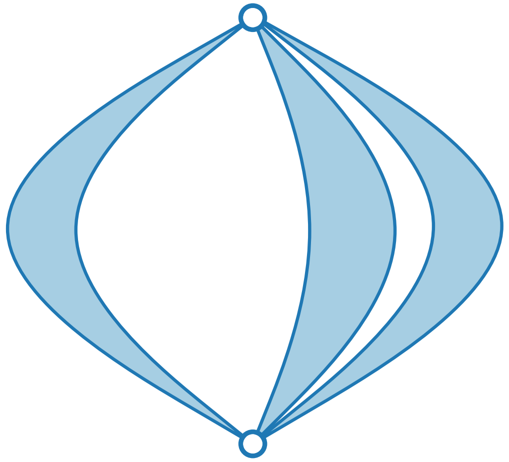
Conquer:

- S-nodes / series composition
- P-nodes / parallel composition





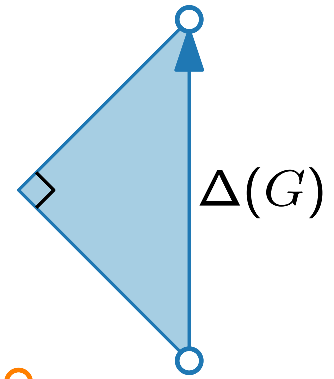




Series-Parallel Graphs – Straight-Line Drawings

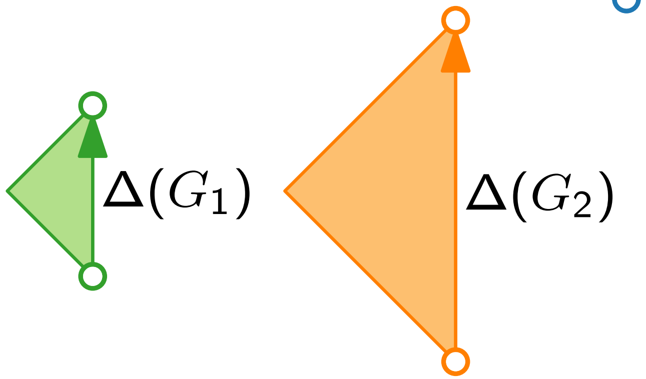
Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$



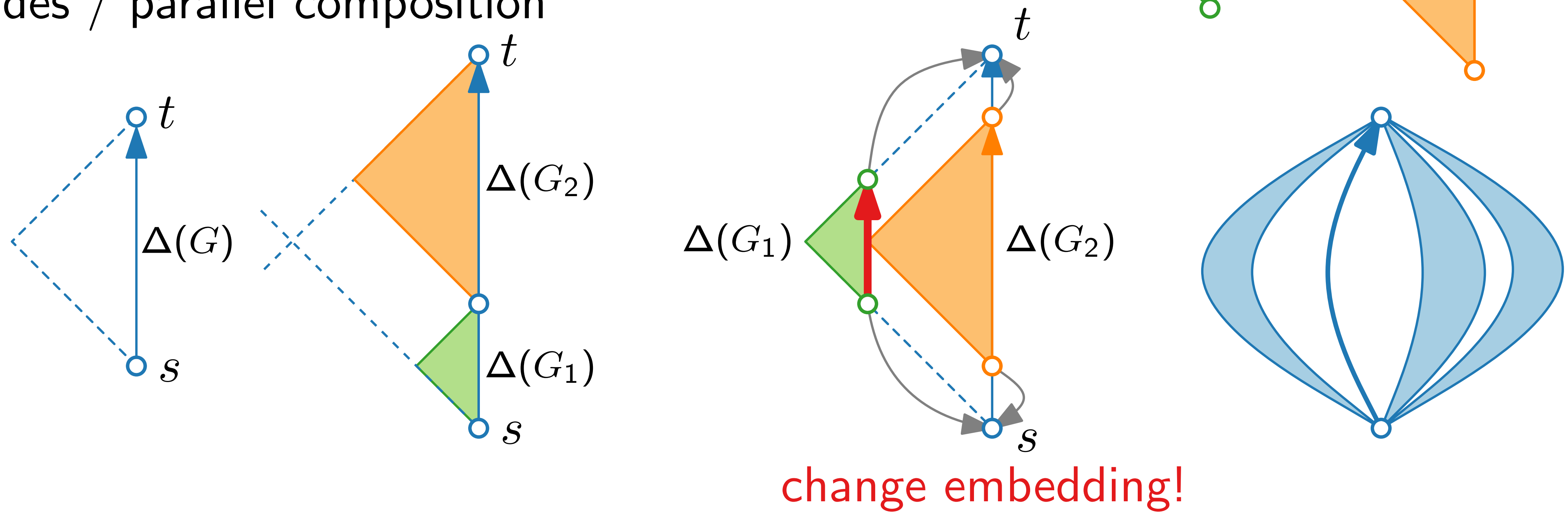
Base case: Q-nodes

Divide: Draw G_1 and G_2 first

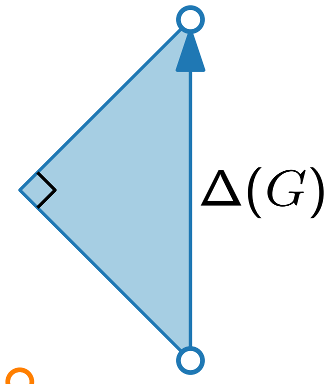


Conquer:

- S-nodes / series composition
- P-nodes / parallel composition



Series-Parallel Graphs – Straight-Line Drawings

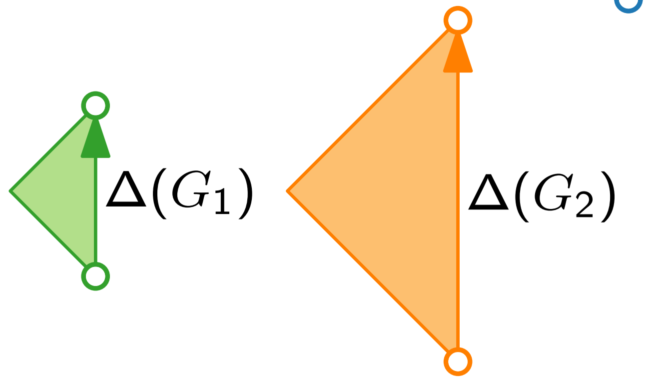


Divide & conquer algorithm using the decomposition tree

- Draw G inside a right-angled isosceles bounding triangle $\Delta(G)$

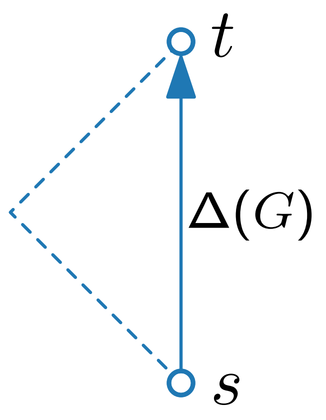
Base case: Q-nodes

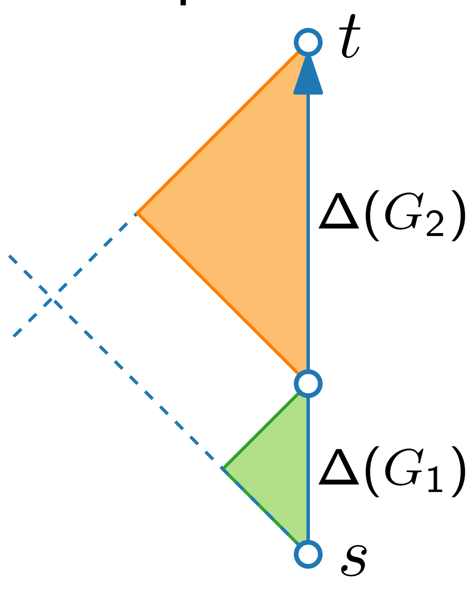
Divide: Draw G_1 and G_2 first

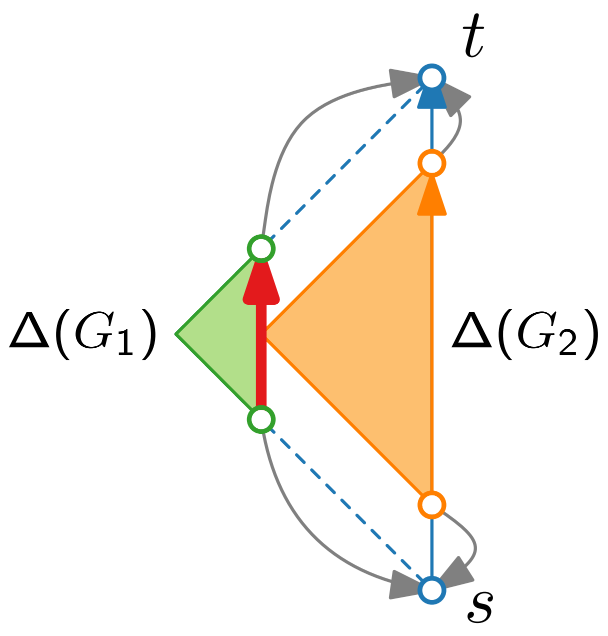


Conquer:

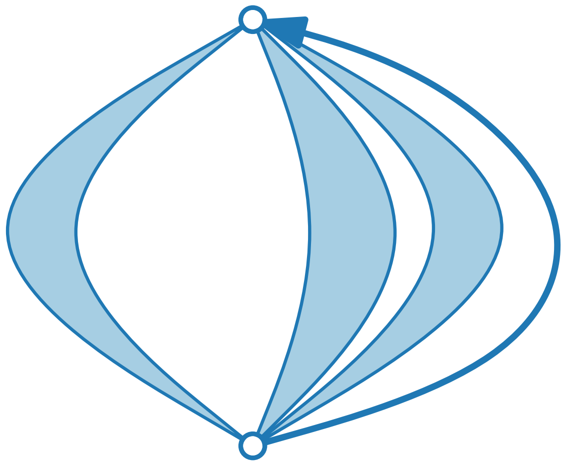
- S-nodes / series composition
- P-nodes / parallel composition







change embedding!

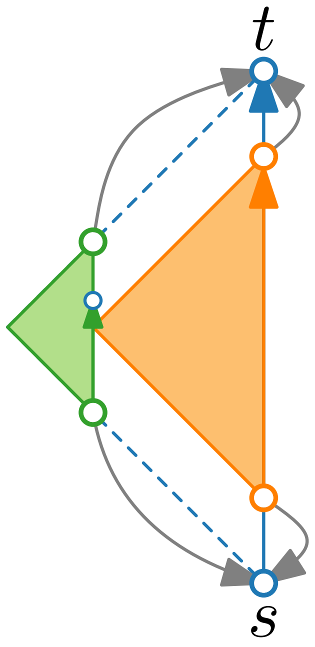


Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?

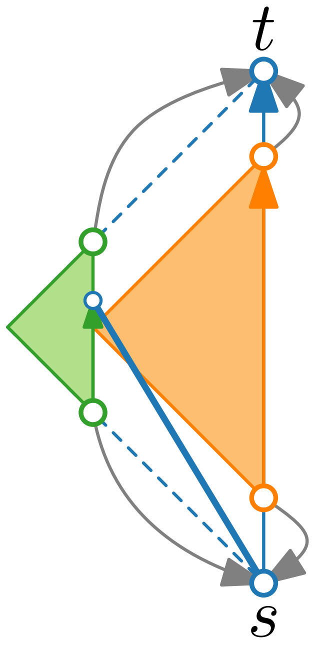
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



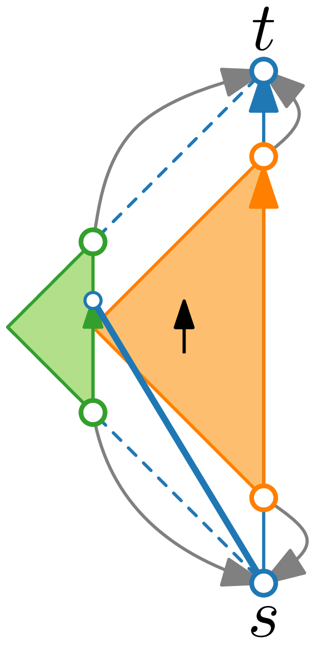
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



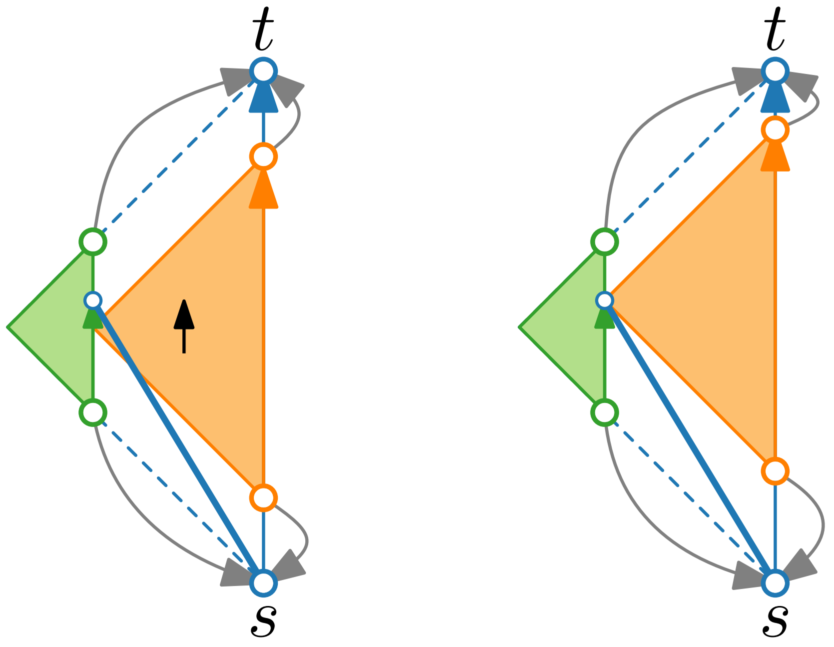
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



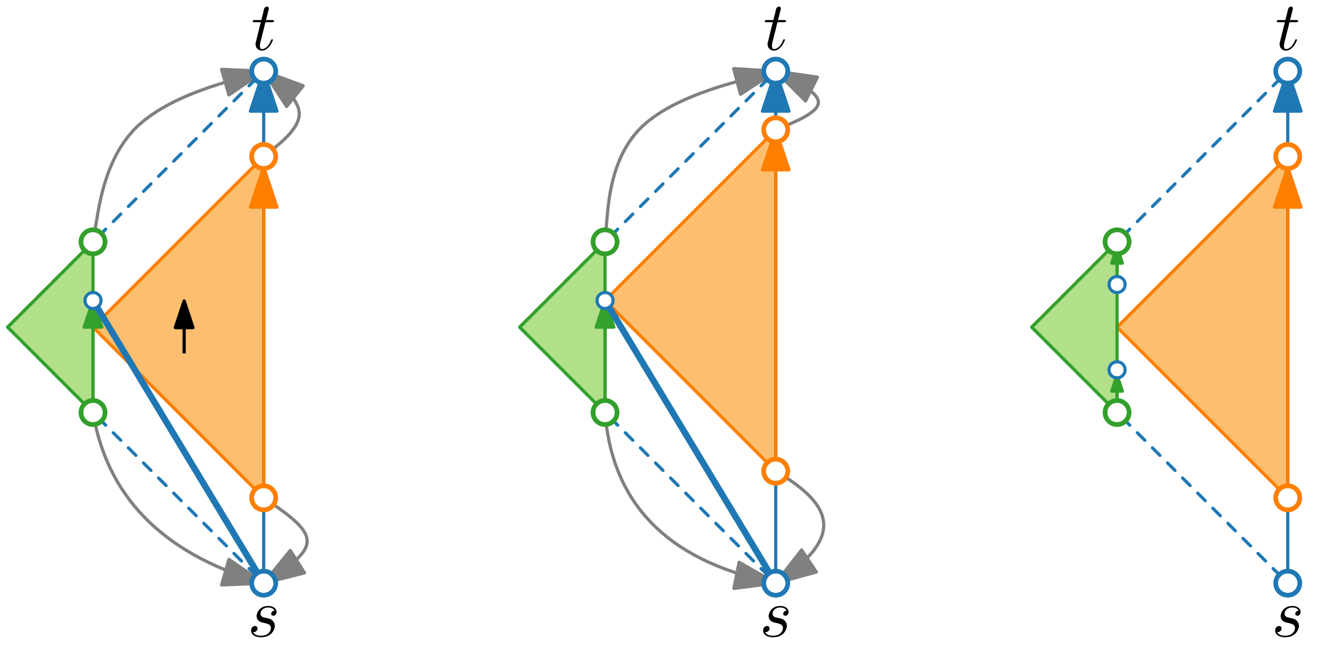
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



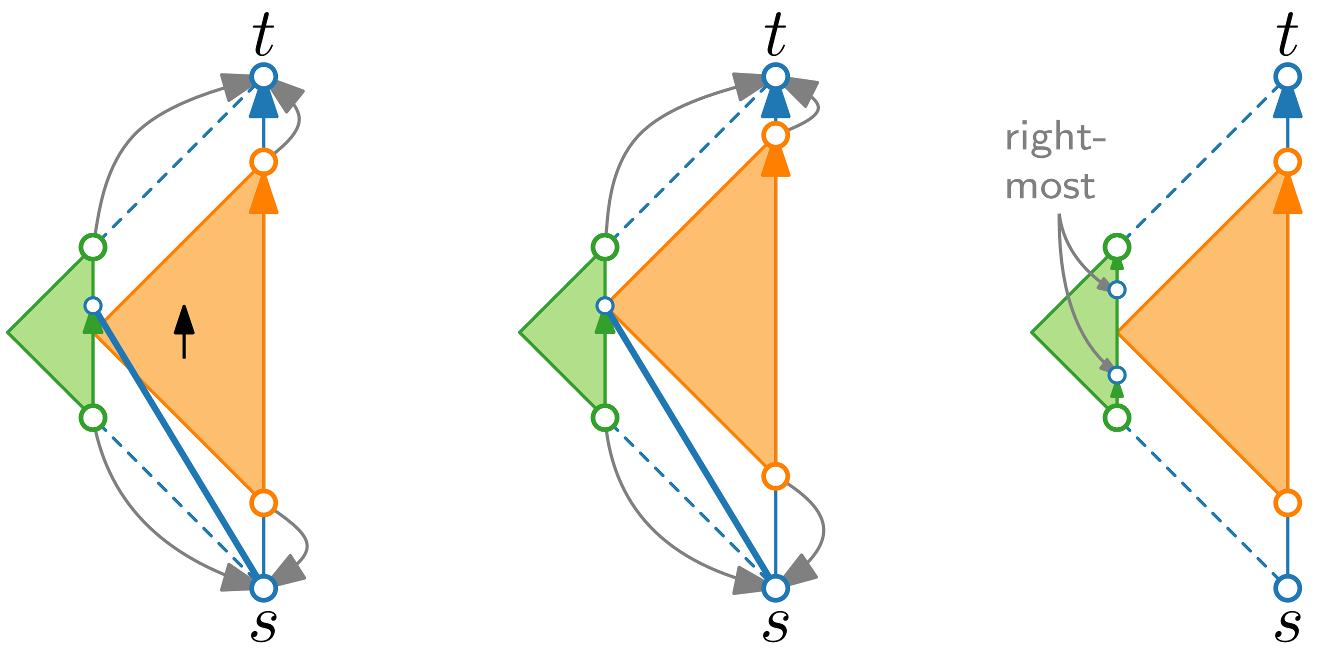
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



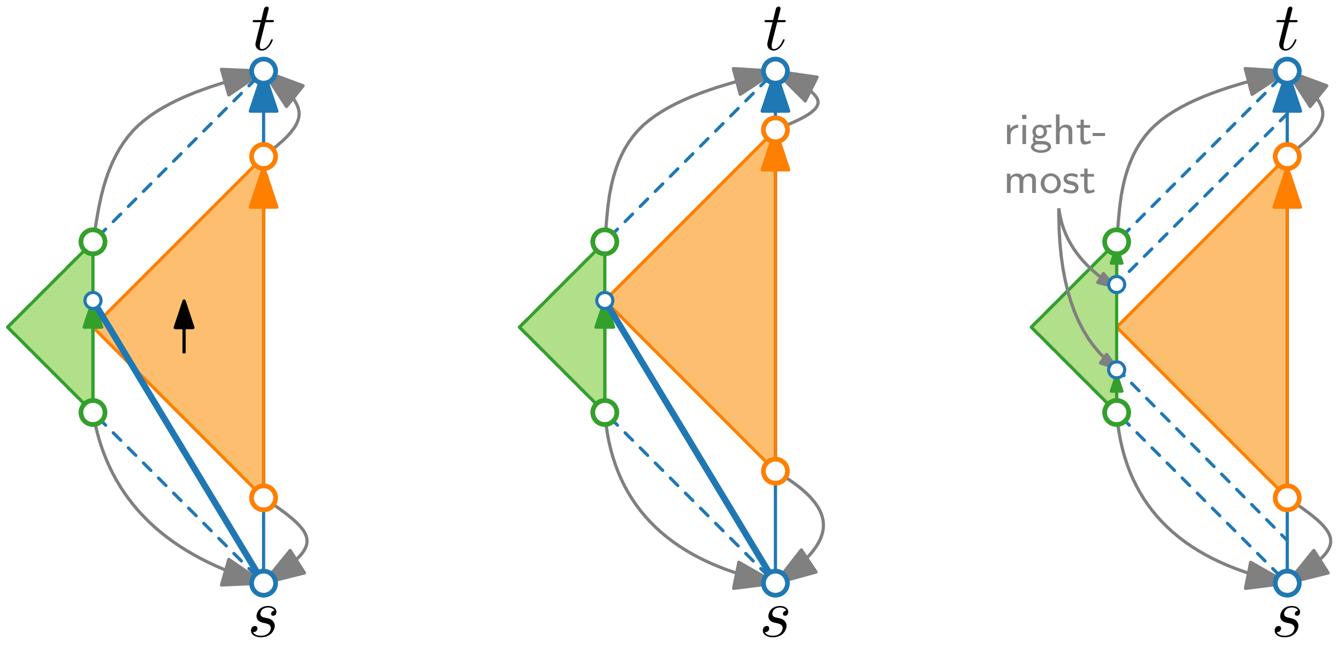
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



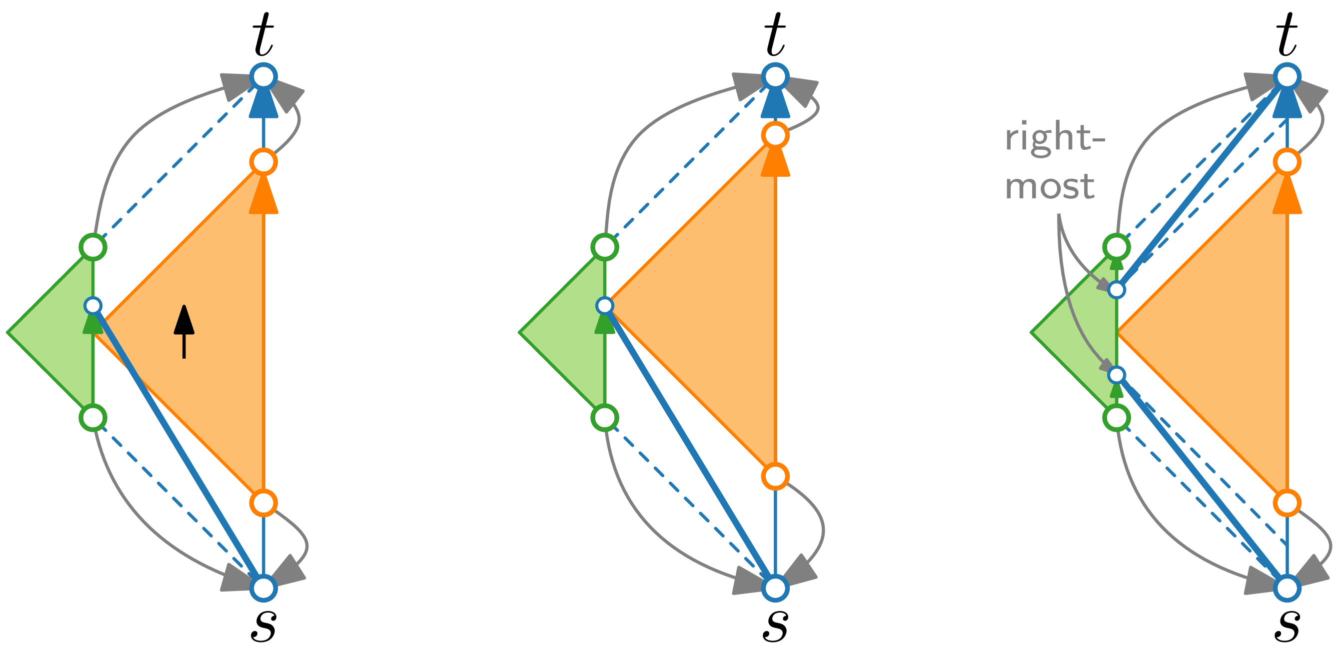
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



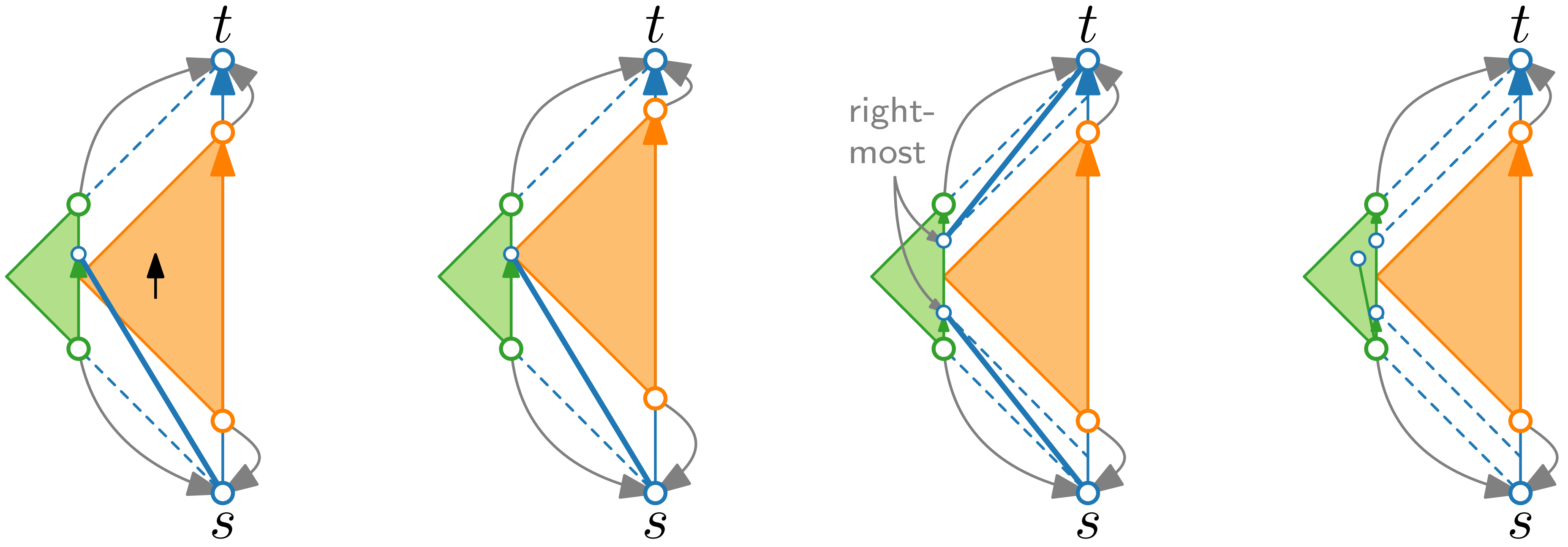
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



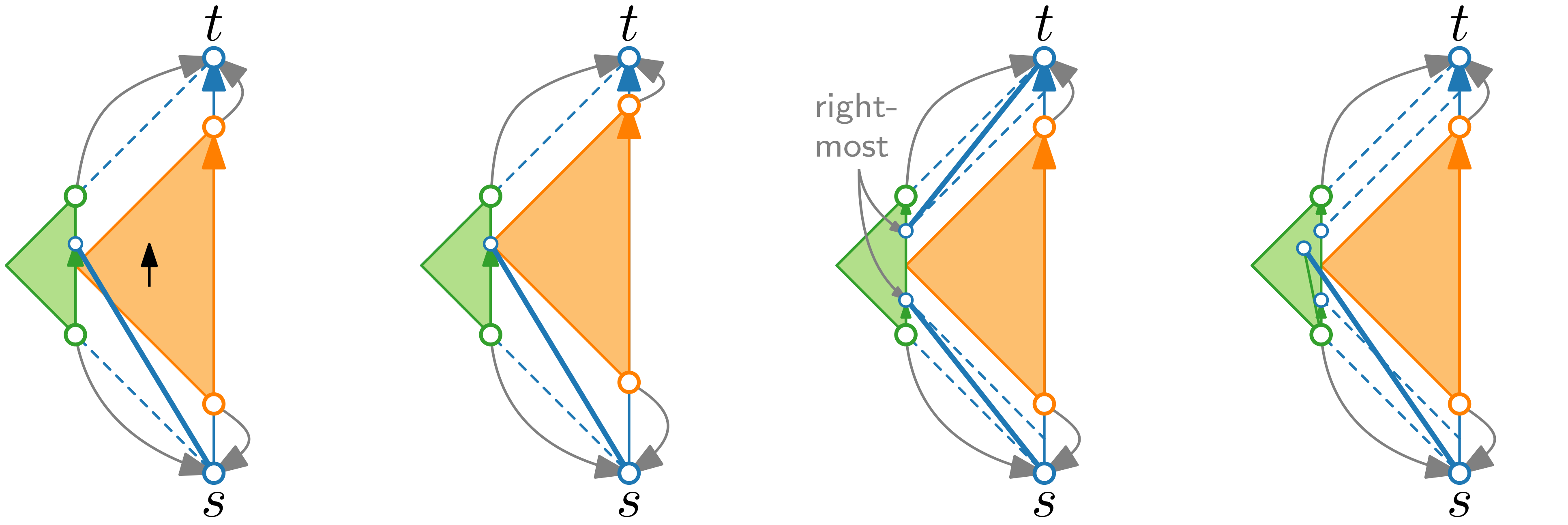
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



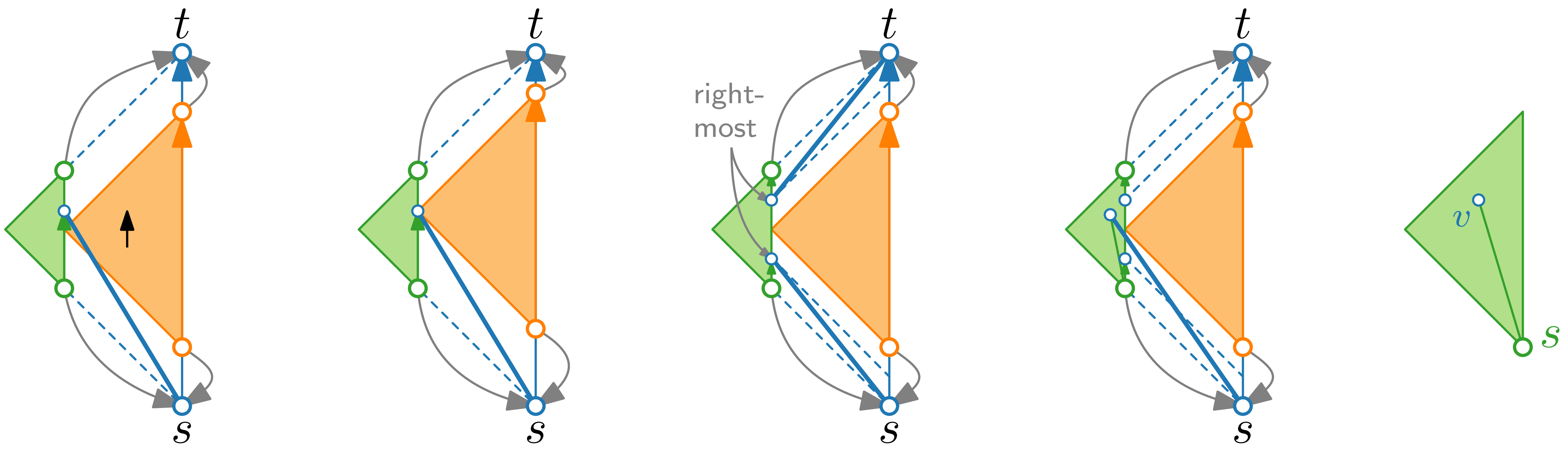
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



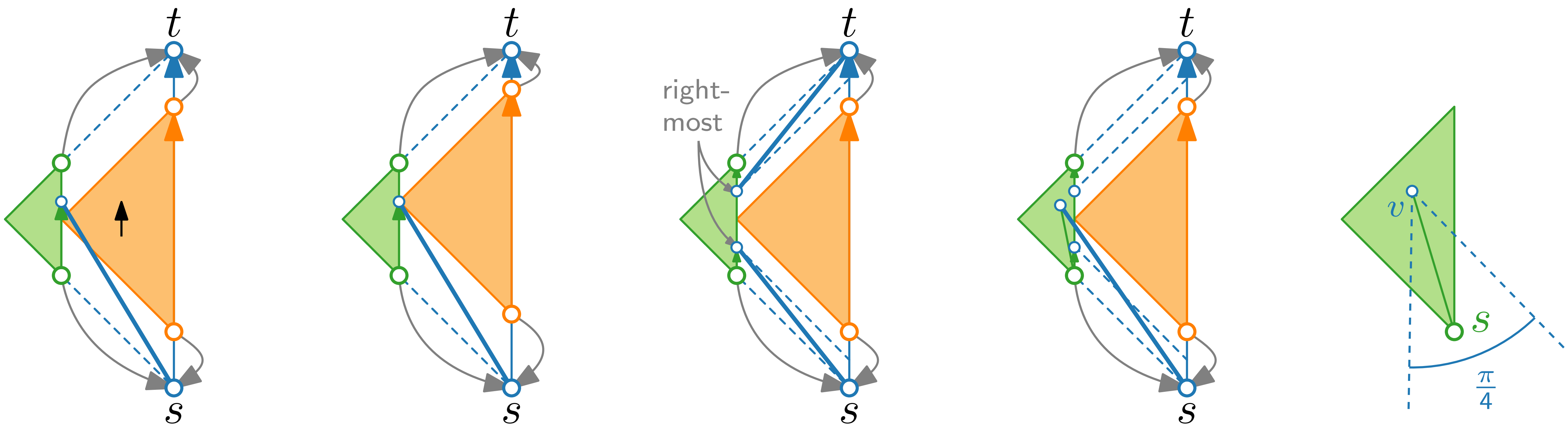
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



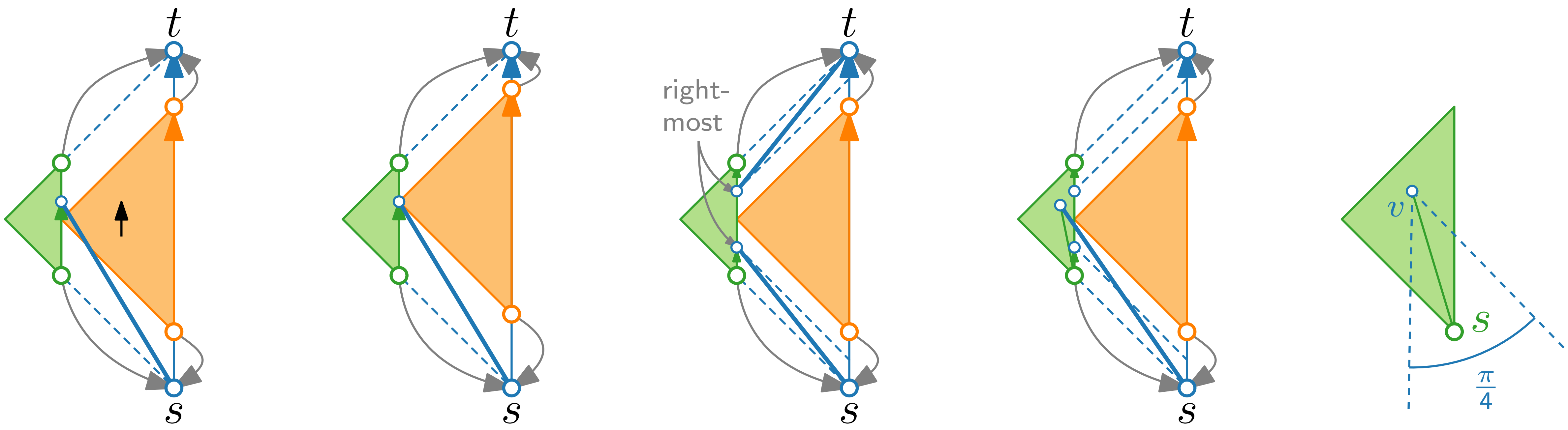
Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



Series-Parallel Graphs – Straight-Line Drawings

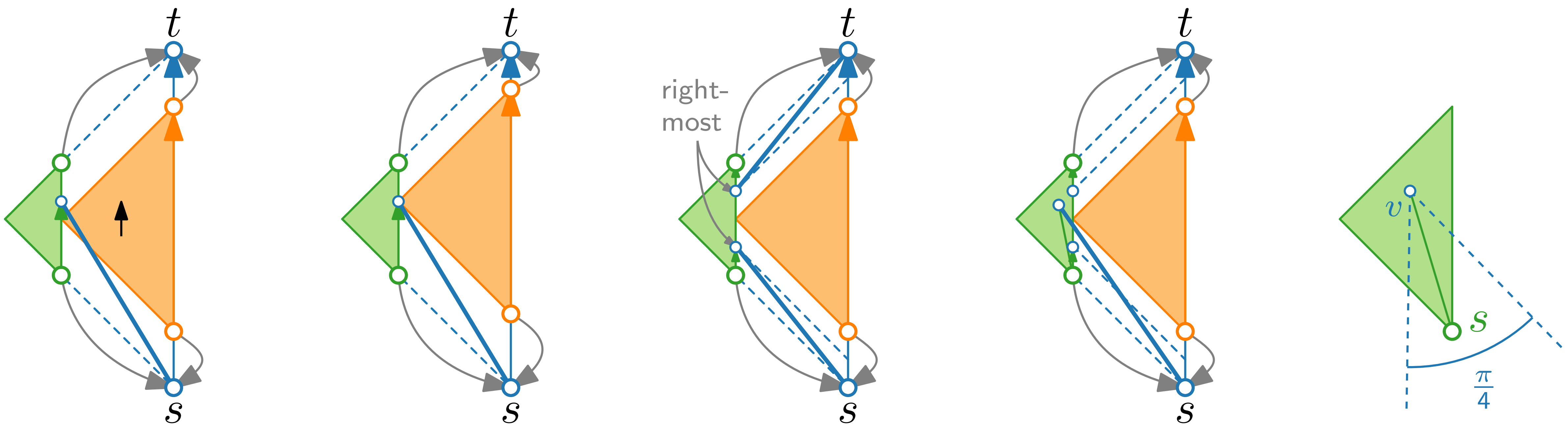
- What makes parallel composition possible without creating crossings?



Assume the following holds:
the only vertex in angle(v) is s

Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?

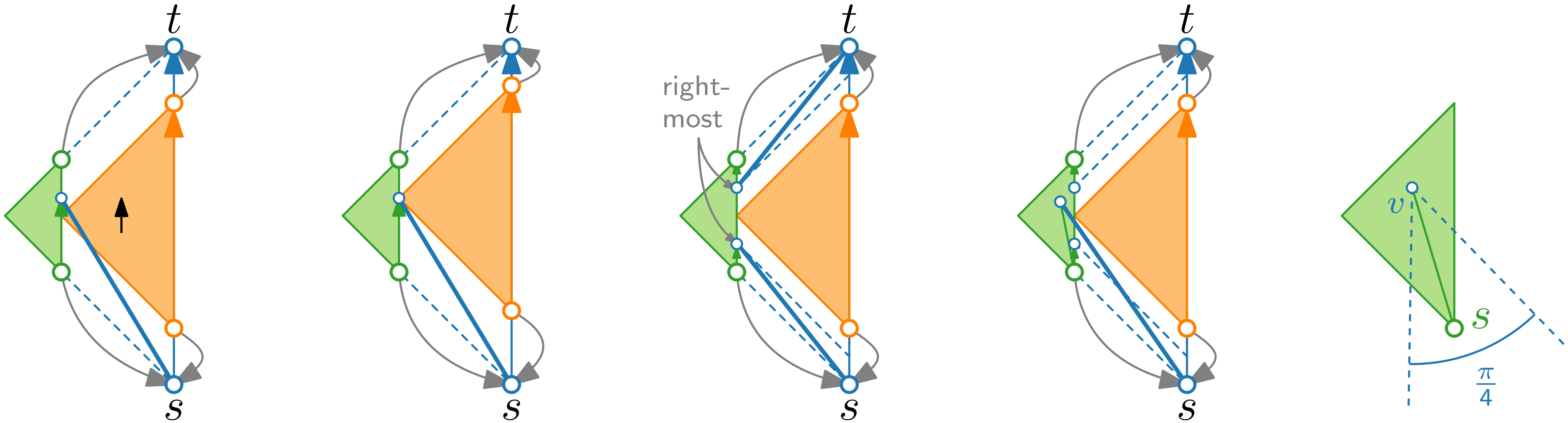


Assume the following holds:
the only vertex in angle(v) is s

- This condition **is** preserved during the induction step.

Series-Parallel Graphs – Straight-Line Drawings

- What makes parallel composition possible without creating crossings?



Assume the following holds:
the only vertex in angle(v) is s

- This condition **is** preserved during the induction step.

> **Lemma.**
> The drawing produced by the algorithm is planar.

Series-Parallel Graphs – Result

Theorem.
Let G be a series-parallel graph. Then G (with **variable embedding**) admits a drawing Γ that

Series-Parallel Graphs – Result

Theorem.
Let G be a series-parallel graph. Then G (with **variable embedding**) admits a drawing Γ that

- is upward planar and

Series-Parallel Graphs – Result

Theorem.
Let G be a series-parallel graph. Then G (with **variable embedding**) admits a drawing Γ that

- is upward planar and
- a straight-line drawing

Series-Parallel Graphs – Result

Theorem.
Let G be a series-parallel graph. Then G (with **variable embedding**) admits a drawing Γ that

- is upward planar and
- a straight-line drawing
- with area in $\mathcal{O}(n^2)$.

Series-Parallel Graphs – Result

Theorem.
Let G be a series-parallel graph. Then G (with **variable embedding**) admits a drawing Γ that

- is upward planar and
- a straight-line drawing
- with area in $\mathcal{O}(n^2)$.
- Isomorphic components of G have congruent drawings up to translation.

Series-Parallel Graphs – Result

Theorem.
Let G be a series-parallel graph. Then G (with **variable embedding**) admits a drawing Γ that

- is upward planar and
- a straight-line drawing
- with area in $\mathcal{O}(n^2)$.
- Isomorphic components of G have congruent drawings up to translation.

Γ can be computed in $\mathcal{O}(n)$ time.

Series-Parallel Graphs – Fixed Embedding

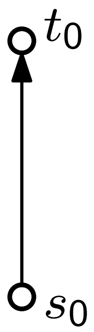
Theorem. [Bertolazzi et al. 94]

There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

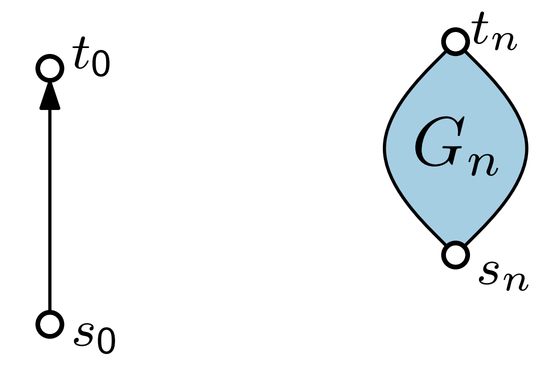


G_0

Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

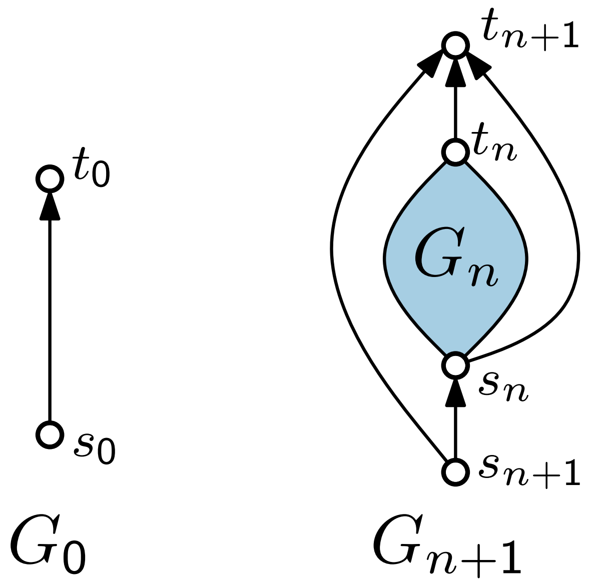


G_0 G_{n+1}

Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

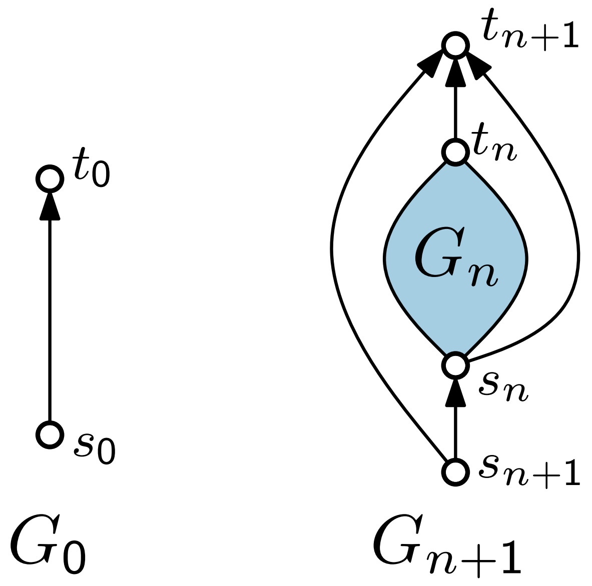
There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

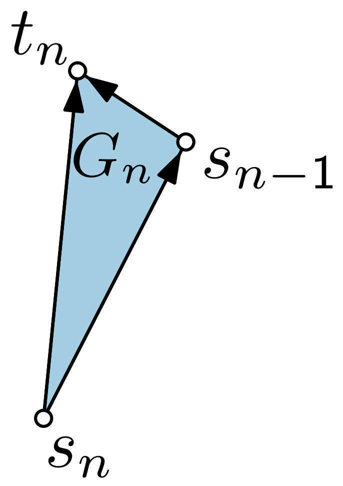


Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

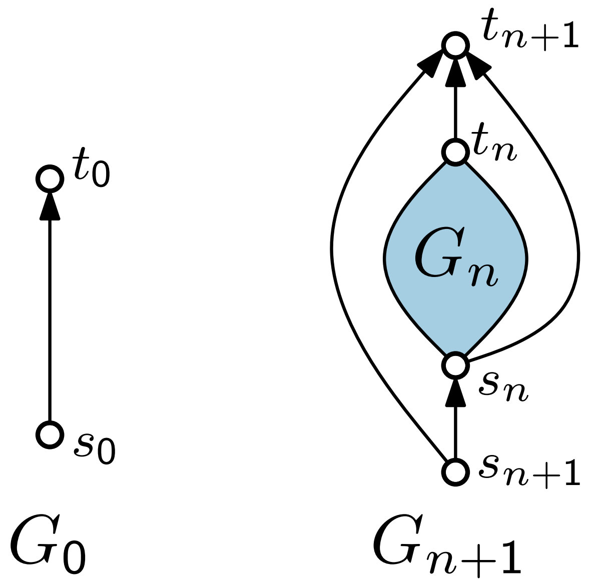


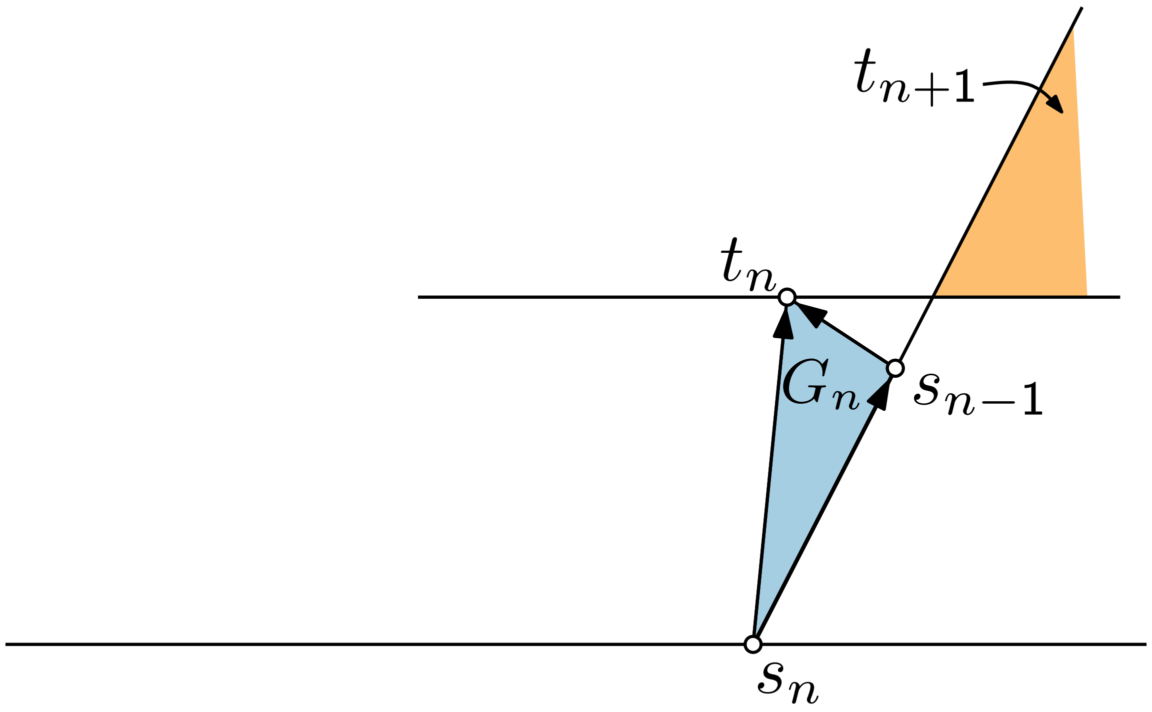


Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

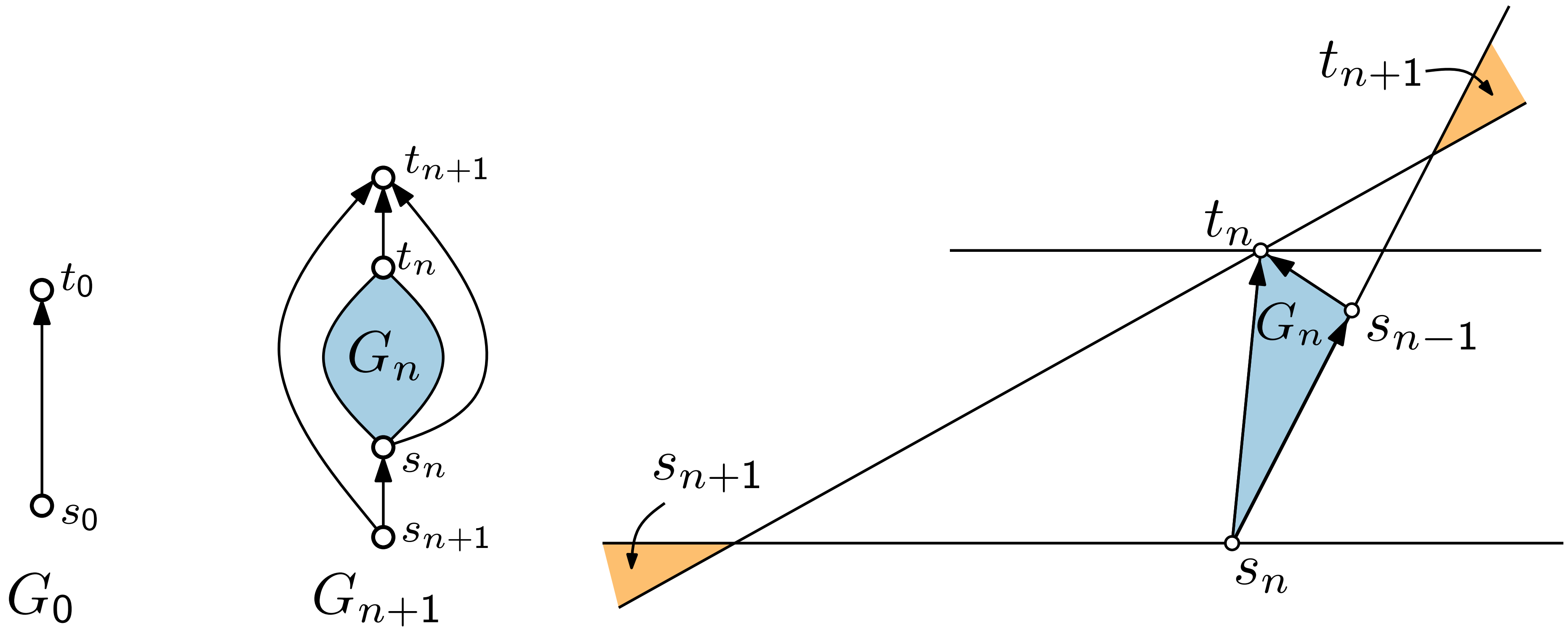




Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

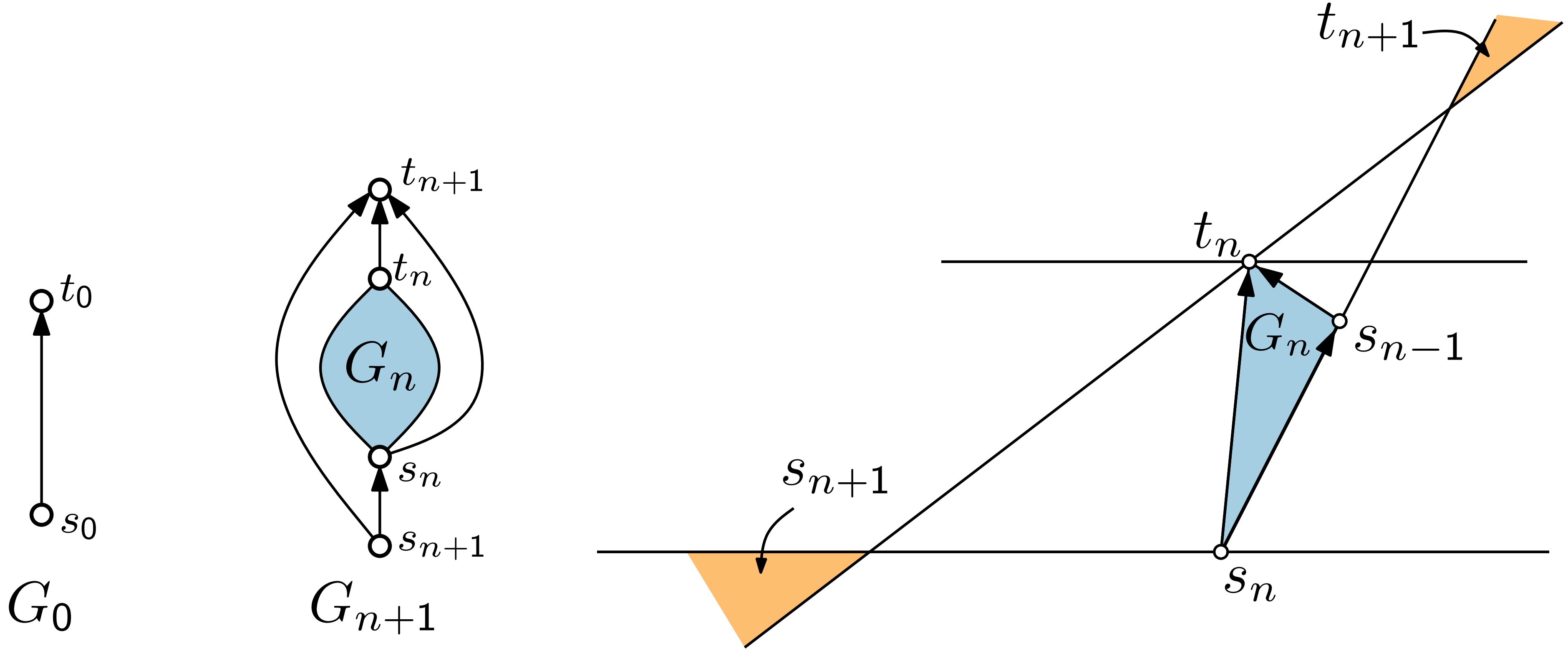
There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.



Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

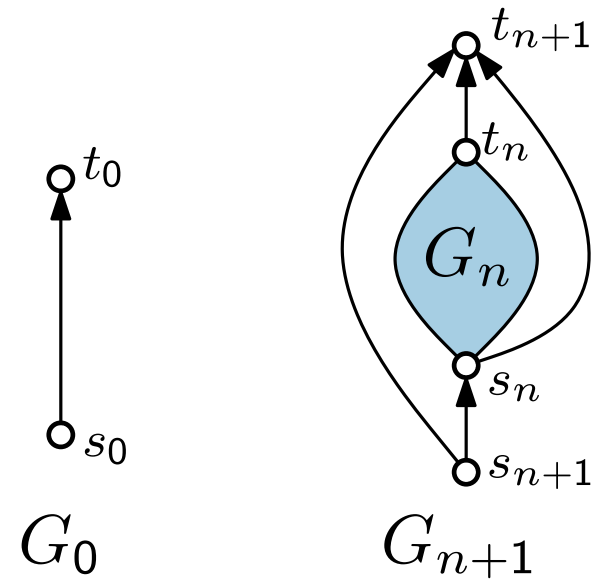
There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

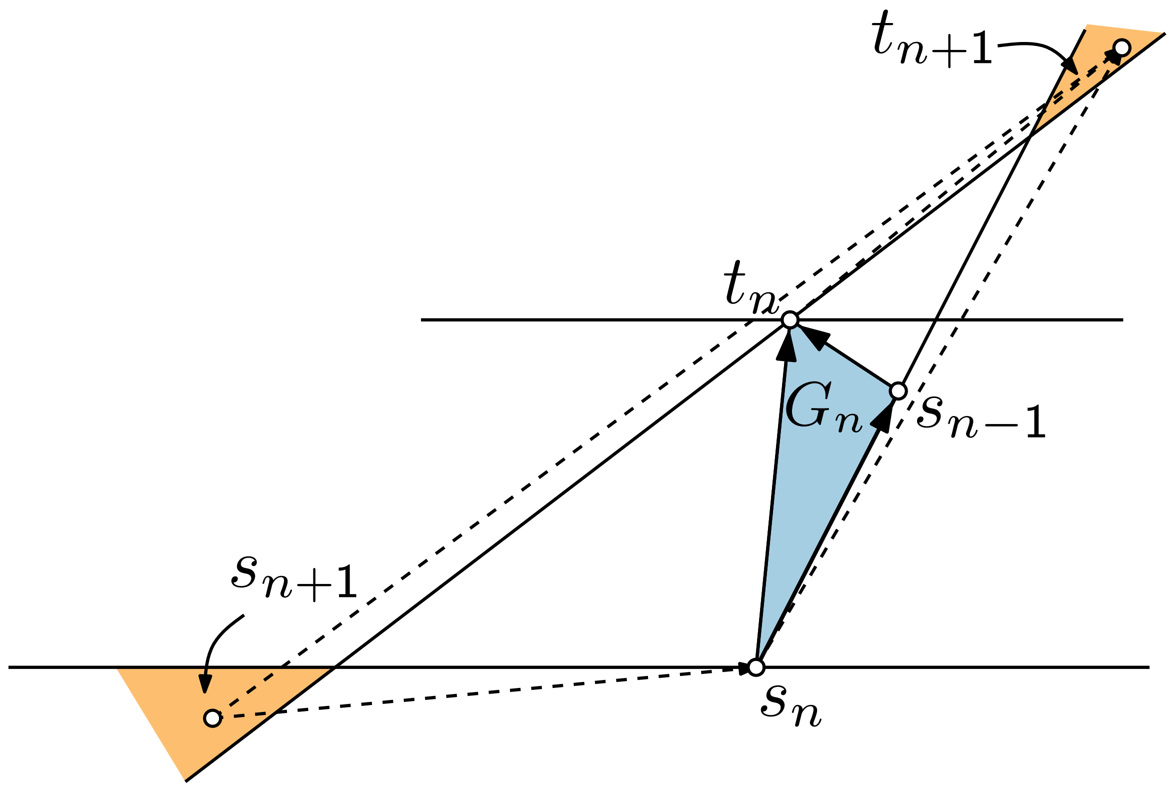


Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

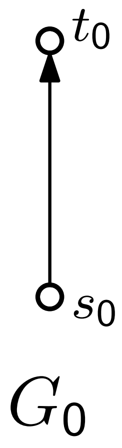


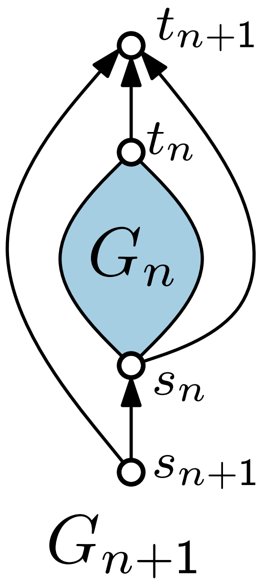


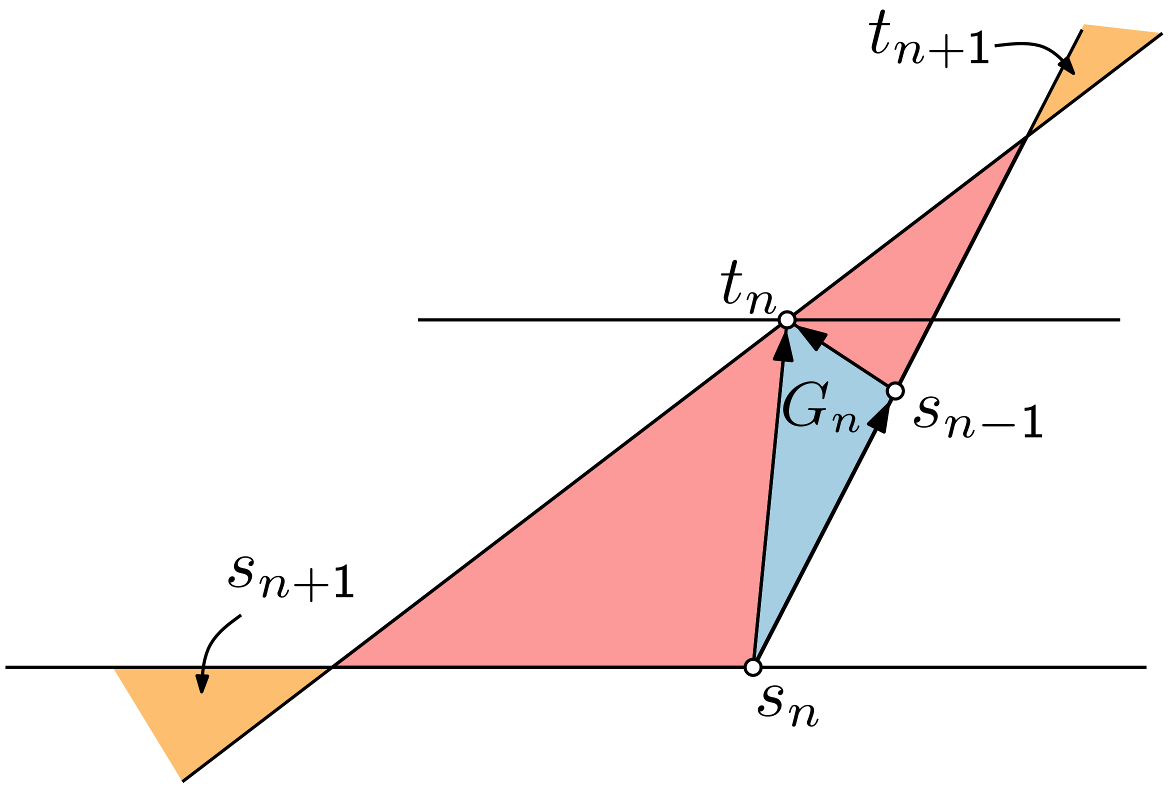
Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.



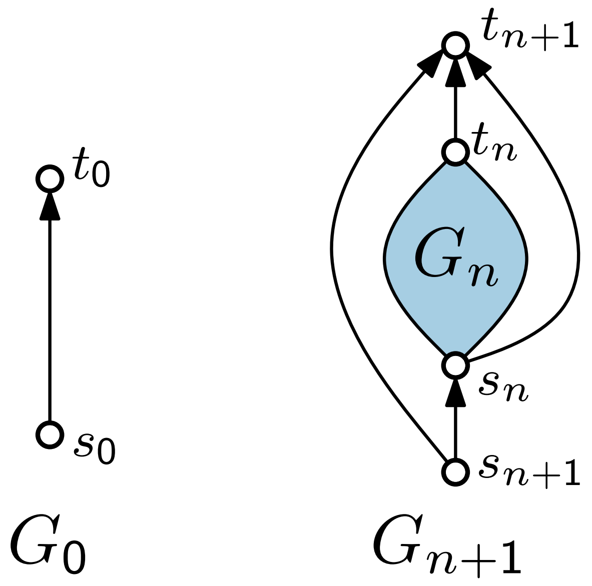


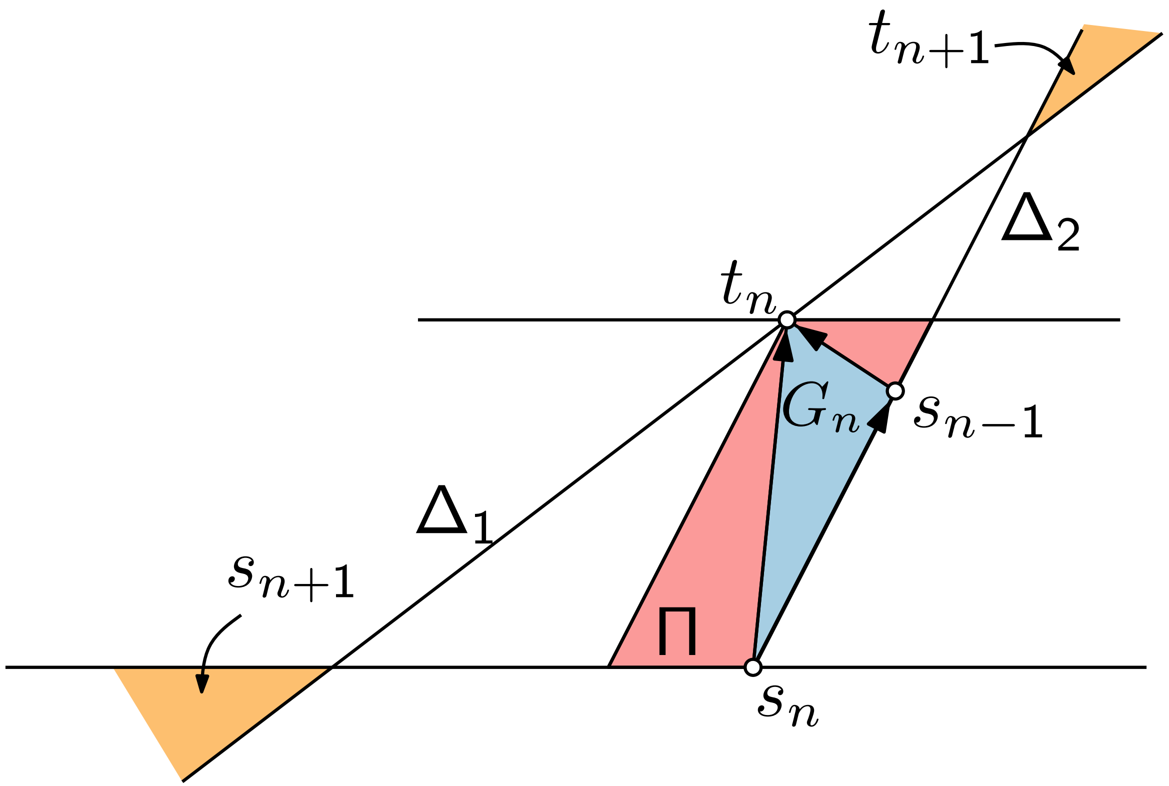


Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.



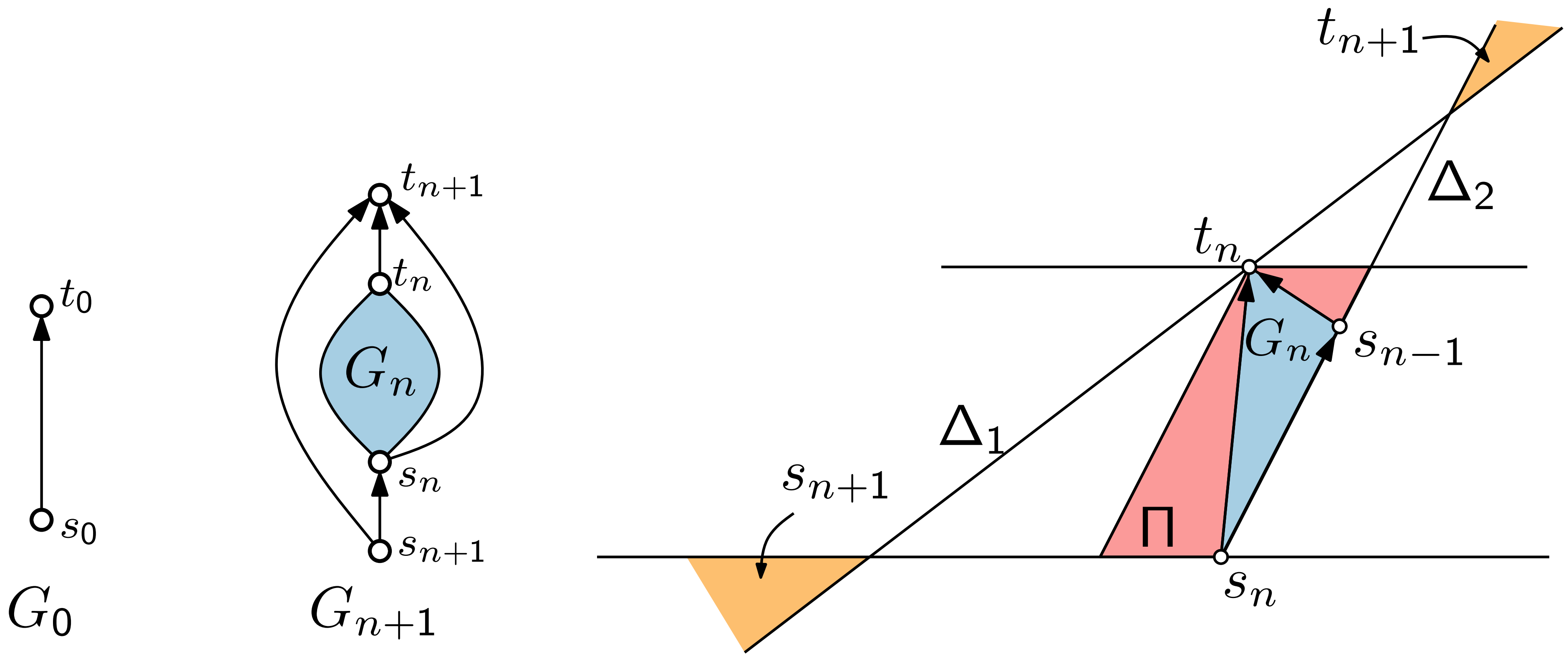


Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]

There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

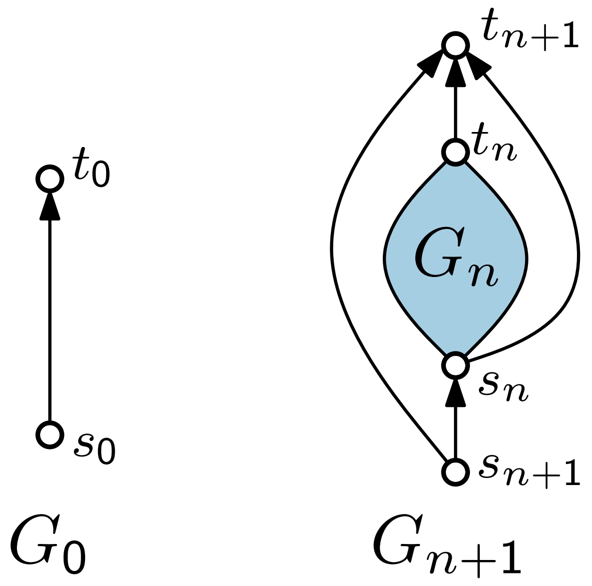
- $2 \cdot \text{Area}(G_n) < \text{Area}(\Pi)$

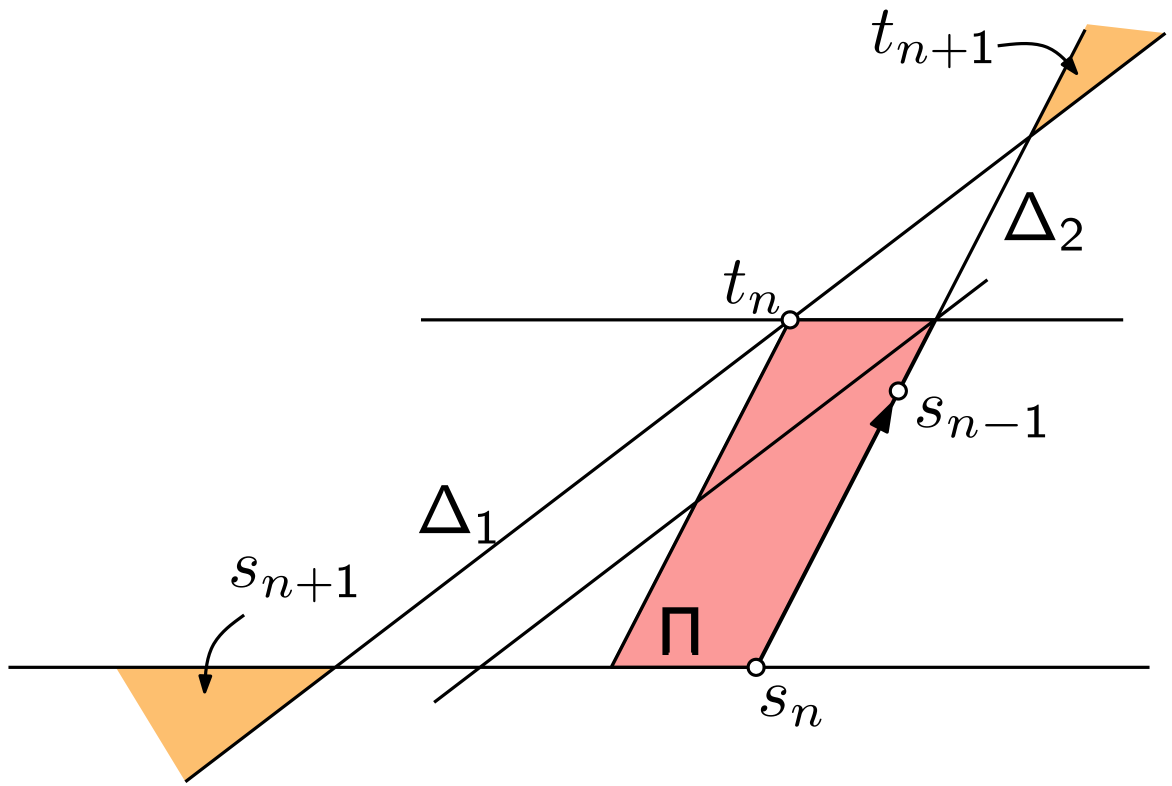


Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]
There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

- $2 \cdot \text{Area}(G_n) < \text{Area}(\Pi)$

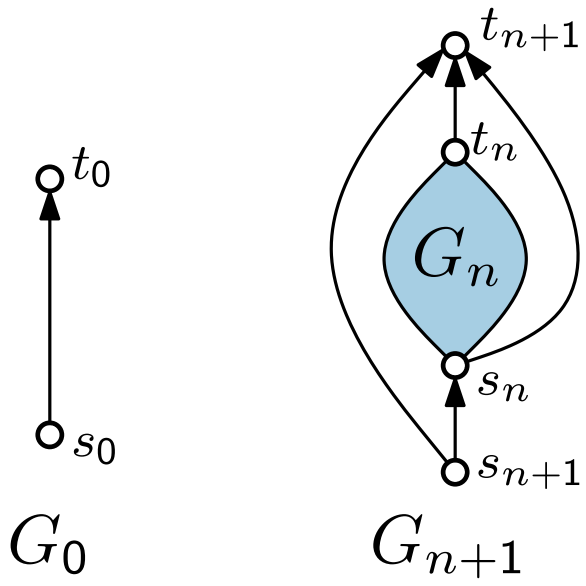


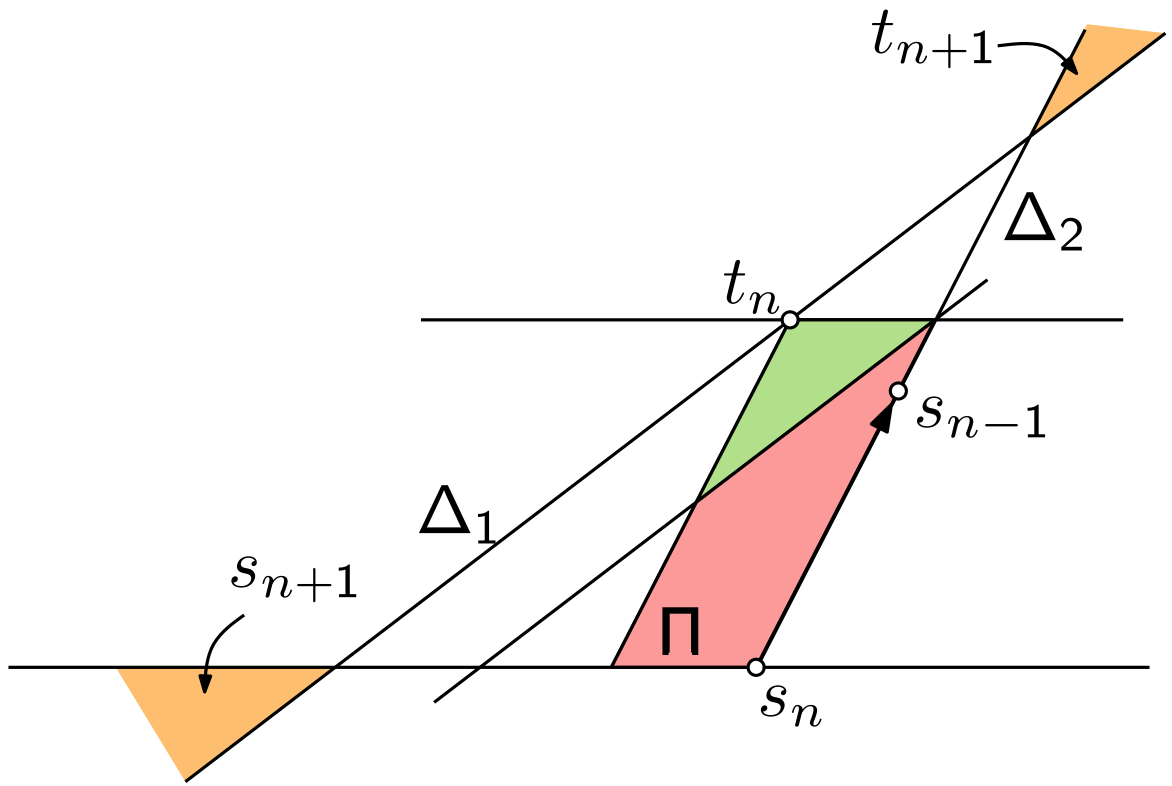


Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]
There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

- $2 \cdot \text{Area}(G_n) < \text{Area}(\Pi)$

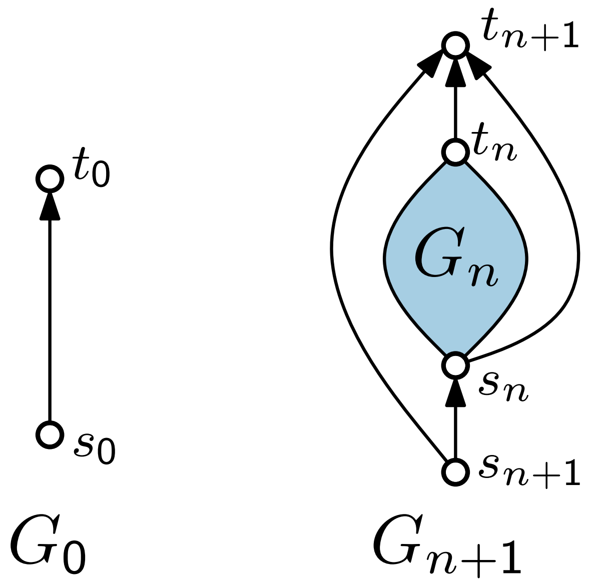


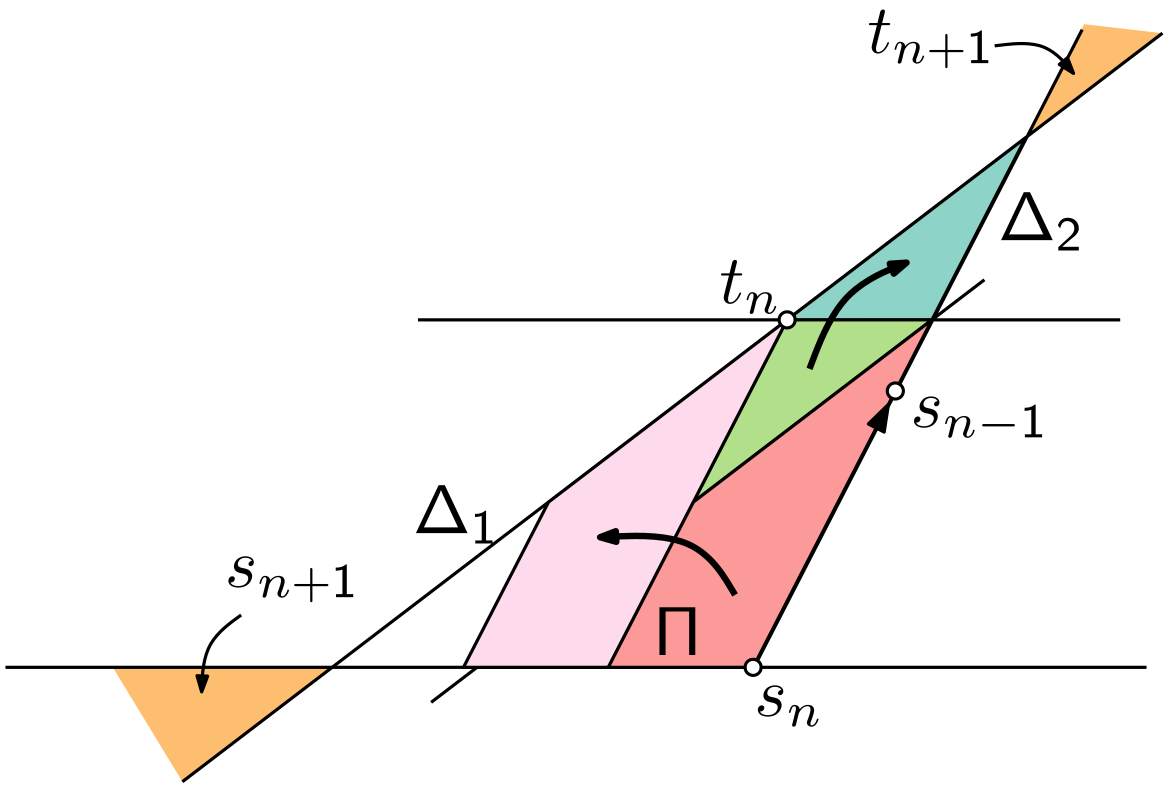


Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]
There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

- $2 \cdot \text{Area}(G_n) < \text{Area}(\Pi)$

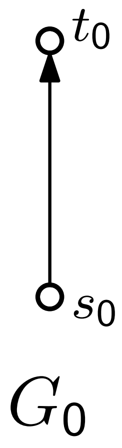


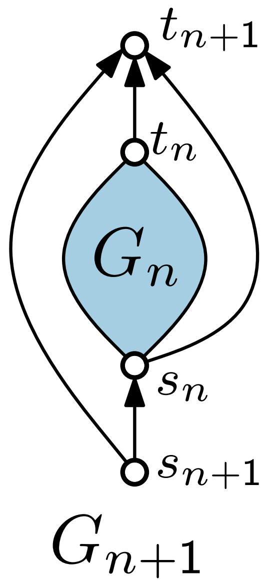


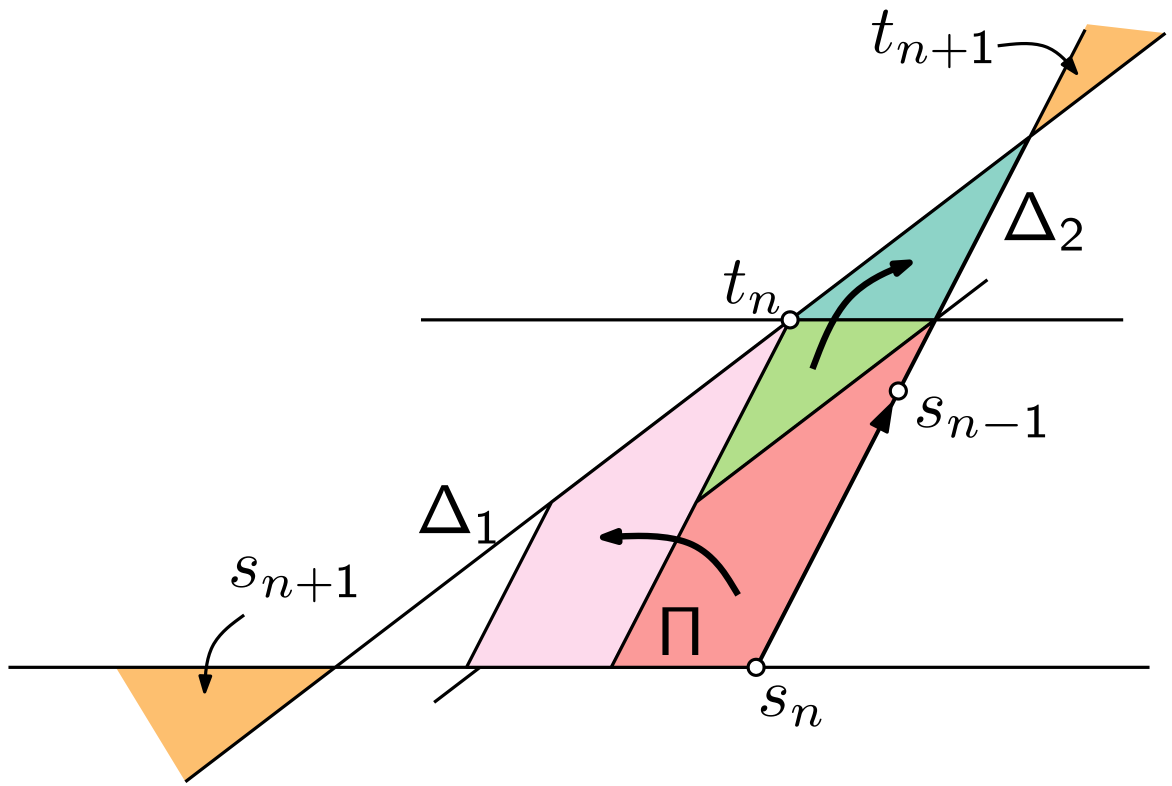
Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]
There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

- $2 \cdot \text{Area}(G_n) < \text{Area}(\Pi)$
- $2 \cdot \text{Area}(\Pi) \leq \text{Area}(G_{n+1})$





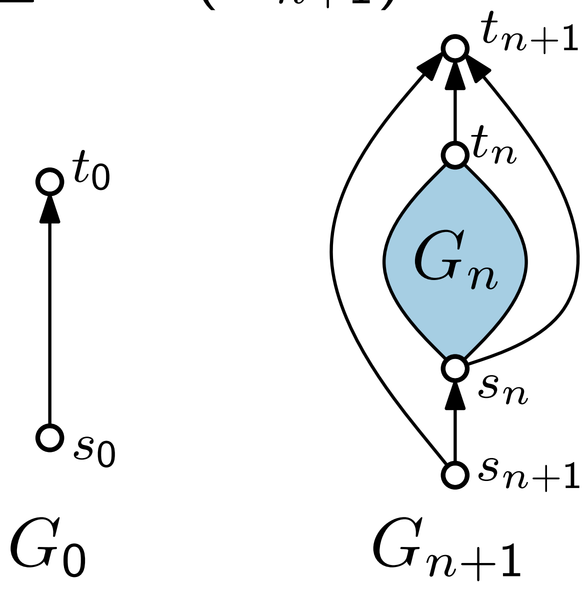


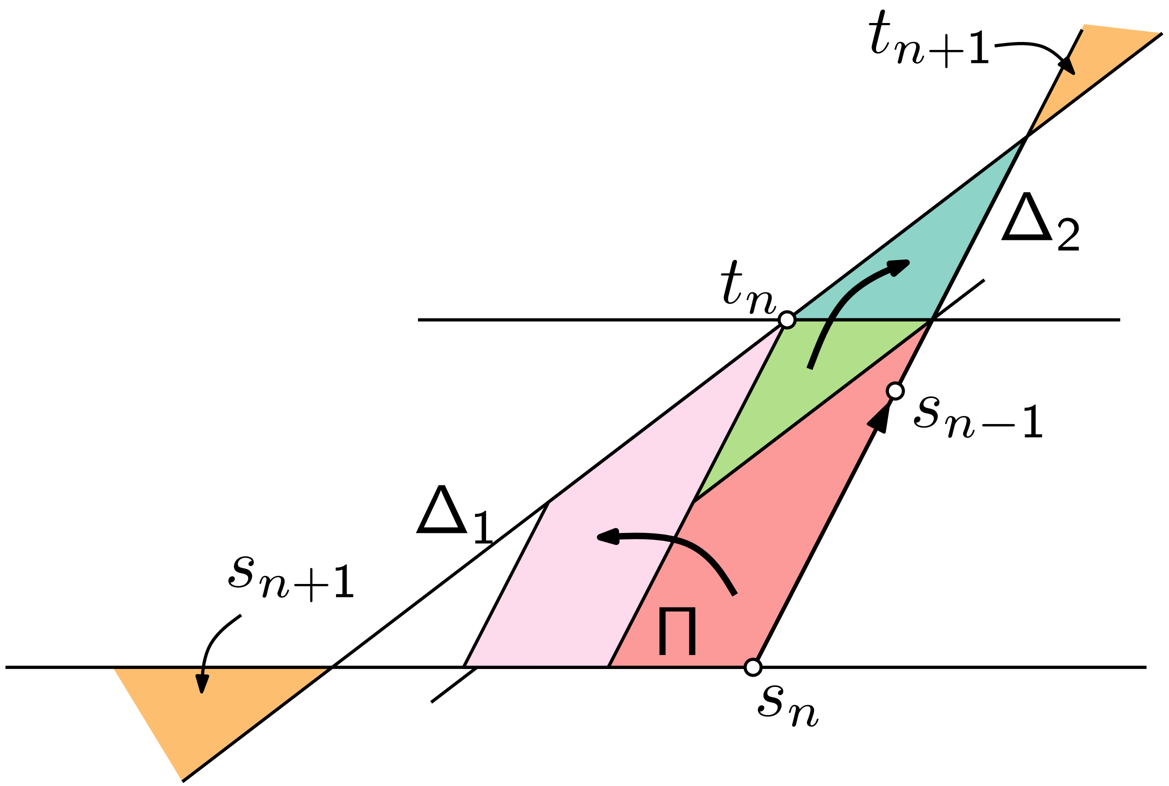
Series-Parallel Graphs – Fixed Embedding

Theorem. [Bertolazzi et al. 94]
There exists a $2n$-vertex series-parallel graph G_n such that any upward planar drawing of G_n that respects the embedding requires $\Omega(4^n)$ area.

- $2 \cdot \text{Area}(G_n) < \text{Area}(\Pi)$
- $2 \cdot \text{Area}(\Pi) \leq \text{Area}(G_{n+1})$

$\Rightarrow 4 \cdot \text{Area}(G_n) \leq \text{Area}(G_{n+1})$





Literature

- [GD, Chapter 3] for 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