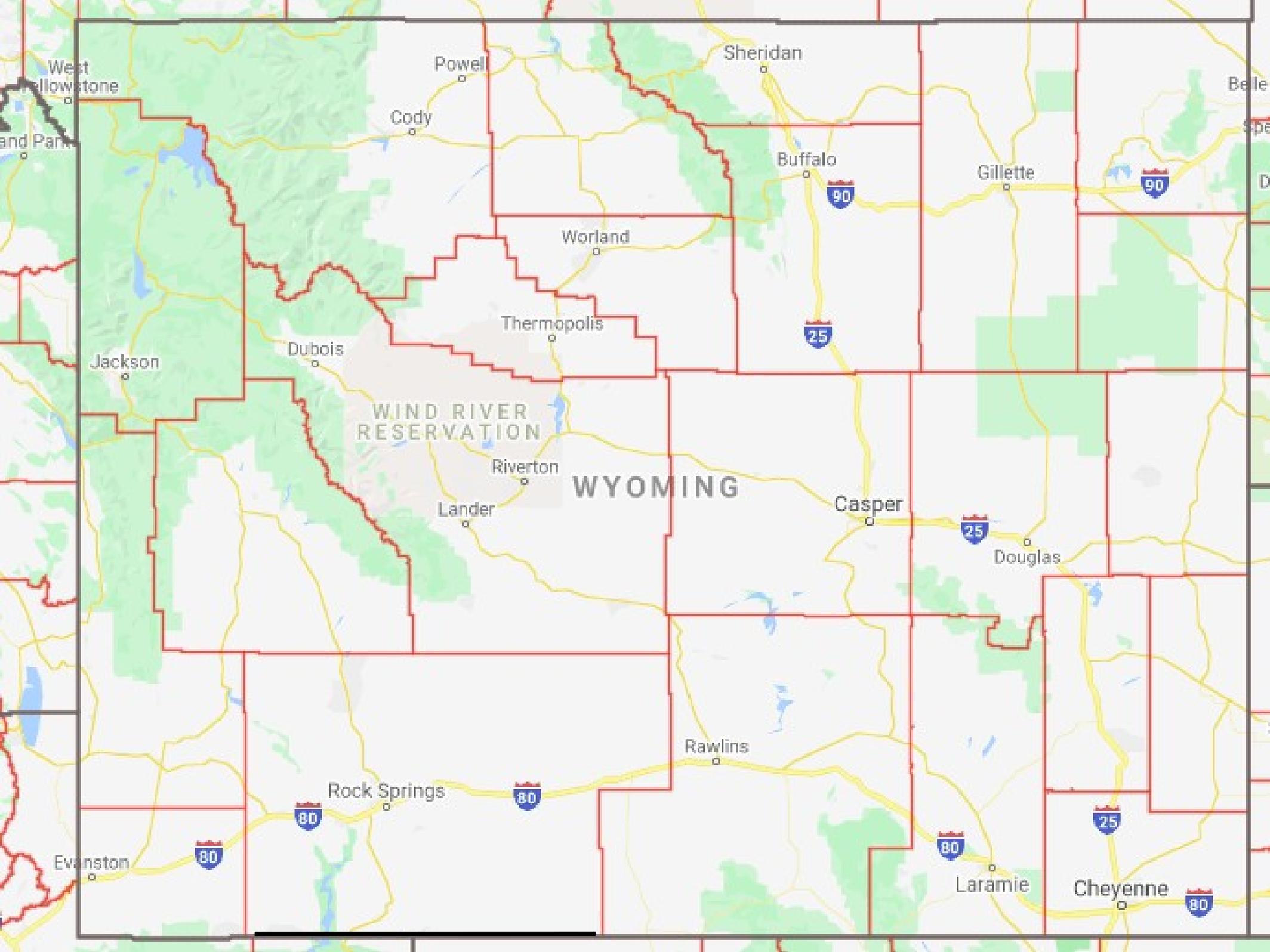# Planar Point Location

Outline for Today

- ***Plane Graphs***
  - Subdividing 2D space into regions.
- ***Point Location***
  - Where are you in 2D space?
- ***Slab Decomposition***
  - Breaking space apart for fast searches.
- ***Persistent Red/Black Trees***
  - Sharing space across trees.

Plane Graphs
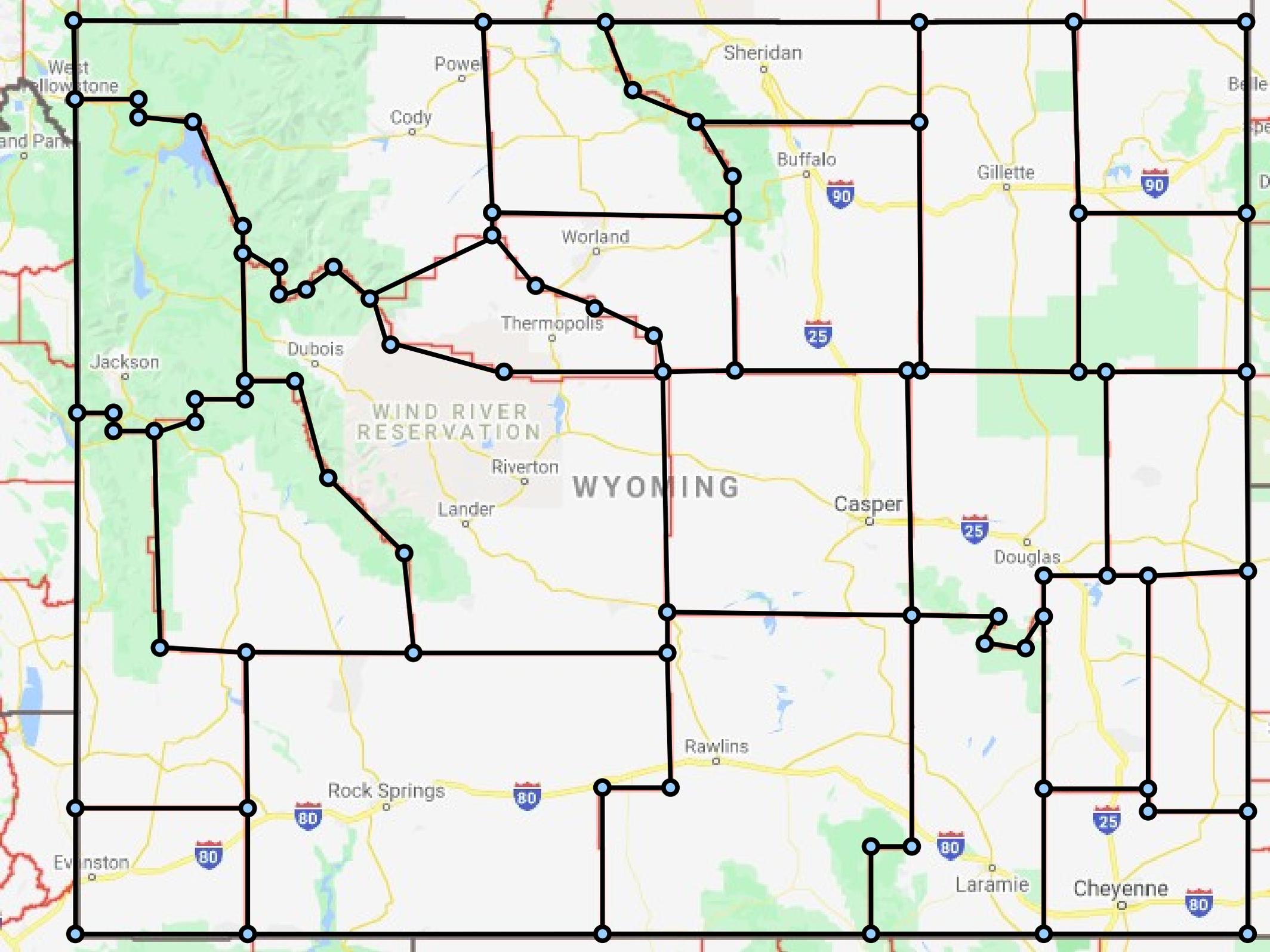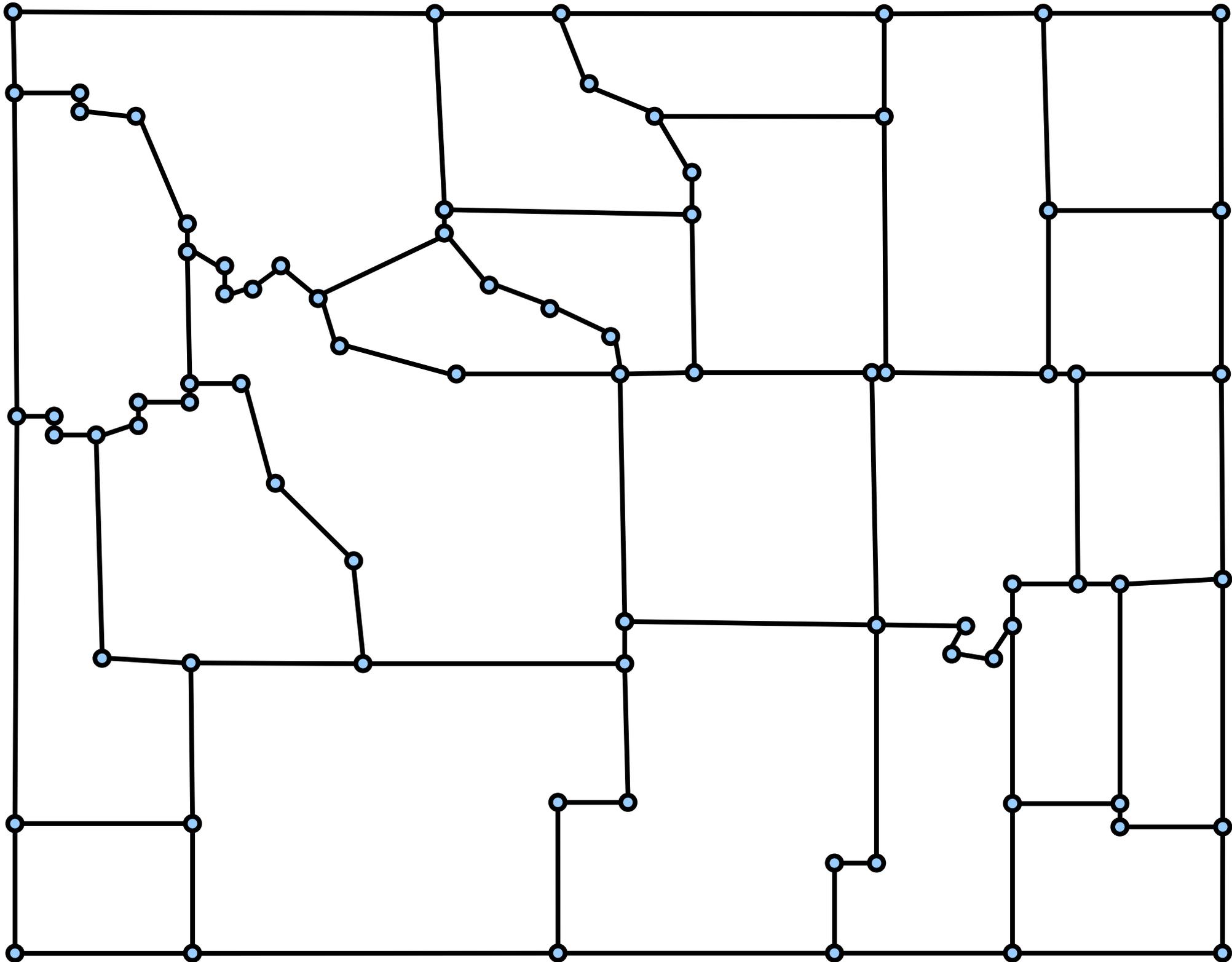
Plane Graphs

- A ***plane graph*** is a drawing of a graph in 2D space such that
  - all edges are represented as simple curves, and
  - excluding their endpoints, no two curves touch.
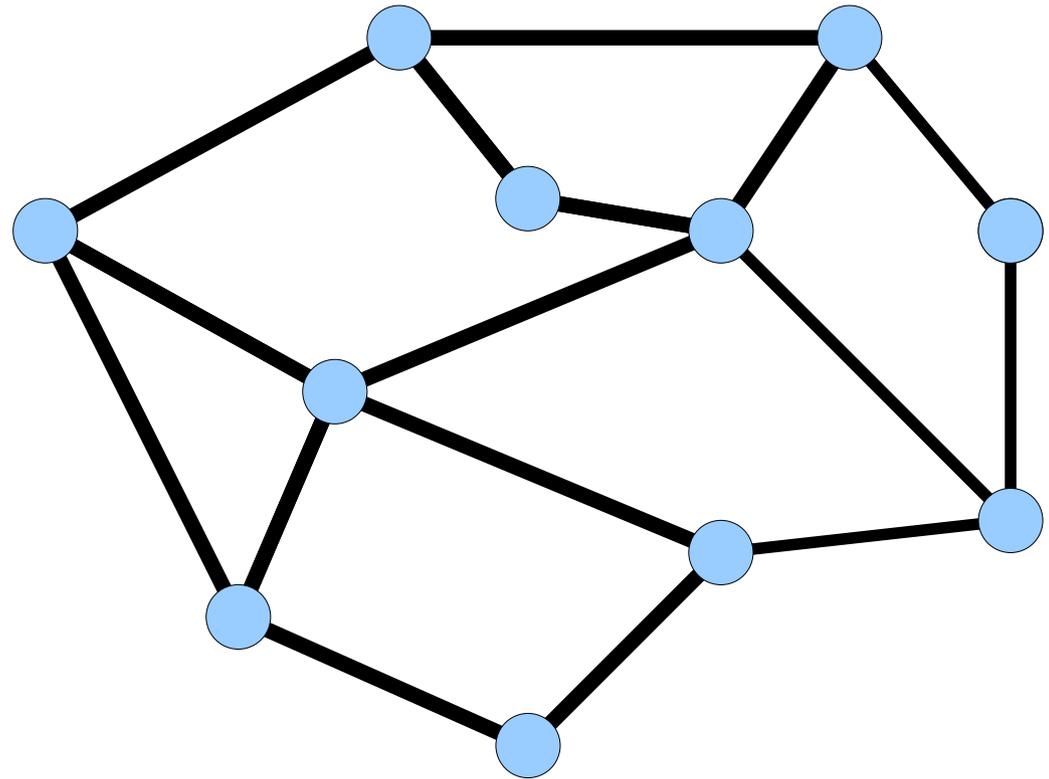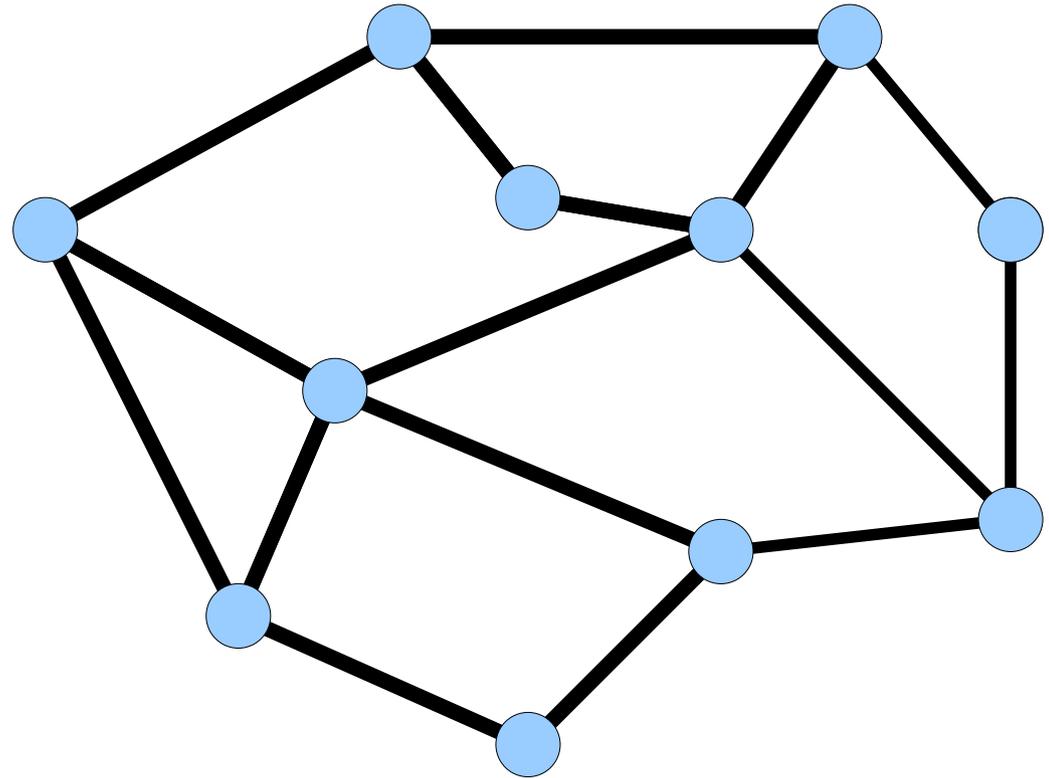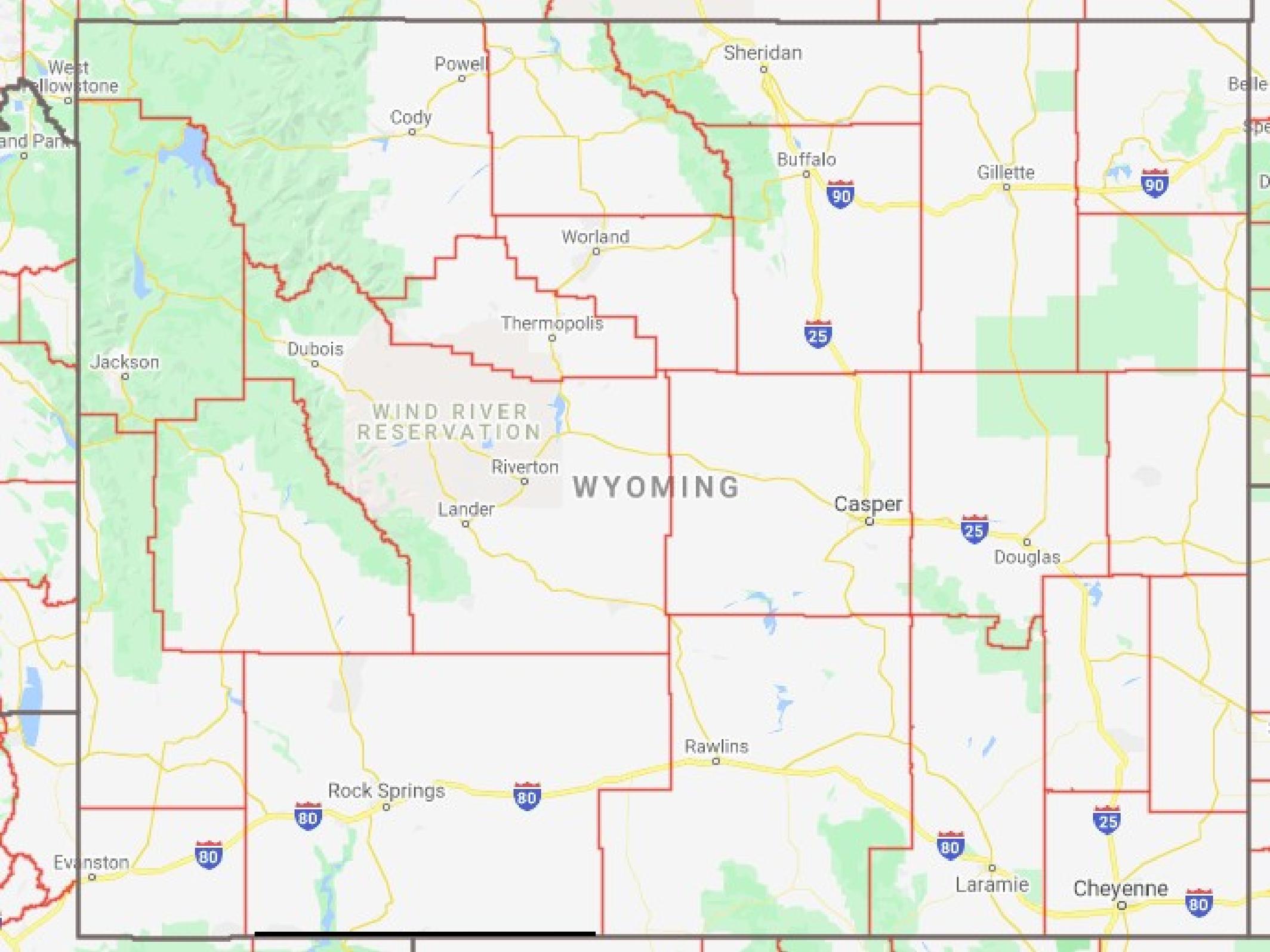- Plane graphs are a natural way of encoding borders on a map or regions in an image.

Plane Graphs

- For most of today, we'll be talking about ***straight-line plane graphs***, plane graphs where each edge is a straight line.

- ***Fáry's theorem*** says that every plane graph can be redrawn as a straight line plane graph.

  - This is a *beautiful* proof by induction; do a quick Google search for more details.

- Many of the techniques from today work in the more general case of arbitrary curves – it's a good exercise to think about how to modify them.

Plane Graphs

- A plane graph subdivides 2D space into regions called **faces**, spaces demarcated by edges.

- The outer region of space outside the plane graph is considered a face as well.

- When working with plane graphs, often the *faces* are more important than the *vertices*.

Plane Graphs

- A plane graph subdivides 2D space into regions called **faces**, spaces demarcated by edges.

- The outer region of space outside the plane graph is considered a face as well.

- When working with plane graphs, often the *faces* are more important than the *vertices*.

Plane Graphs

- A plane graph subdivides 2D space into regions called **faces**, spaces demarcated by edges.

- The outer region of space outside the plane graph is considered a face as well.

- When working with plane graphs, often the *faces* are more important than the *vertices*.
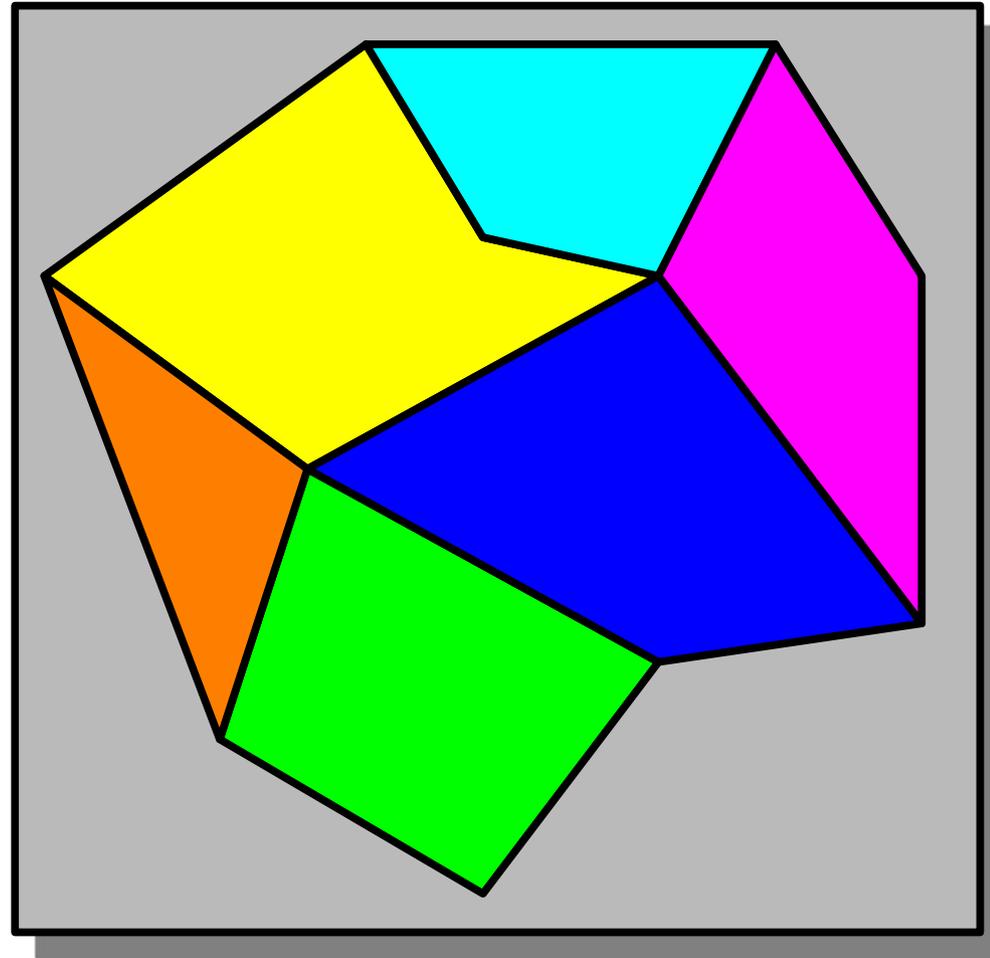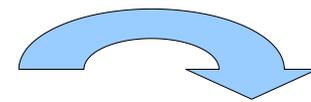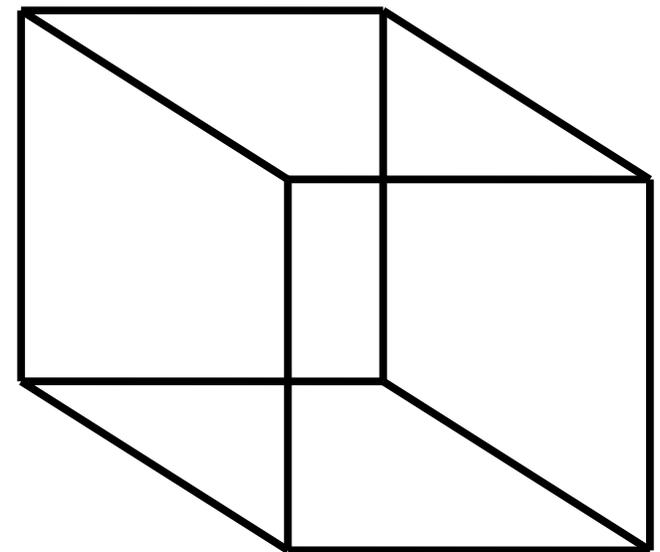
Plane Graphs

- A plane graph subdivides 2D space into regions called **faces**, spaces demarcated by edges.

- The outer region of space outside the plane graph is considered a face as well.

- When working with plane graphs, often the *faces* are more important than the *vertices*.
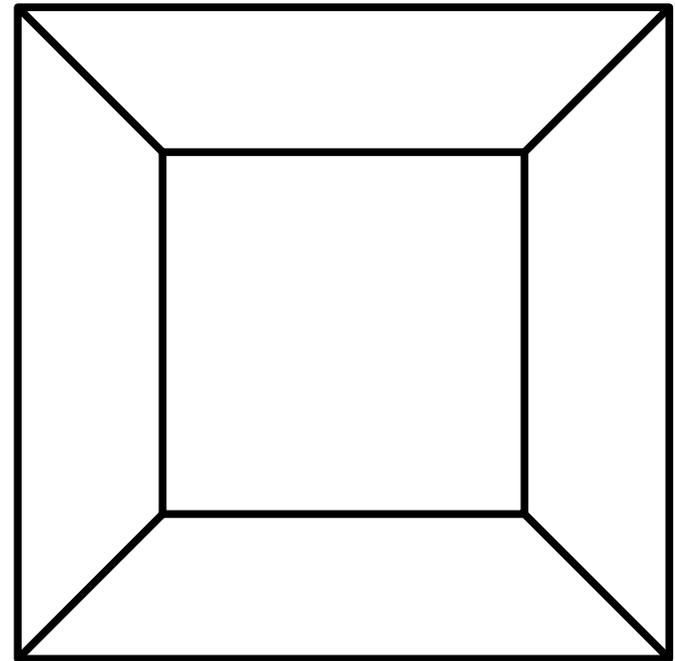
A Key Theorem

- ***Theorem:*** A plane graph with n nodes has O(n) edges and O(n) faces.

- This is a consequence of ***Euler's theorem***, which relates the number of edges, vertices, and faces of a plane graph.

- This means that plane graphs are sparse.

- We will typically use n, the number of nodes, as a measure of the size of the input.
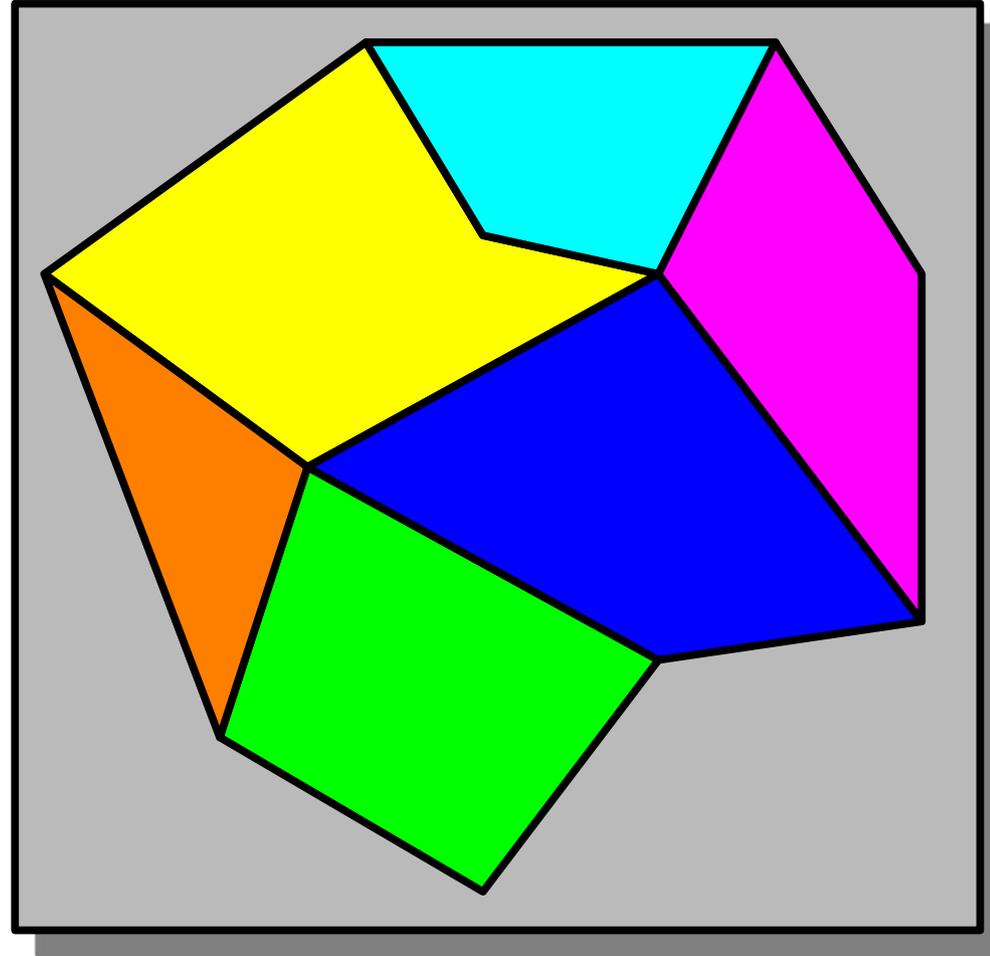
The Story So Far

- A plane graph is a drawing of a graph in the plane. We'll assume we're working with straight-line drawings today.

- A plane graph with n nodes has O(n) edges (they're *sparse*).

- Plane graphs decompose the plane into O(n) faces, including the infinite external face.

Planar Subdivisions
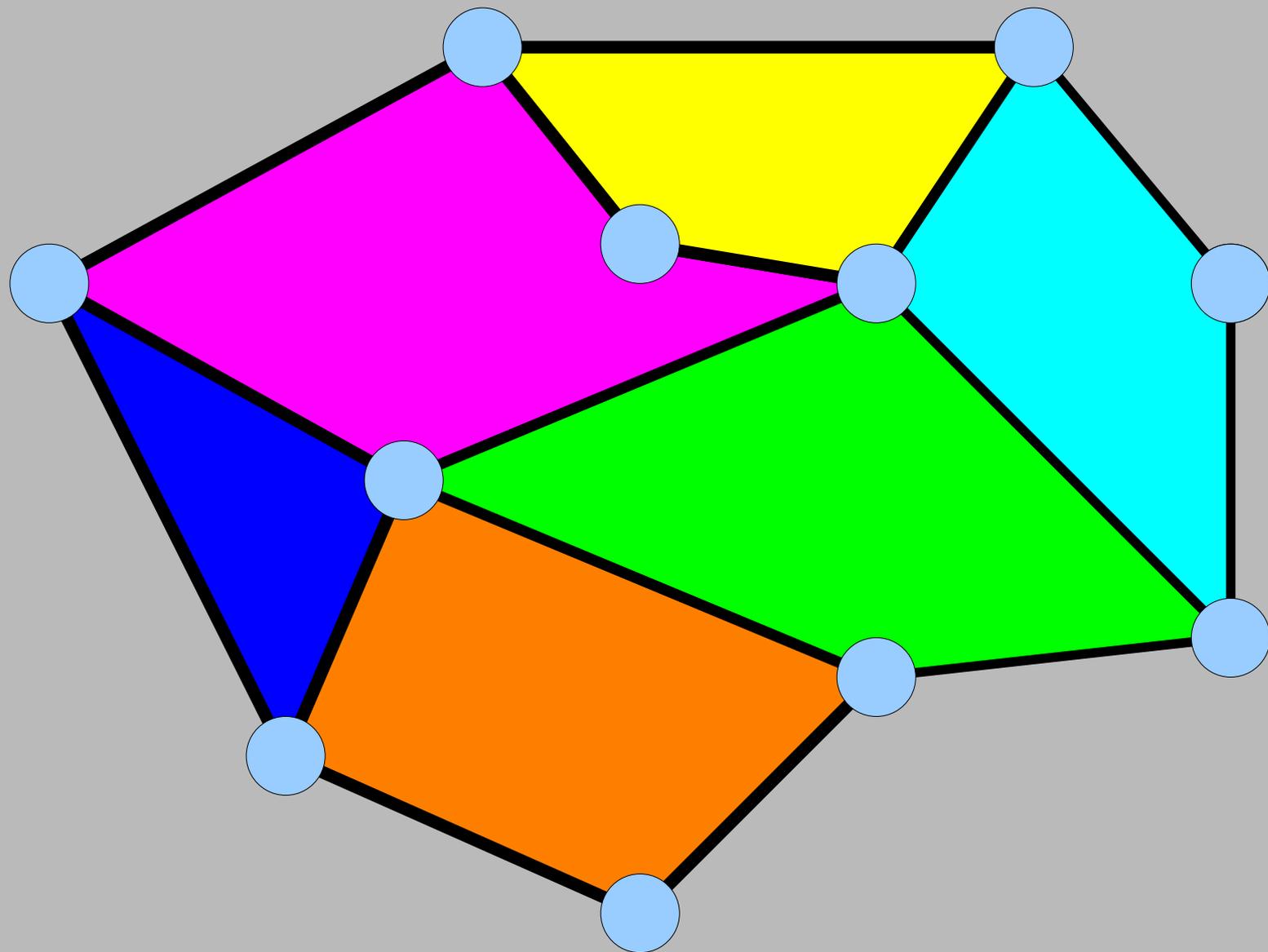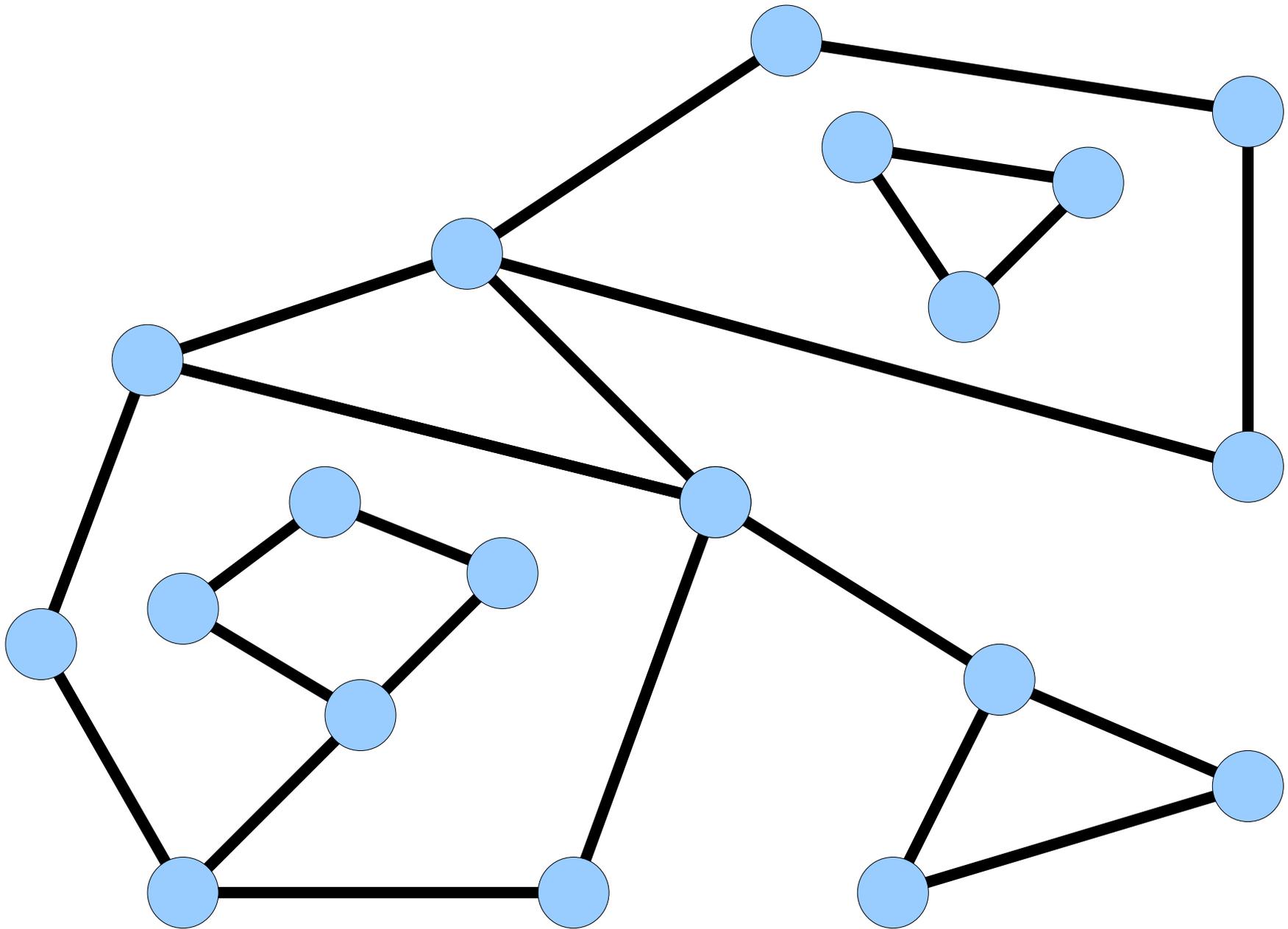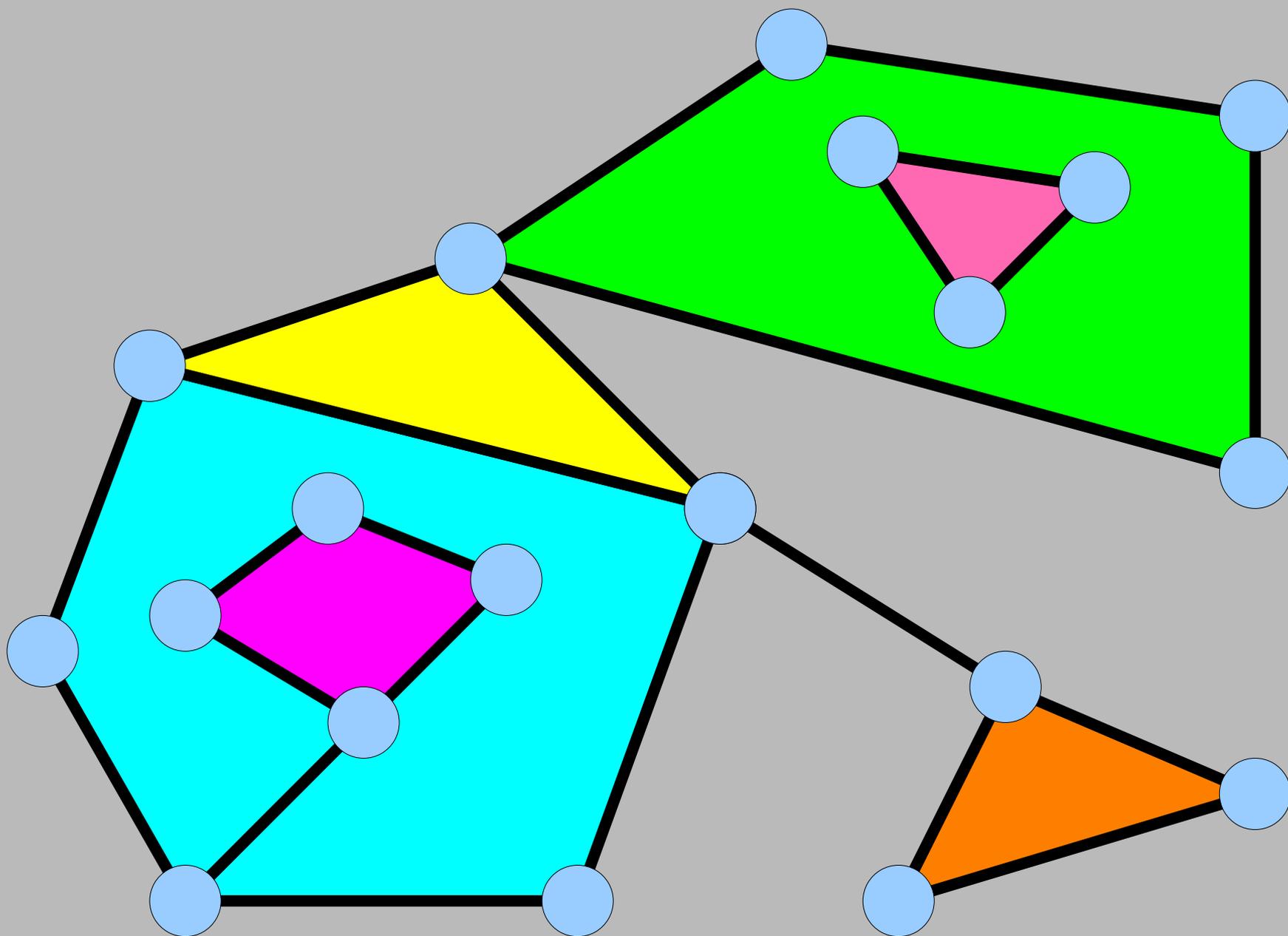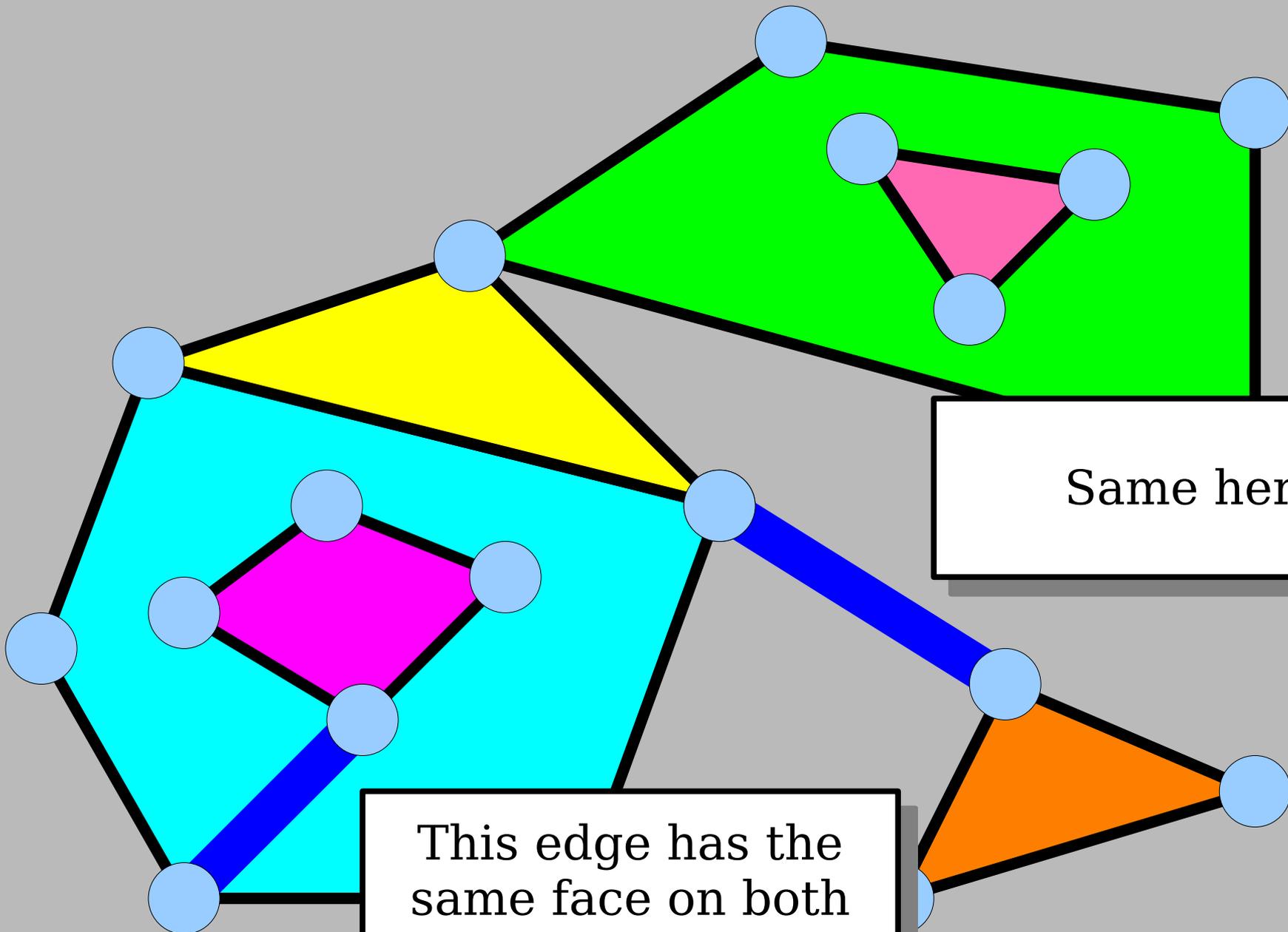
This edge has the same face on both sides.

Same here.

Planar Subdivisions

- A ***planar subdivision*** is a plane graph where each edge borders two different faces.

- Equivalently, a planar subdivision is a plane graph with no bridges.
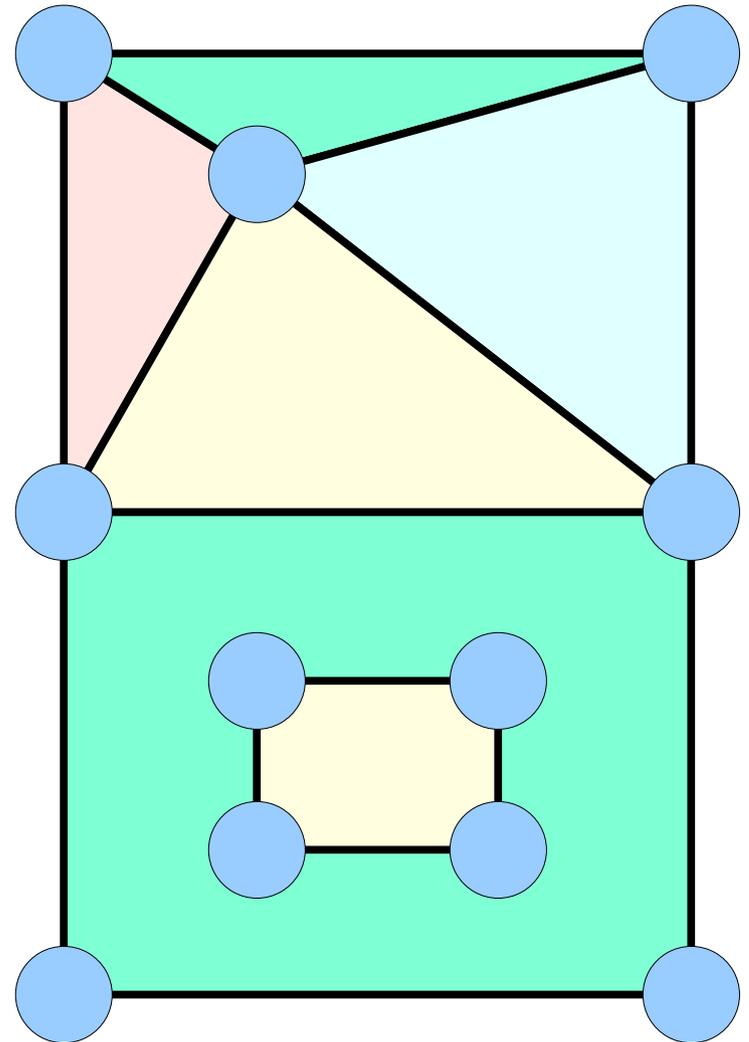
Planar Point Location

Planar Point Location

- The ***planar point location problem*** is the following:

  Preprocess a planar subdivision to efficiently answer queries of the form "which face is point *p* in?"

- It's yet another place where we'd expect to get a tradeoff between preprocessing time and runtime.

- ***Question:*** How quickly can we solve this problem?
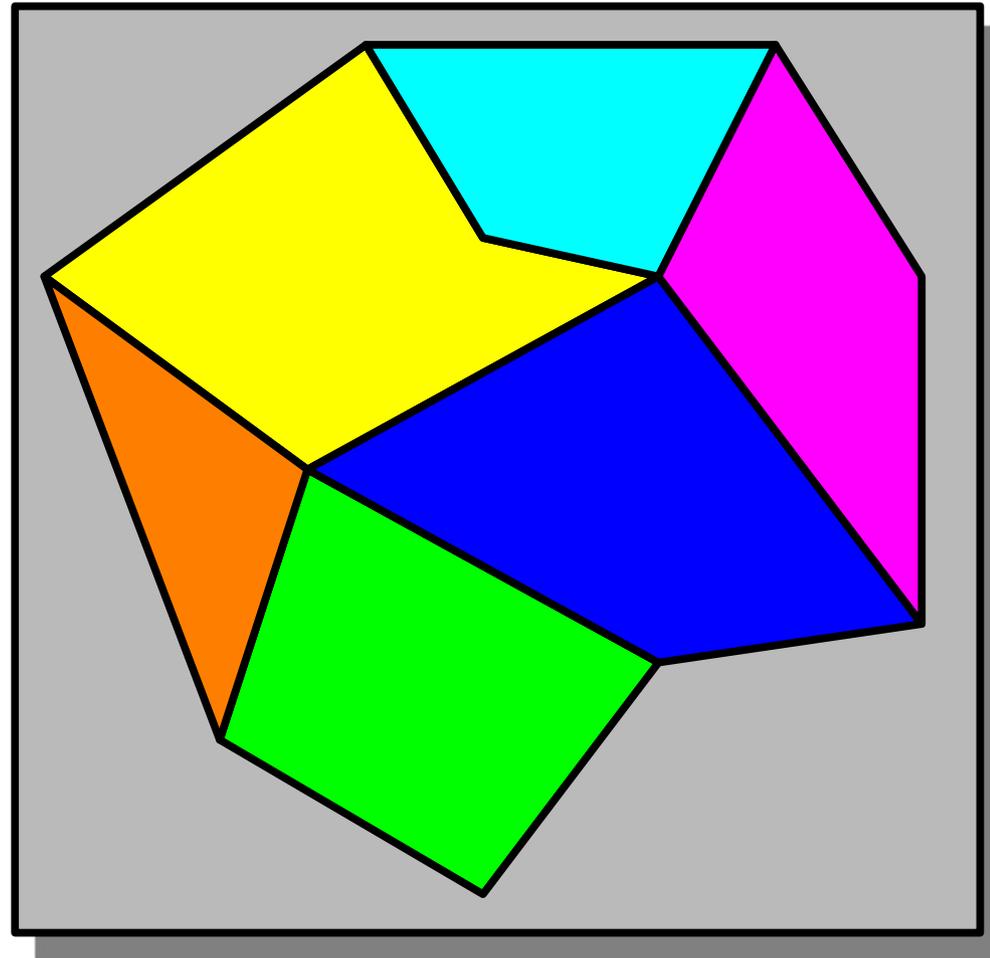
A Naive Solution

- What would the "no preprocessing" solution to this problem look like?

- **Basic idea:** Iterate over all closed faces and see if the point is in any of them. If so, return it. If not, return the external face.

- **Question:** How do you check whether a point is contained within a face?
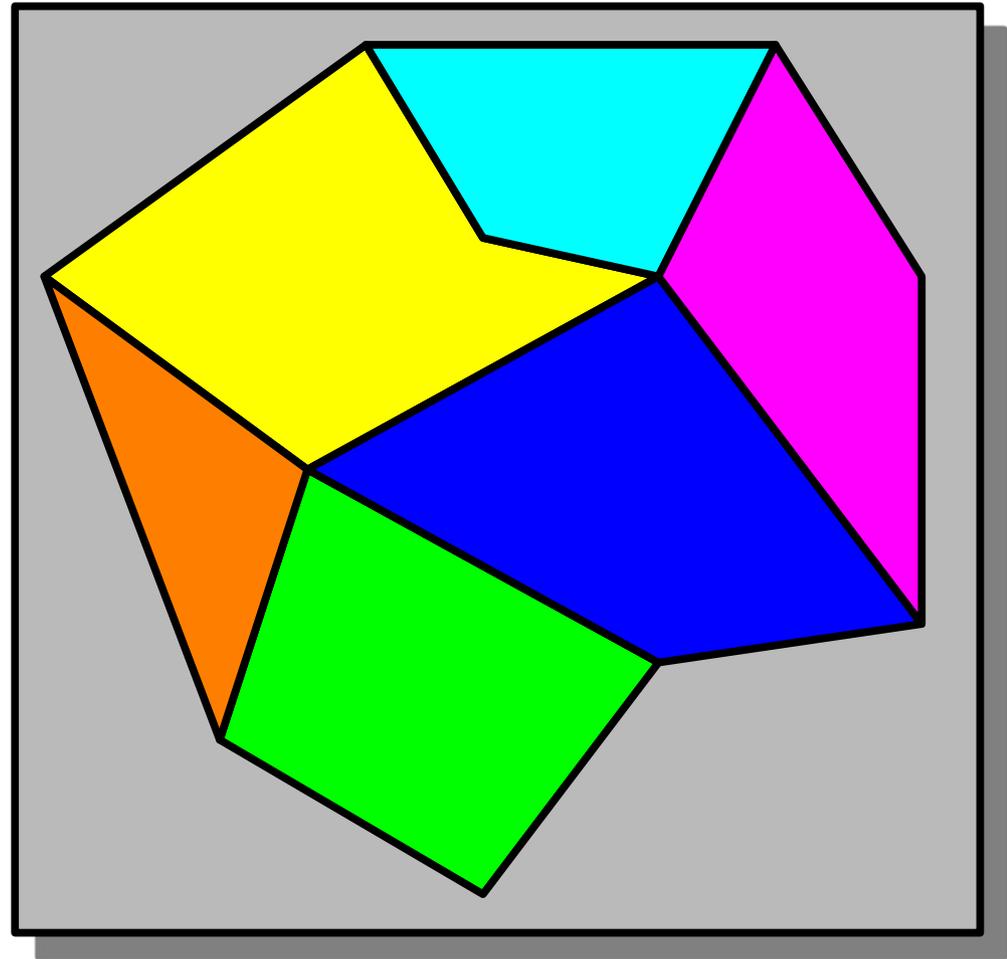
Point-In-Face Queries

- Faces in planar subdivisions can have irregular shapes.

- Simply testing if a point is in such a polygon seems challenging!

- However, there's a really clever way to solve this problem.
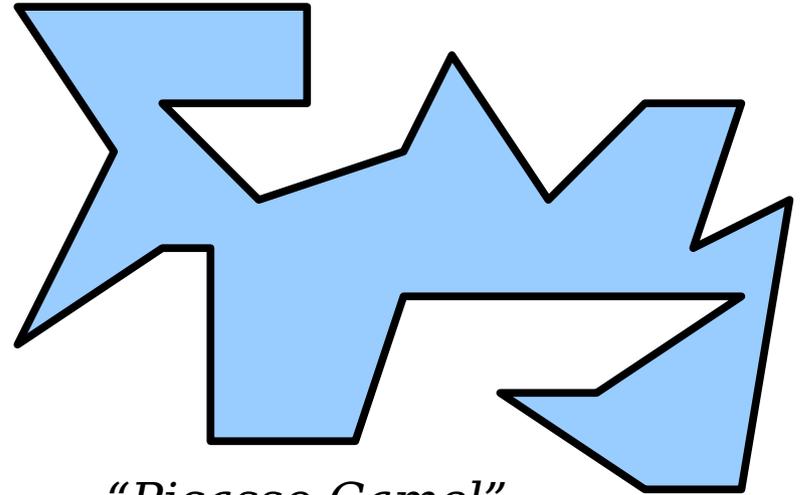
"Picasso Camel"

"Jack-O-Lantern"
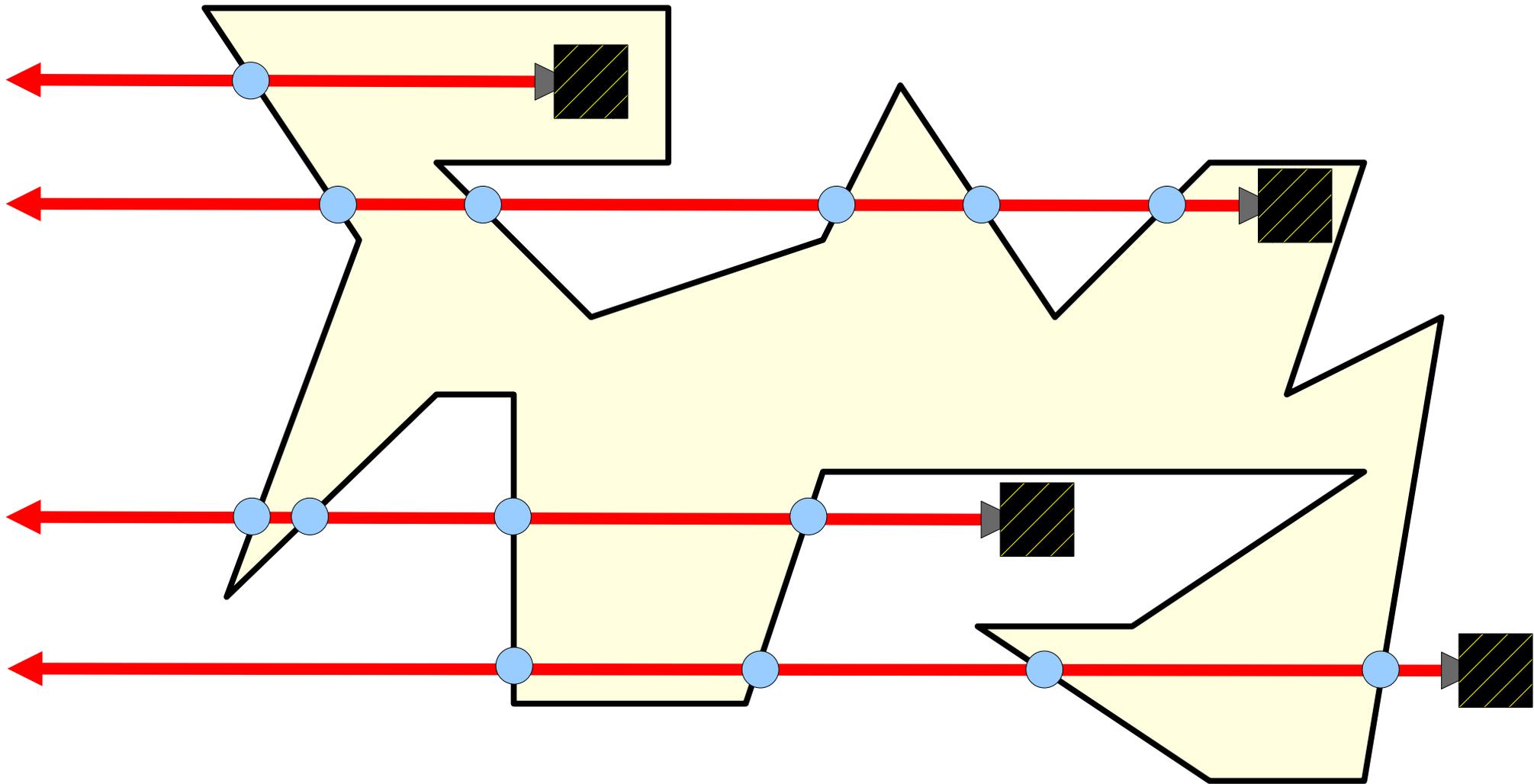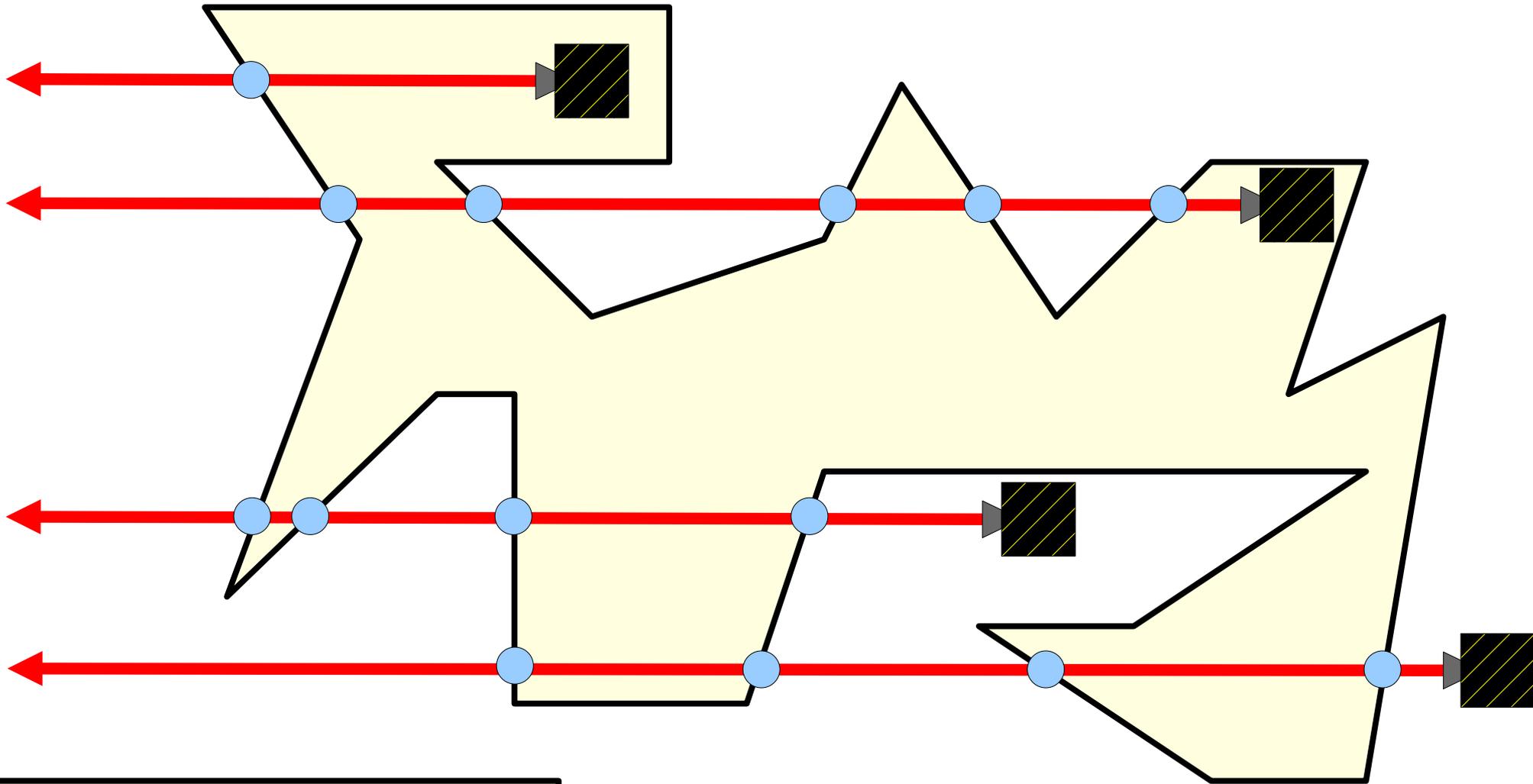
Notice a pattern in the number of intersections?

Formulate a hypothesis!
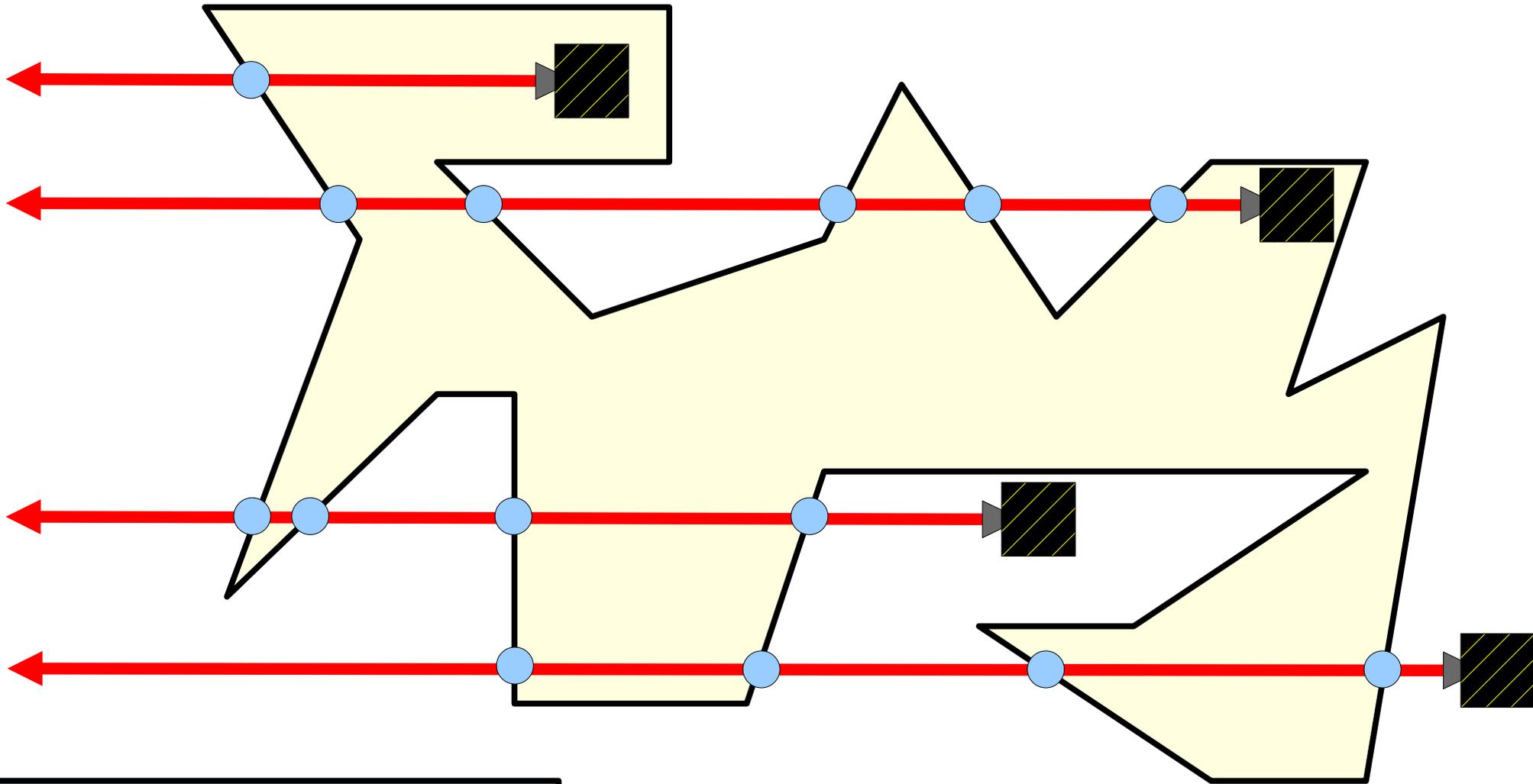
Notice a pattern in the number of intersections?

Discuss with your neighbors!
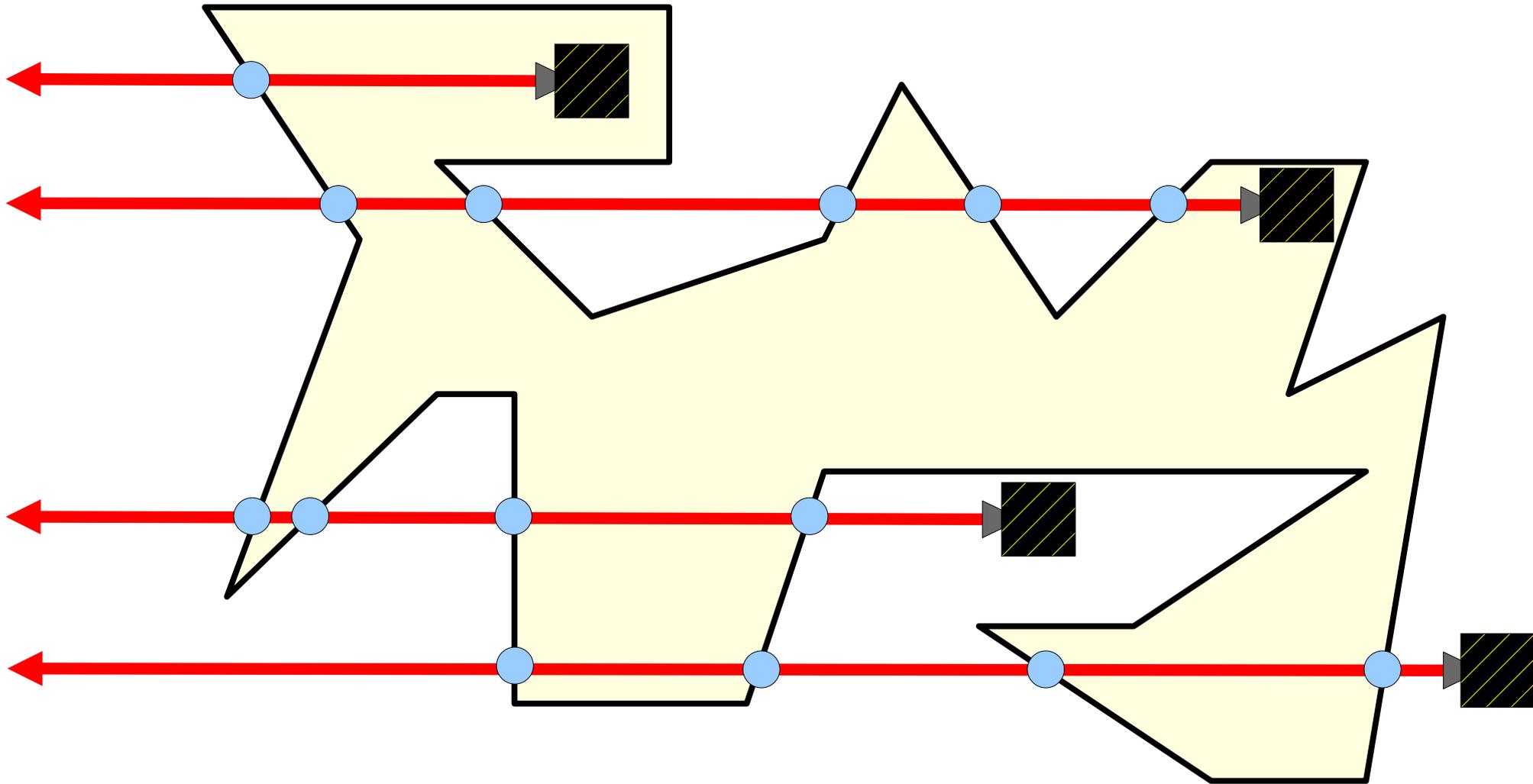
Each line crossed toggles whether we're outside or inside the polygon.

Suppose we cross k line segments. If k is even, the point is outside. If k is odd, it's inside.
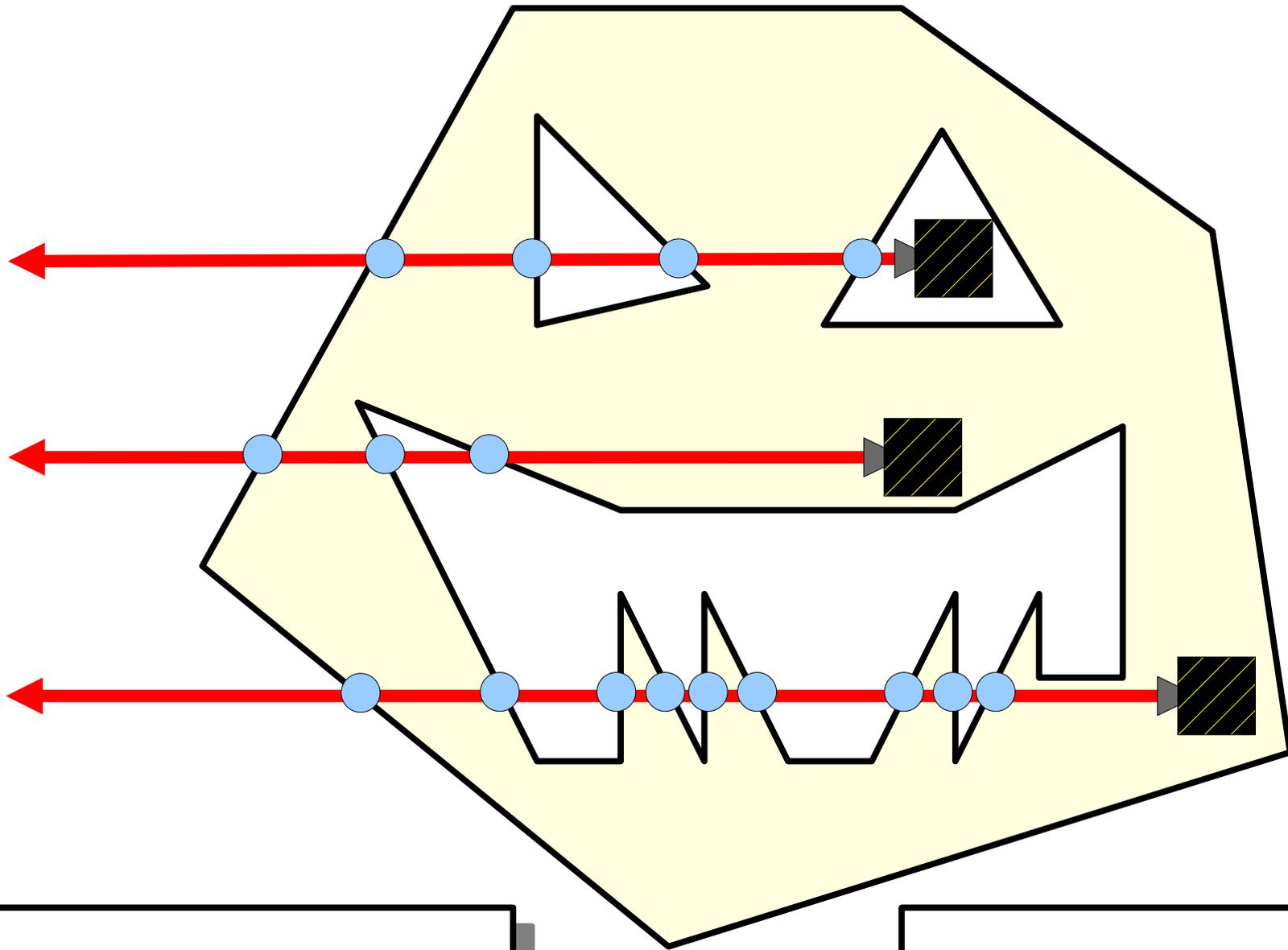
Each line crossed toggles whether we're outside or inside the polygon.

Suppose we cross k line segments. If k is even, the point is outside. If k is odd, it's inside.

A Naive Solution

- For each polygon, count the number of edges that intersect a horizontal line passing through the query point.
  - This intersection test boils down to some linear algebra and can be done in time O(1).
- Each edge is processed at most twice, once for each face it borders.
- Total query time: **O(n)**.
- ***Question:*** Can we do better?

	Preprocessing Time	Query Time	Space Usage
Test All Faces	O($n \log n$)	O(n)	O(n)

Slab Decomposition
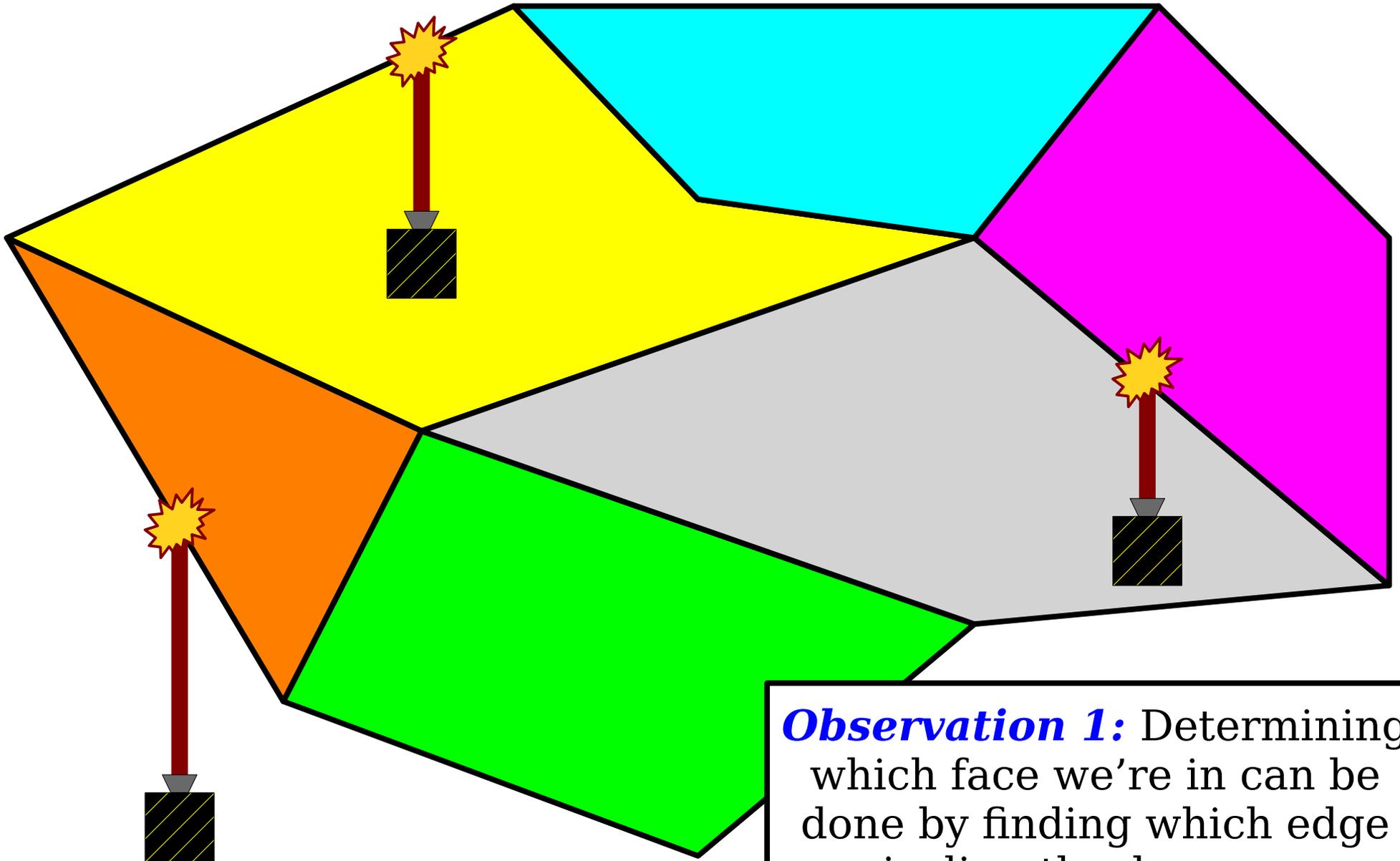
Observation 1: Determining which face we're in can be done by finding which edge is directly above us.

Observation 2: Nudging a point left or right usually doesn't change which segment is above it. That can only change if we cross the *x* coordinate of a vertex.
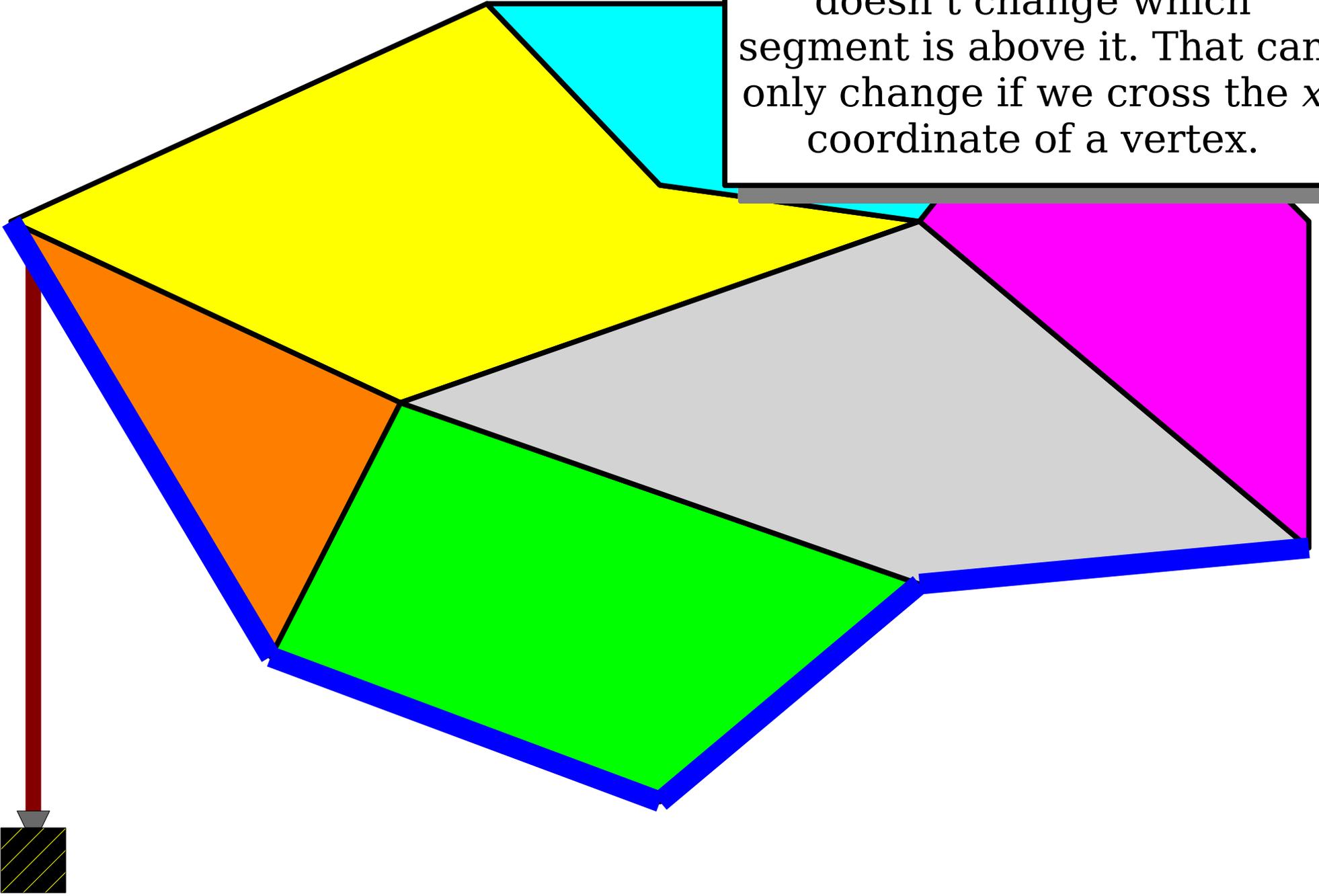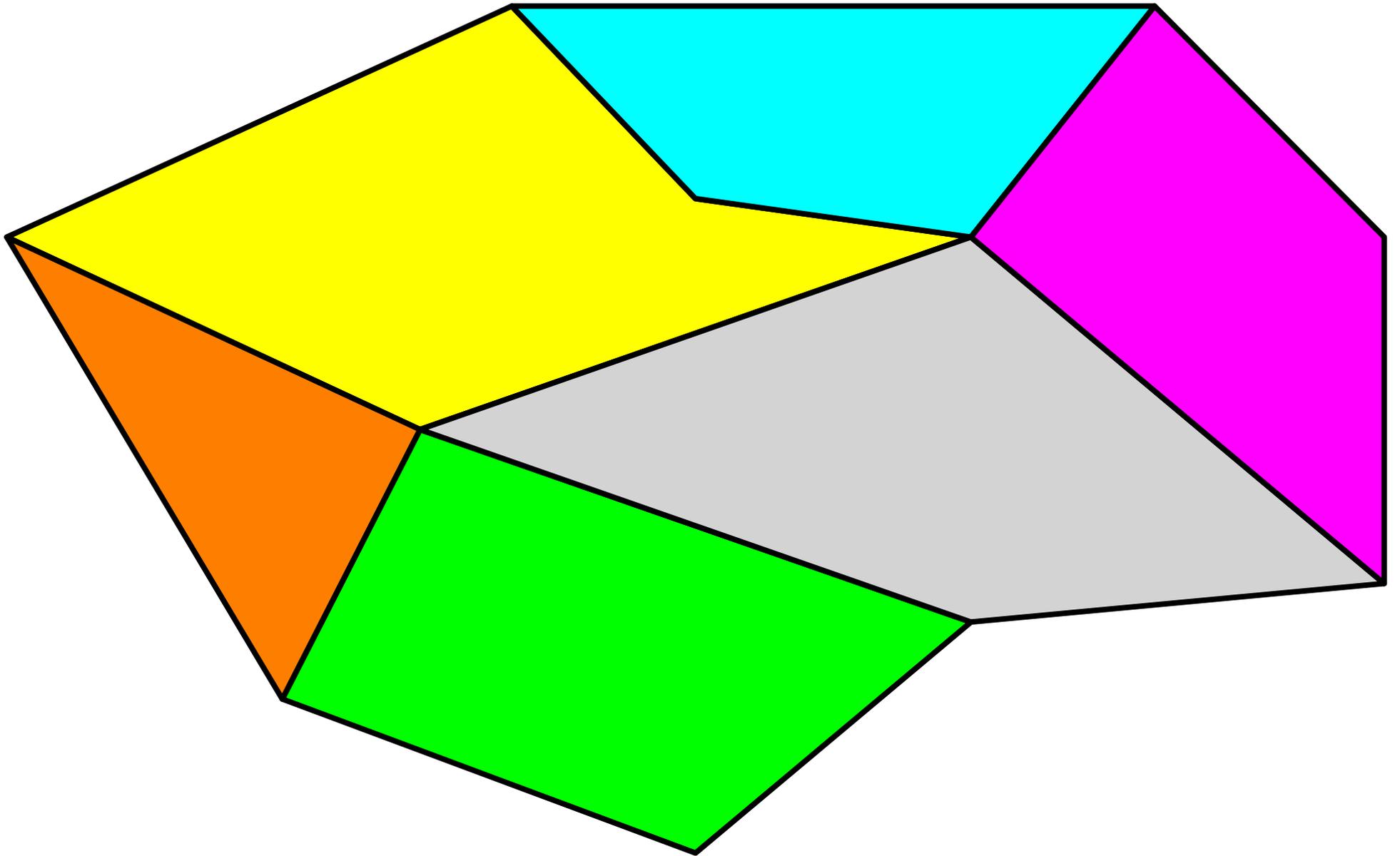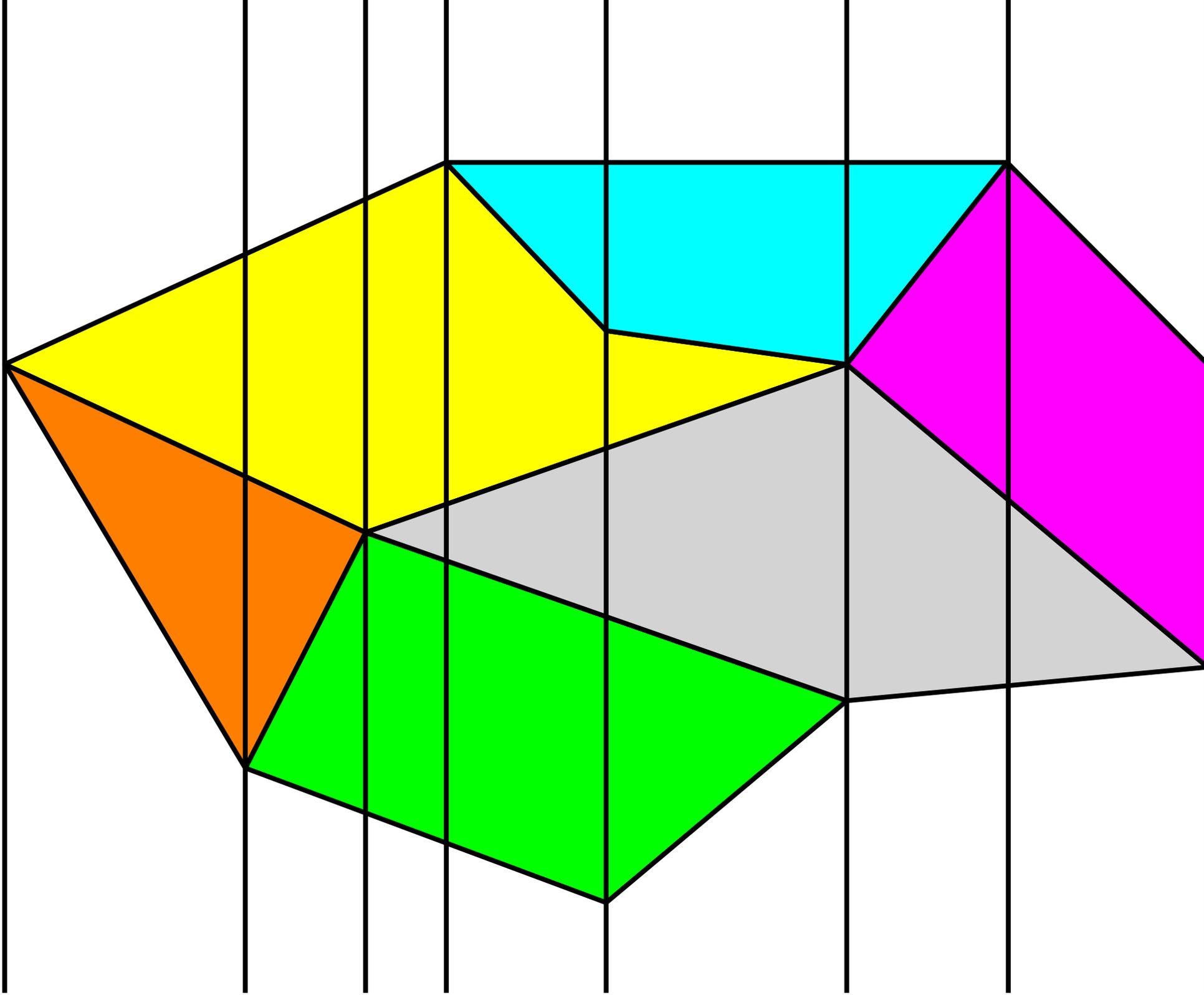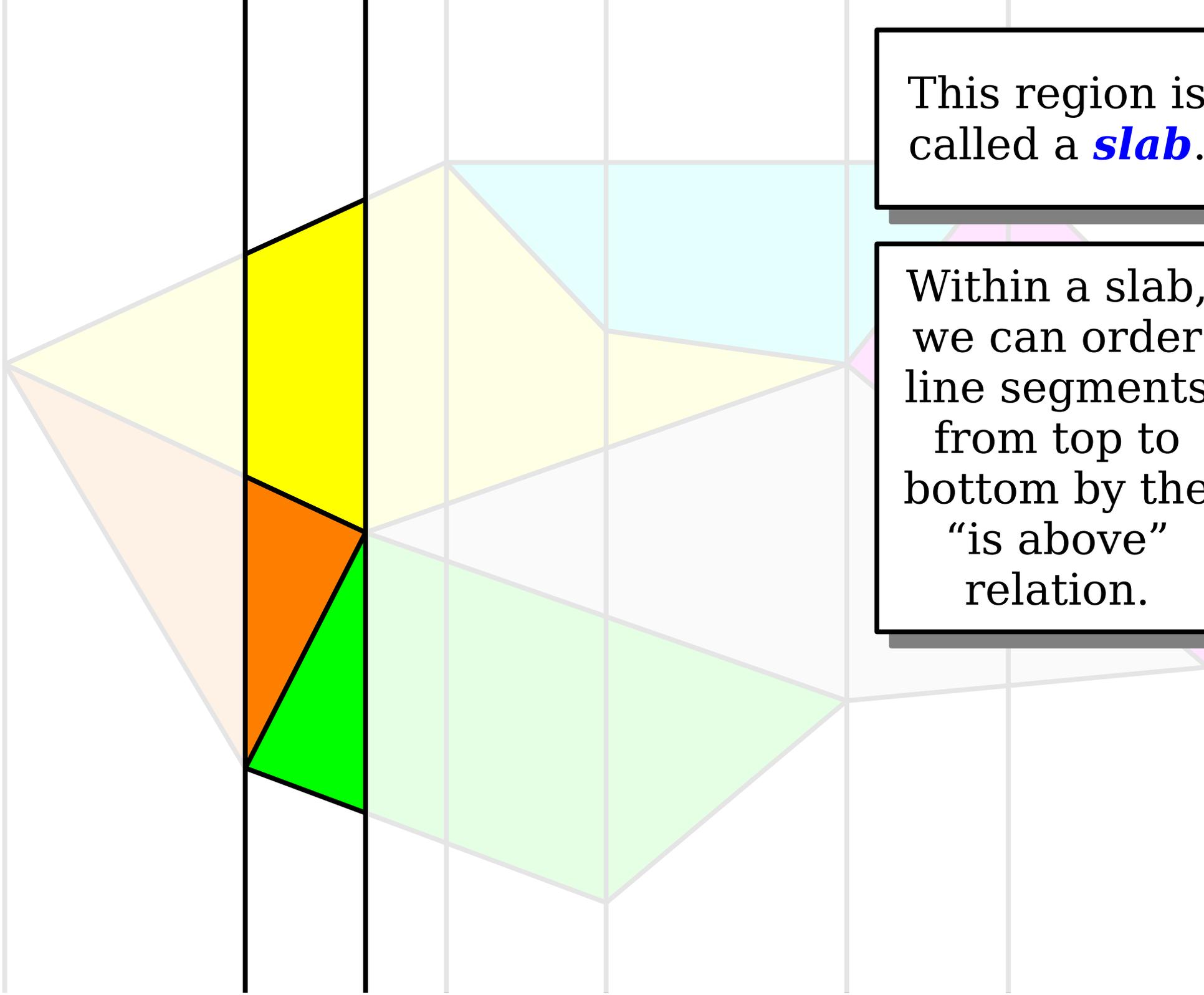
This region is called a *slab*.

Within a slab, we can order line segments from top to bottom by the "is above" relation.
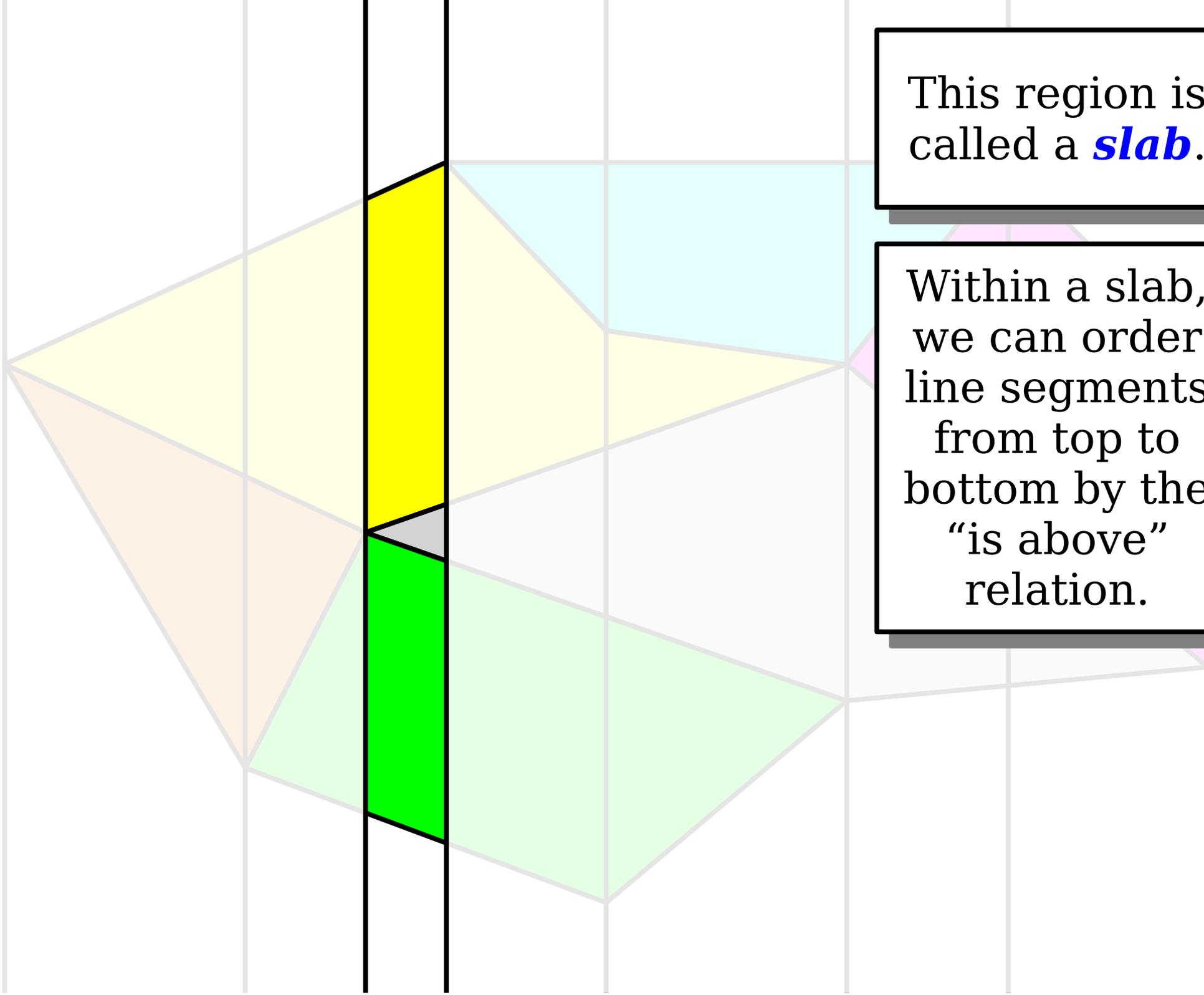
This region is called a *slab*.

Within a slab, we can order line segments from top to bottom by the "is above" relation.
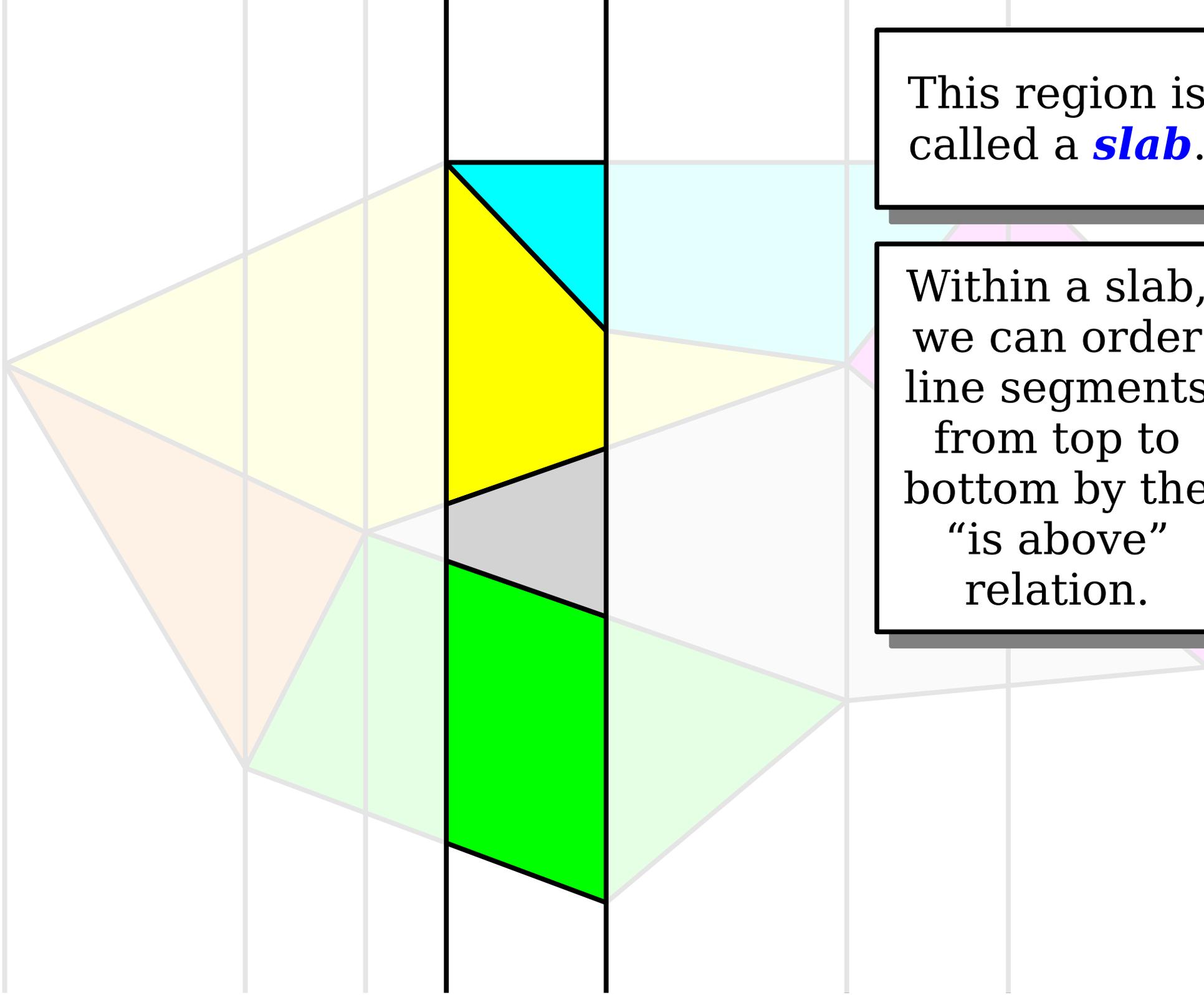
This region is called a **slab**.

Within a slab, we can order line segments from top to bottom by the "is above" relation.
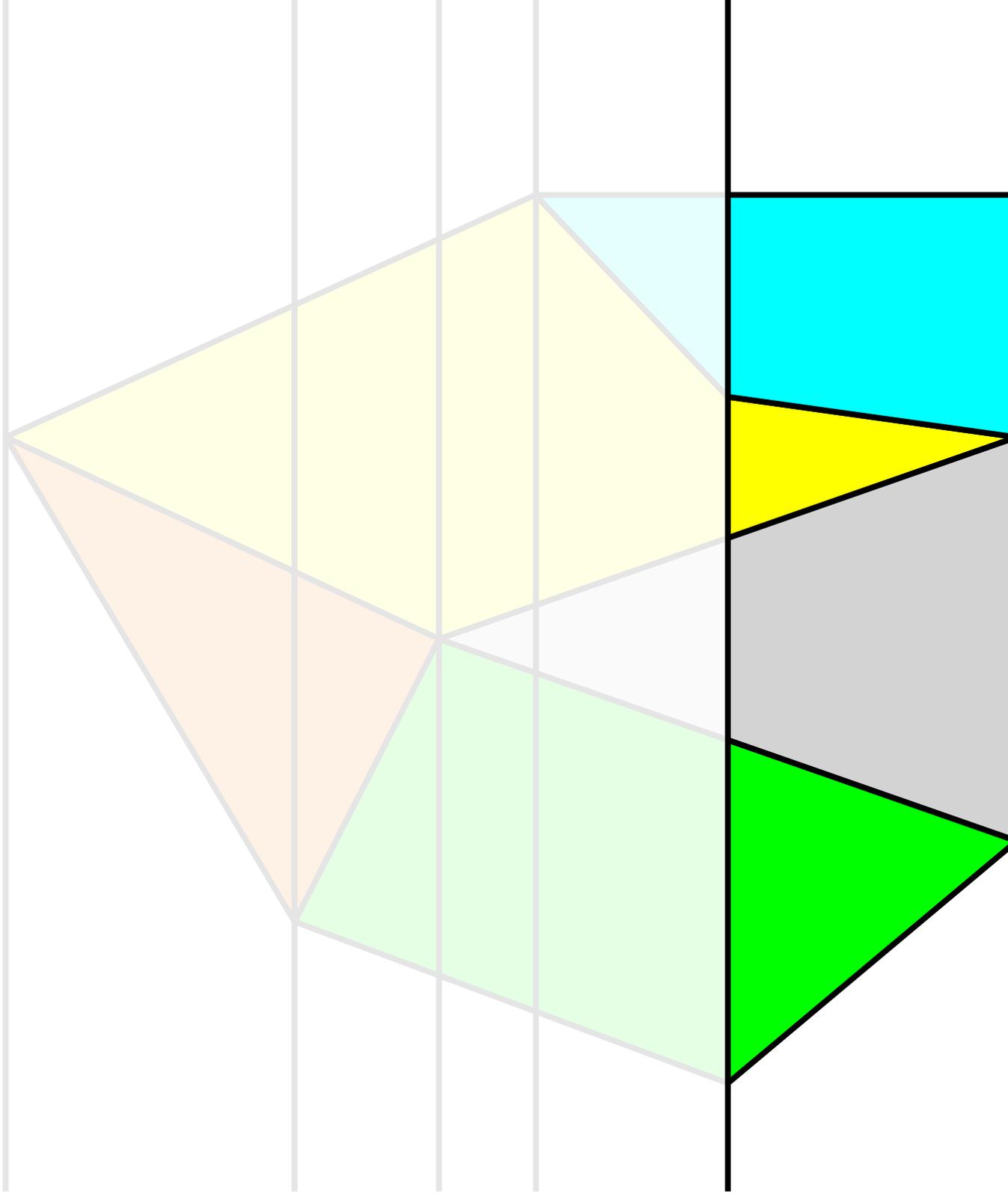
This region is called a **slab**.

Within a slab, we can order line segments from top to bottom by the "is above" relation.
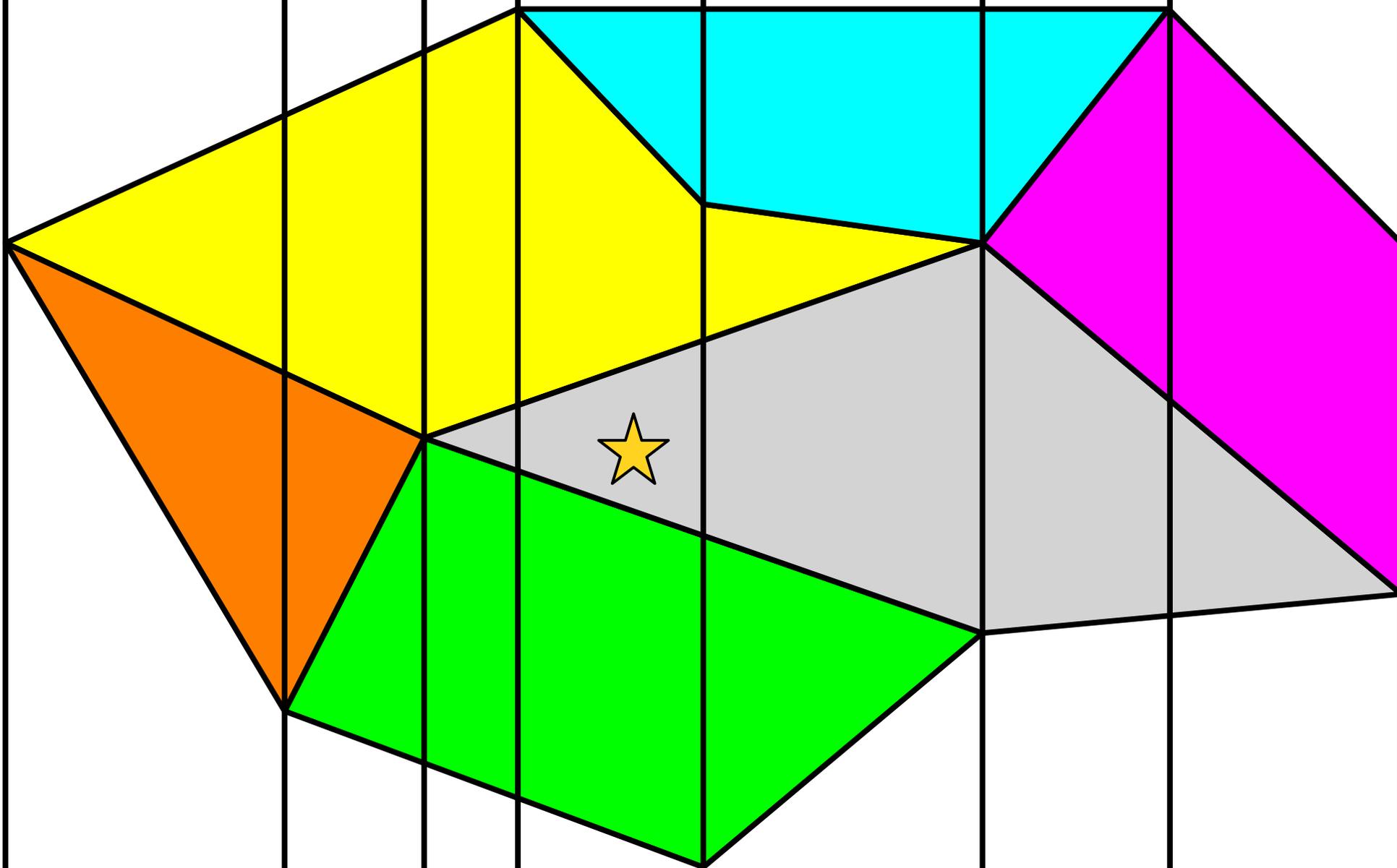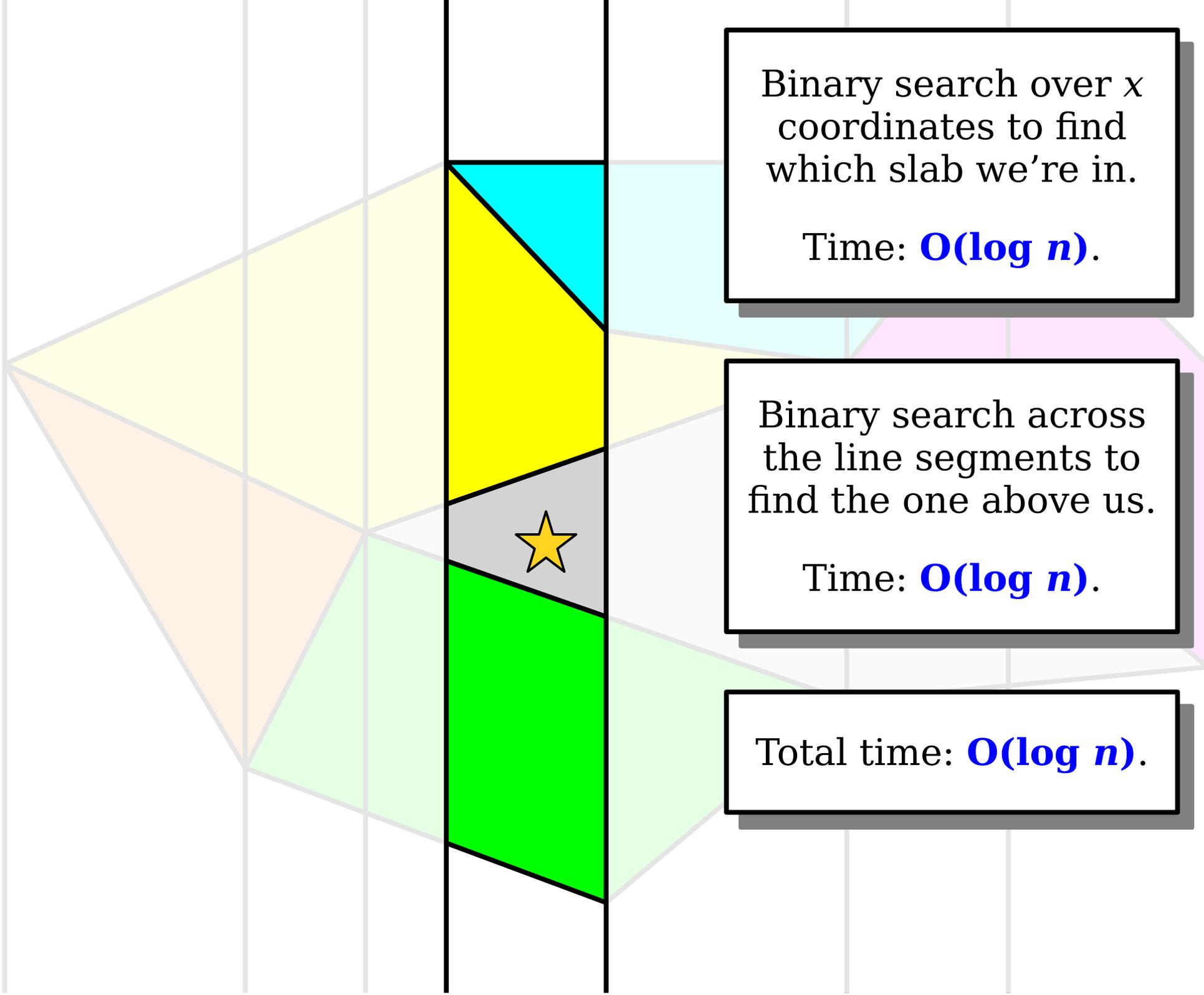
Binary search over *x* coordinates to find which slab we're in.

Time: **O(log *n*)**.

Binary search across the line segments to find the one above us.

Time: **O(log *n*)**.

Total time: **O(log *n*)**.

Slab Decomposition

- **Idea:** Draw a vertical line through each line segment endpoint to cut the space into *slabs*.

- Each slab consists of some number of line segments, which can be ordered from top to bottom.

- We can then solve point location in time **O(log n)** by binary searching in the *x* direction to find which slab we're in, then binary searching in the *y* direction to see which face we're in.
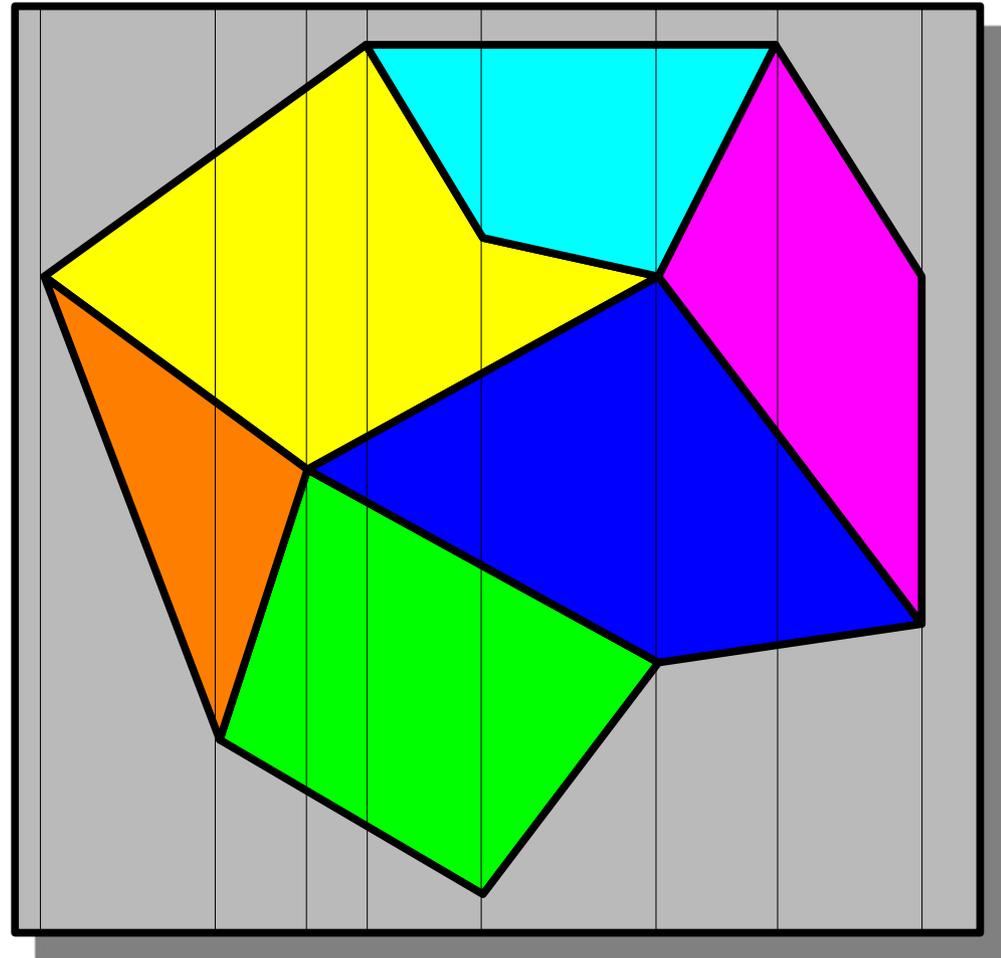
Slab Decomposition

- Some remaining questions:
  - How do you construct the slabs?
  - How much preprocessing time will this take?
  - How much space is needed?
- Let's address each of these questions in turn.
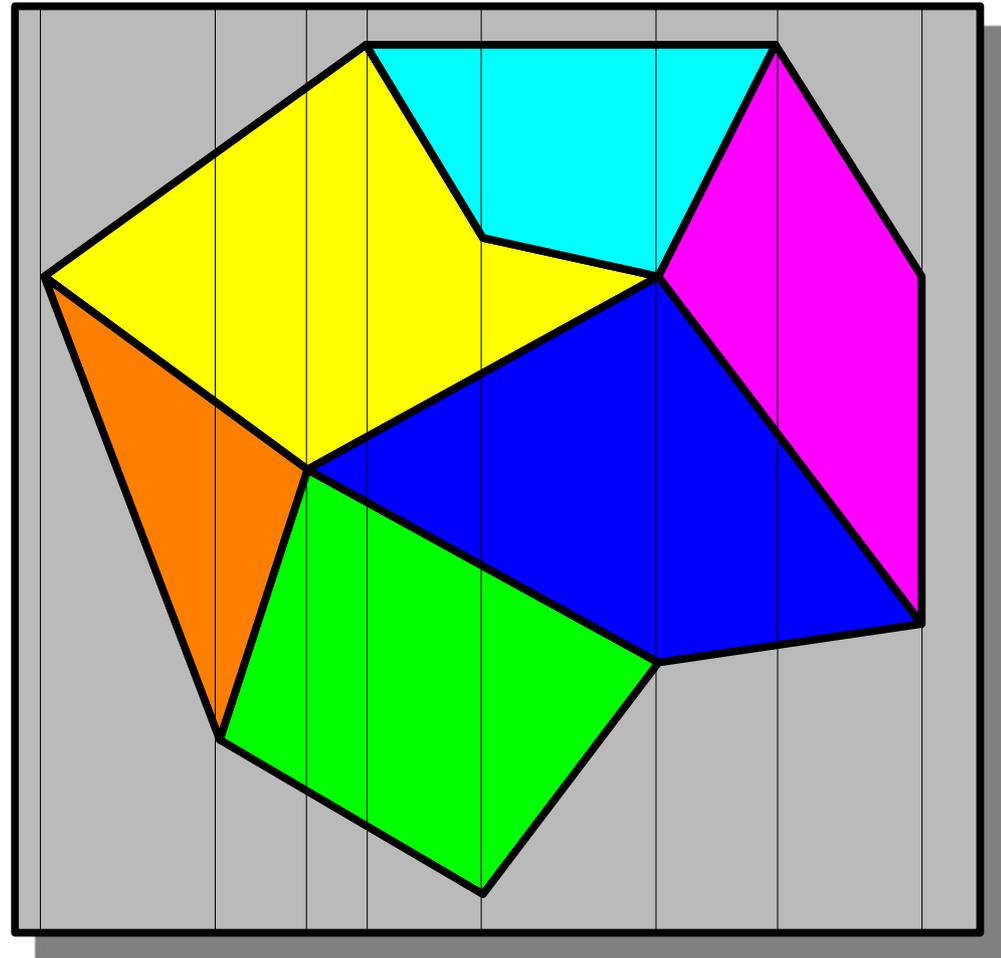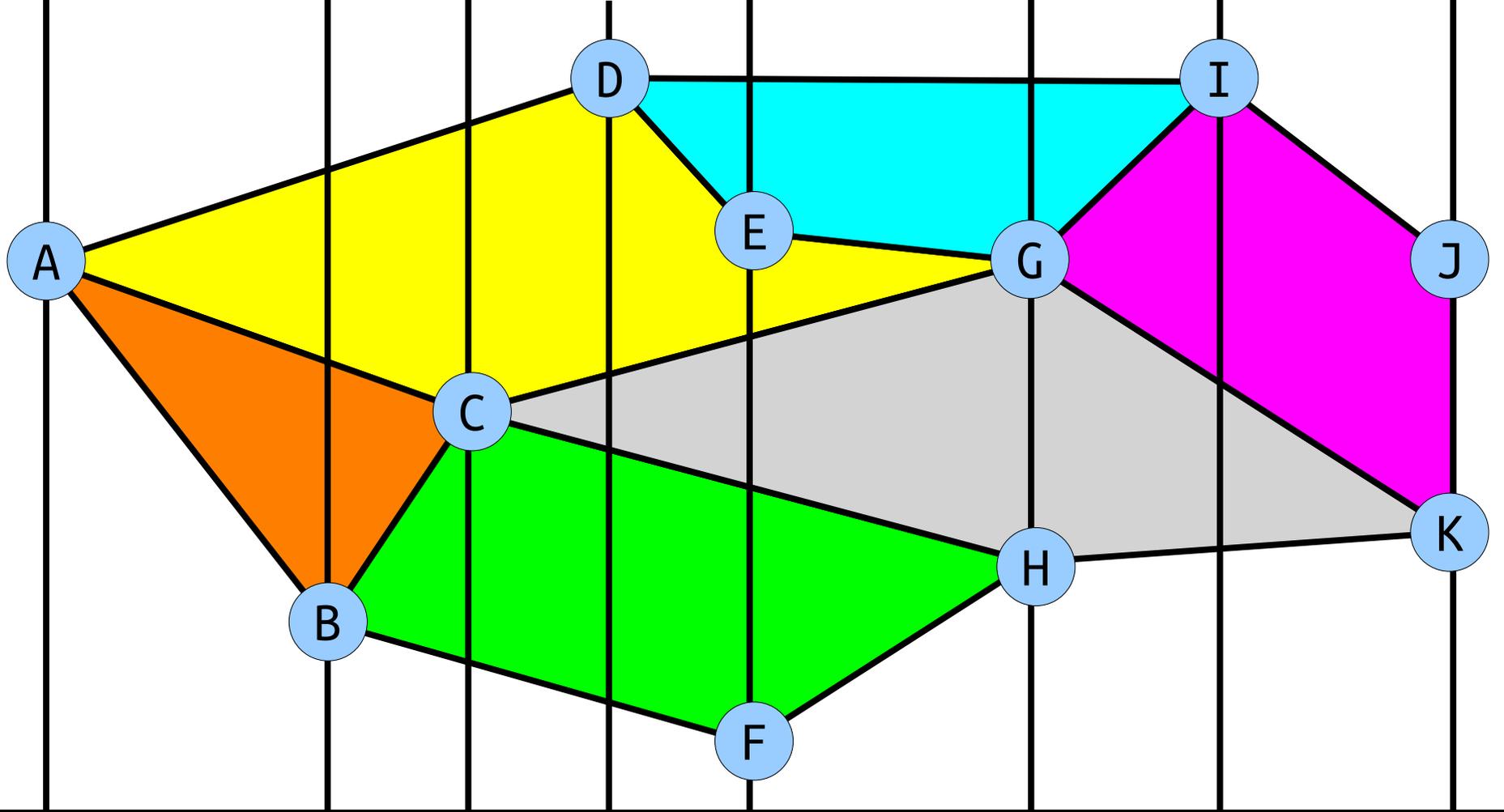
Slab 1

The slab to the left of all polygons has no line segments in it.
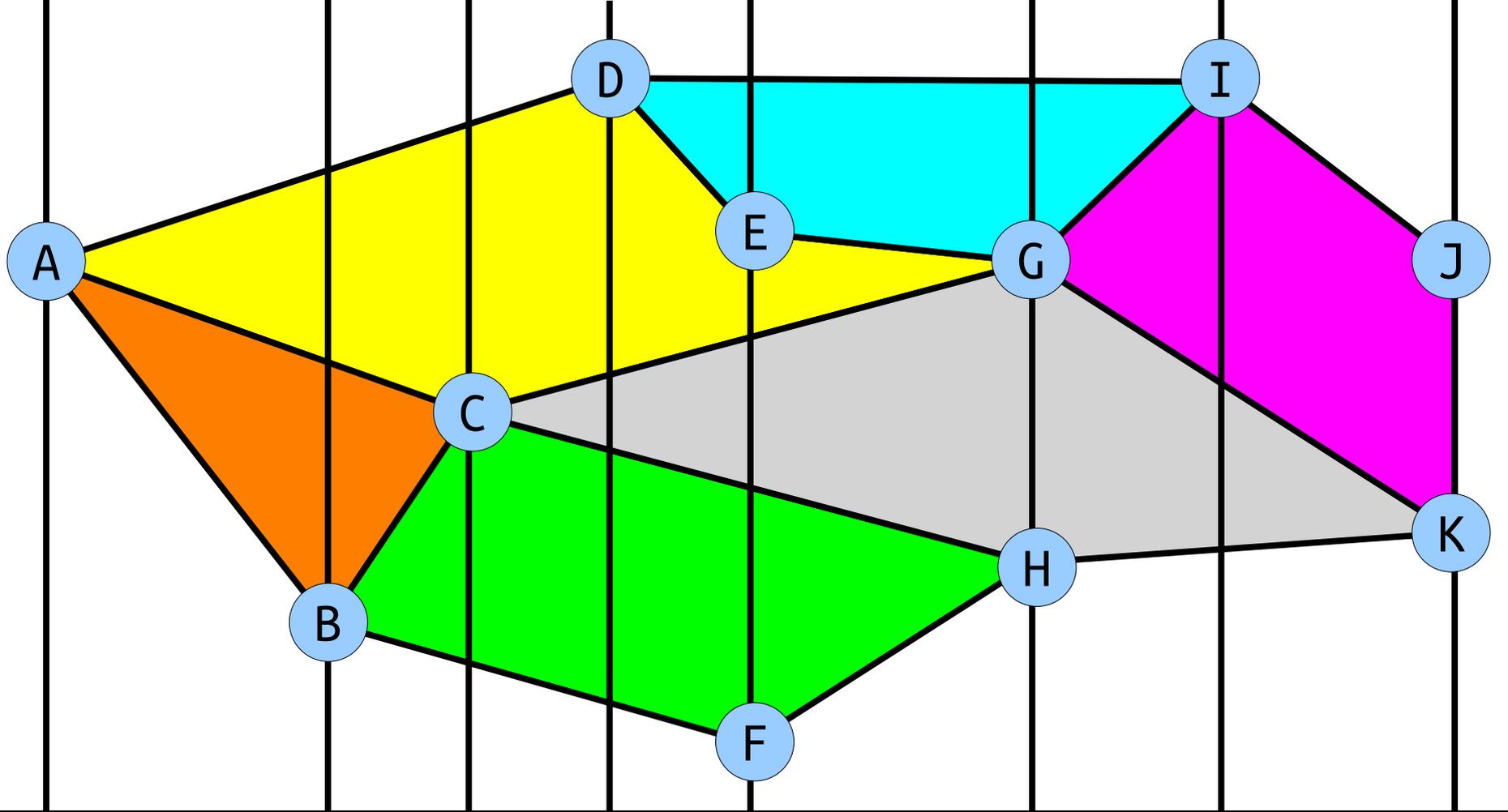
Slab 1 | Slab 2
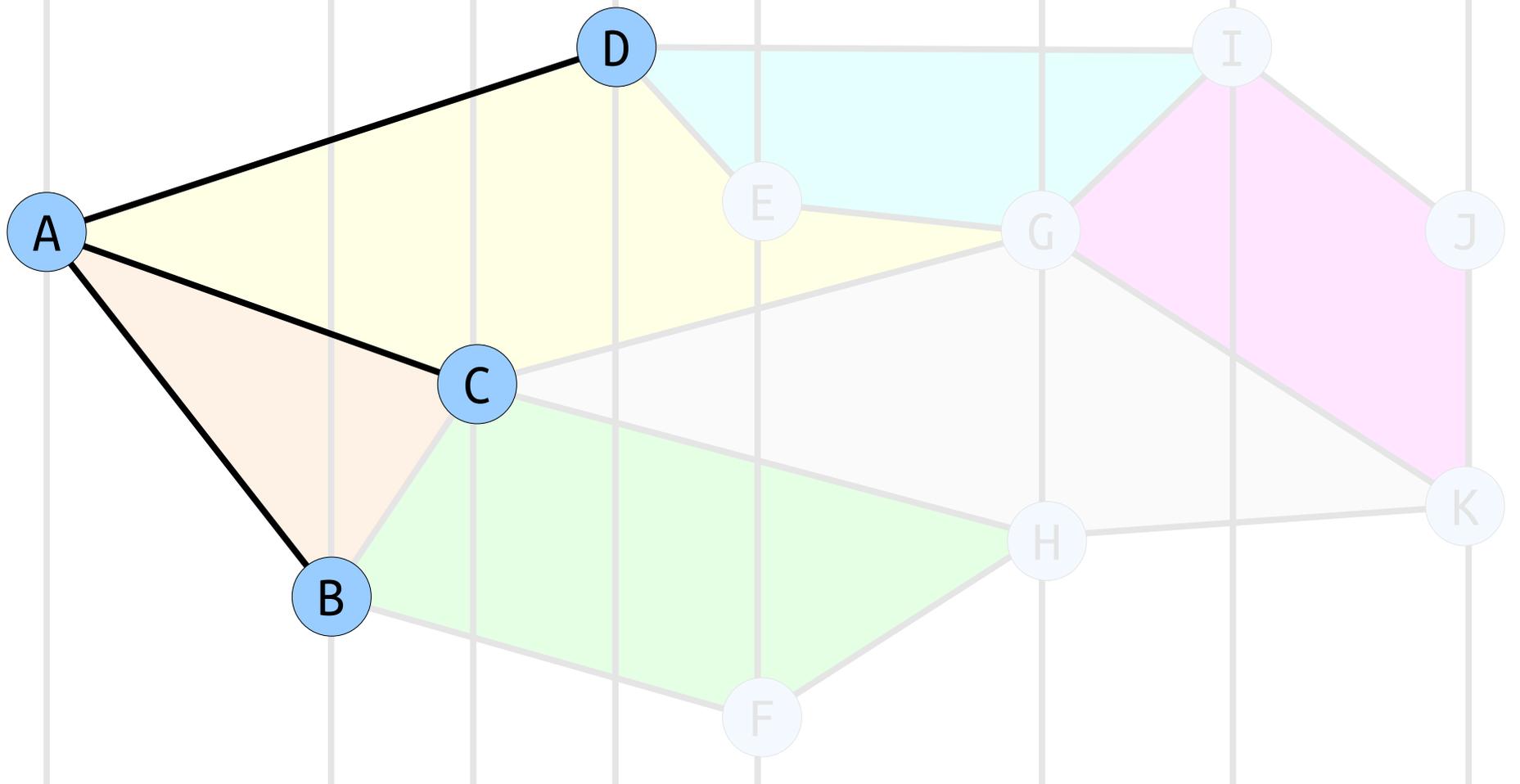
Slab 1	Slab 2
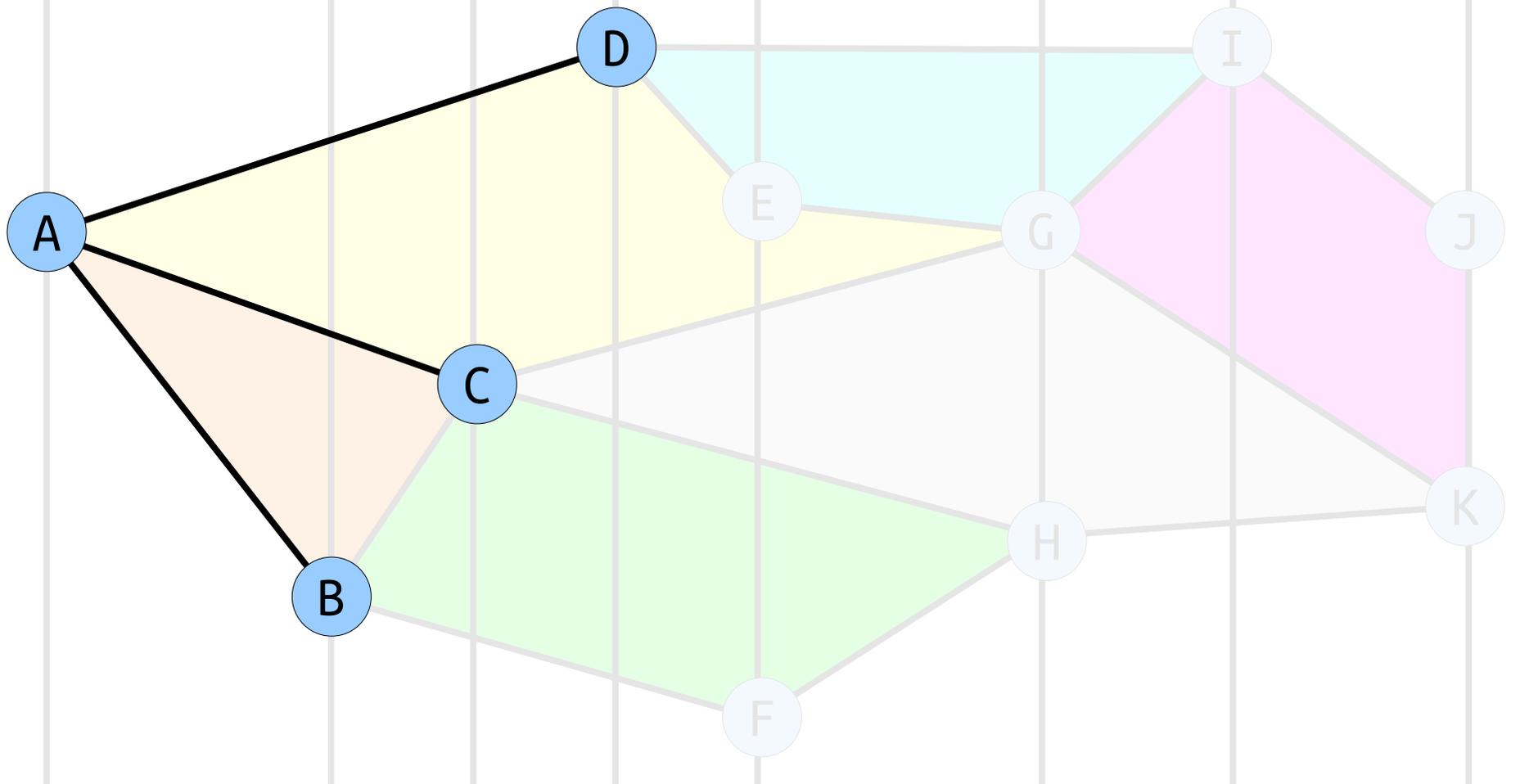
Slab 1	Slab 2

Sort all line segments emanating from *A* from top to bottom, and add them into the slab.
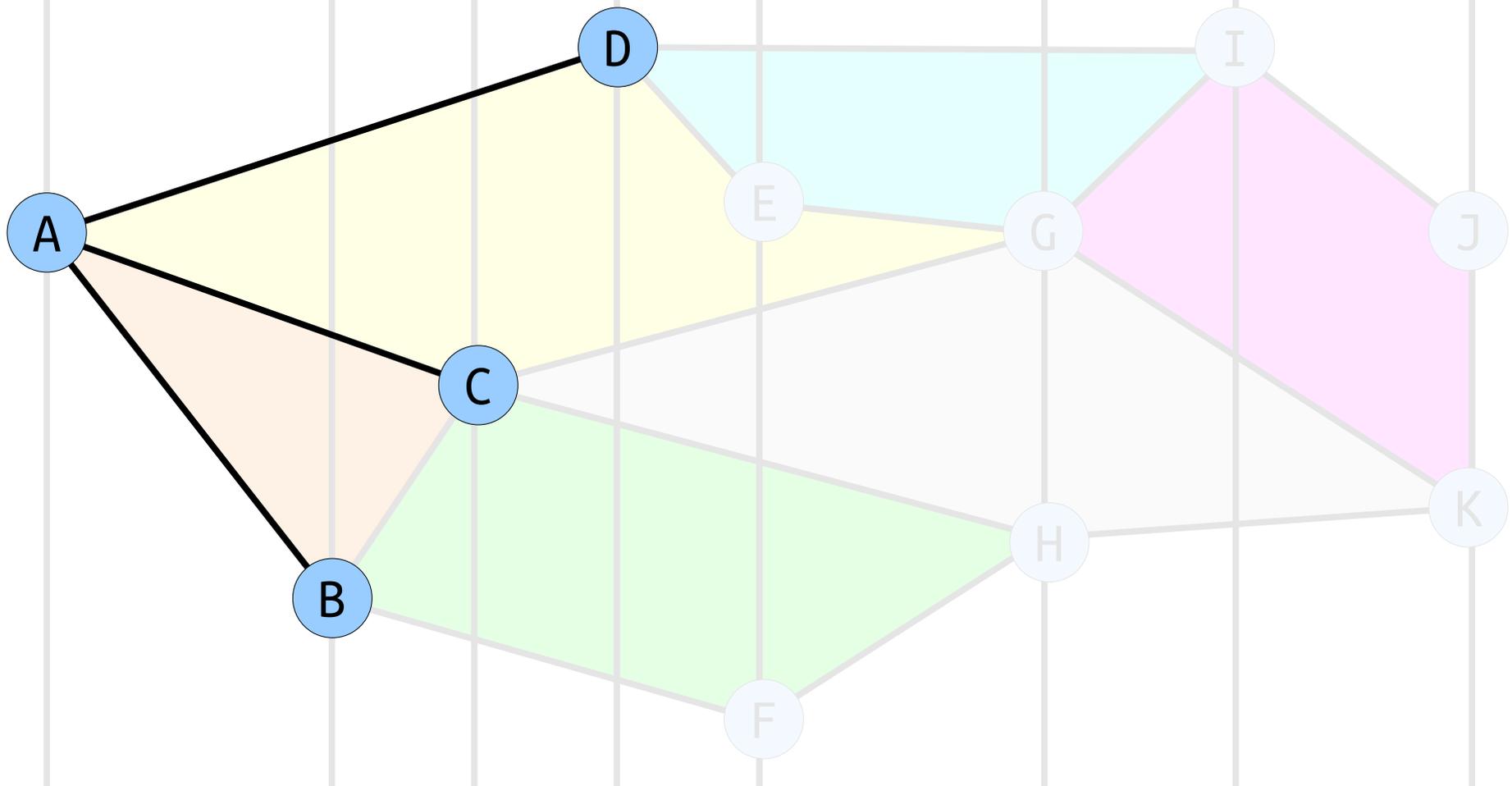
Slab 1	Slab 2
	AD AC AB

Sort all line segments emanating from *A* from top to bottom, and add them into the slab.

Slab 1	Slab 2
	AD
	AC
	AB

D I

A

E G

Many of the line segments from
before are still in this slab.

Some of the line segments from
before are no longer in this slab.

Some new line segments will be
added to this slab.

C

B
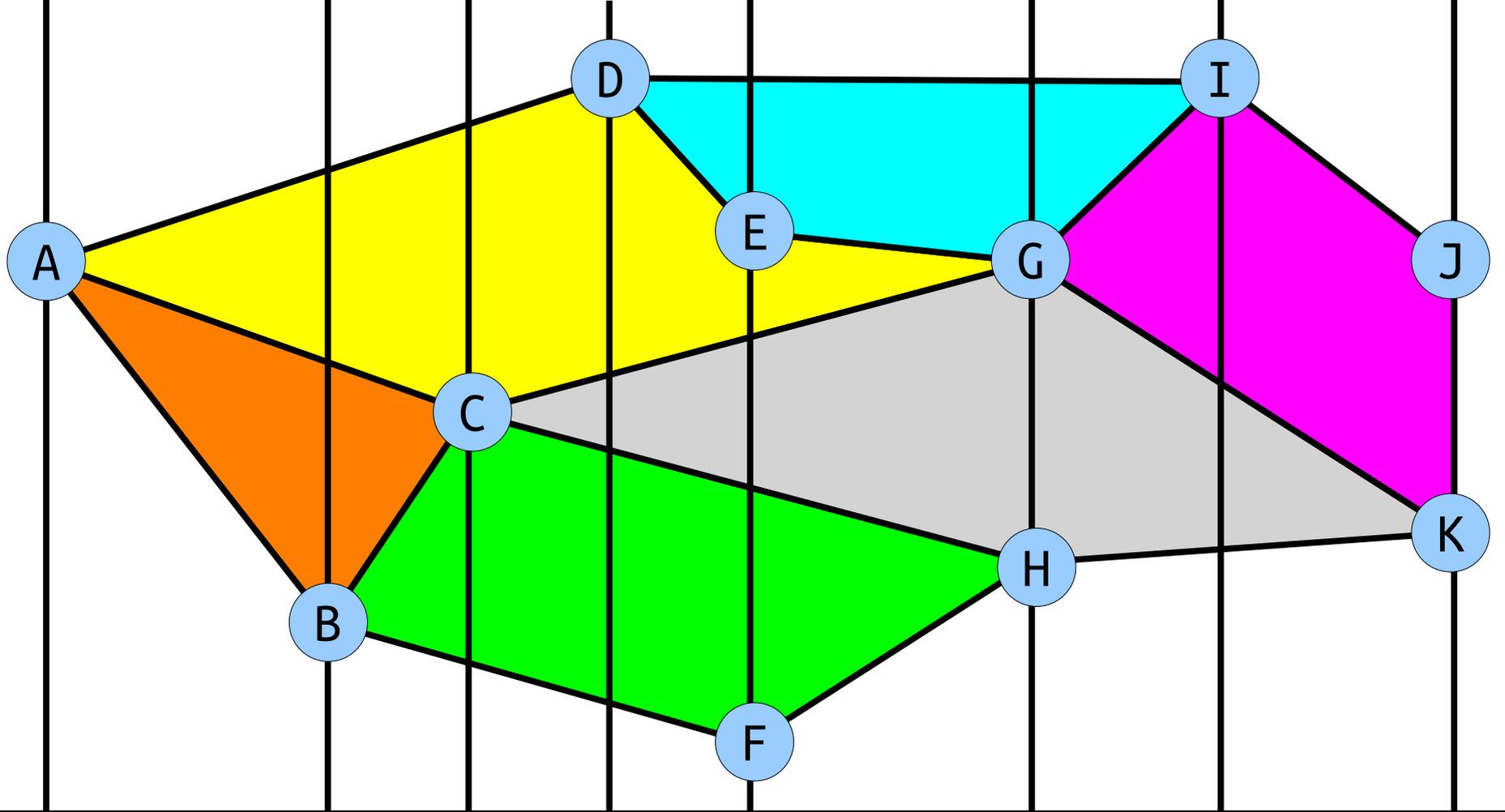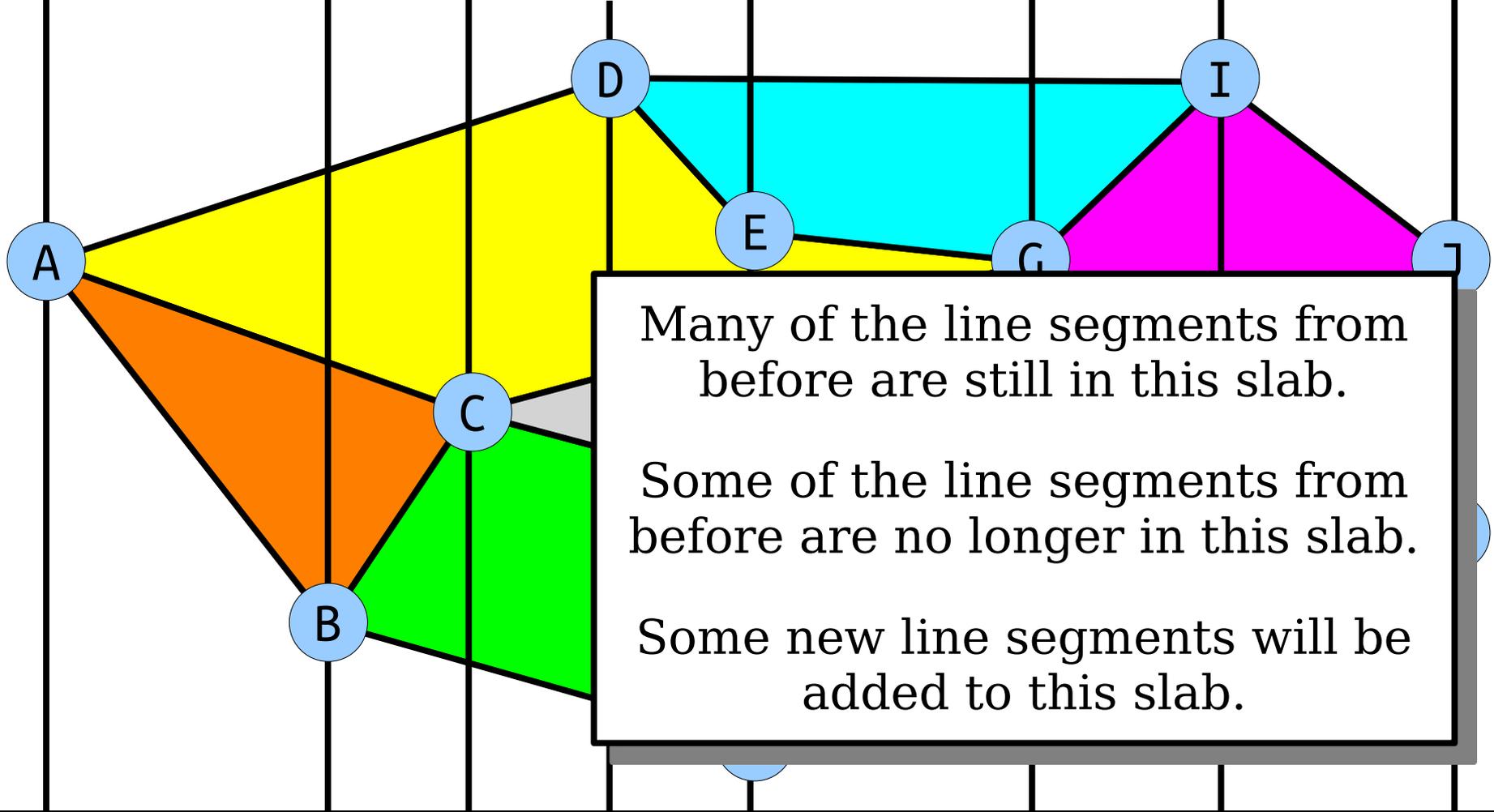
Slab 1	Slab 2	Slab 3
	AD	
	AC	
	AB	
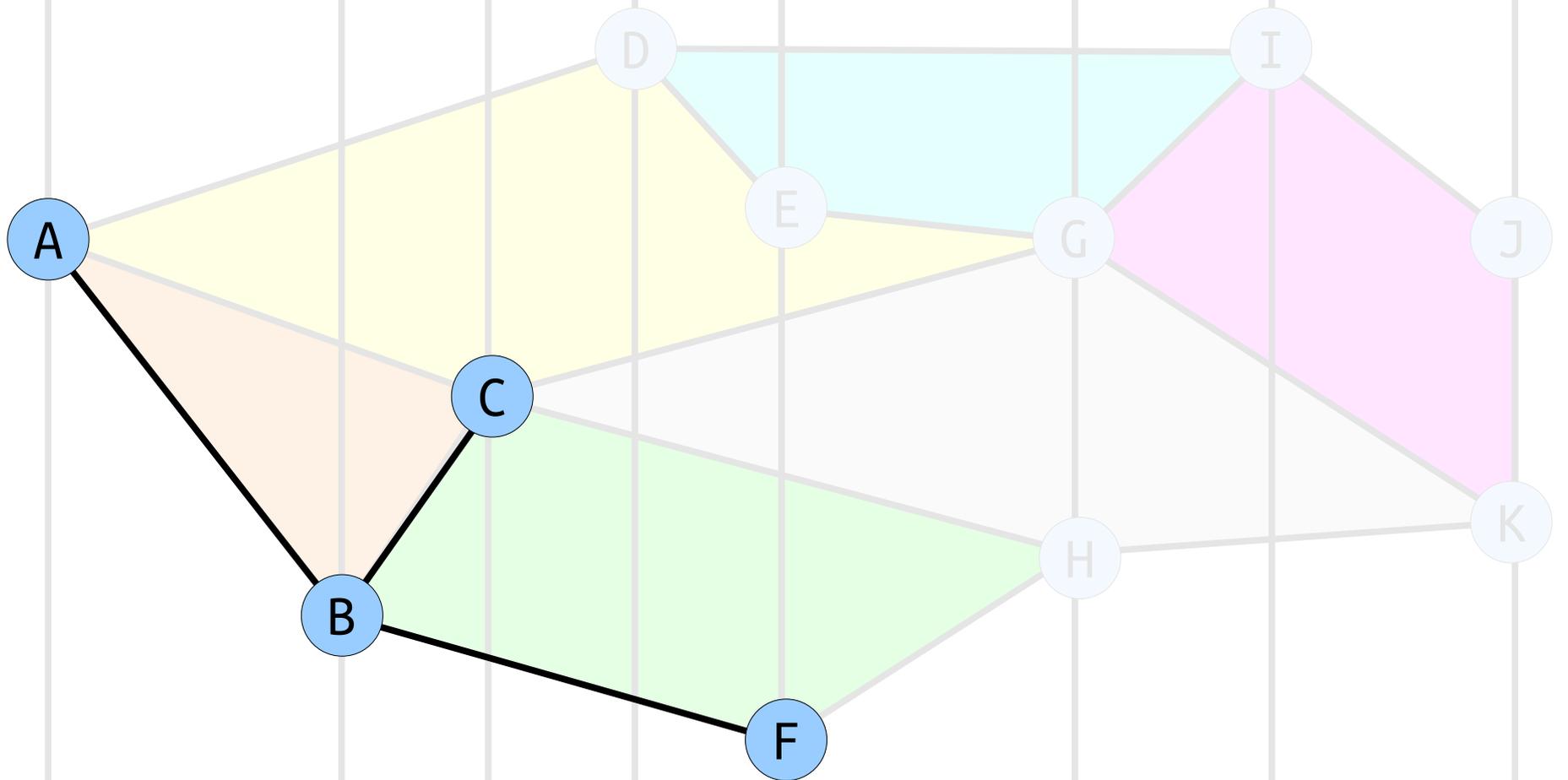
Slab 1	Slab 2	Slab 3
	AD	
	AC	
	AB	
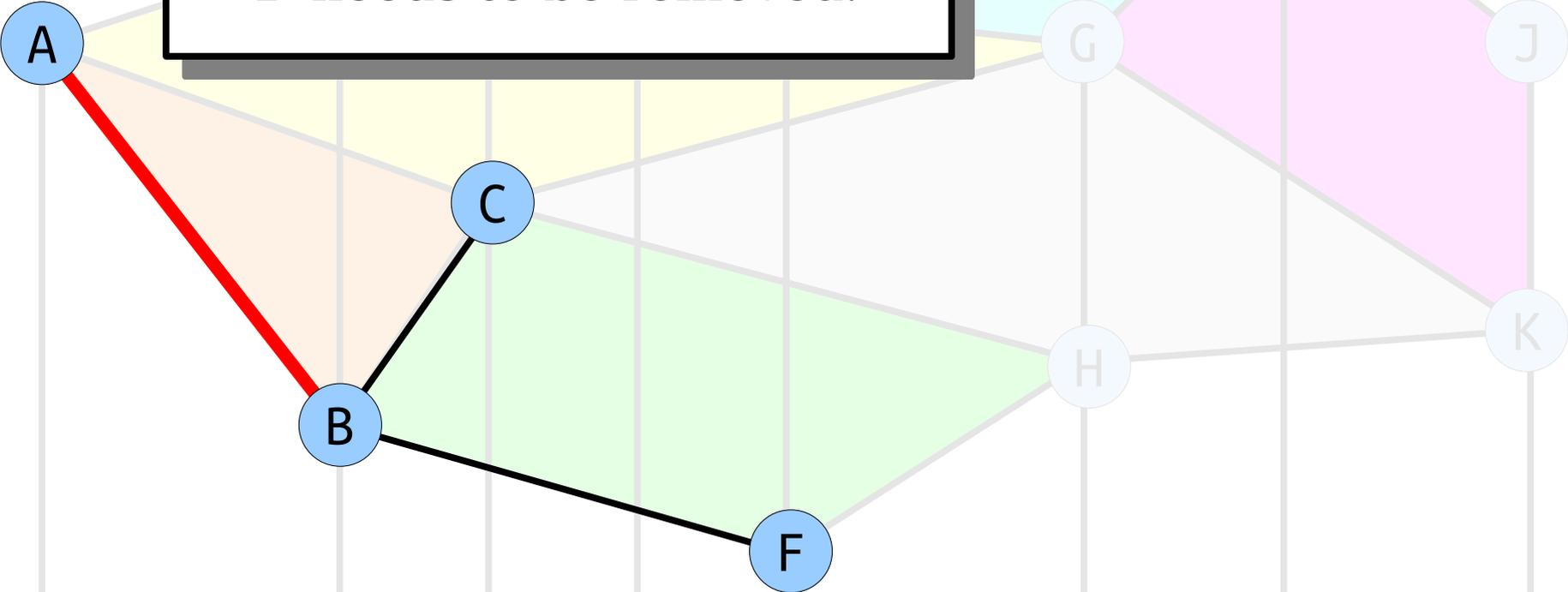
Any line segment ending in *B* needs to be removed.

Slab 1	Slab 2	Slab 3
	AD AC AB	

Any line segment ending in *B* needs to be removed.

A

C

B

F

Slab 1	Slab 2	Slab 3
	AD	AD
	AC	AC
	AB	AB

Any line segment ending in *B* needs to be removed.

A

C

B

F

Slab 1	Slab 2	Slab 3
	AD AC AB	AD AC

Any line segment ending in *B* needs to be removed.

Any line segment starting at *B* needs to be added in sorted order.

Slab 1	Slab 2	Slab 3
	AD AC AB	AD AC

Any line segment ending in B needs to be removed.

Any line segment starting at B needs to be added in sorted order.
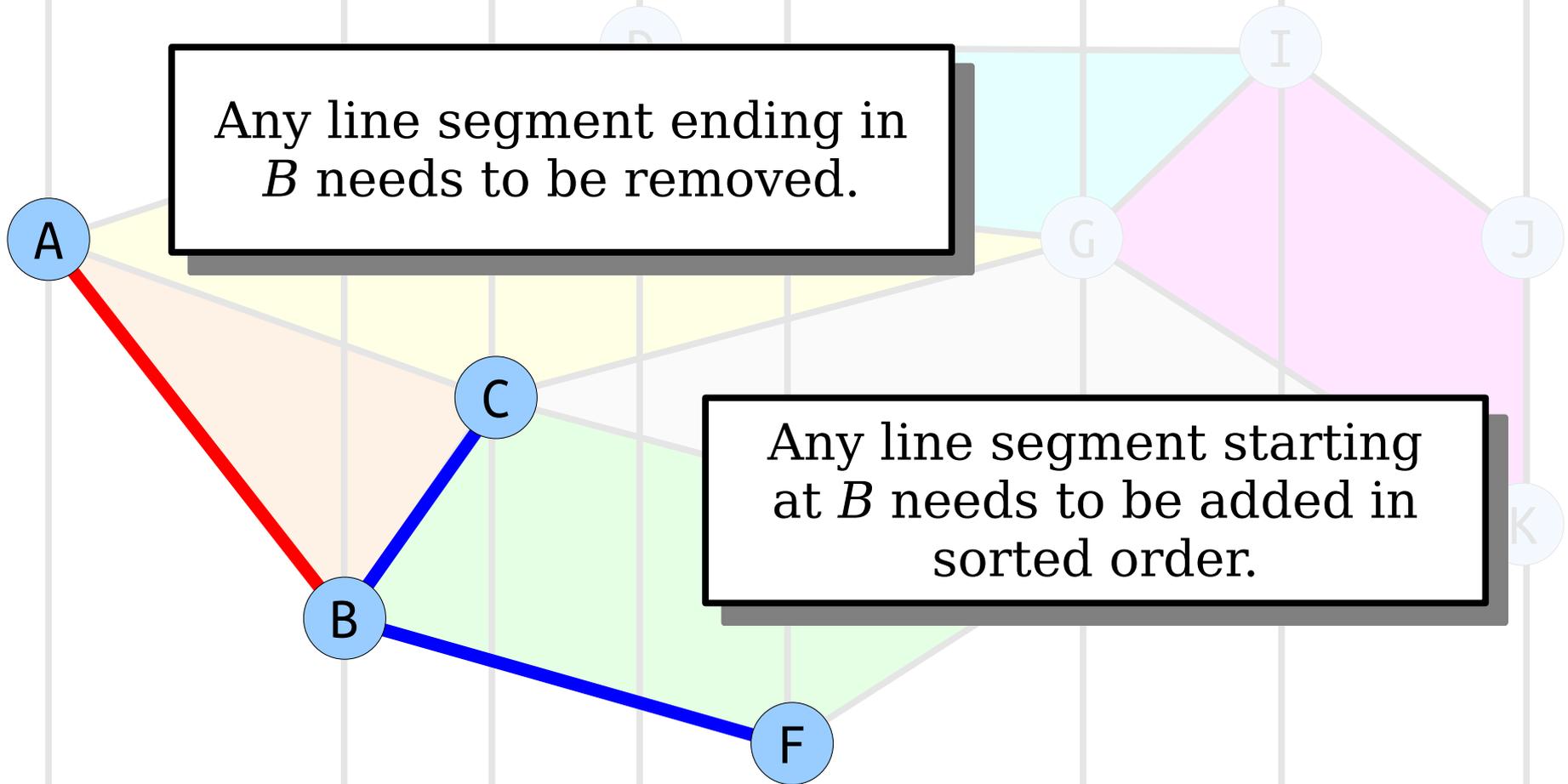
Slab 1	Slab 2	Slab 3
	AD	AD
	AC	AC
	AB	BC
		BF
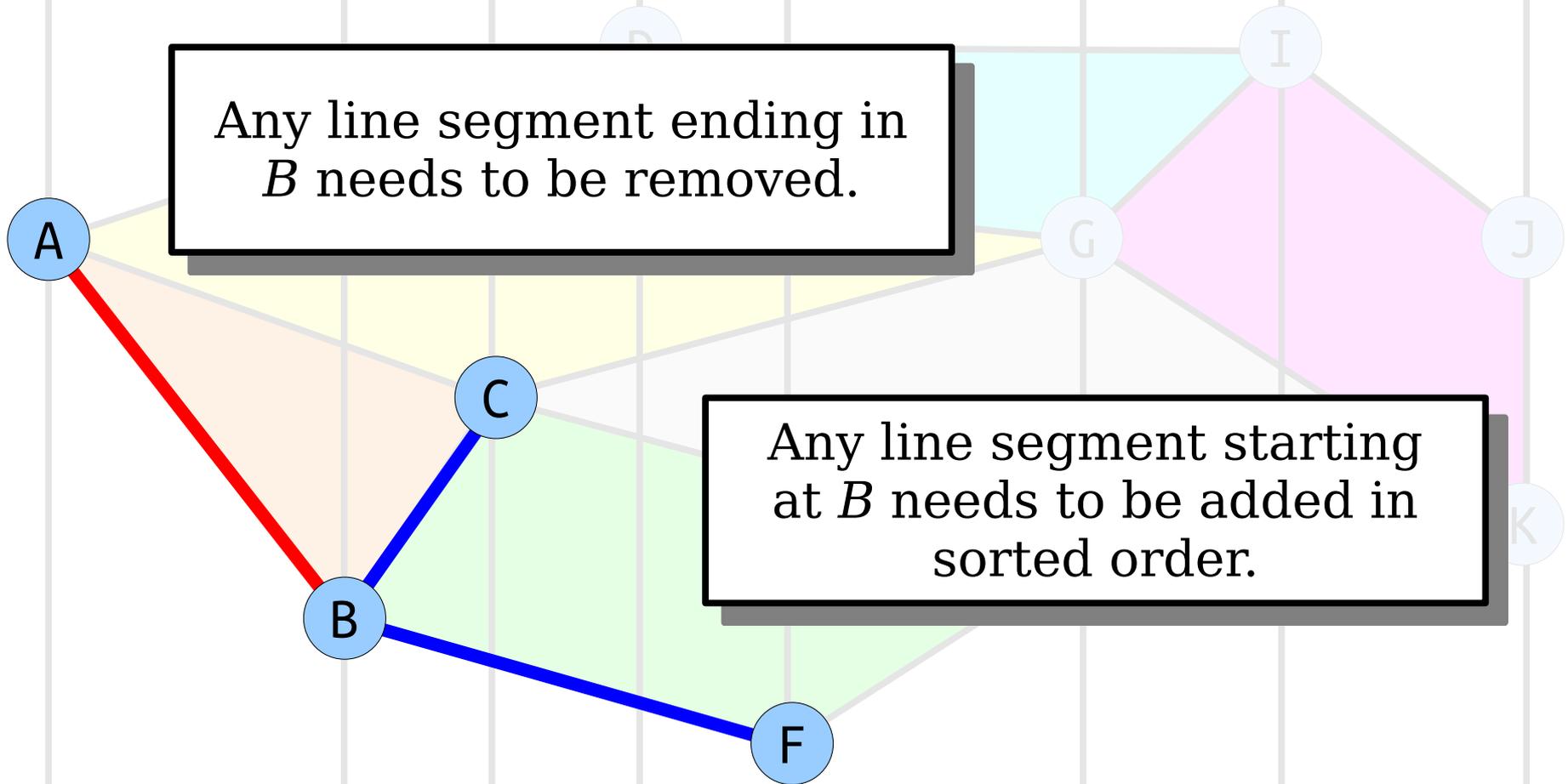
Slab 1	Slab 2	Slab 3
	AD AC AB	AD AC BC BF
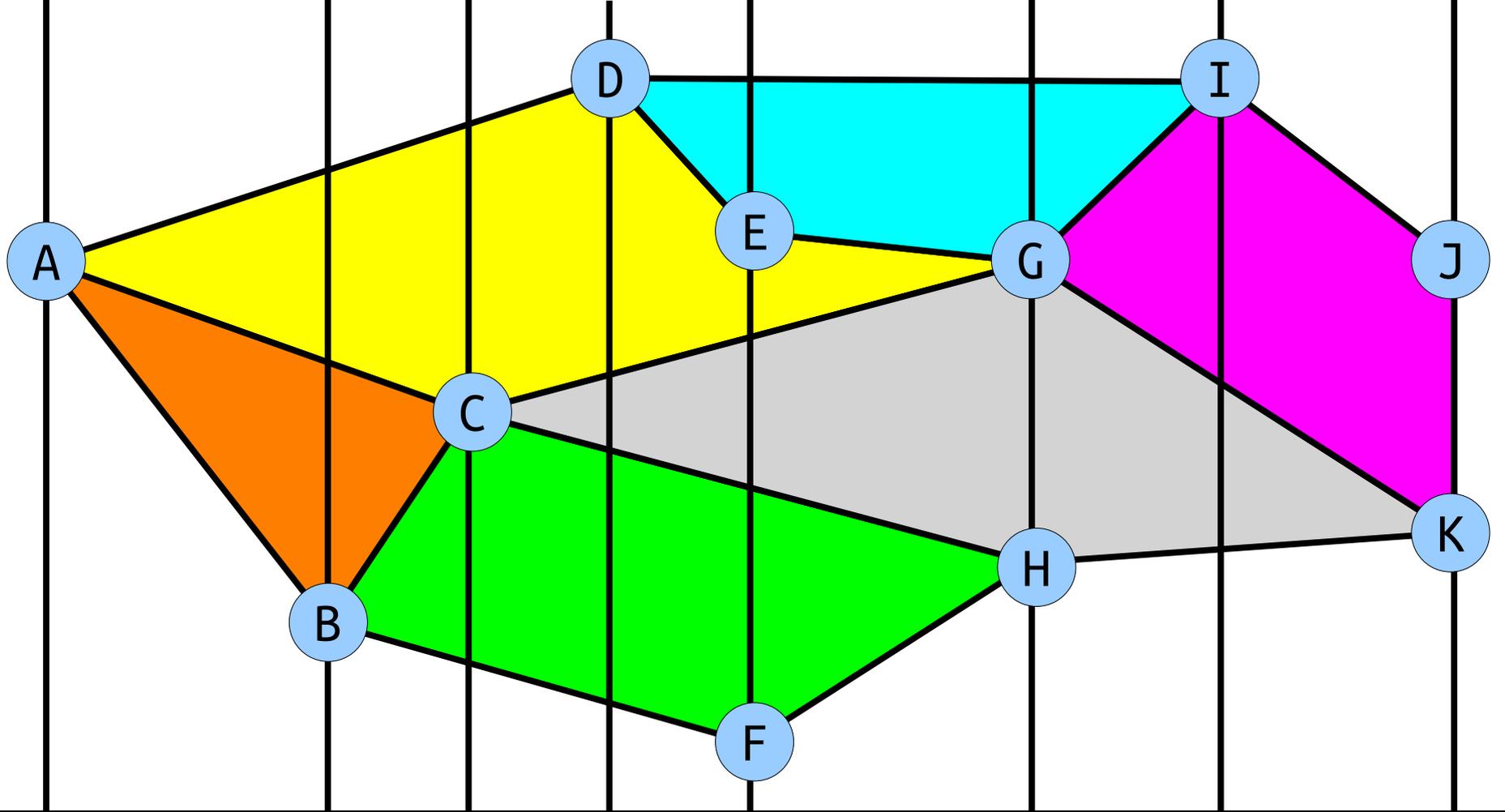
Slab 1	Slab 2	Slab 3	Slab 4
	AD AC AB	AD AC BC BF	
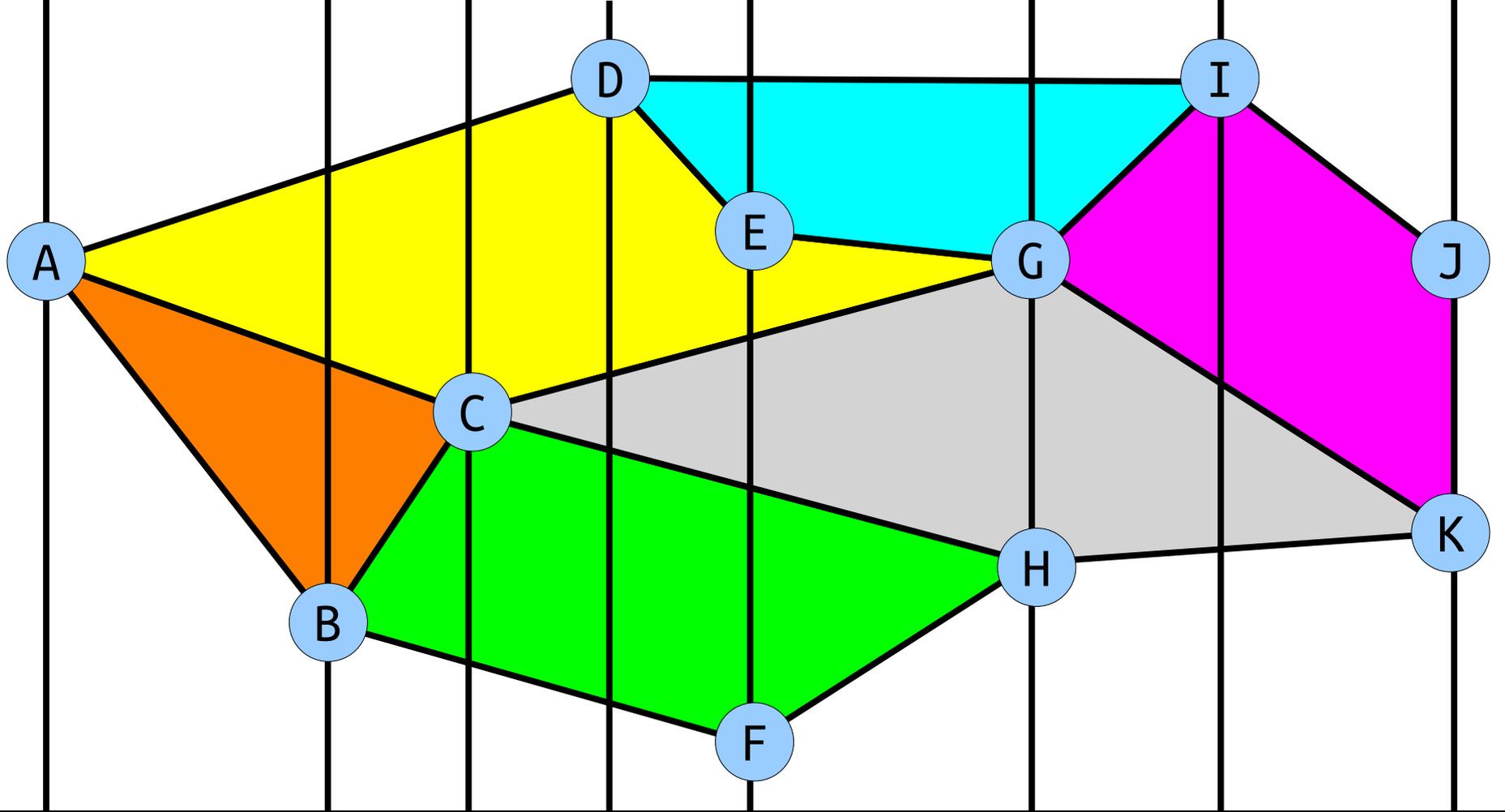
Slab 1	Slab 2	Slab 3	Slab 4
	AD AC AB	AD AC BC BF	

Copy the previous slab.

Slab 1	Slab 2	Slab 3	Slab 4
	AD AC AB	AD AC BC BF	AD AC BC BF
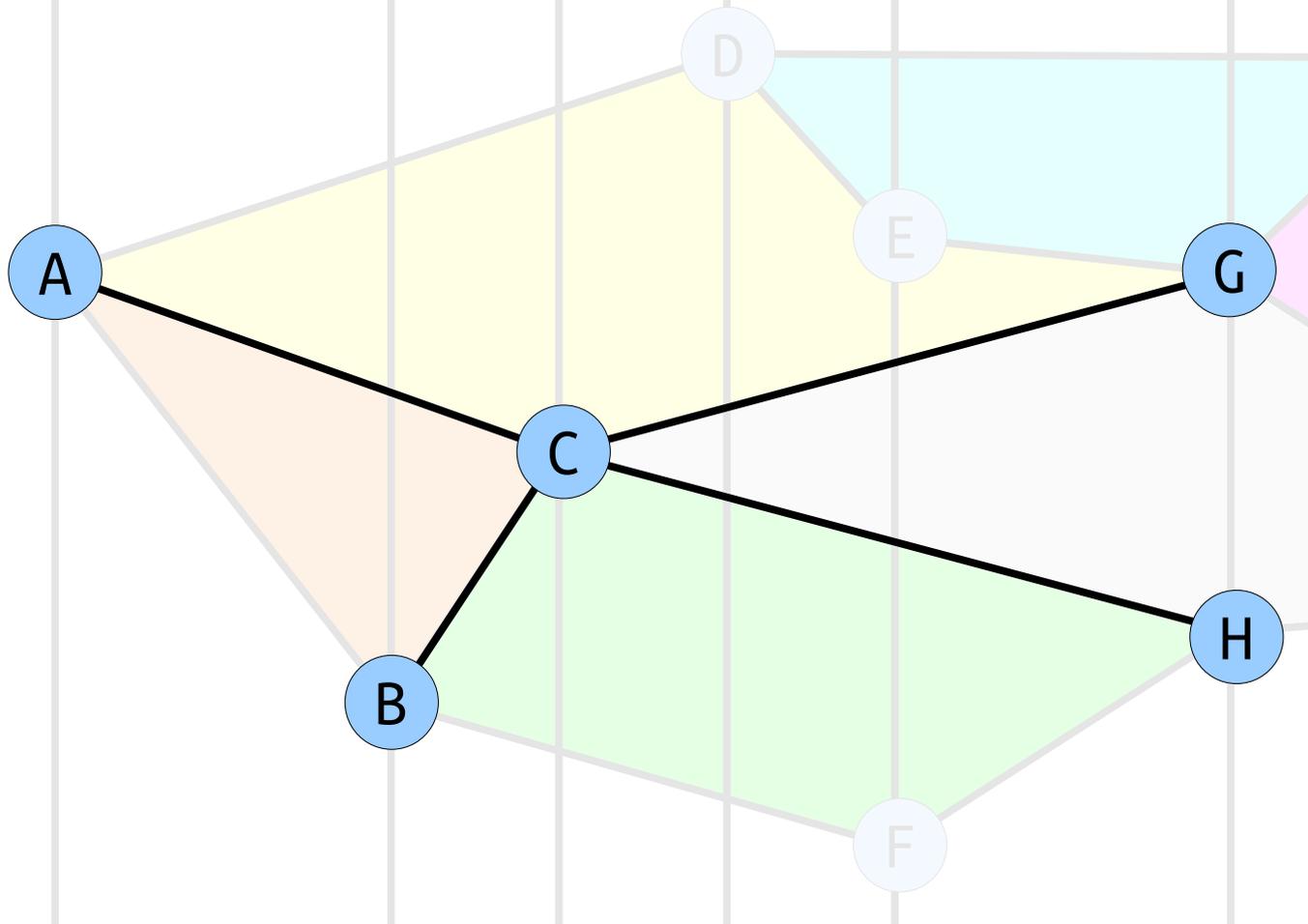
A B C D E F G H

Copy the previous slab.

Remove all segments ending at C.

Slab 1	Slab 2	Slab 3	Slab 4
	AD	AD	AD
	AC	AC	AC
	AB	BC	BC
		BF	BF

Copy the previous slab.

Remove all segments ending at C.

Slab 1	Slab 2	Slab 3	Slab 4
	AD AC AB	AD AC BC BF	AD AC BC BF

Copy the previous slab.

Remove all segments ending at C.

Slab 1	Slab 2	Slab 3	Slab 4
	AD	AD	
	AC	AC	AD
	AB	BC	BF
		BF	

Copy the previous slab.

Remove all segments ending at C.

Add all segments starting at C, in sorted order.

Slab 1	Slab 2	Slab 3	Slab 4
	AD AC AB	AD AC BC BF	AD BF

Copy the previous slab.

Remove all segments ending at C.
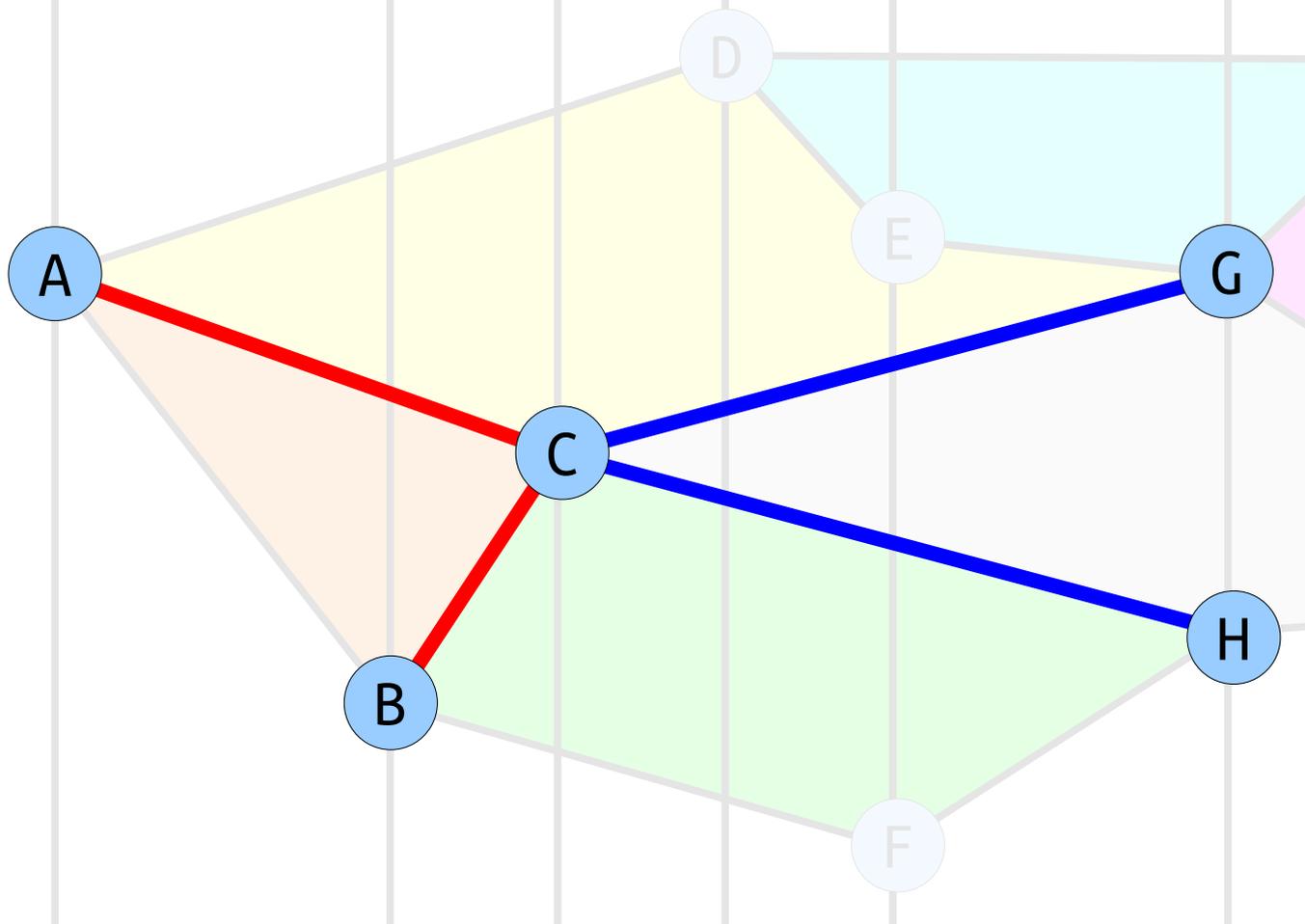
Add all segments starting at C, in sorted order.

Slab 1	Slab 2	Slab 3	Slab 4
	AD AC AB	AD AC BC BF	AD BF

Copy the previous slab.

Remove all segments ending at C.

Add all segments starting at C, in sorted order.

Slab 1	Slab 2	Slab 3	Slab 4
	AD	AD	AD
	AC	AC	CG
	AB	BC	CH
		BF	BF

Slab 1	Slab 2	Slab 3	Slab 4
	AD AC AB	AD AC BC BF	AD CG CH BF
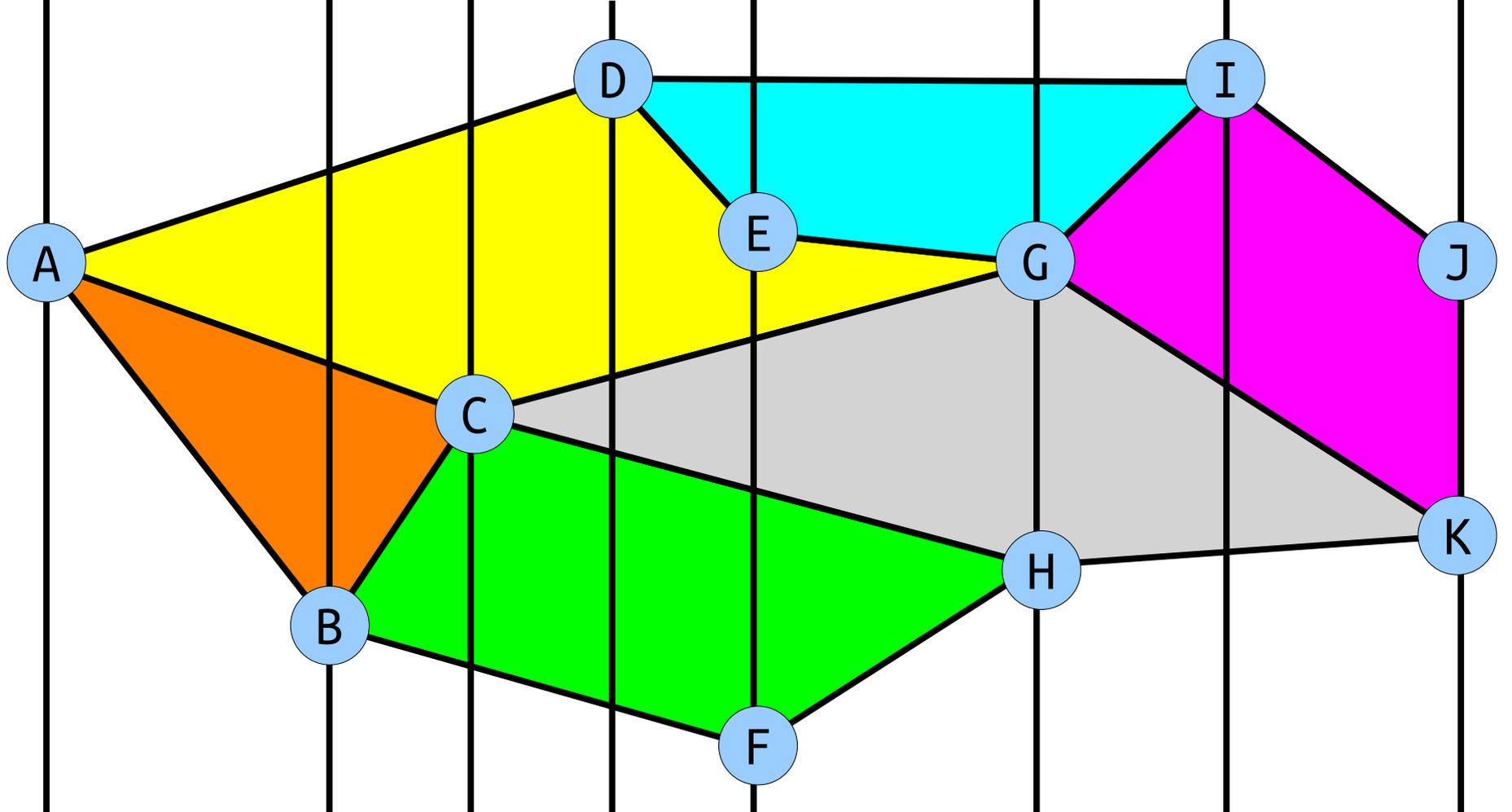
Slab 1	Slab 2	Slab 3	Slab 4	Slab 5	Slab 6	Slab 7	Slab 8	Slab 9
	AD AC AB	AD AC BC BF	AD CG CH BF	DI DE CG CH BF	DI EG CG CH FH	DI GI GK HK	IJ GK HK	

Building Slabs

- Sort vertices from left to right, grouping vertices with equal x coordinates together.

- Create an initial, empty slab.

- For each distinct x coordinate, from left to right:

  - Copy the previous slab.

  - Delete all segments that end at this x coordinate.

  - Add all segments that start at this x coordinate, keeping segments sorted from top to bottom.

- *Question:* How fast is this algorithm?

Observation: These vertical stripes cut line segments into *fragments* that run from the start of a slab to the end.

Each line segment appears in k slabs, where k is the number of fragments it's cut into.

Claim: There are families of planar subdivisions with $O(n)$ edges that produce $\Theta(n^2)$ fragments, and this is a tight upper bound.

Slab 1	Slab 2	Slab 3	Slab 4	Slab 5	Slab 6	Slab 7	Slab 8	Slab 9	Slab 10	Slab 11
	AX AB	AX BX BC	AX BX CX CD	AX BX CX DX DE	AX BX CX DX EX EF	AX BX CX DX EX FX FG	AX BX CX DX EX FX GX GH	AX BX CX DX EX FX GX HX HI	AX BX CX DX EX FX GX HX IX IJ	

Slabs: The Drawback

- In the worst case, a slab decomposition requires $\Theta(n^2)$ space.

- If you're clever with your preprocessing, you can construct a slab decomposition in time $O(n \log n) + O(S)$, where S is the number of fragments.

- ***Question:*** Can we do better?

	Preprocessing Time	Query Time	Space Usage
Test All Faces	$O(n \log n)$	$O(n)$	$O(n)$
Slab Decomposition	$O(n^2)$	$O(\log n)$	$O(n^2)$

Slabs: The Drawback

- In the worst case, a slab decomposition requires $\Theta(n^2)$ space.

- If you're clever with your preprocessing, you can construct a slab decomposition in time $O(n \log n) + O(S)$, where S is the number of fragments.

- **_Question:_** Can we do better?

	Preprocessing Time	Query Time	Space Usage
Test All Faces	$O(n \log n)$	$O(n)$	$O(n)$
Slab Decomposition	$O(n^2)$	$O(\log n)$	$O(n^2)$

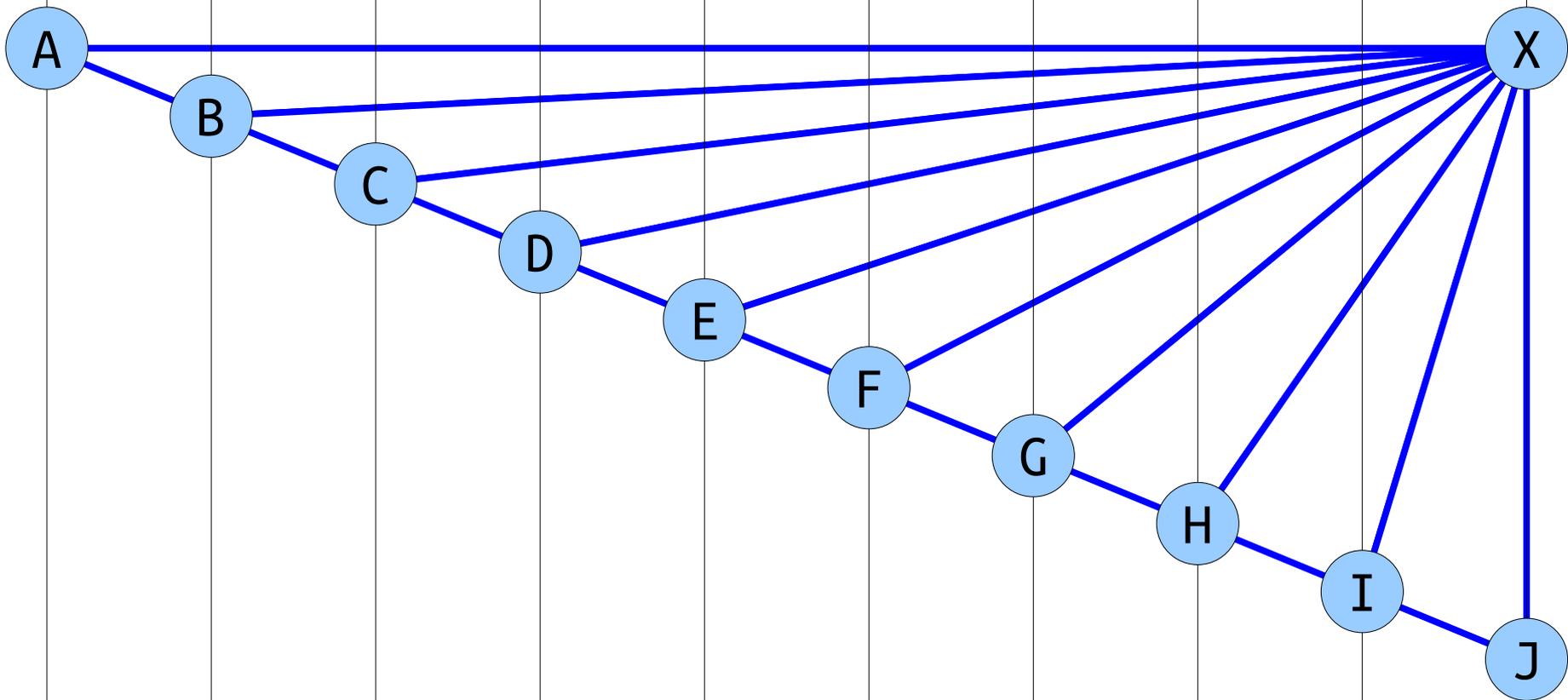| Slab 1	Slab 2	Slab 3	Slab 4	Slab 5	Slab 6	Slab 7	Slab 8	Slab 9	Slab 10	Slab 11
	AX AB	AX BX BC	AX BX CX CD	AX BX CX DX DE	AX BX CX DX EX EF	AX BX CX DX EX FX FG	AX BX CX DX EX FX GX GH	AX BX CX DX EX FX GX HX HI	AX BX CX DX EX FX GX HX IX IJ	

Slab 1	Slab 2	Slab 3	Slab 4	Slab 5	Slab 6	Slab 7	Slab 8	Slab 9	Slab 10	Slab 11
	Add AX Add AB	Add BX Add BC Del AB	Add CX Add CD Del BC	Add DX Add DE Del CD	Add EX Add EF Del DE	Add FX Add FG Del EF	Add GX Add GH Del FG	Add HX Add HI Del GH	Add IX Add IJ Del HI	Del AX Del BX ... Del JX

Slab 1	Slab 2	Slab 3	Slab 4	Slab 5	Slab 6	Slab 7	Slab 8	Slab 9	Slab 10	Slab 11
	Add AX Add AB	Add BX Add BC Del AB	Add CX Add CD Del BC	Add DX Add DE Del CD	Add EX Add EF Del DE	Add FX Add FG Del EF	Add GX Add GH Del FG	Add HX Add HI Del GH	Add IX Add IJ Del HI	Del AX Del BX ... Del JX

Saving Space

- Each segment is first added in some slab and later removed from a slab.

- Total number of edits made across all slabs: O(n).

- **_Question:_** Can we use O(n) storage for our slabs by just encoding the deltas?

| Slab 1	Slab 2	Slab 3	Slab 4	Slab 5	Slab 6	Slab 7	Slab 8	Slab 9	Slab 10	Slab 11
	Add AX Add AB	Add BX Add BC Del AB	Add CX Add CD Del BC	Add DX Add DE Del CD	Add EX Add EF Del DE	Add FX Add FG Del EF	Add GX Add GH Del FG	Add HX Add HI Del GH	Add IX Add IJ Del HI	Del AX Del BX ... Del JX

Saving Space

- Just storing the deltas themselves breaks our O(log *n*) query time – how can you binary search over a group of segments if you don't know which ones are there?

- We're going to need to change our approach.

| Slab 1	Slab 2	Slab 3	Slab 4	Slab 5	Slab 6	Slab 7	Slab 8	Slab 9	Slab 10	Slab 11

	`Add AX`
	`Add AB`
		`Add BX`
		`Add BC`
		`Del AB`
			`Add CX`
			`Add CD`
			`Del BC`
				`Add DX`
				`Add DE`
				`Del CD`
					`Add EX`
					`Add EF`
					`Del DE`
						`Add FX`
						`Add FG`
						`Del EF`
							`Add GX`
							`Add GH`
							`Del FG`
								`Add HX`
								`Add HI`
								`Del GH`
									`Add IX`
									`Add IJ`
									`Del HI`
										`Del AX`
										`Del BX`
										`…`
										`Del JX`

Abstracting Away

- We need to store several collections of sorted sequences.

- Each sequence is formed by making some number of edits to the previous one.

- We want to do so with space usage proportional to the number of edits.
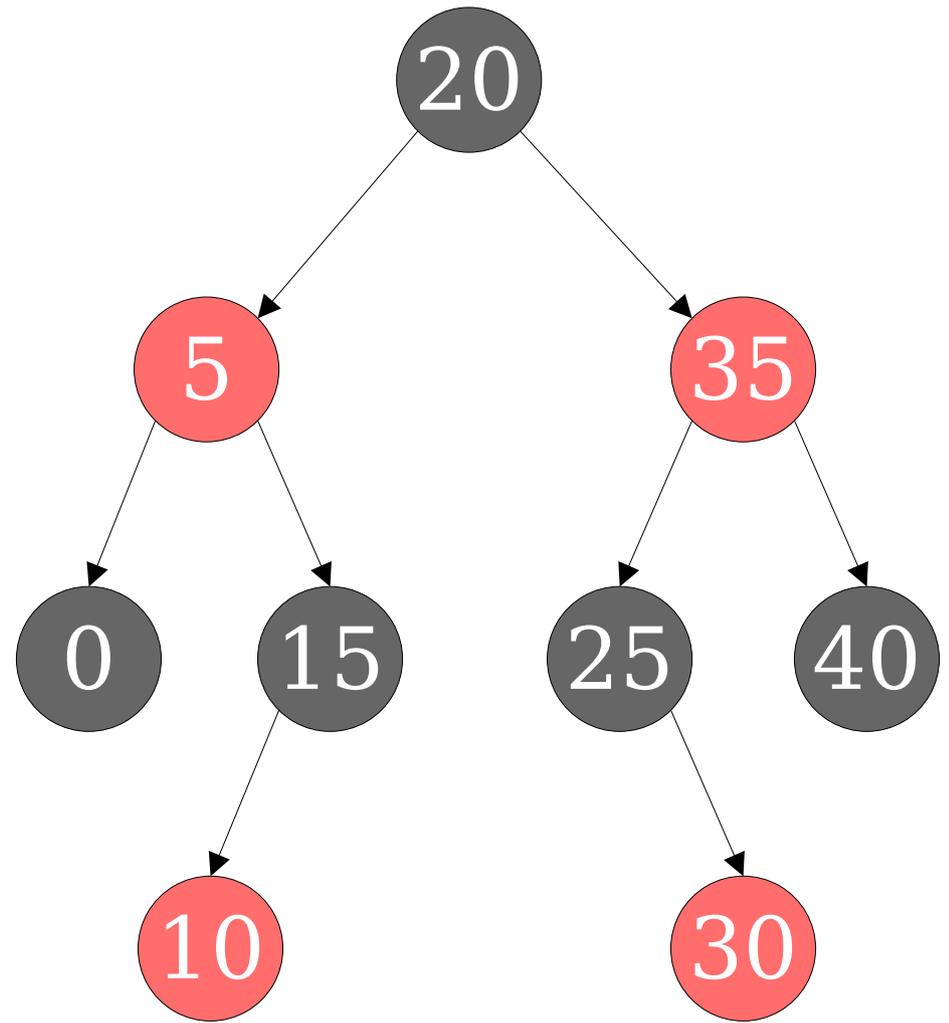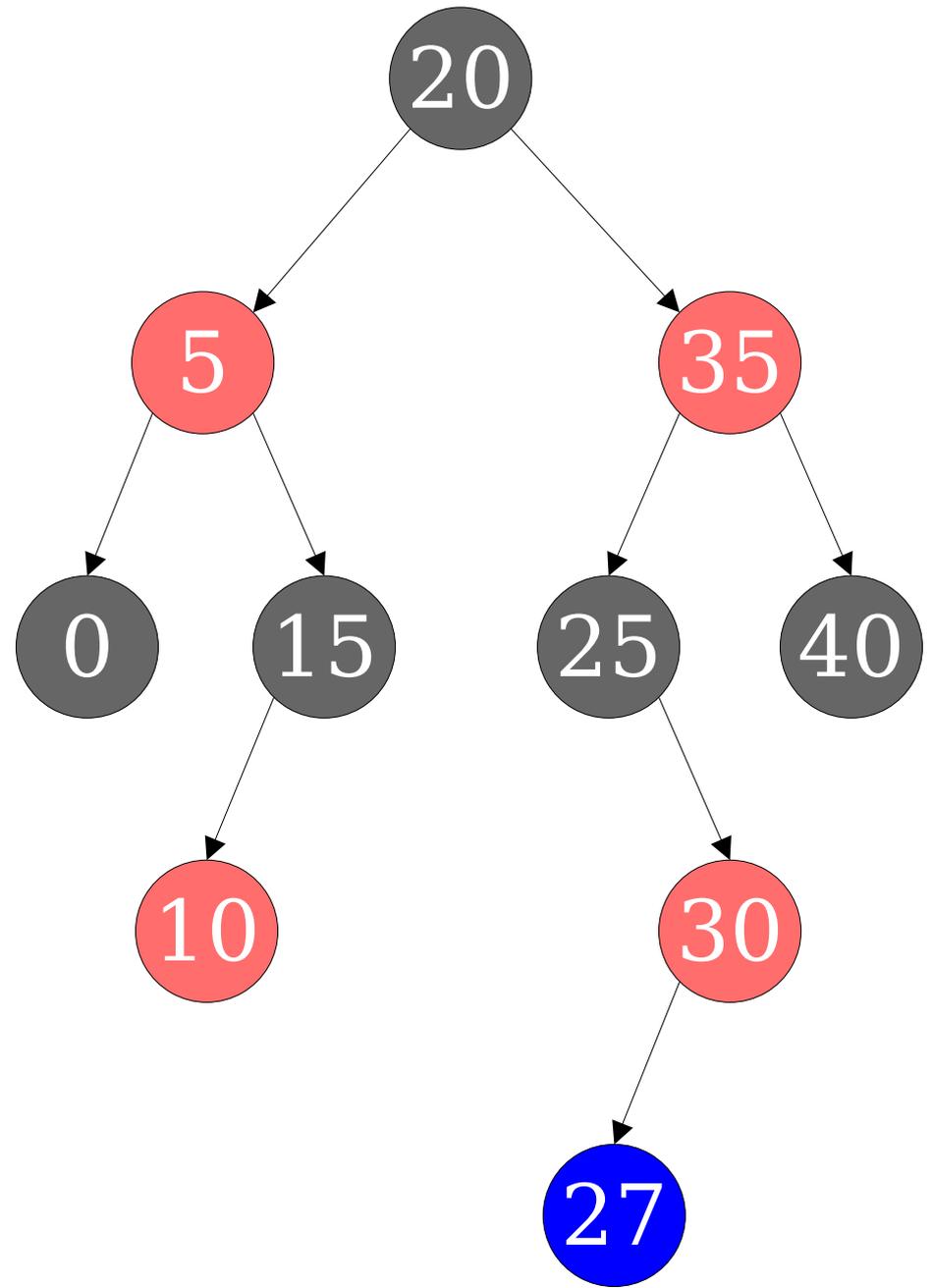
- *Question:* How do we do this?
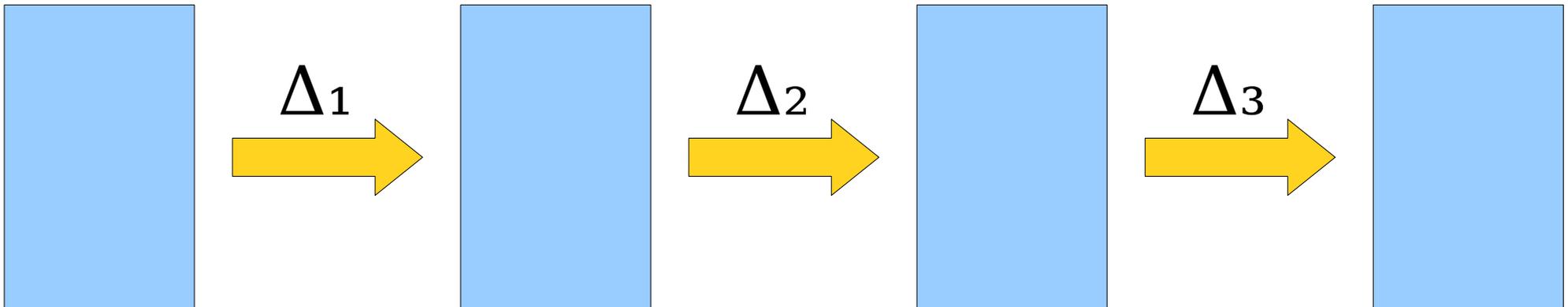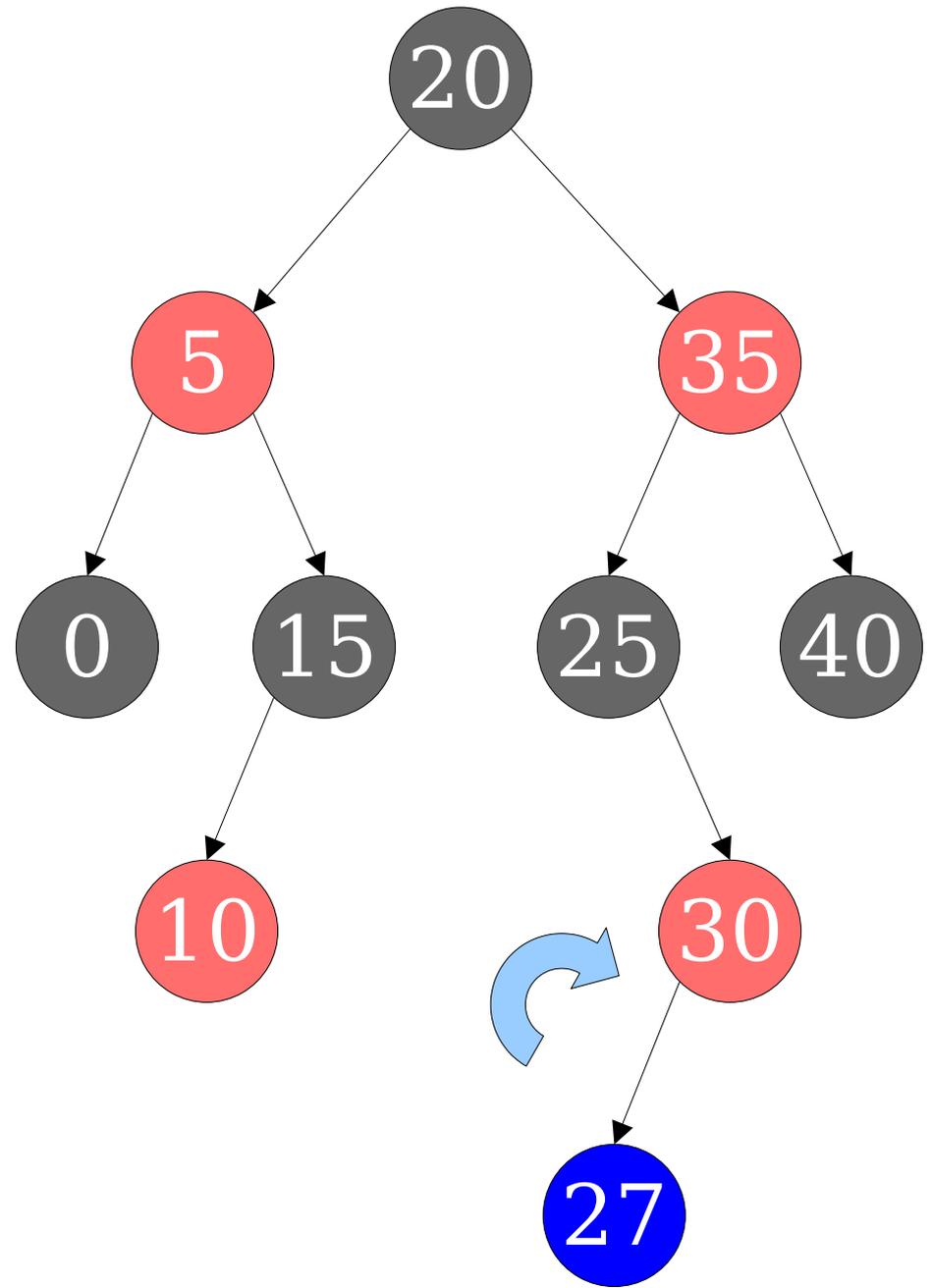
Abstracting Away

- **Reasonable Guess:** Store our sorted sequences as red/black trees to support fast edits (insertions and removals).

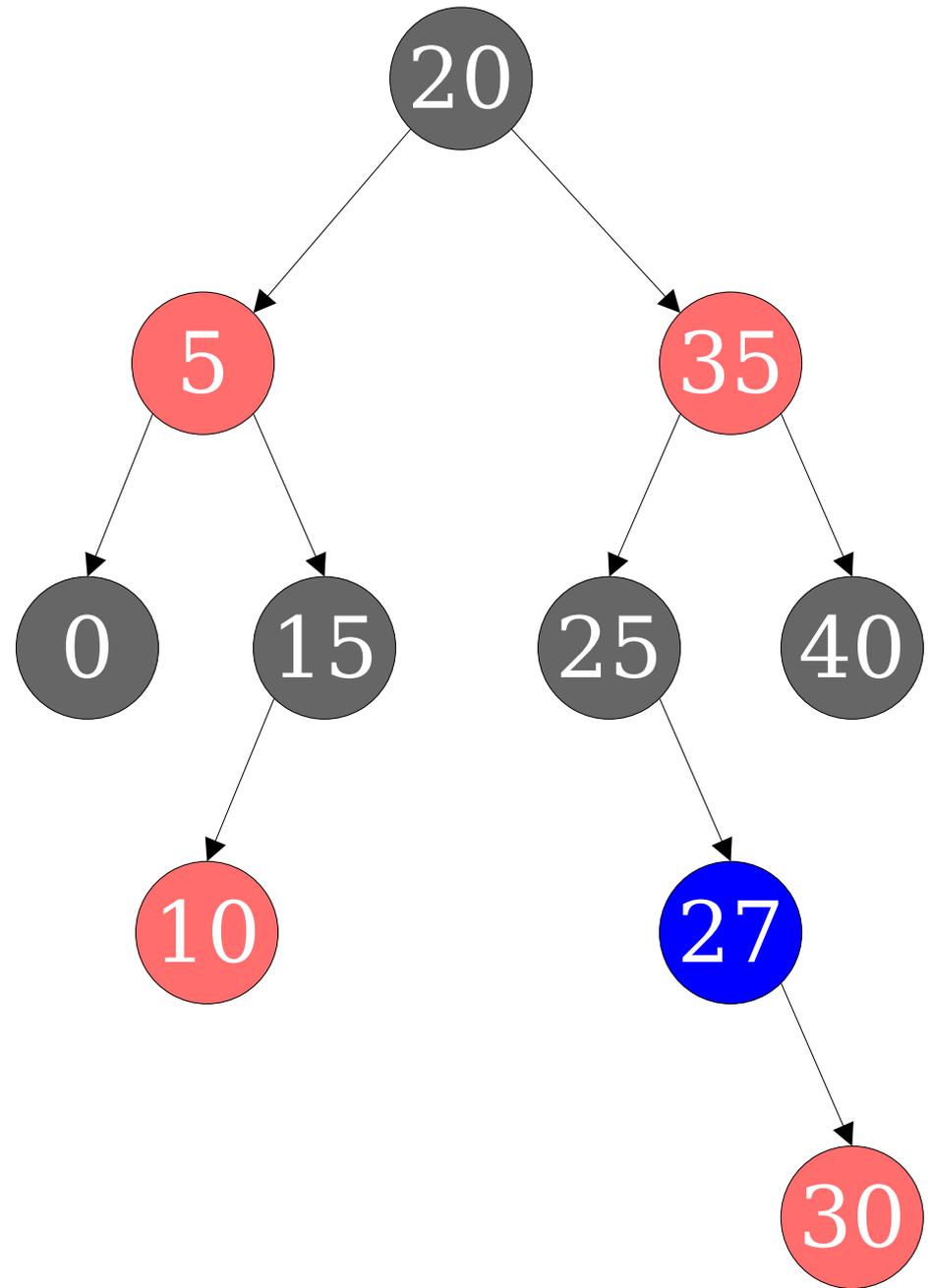- We're now left with the following goal:

  Modify a red/black tree so that, after making changes, you're left with two trees: the tree you started with, plus the resulting tree.
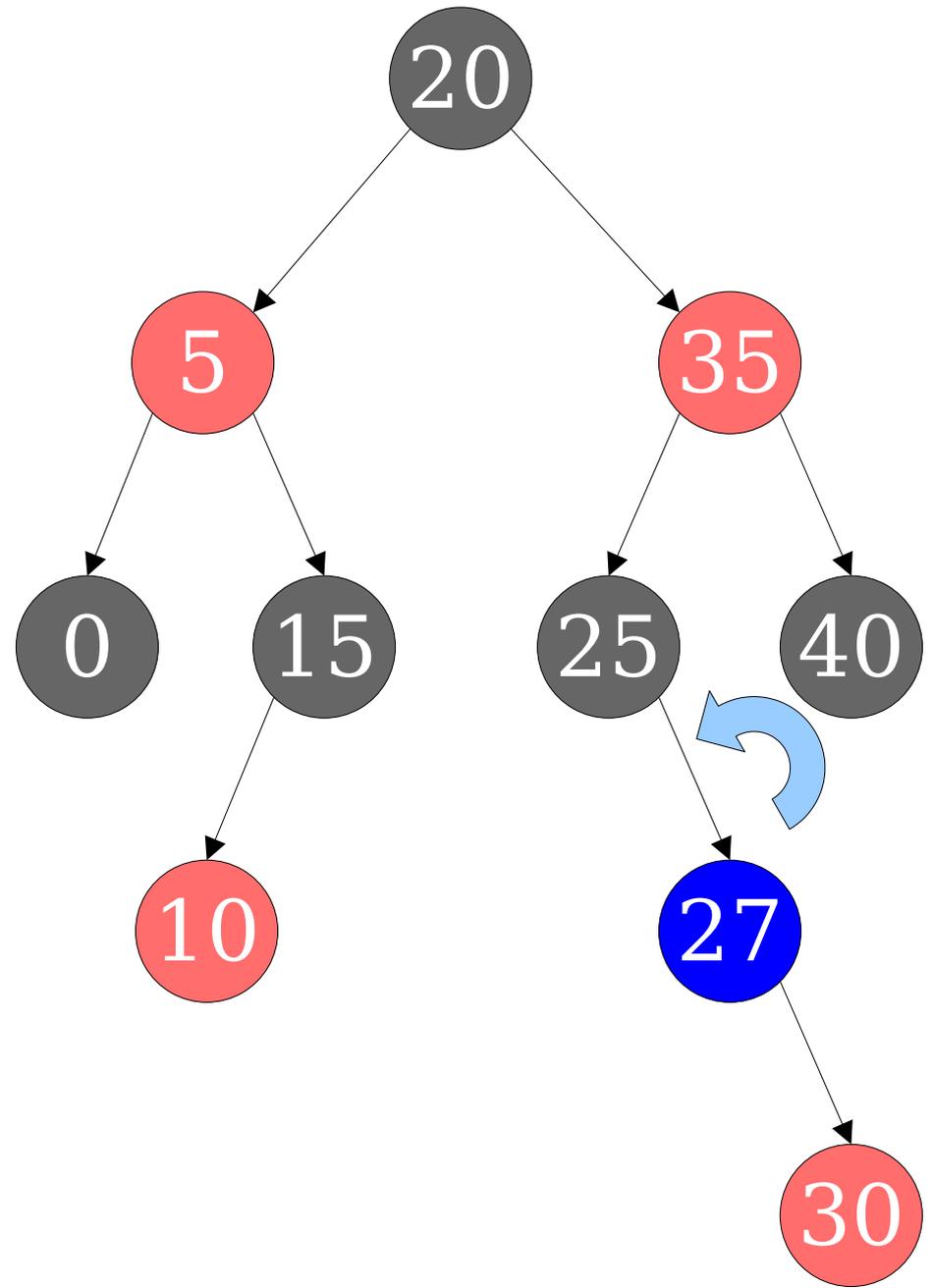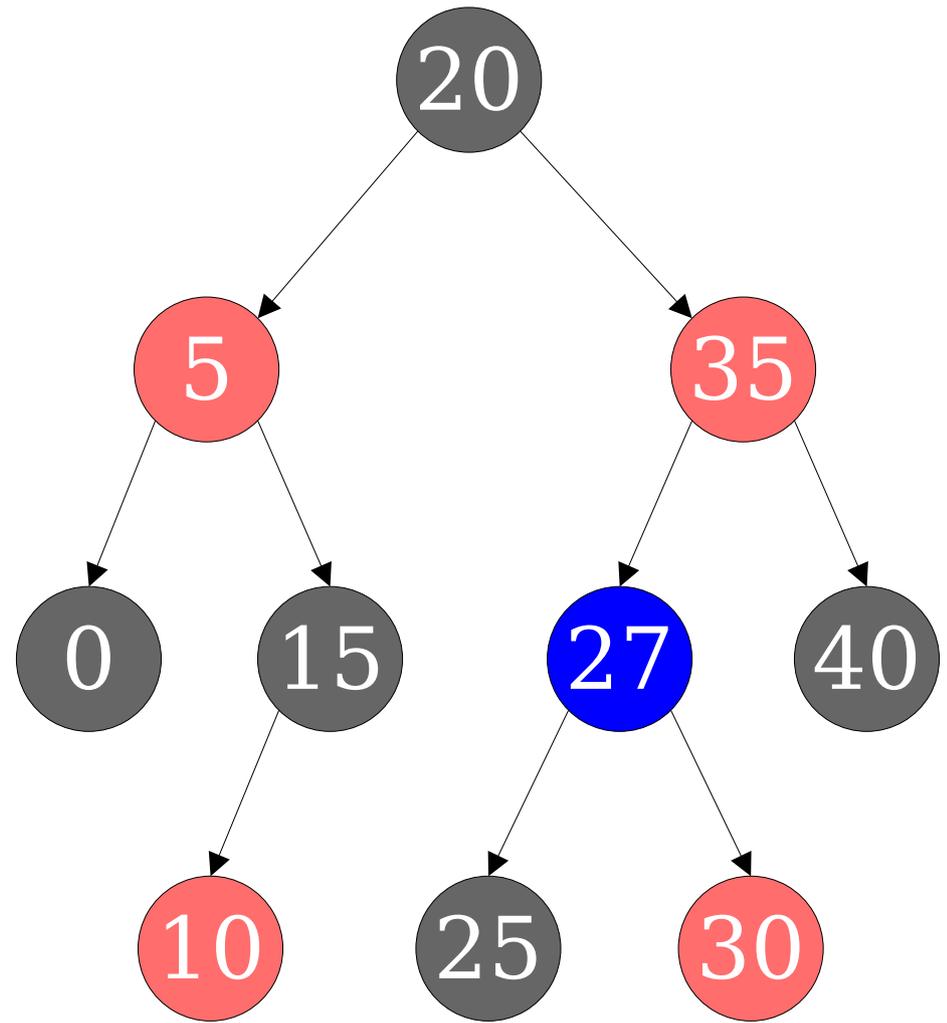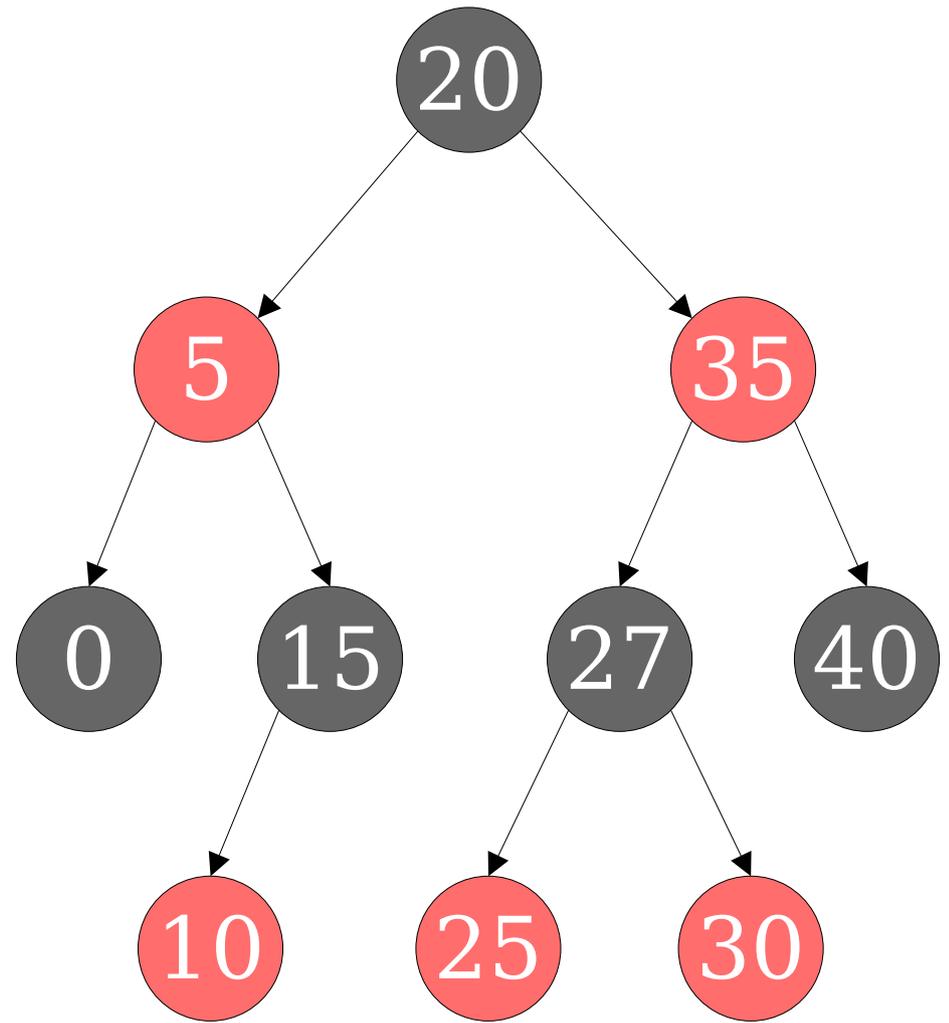
$\Delta_1 \qquad \Delta_2 \qquad \Delta_3$
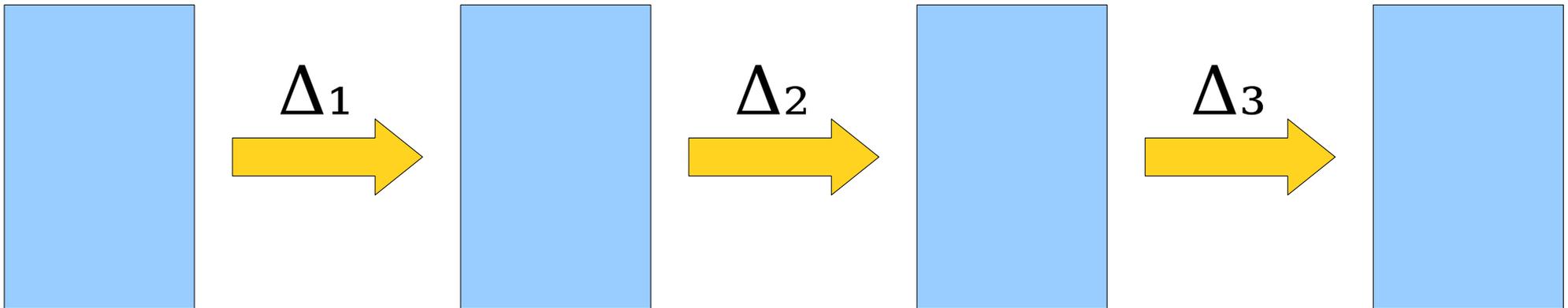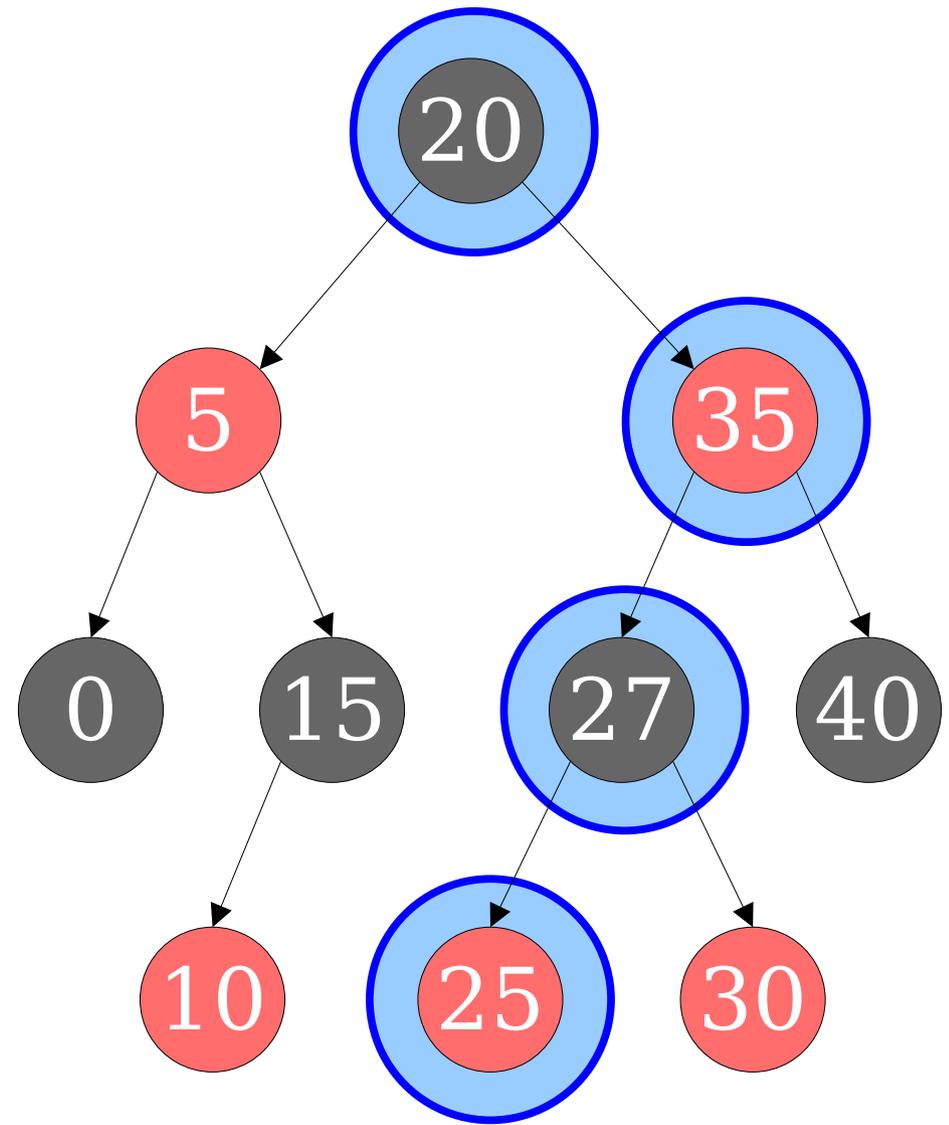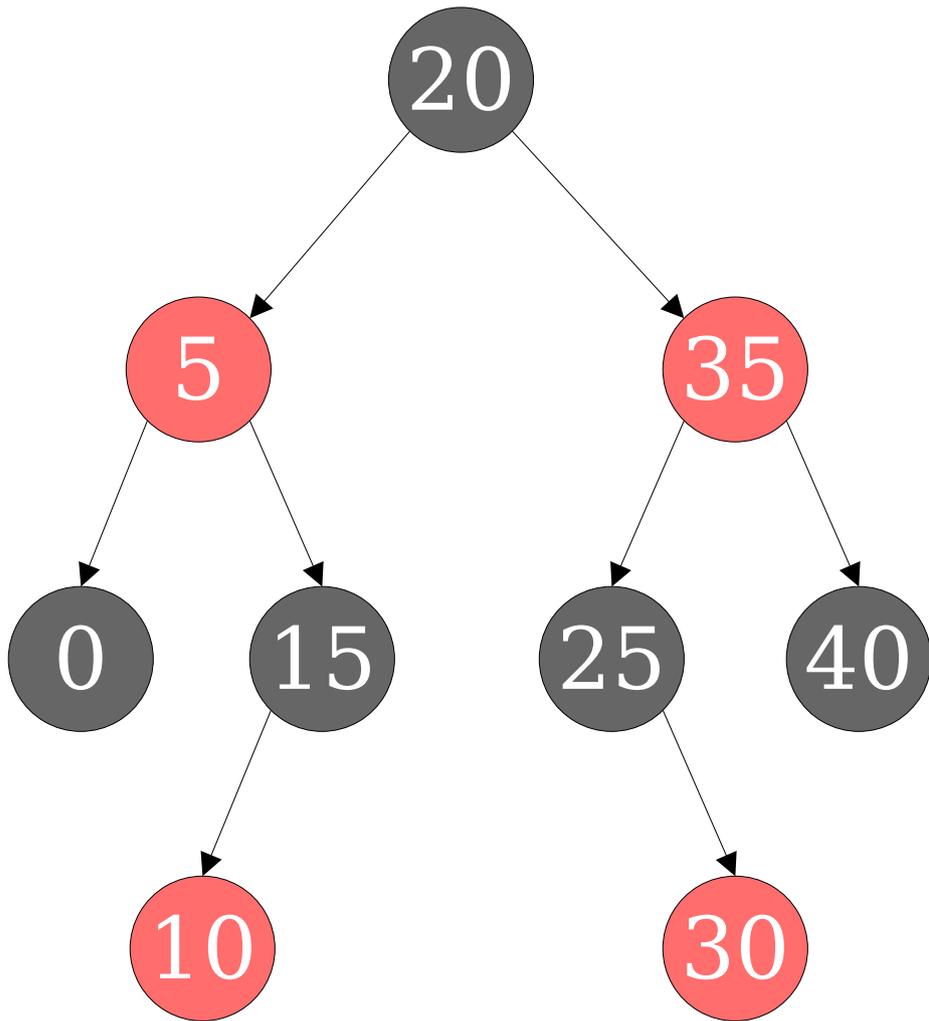
Which nodes in the red/black tree can see the changes that were just made?
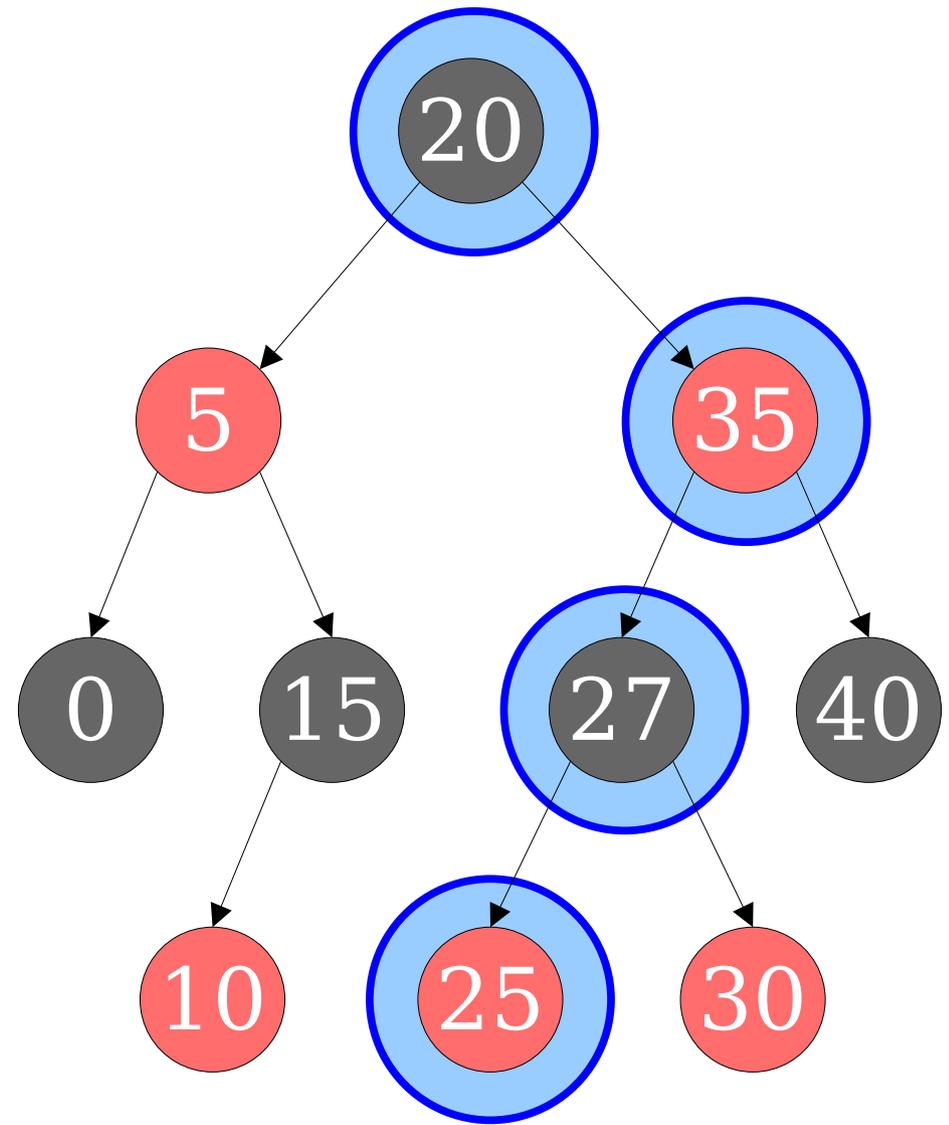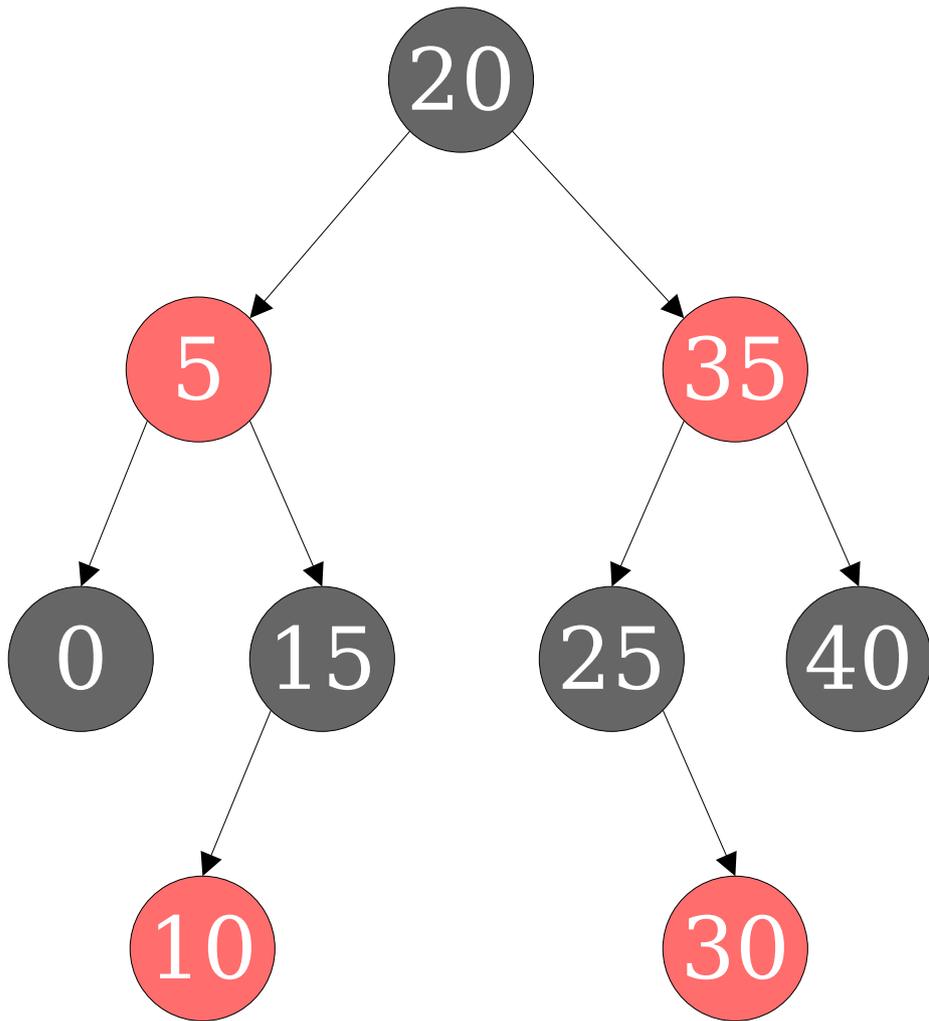
Idea: Keep the old tree around. Clone nodes that can see a part of the tree that changed.
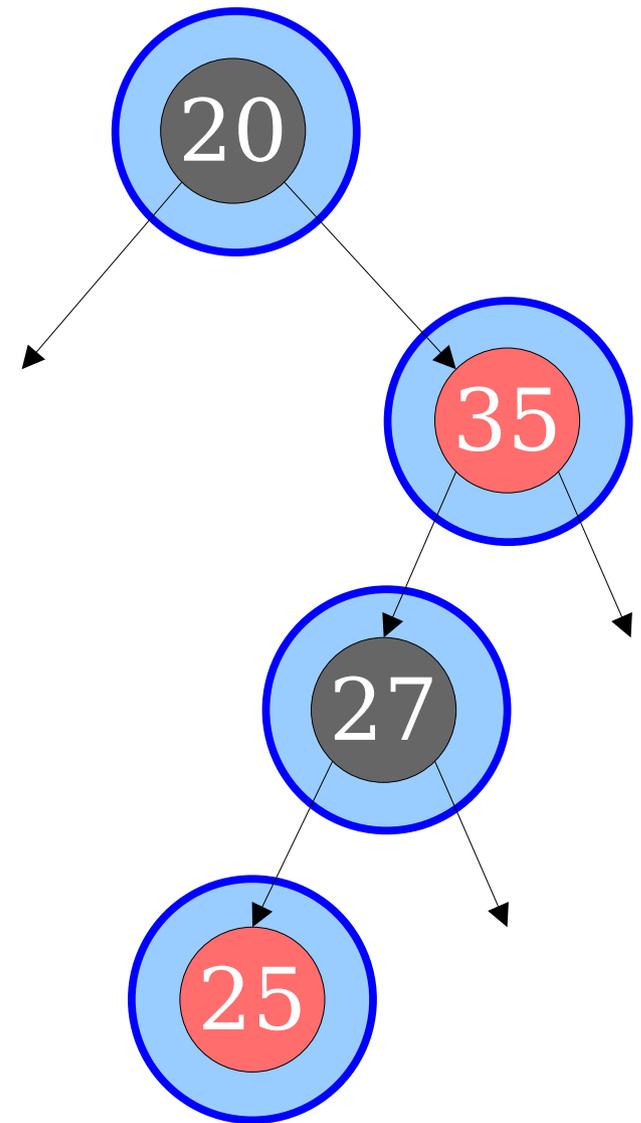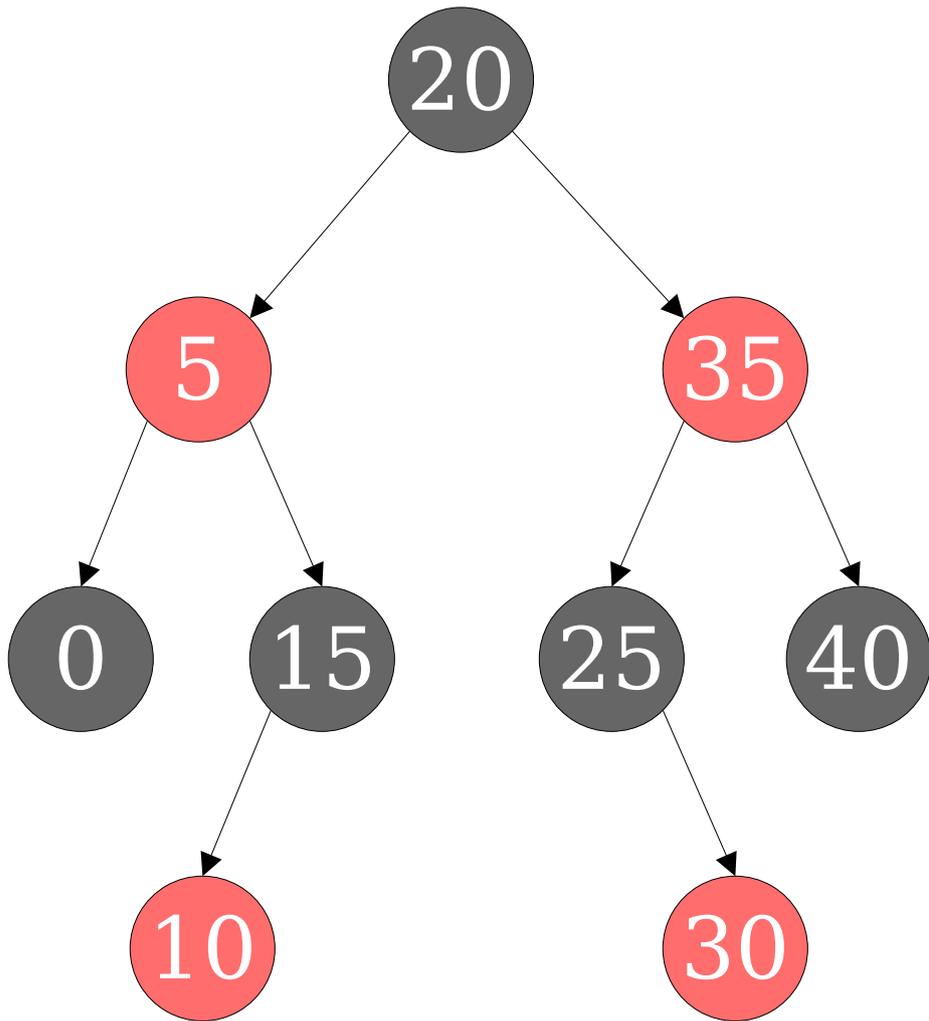
Idea: Keep the old tree around. Clone nodes that can see a part of the tree that changed.
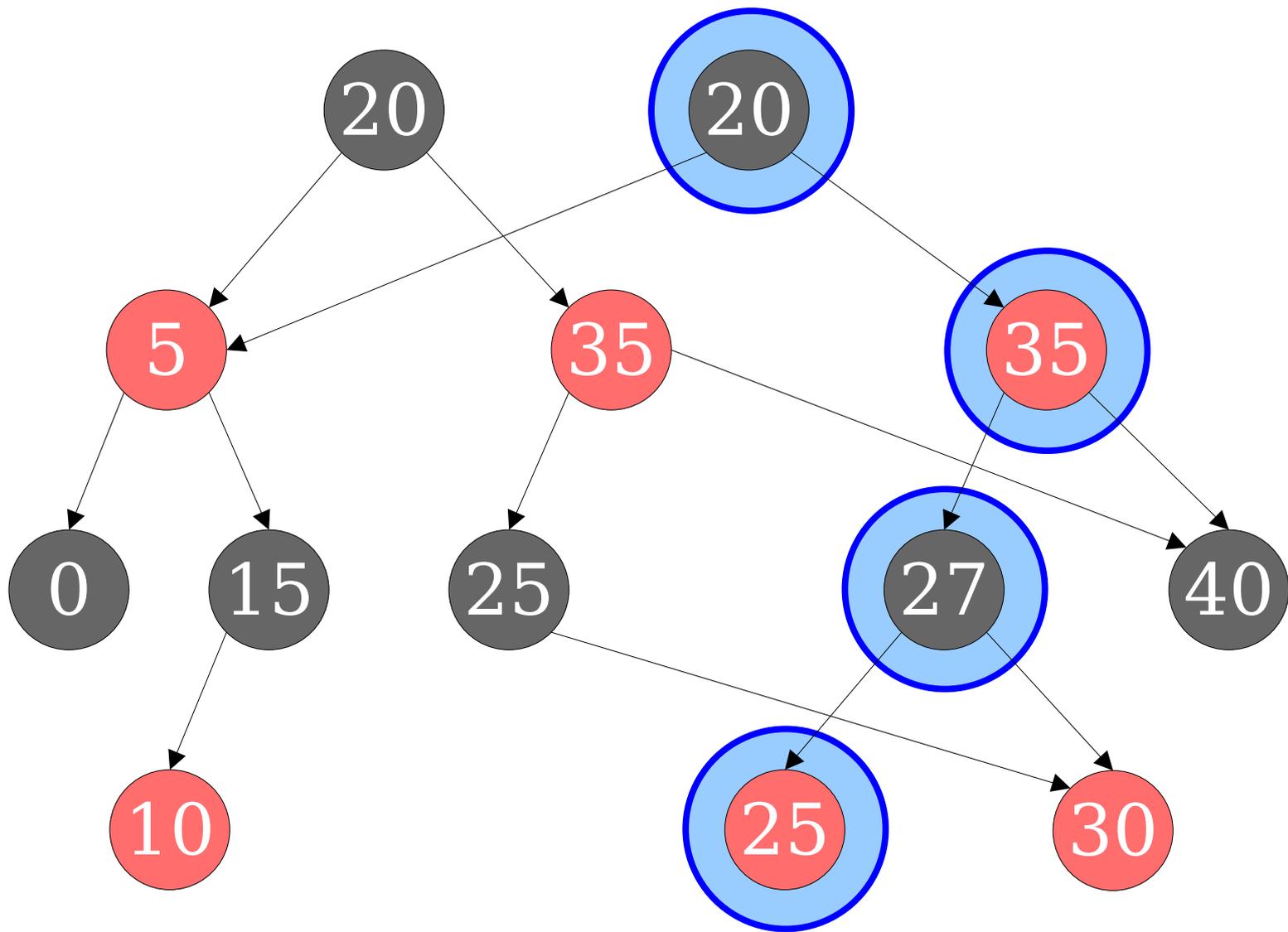
Idea: Keep the old tree around. Clone nodes that can see a part of the tree that changed.
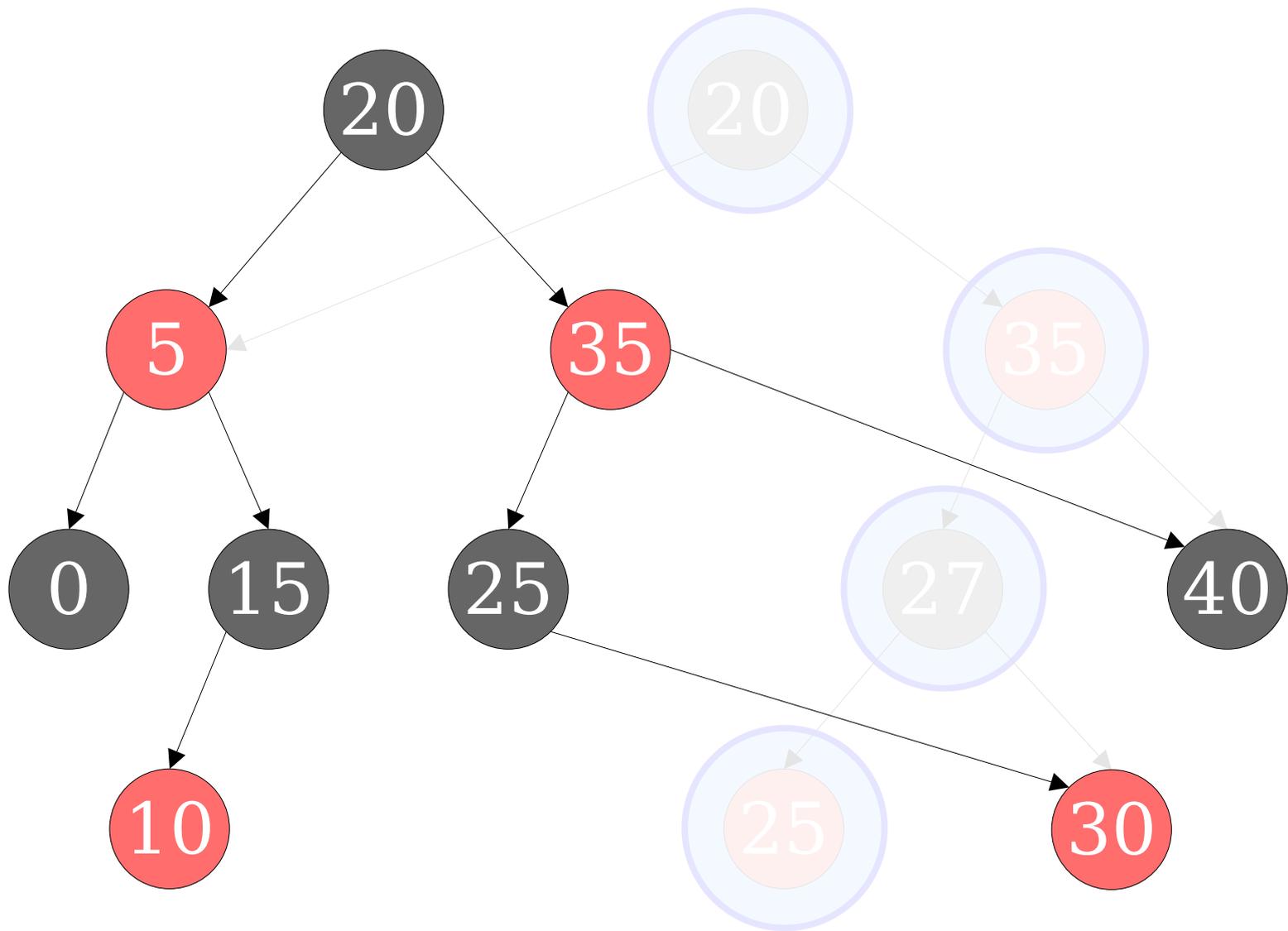
Idea: Keep the old tree around. Clone nodes that can see a part of the tree that changed.
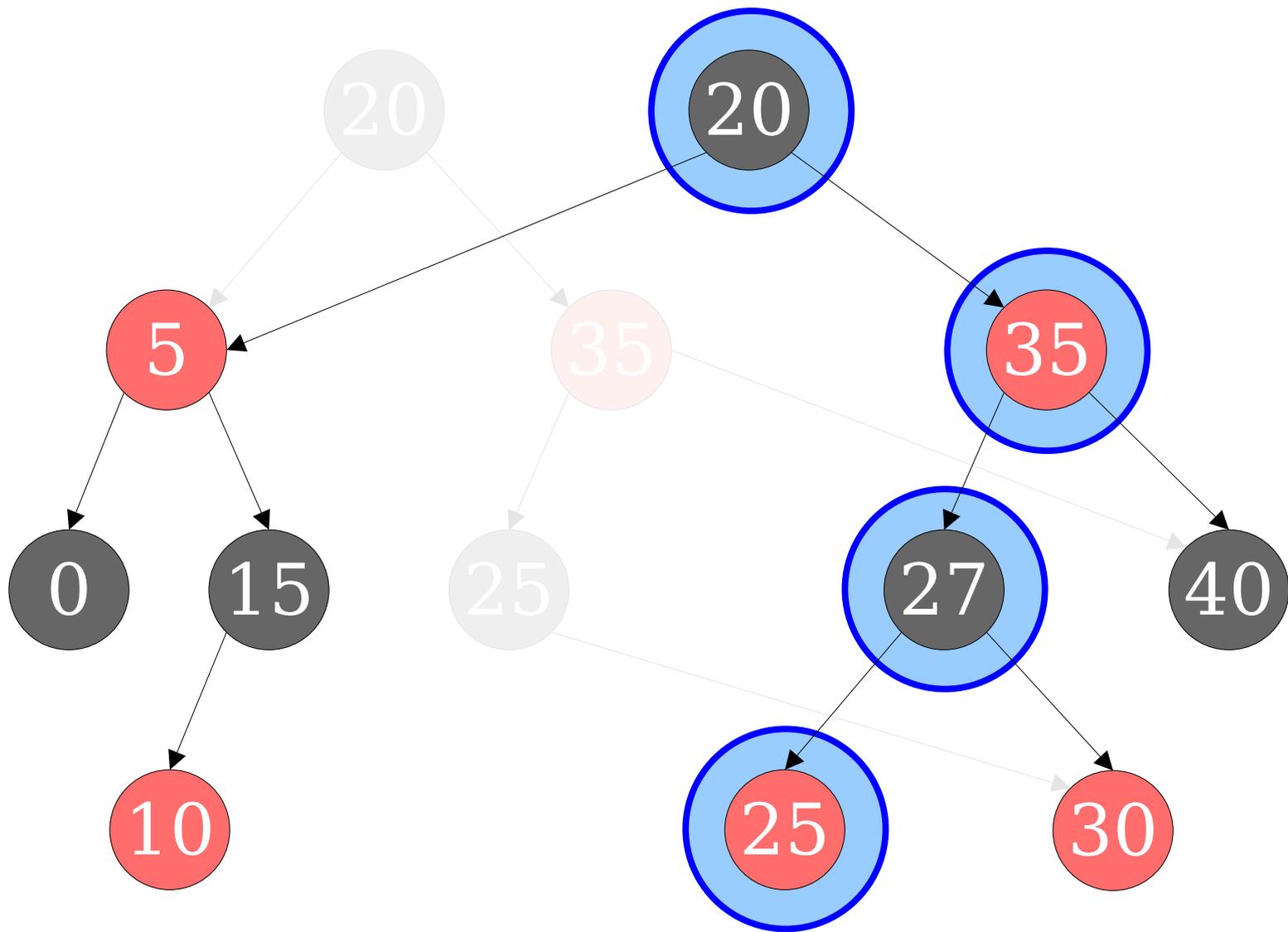
Idea: Keep the old tree around. Clone nodes that can see a part of the tree that changed.

Purely Functional Red/Black Trees

- A ***purely functional red/black tree*** (PFRBT) is a BST where nodes cannot be modified after they're created.

- Operations that would normally mutate the tree instead create new nodes representing the changed parts and share some of the original tree structure.

- The overall shape of the forest of trees is a DAG where each node and its descendants form a red/black tree.
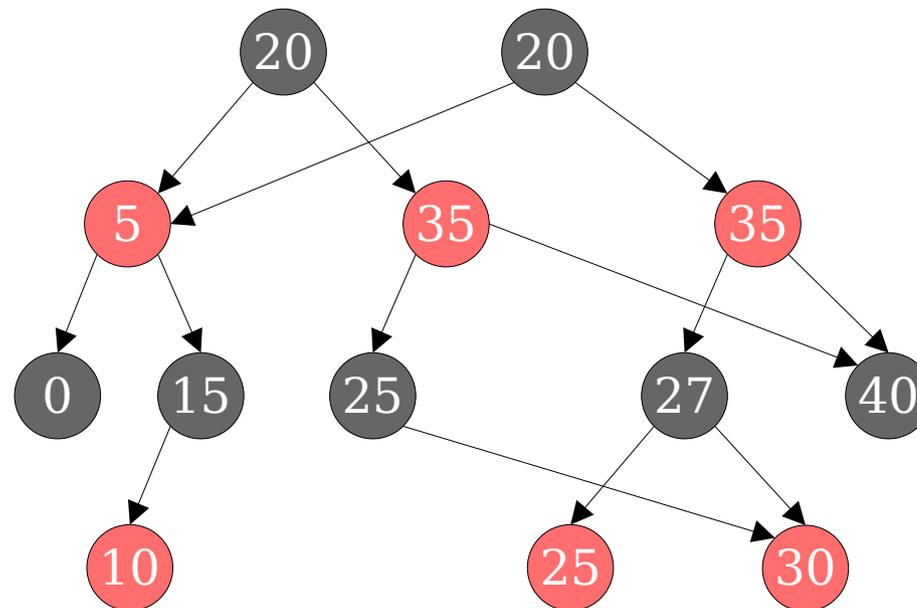
Purely Functional Red/Black Trees

- An insertion or deletion on PFRBTs can be done in time $O(\log n)$ and creates $\Theta(\log n)$ new nodes.

- ***Proof idea:*** We only touch nodes on the access path, plus children of nodes on the access path.

Purely Functional Red/Black Trees

- An insertion or deletion on PFRBTs can be done in time O(log n) and creates Θ(log n) new nodes.

- ***Proof idea:*** We only touch nodes on the access path, plus children of nodes on the access path.

Purely Functional Red/Black Trees

- An insertion or deletion on PFRBTs can be done in time O(log n) and creates $\Theta(\log n)$ new nodes.

- ***Proof idea:*** We only touch nodes on the access path, plus children of nodes on the access path.
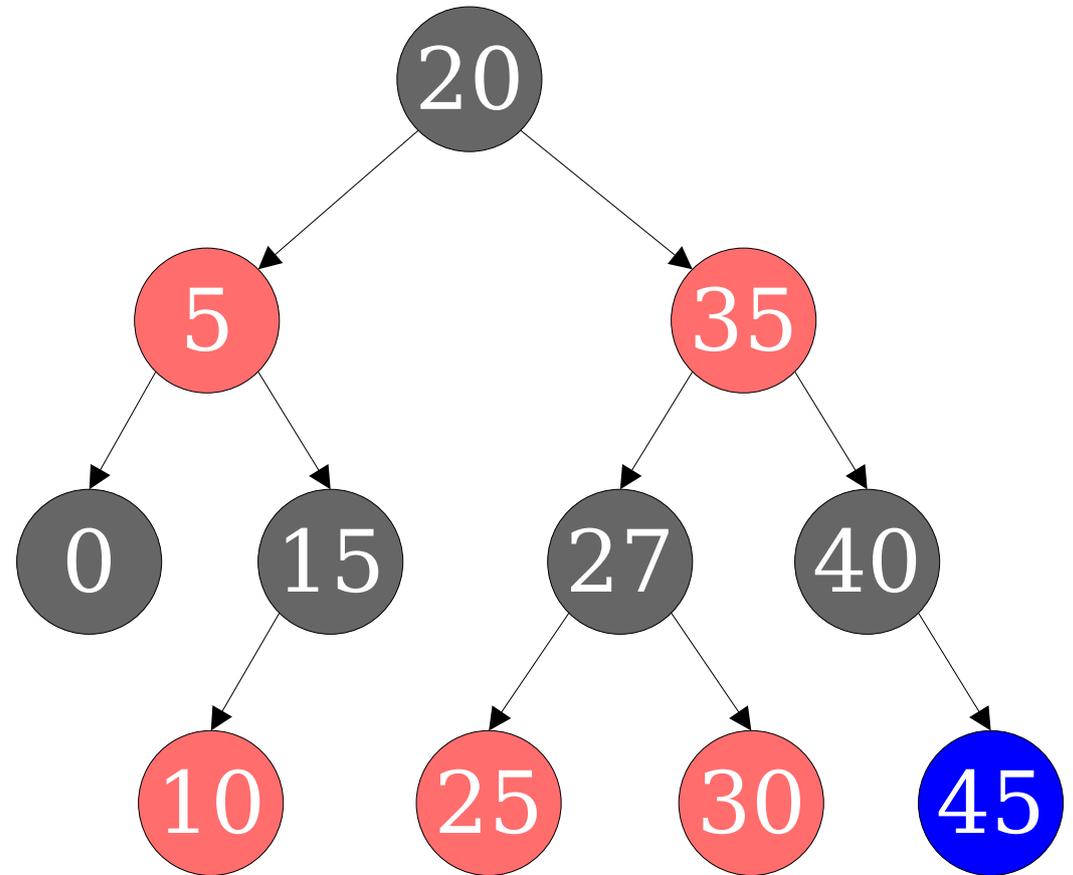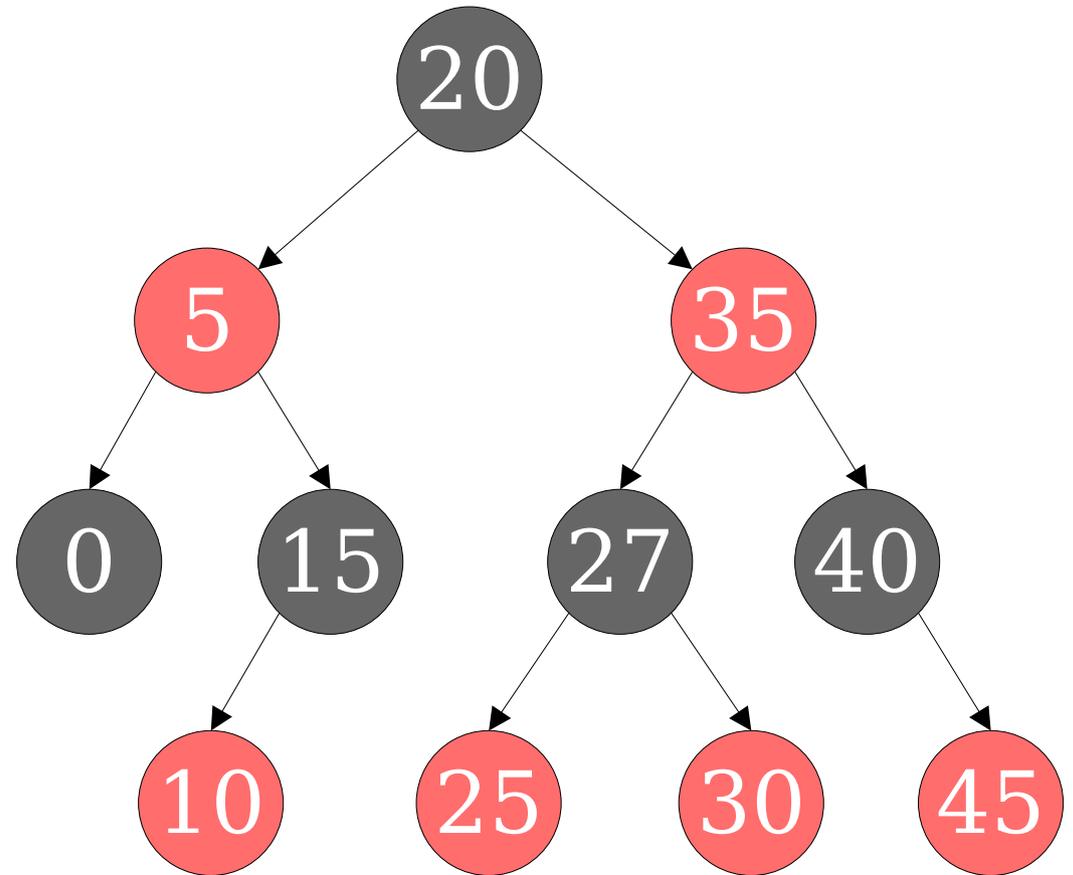
Purely Functional Red/Black Trees

- An insertion or deletion on PFRBTs can be done in time O(log n) and creates Θ(log n) new nodes.

- ***Proof idea:*** We only touch nodes on the access path, plus children of nodes on the access path.

Functional Slabs

- ***Idea:*** Use the slab decomposition, storing slabs as PFRBTs.

- Each line segment is added into some slab, then removed from some later slab.

- Total number of insertions and deletions: O(n).

- Time cost per insertion or deletion: O(log n).

- Space cost per insertion or deletion: O(log n).

- Total time and space for preprocessing: **O(n log n)**.
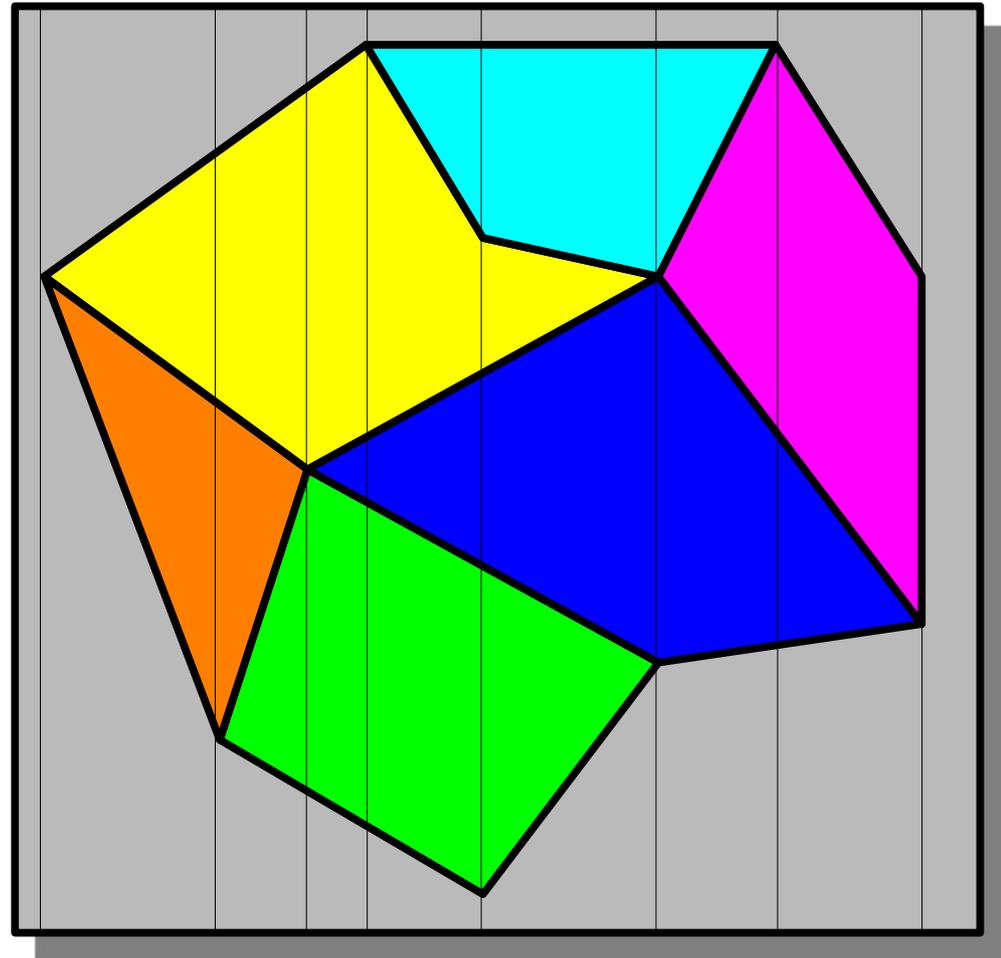
The Story So Far

- Using PFRBTs gives us markedly better worst-case preprocessing and space costs while preserving queries.

- Our space usage is O(n log n), which is realizable in the worst case.

- Just storing the deltas themselves requires space O(n).

- ***Question:*** Can we do better?

	Preprocessing Time	Query Time	Space Usage
Test All Faces	O(n log n)	O(n)	O(n)
Slab Decomposition	O(n^2)	O(log n)	O(n^2)
Slabs With PFRBTs	O(n log n)	O(log n)	O(n log n)

The Story So Far

- Using PFRBTs gives us markedly better worst-case preprocessing and space costs while preserving queries.
- Our space usage is $O(n \log n)$, which is realizable in the worst case.
- Just storing the deltas themselves requires space $O(n)$.
- **Question:** Can we do better?

	Preprocessing Time	Query Time	Space Usage
Test All Faces	$O(n \log n)$	$O(n)$	$O(n)$
Slab Decomposition	$O(n^2)$	$O(\log n)$	$O(n^2)$
Slabs With PFRBTs	$O(n \log n)$	$O(\log n)$	$O(n \log n)$

Saving Space in PFRBTs

- We are currently using O(n log n) space. We want to use O(n) space. Where is the extra log factor from?

- ***Observation:*** Inserting into a PFRBT copies a full path of size O(log n) after each operation, but in many cases only O(1) nodes are updated.

Saving Space in PFRBTs

- We are currently using O(n log n) space. We want to use O(n) space. Where is the extra log factor from?

- ***Observation:*** Inserting into a PFRBT copies a full path of size O(log n) after each operation, but in many cases only O(1) nodes are updated.
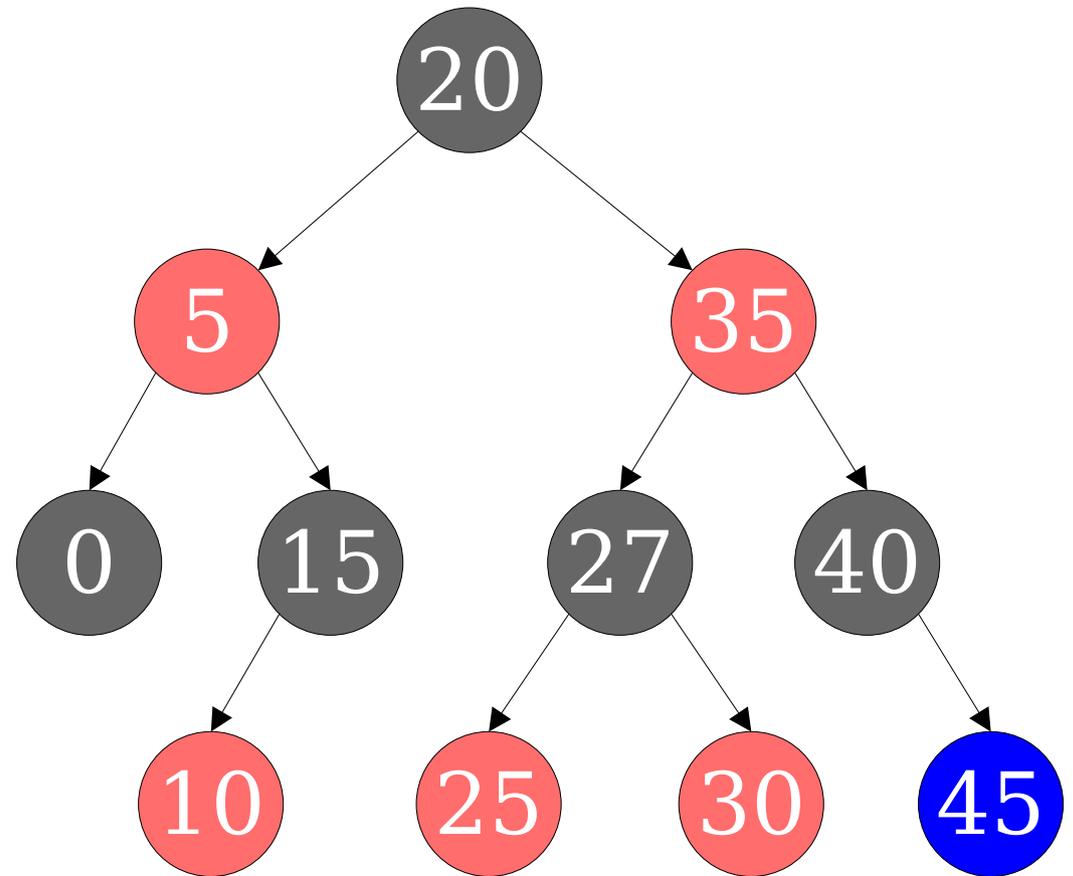
Saving Space in PFRBTs

- We are currently using O(n log n) space. We want to use O(n) space. Where is the extra log factor from?

- ***Observation:*** Inserting into a PFRBT copies a full path of size O(log n) after each operation, but in many cases only O(1) nodes are updated.

Saving Space in PFRBTs

- We are currently using O(n log n) space. We want to use O(n) space. Where is the extra log factor from?

- ***Observation:*** Inserting into a PFRBT copies a full path of size O(log n) after each operation, but in many cases only O(1) nodes are updated.
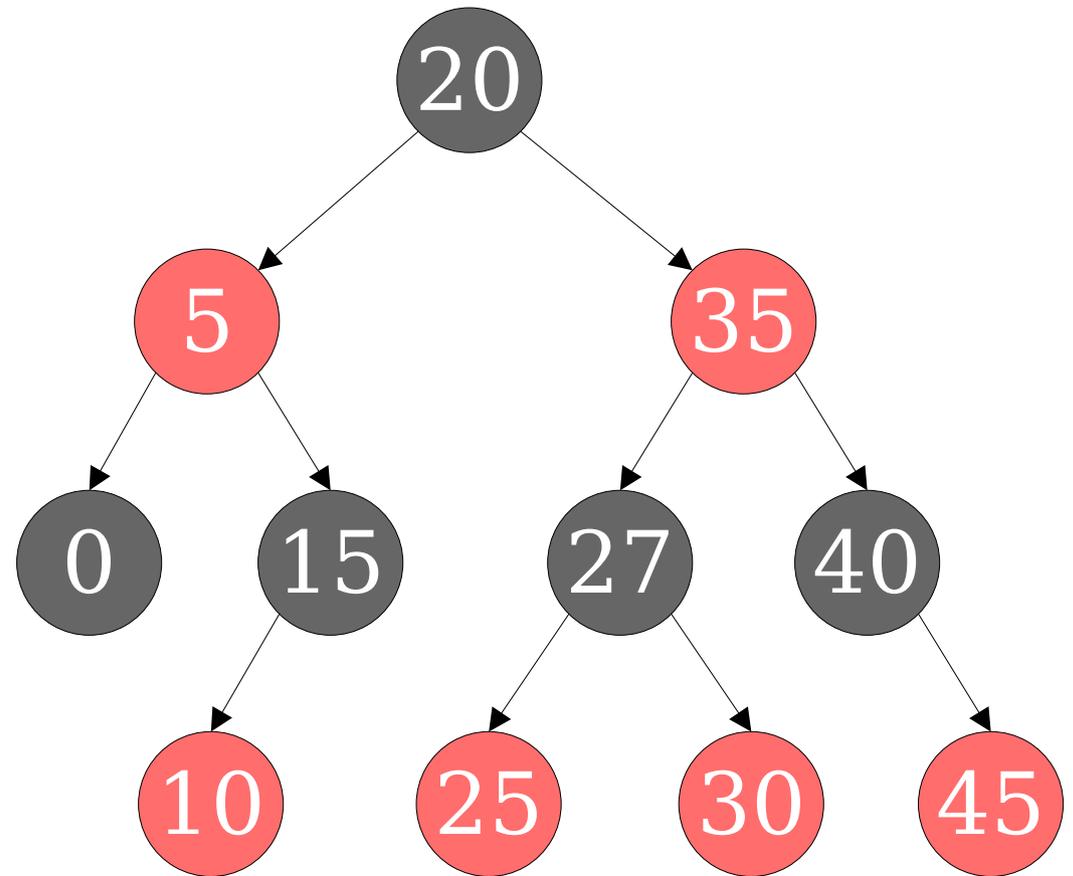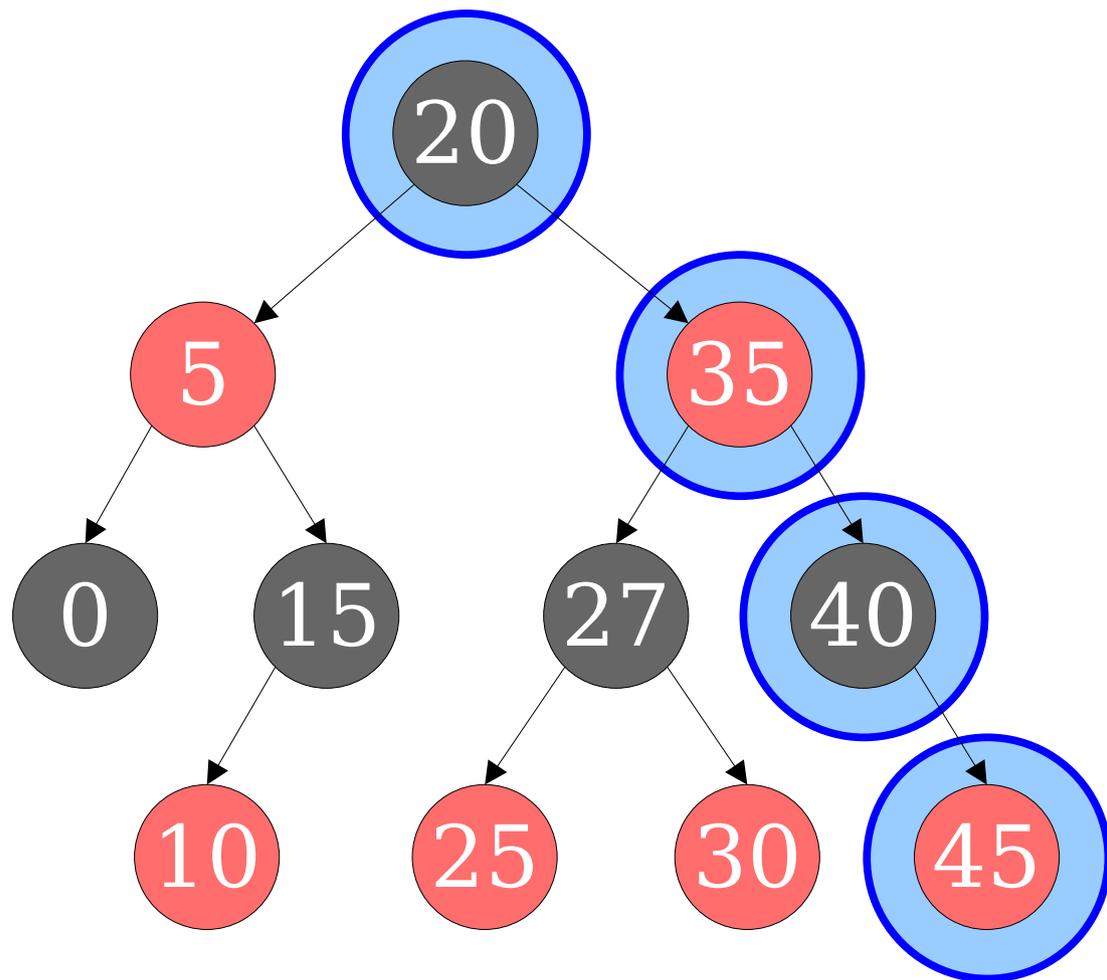
Saving Space

- **_Fact:_** The amortized number of modifications made to a red/black tree after an insert or remove operation is O(1).

- In other words, a series of n operations on a red/black tree never makes more than O(n) total edits to the nodes in that tree.

- **_Goal:_** Store trees such that

  - each operation still lets us access past versions of the tree, but

  - the space usage is proportional to the number of edits made to the data structure.
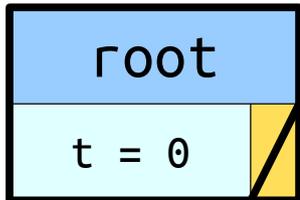
- **_Question:_** Is this possible?

Saving Space

- *Fact:* The amortized number of modifications made to a red/black tree after an insert or remove operation is O(1).

- In other words, red/black tree n edits to the nod

- *Goal:* Store tree

  - each operatio of the tree, but

> Where would it make sense to store the information about the edits?
>
> *Idea:* Write the edit down at the place that you make it.

- the space usage is proportional to the number of edits made to the data structure.

- *Question:* Is this possible?

Journal Trees

- A *journal tree* is a BST where each field in each node is replaced by a *journal* describing its values over time.

- Each operation has an *timestamp* of when it happened.
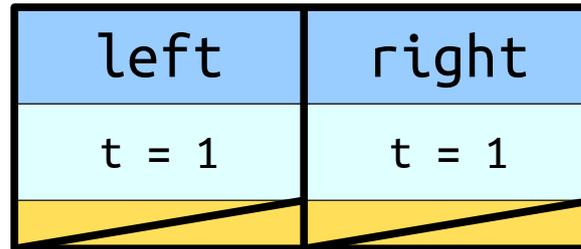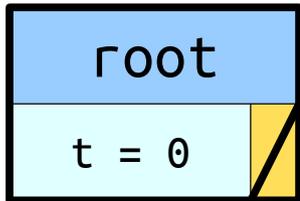
```
root
t = 0
```

Journal Trees

- A ***journal tree*** is a BST where each field in each node is replaced by a ***journal*** describing its values over time.

- Each operation has an ***timestamp*** of when it happened.

root
t = 0

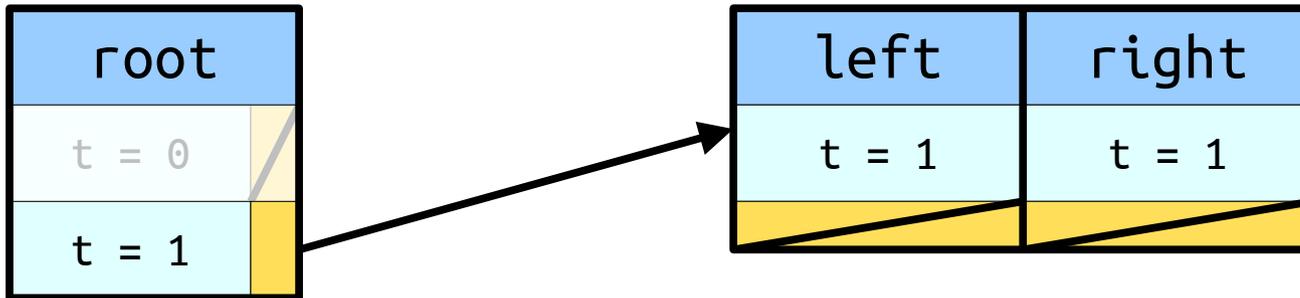| left	right
t = 1	t = 1

Journal Trees

- A ***journal tree*** is a BST where each field in each node is replaced by a ***journal*** describing its values over time.

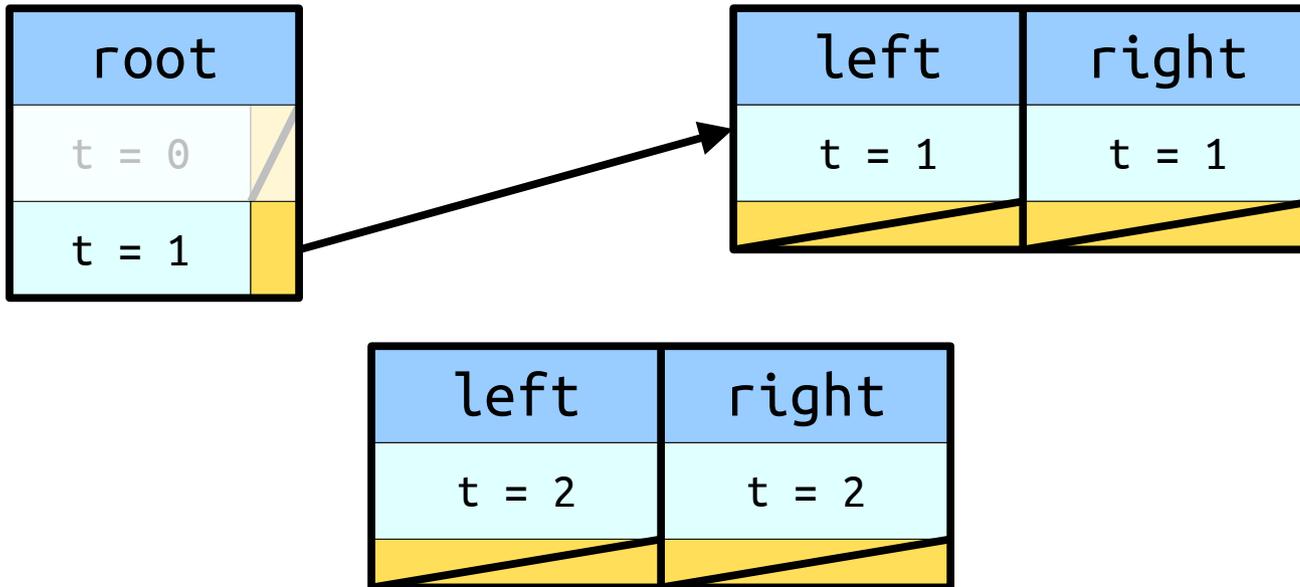- Each operation has an ***timestamp*** of when it happened.

Journal Trees

- A *journal tree* is a BST where each field in each node is replaced by a *journal* describing its values over time.

- Each operation has an *timestamp* of when it happened.

Journal Trees

- A *journal tree* is a BST where each field in each node is replaced by a *journal* describing its values over time.
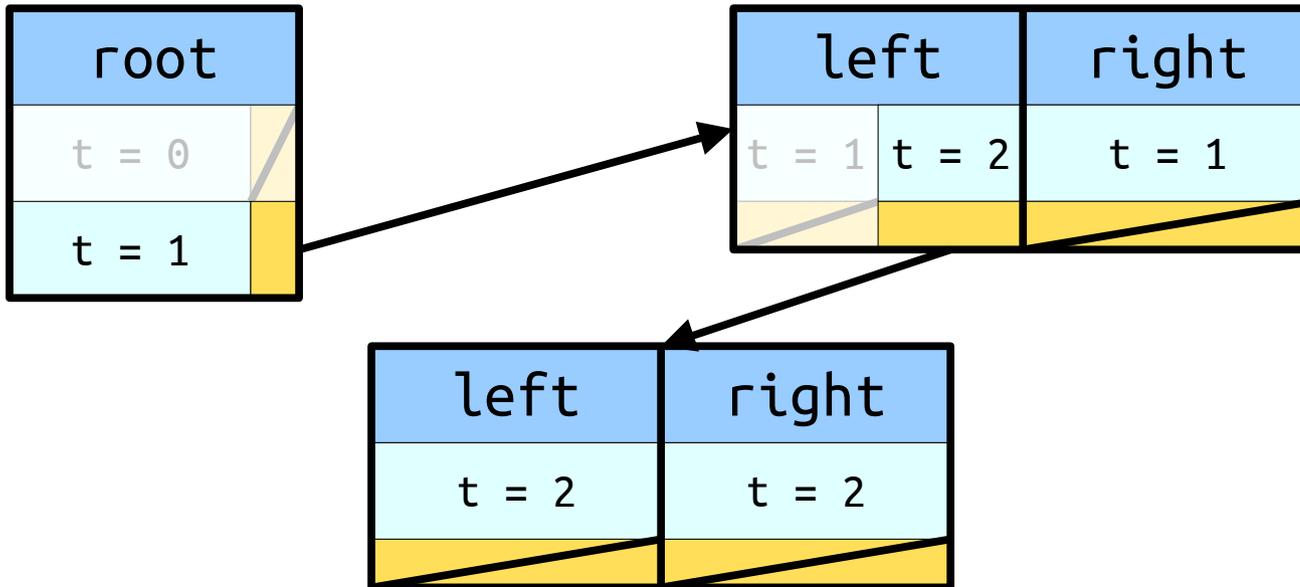
- Each operation has an *timestamp* of when it happened.

Journal Trees

- A *journal tree* is a BST where each field in each node is replaced by a *journal* describing its values over time.
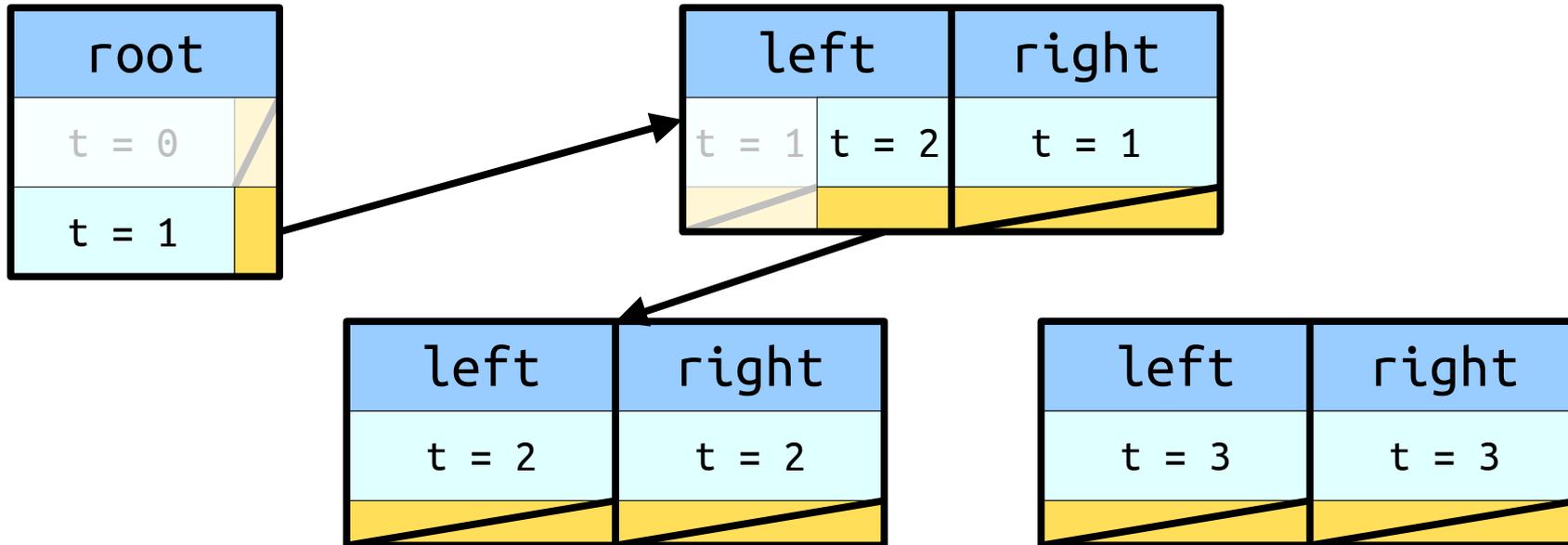
- Each operation has an *timestamp* of when it happened.

Journal Trees

- A *journal tree* is a BST where each field in each node is replaced by a *journal* describing its values over time.
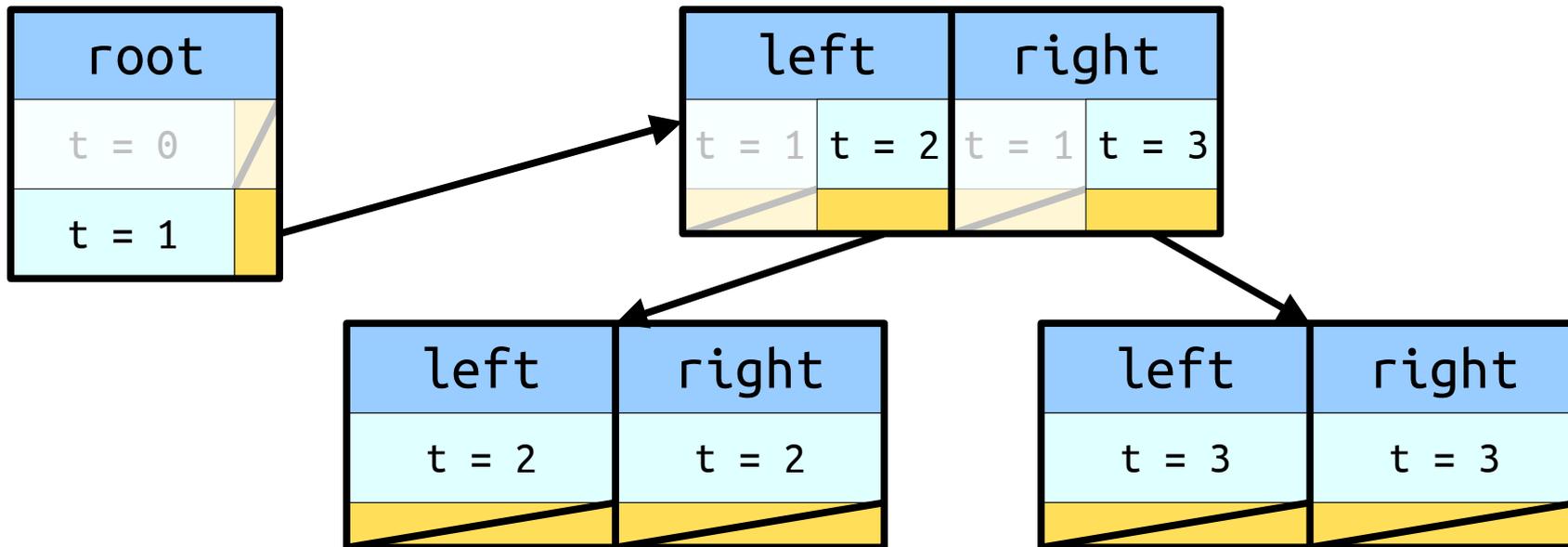
- Each operation has an *timestamp* of when it happened.

Journal Trees

- A **_journal tree_** is a BST where each field in each node is replaced by a **_journal_** describing its values over time.

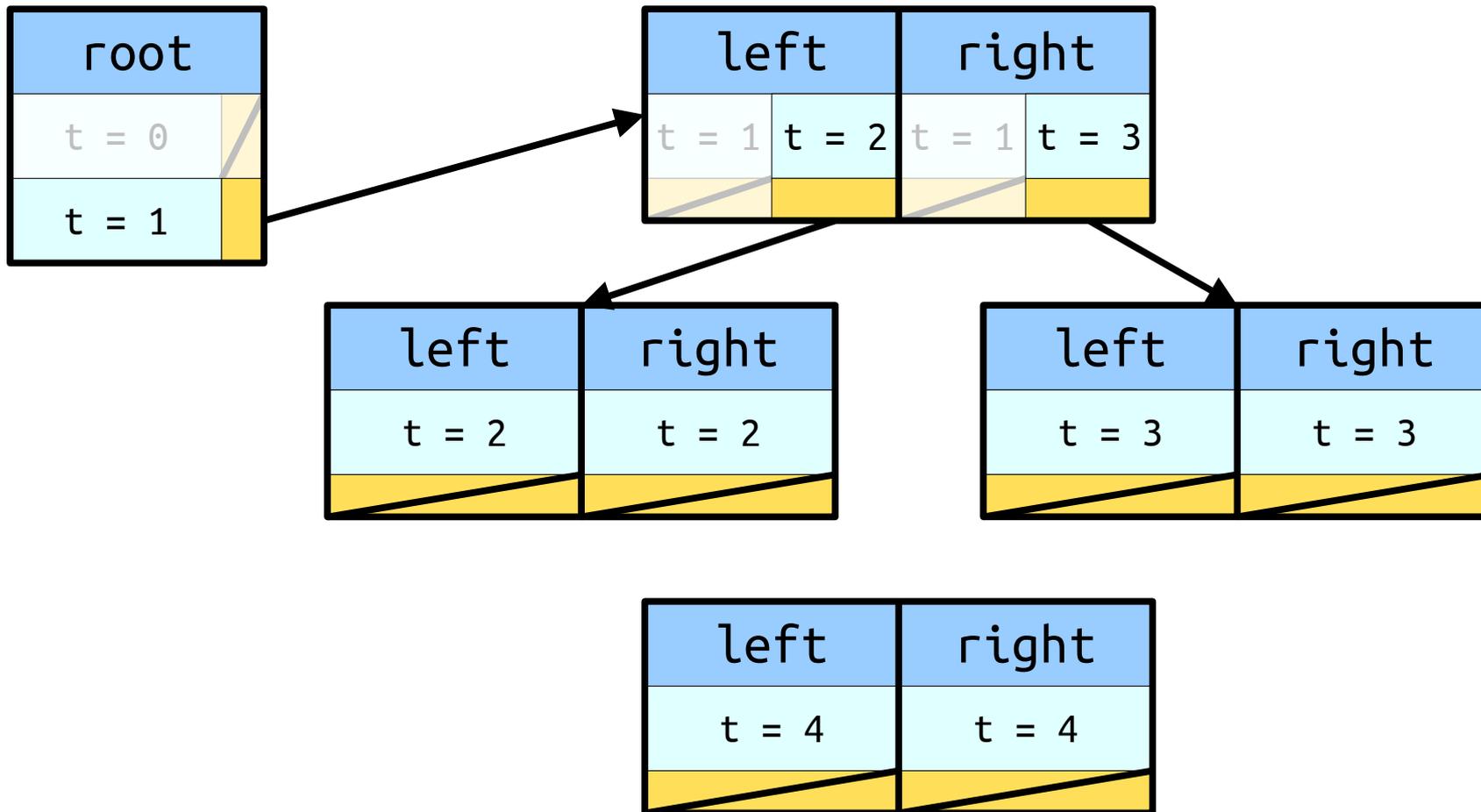- Each operation has an **_timestamp_** of when it happened.

Journal Trees

- A *journal tree* is a BST where each field in each node is replaced by a *journal* describing its values over time.

- Each operation has an *timestamp* of when it happened.

Journal Trees

- A *journal tree* is a BST where each field in each node is replaced by a *journal* describing its values over time.

- Each operation has an *timestamp* of when it happened.

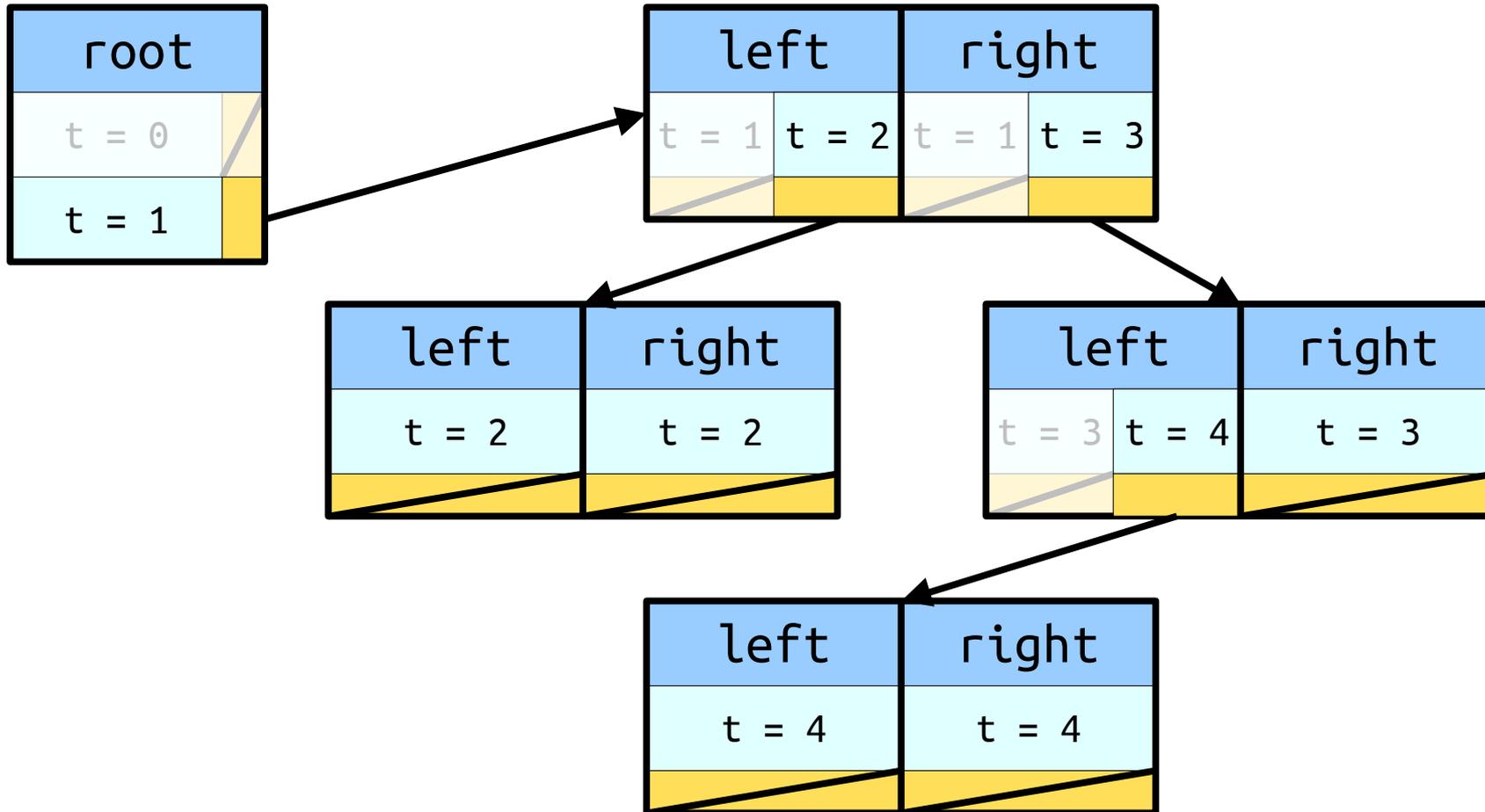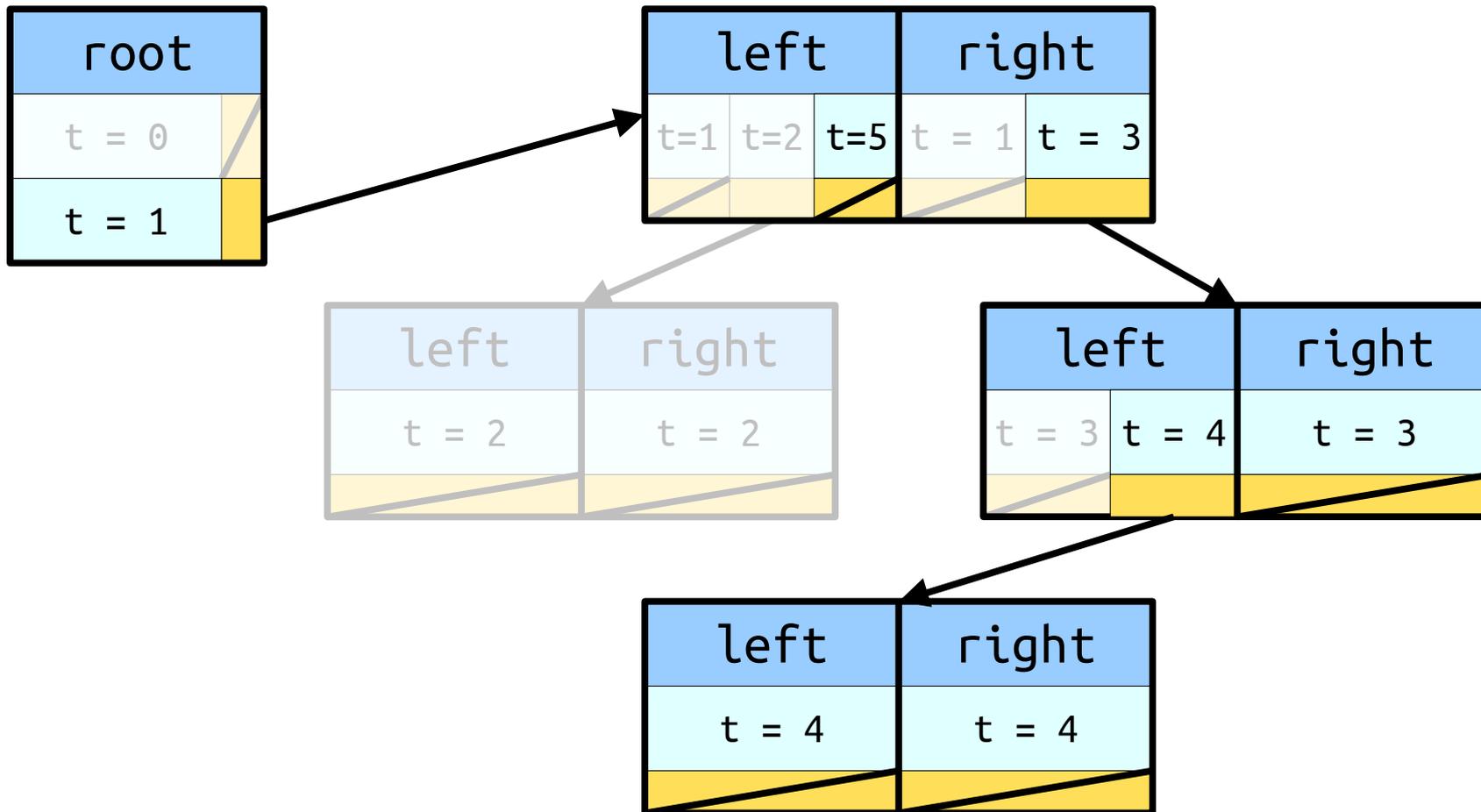# Journal Trees

- A ***journal tree*** is a BST where each field in each node is replaced by a ***journal*** describing its values over time.

- Each operation has an ***timestamp*** of when it happened.

root	
t = 0	
t = 1	

left			right	
t=1	t=2	t=5	t = 1	t = 3

left	right
t = 2	t = 2

left		right
t = 3	t = 4	t = 3

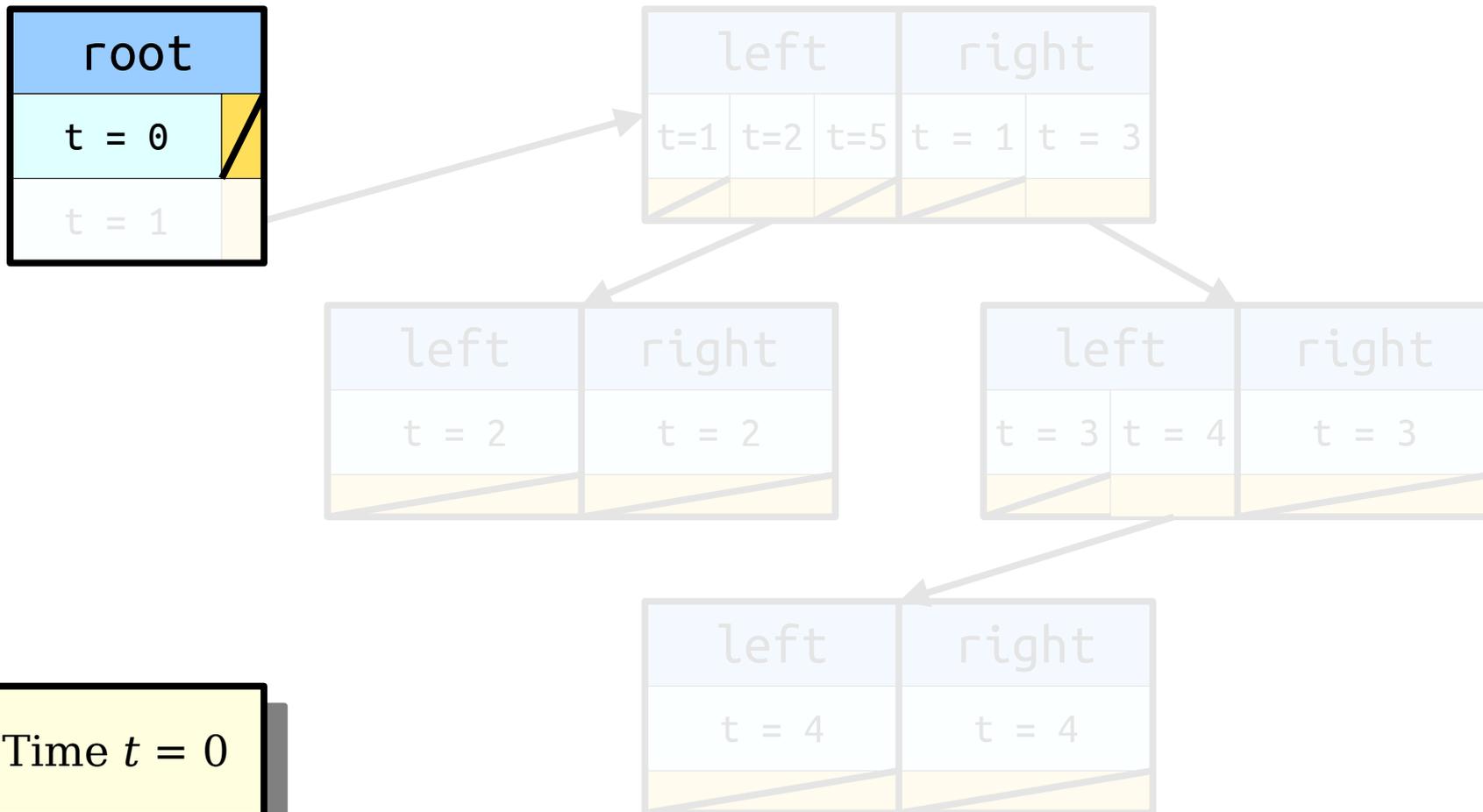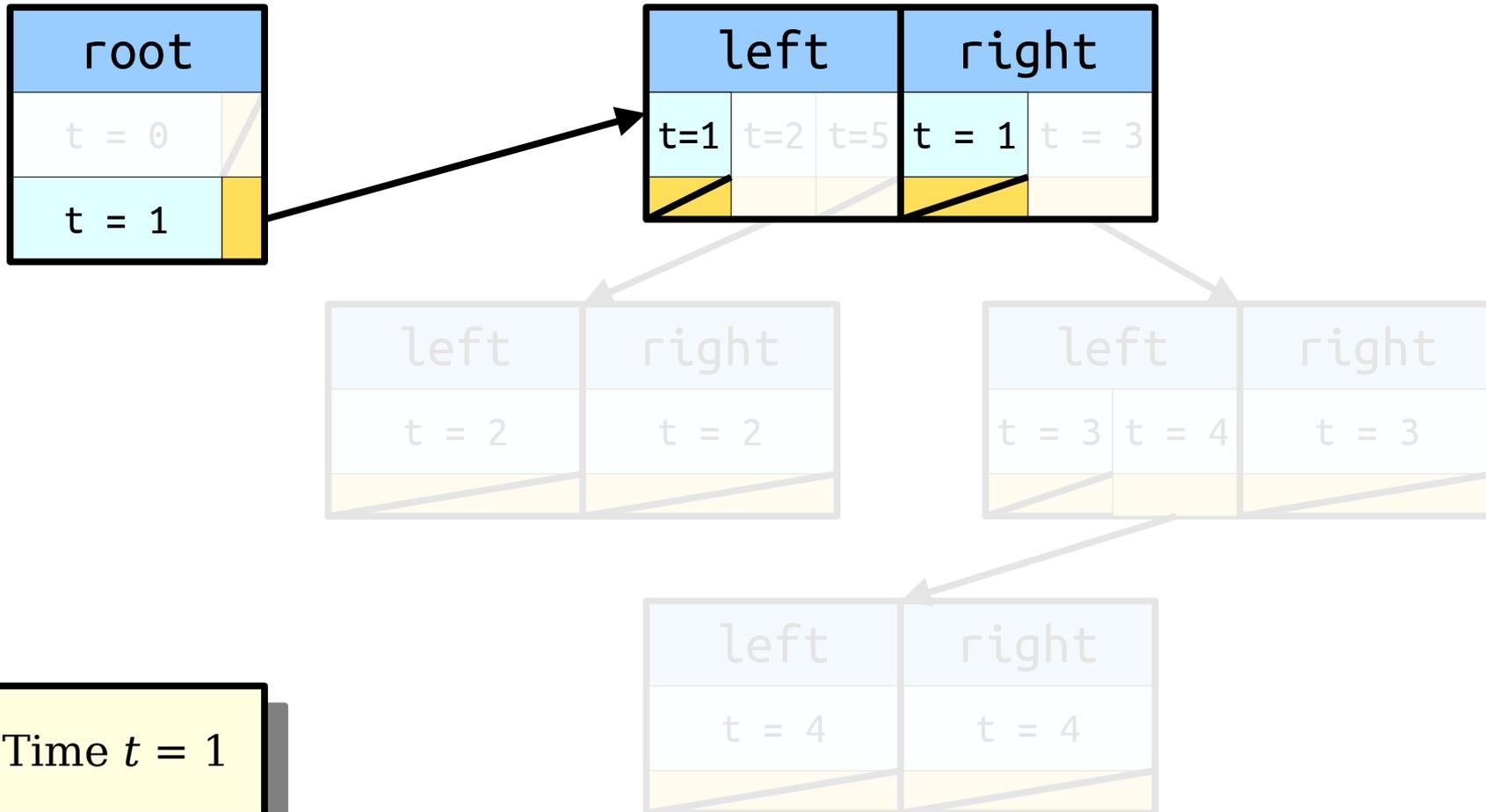| left	right
t = 4	t = 4

Time *t* = 0

Journal Trees

- A ***journal tree*** is a BST where each field in each node is replaced by a ***journal*** describing its values over time.

- Each operation has an ***timestamp*** of when it happened.
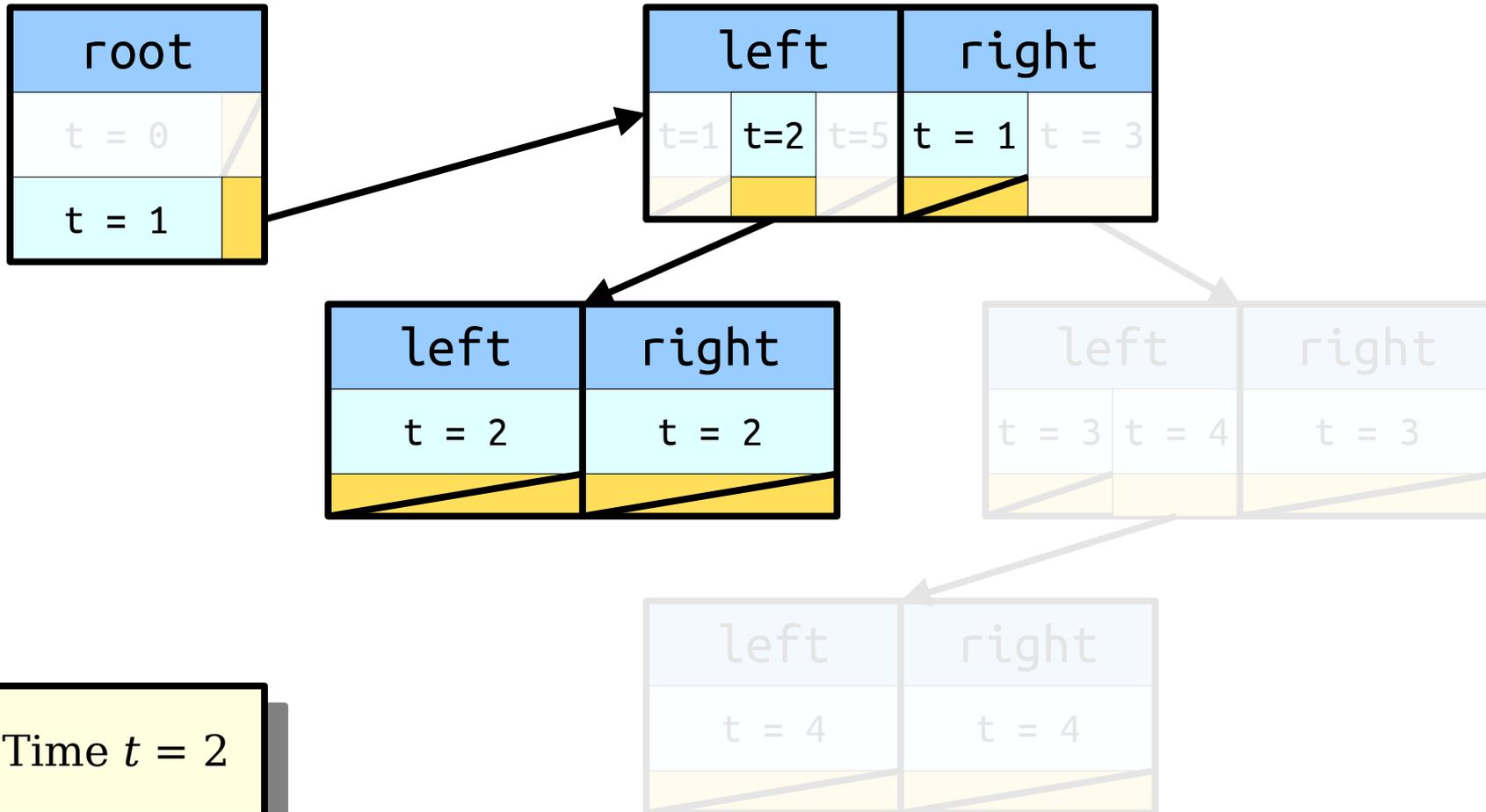

Time *t* = 1

Journal Trees

- A ***journal tree*** is a BST where each field in each node is replaced by a ***journal*** describing its values over time.

- Each operation has an ***timestamp*** of when it happened.

root	
t = 0	
t = 1	

left	right
t=1	t=2	t=5	t = 1	t = 3

left	right
t = 2	t = 2

left	right
t = 3	t = 4	t = 3

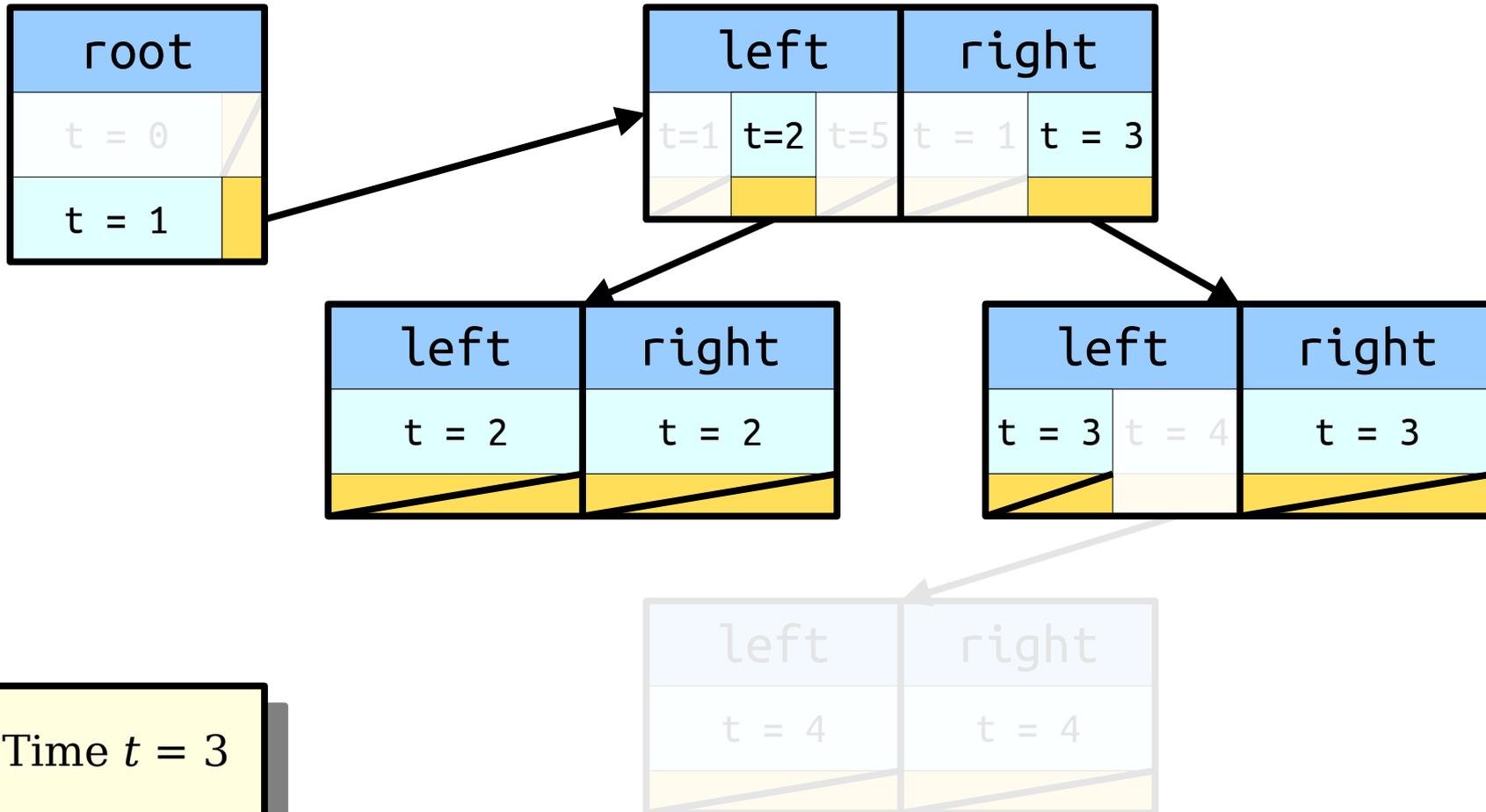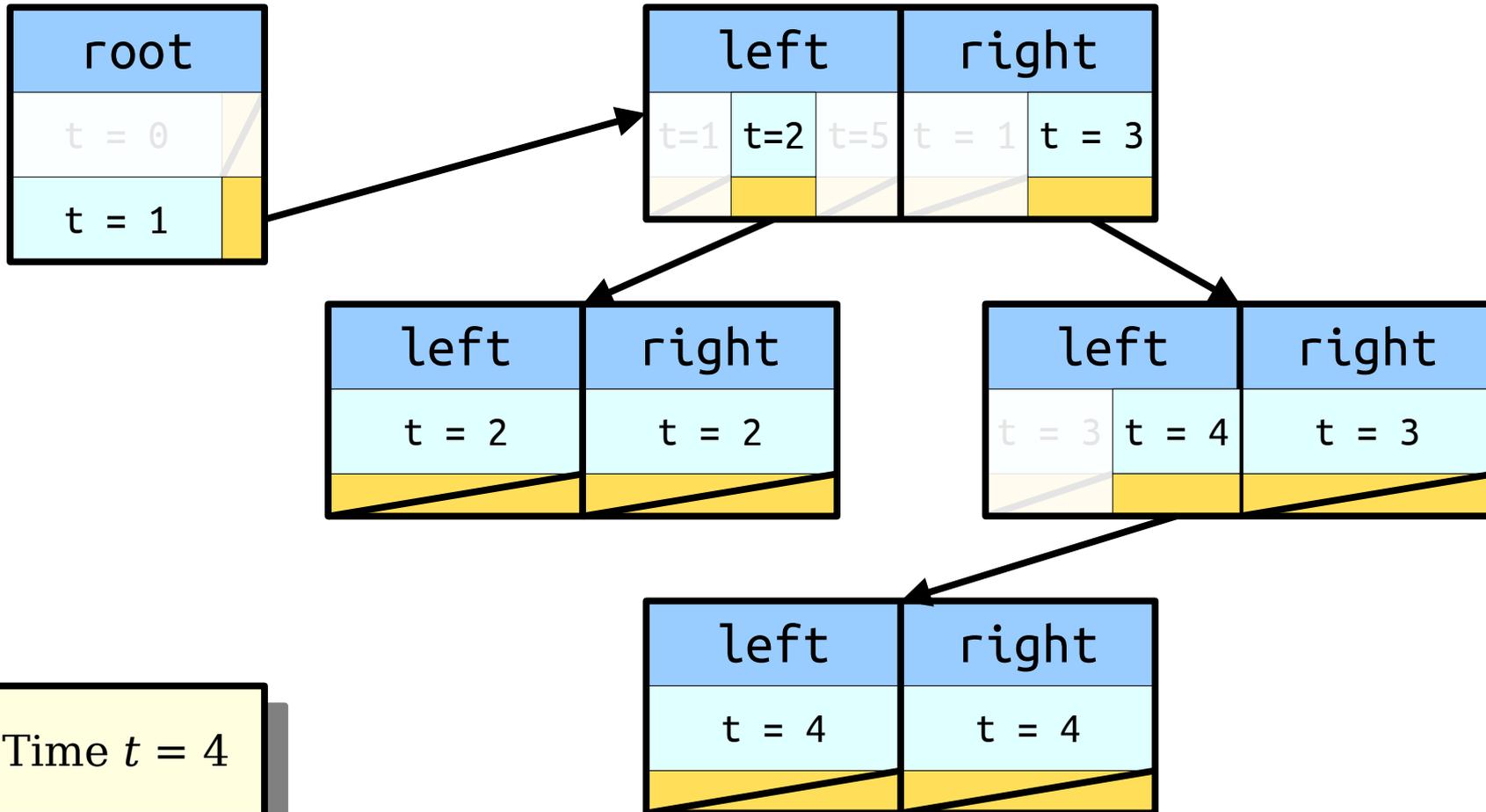| left	right
t = 4	t = 4

Time $t = 2$

Journal Trees

- A **_journal tree_** is a BST where each field in each node is replaced by a **_journal_** describing its values over time.

- Each operation has an **_timestamp_** of when it happened.
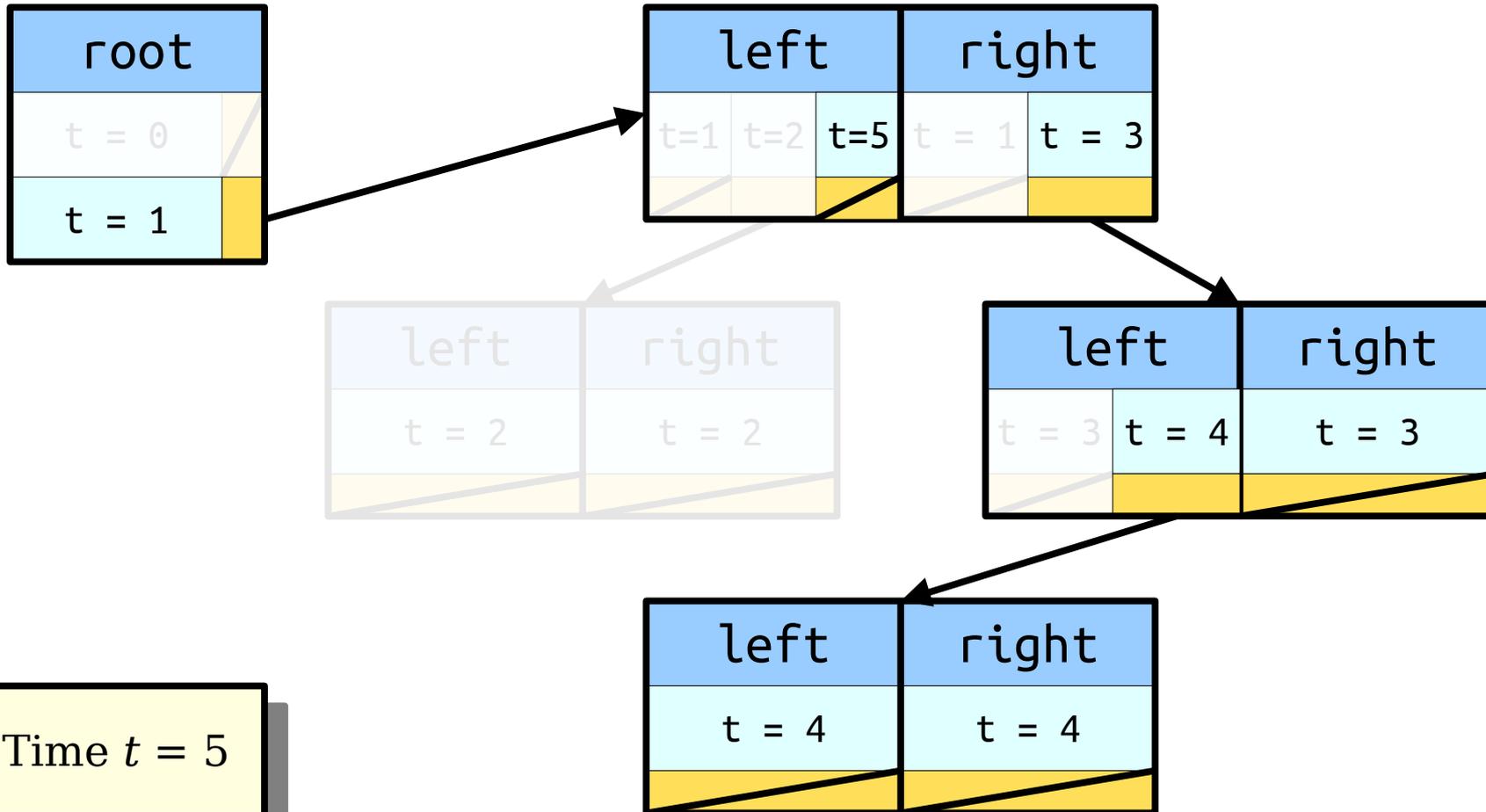

Time *t* = 3

Journal Trees

- A **journal tree** is a BST where each field in each node is replaced by a **journal** describing its values over time.

- Each operation has an **timestamp** of when it happened.

root
t = 0
t = 1

left	right
t=1 t=2 t=5	t = 1 t = 3

left	right
t = 2	t = 2

left	right
t = 3 t = 4	t = 3

| left	right
t = 4	t = 4

Time $t = 4$

Journal Trees

- A **_journal tree_** is a BST where each field in each node is replaced by a **_journal_** describing its values over time.

- Each operation has an **_timestamp_** of when it happened.

Time $t = 5$

Journal Trees

- Space usage: O(m), where m is the number of updates made to the data structure.

- Each insertion and deletion takes (asymptotically) the same amount of time as before.

  - Use the most recent timestamps to see the current version of the tree, then write down any changes made.

- Using a red/black tree, any sequence of n operations requires only **O(n)** storage, since only O(n) updates are made.
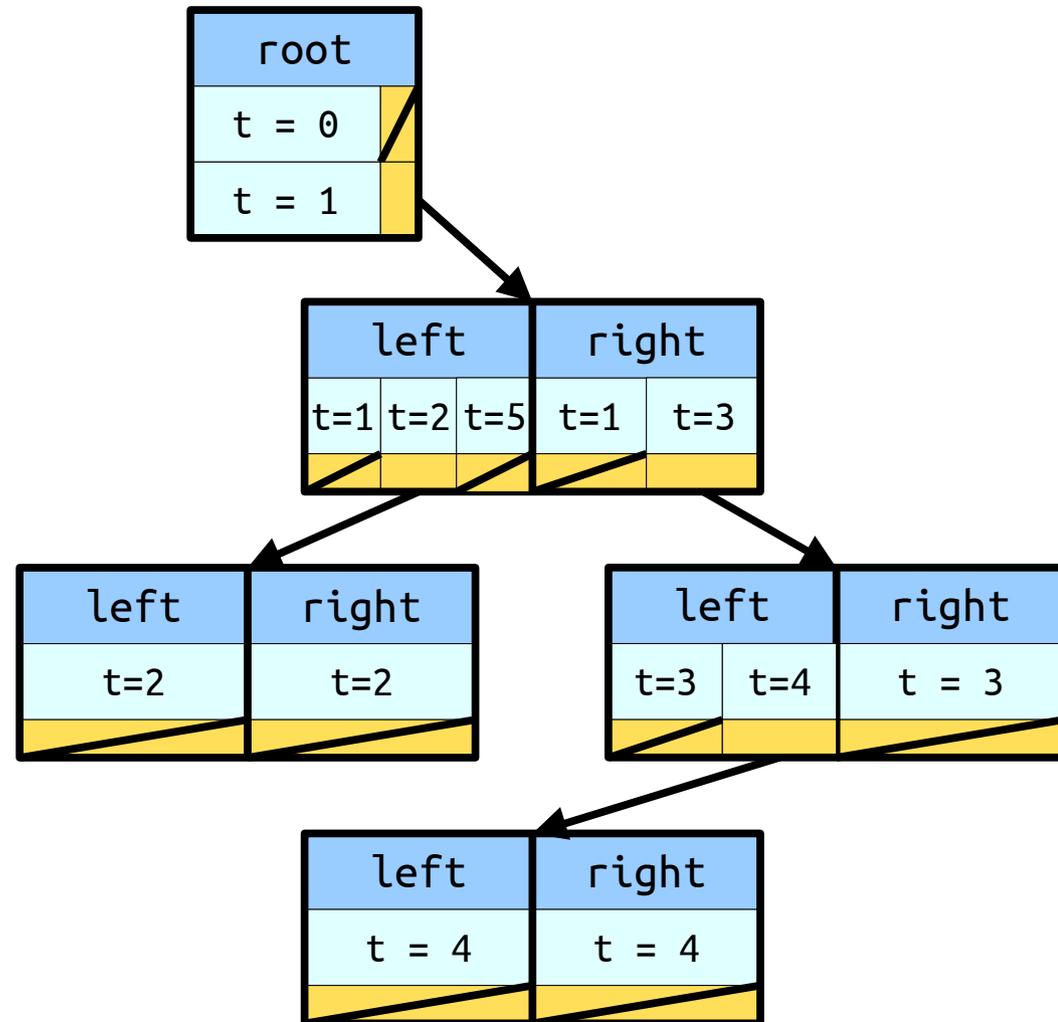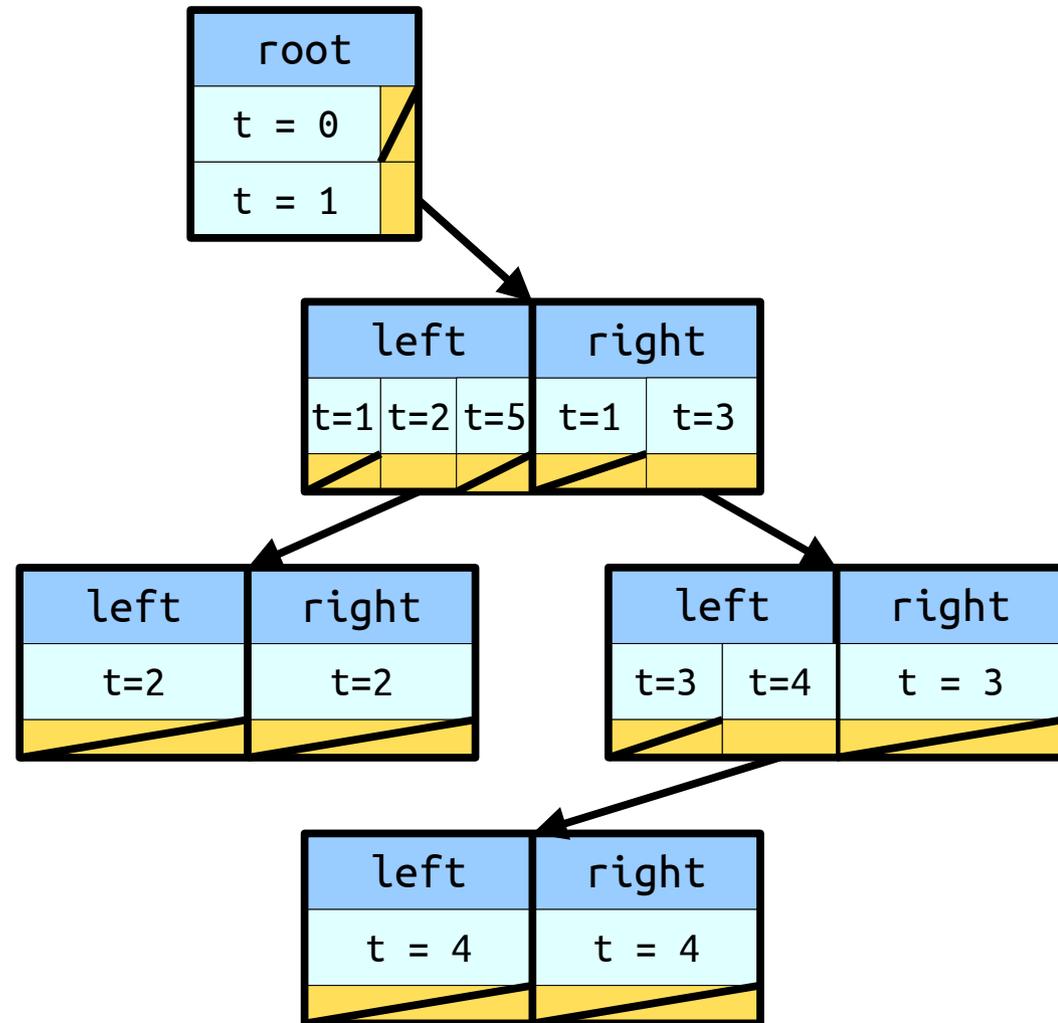
Journal Trees

- Suppose you're given a time t and a key k. What is the cost of seeing whether k is in the tree at time t?
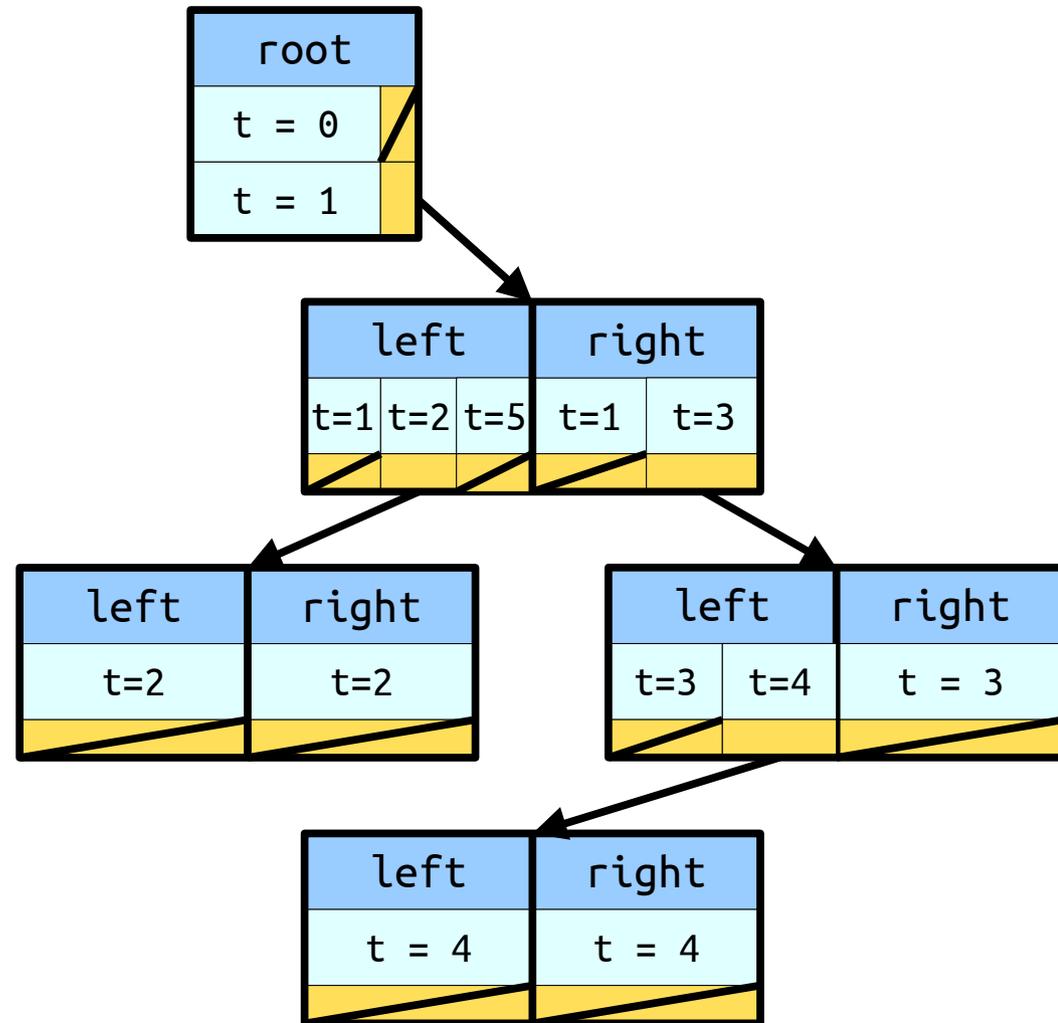
Journal Trees

- Suppose you're given a time t and a key k. What is the cost of seeing whether k is in the tree at time t?
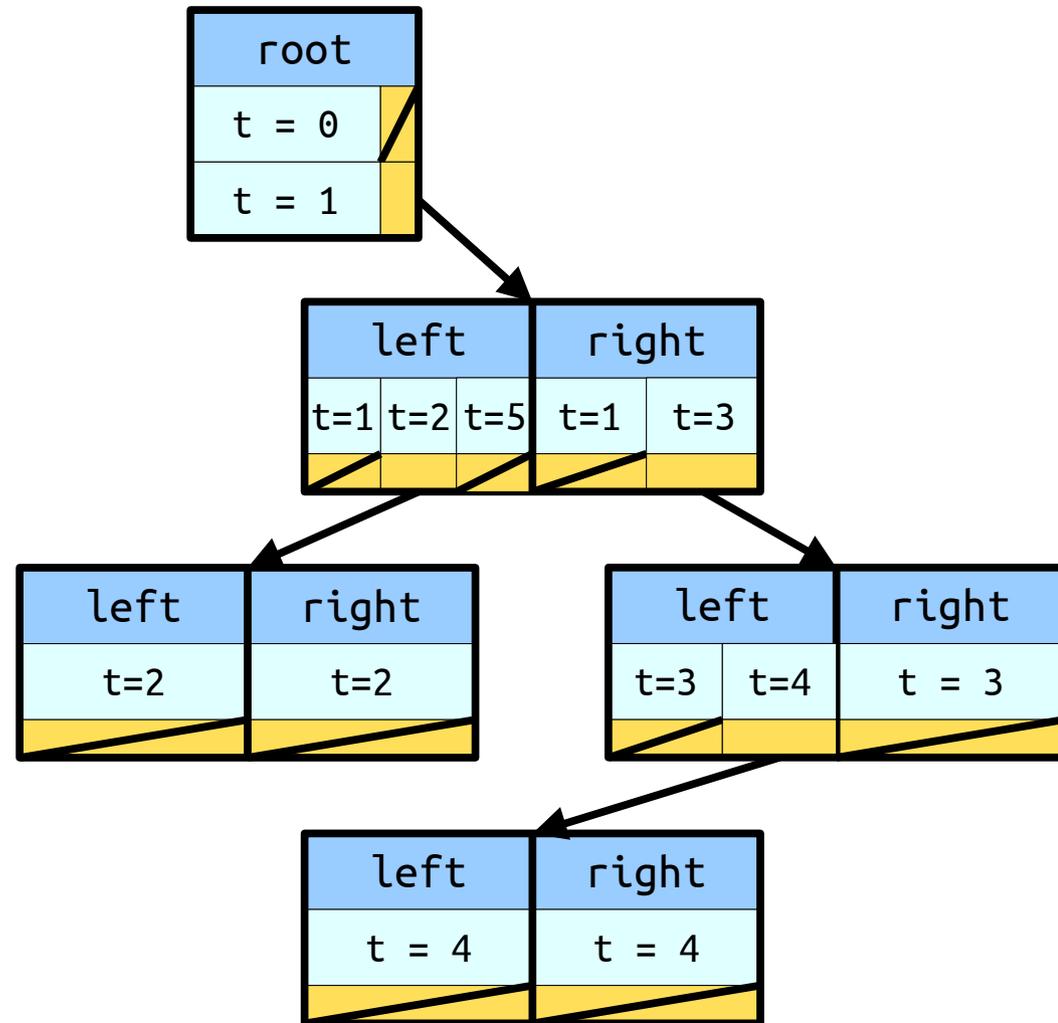
Formulate a hypothesis!

Journal Trees

- Suppose you're given a time t and a key k. What is the cost of seeing whether k is in the tree at time t?

Discuss with your neighbors!

Journal Trees

- Suppose you're given a time t and a key k. What is the cost of seeing whether k is in the tree at time t?

- At each node, we have to do a binary search over timestamps whenever we move left or right.

  - Cost of each binary search: $O(\log t) = O(\log n)$.

  - Number of binary searches: $O(\log t) = O(\log n)$.

- Total cost: **$O(\log^2 n)$**.
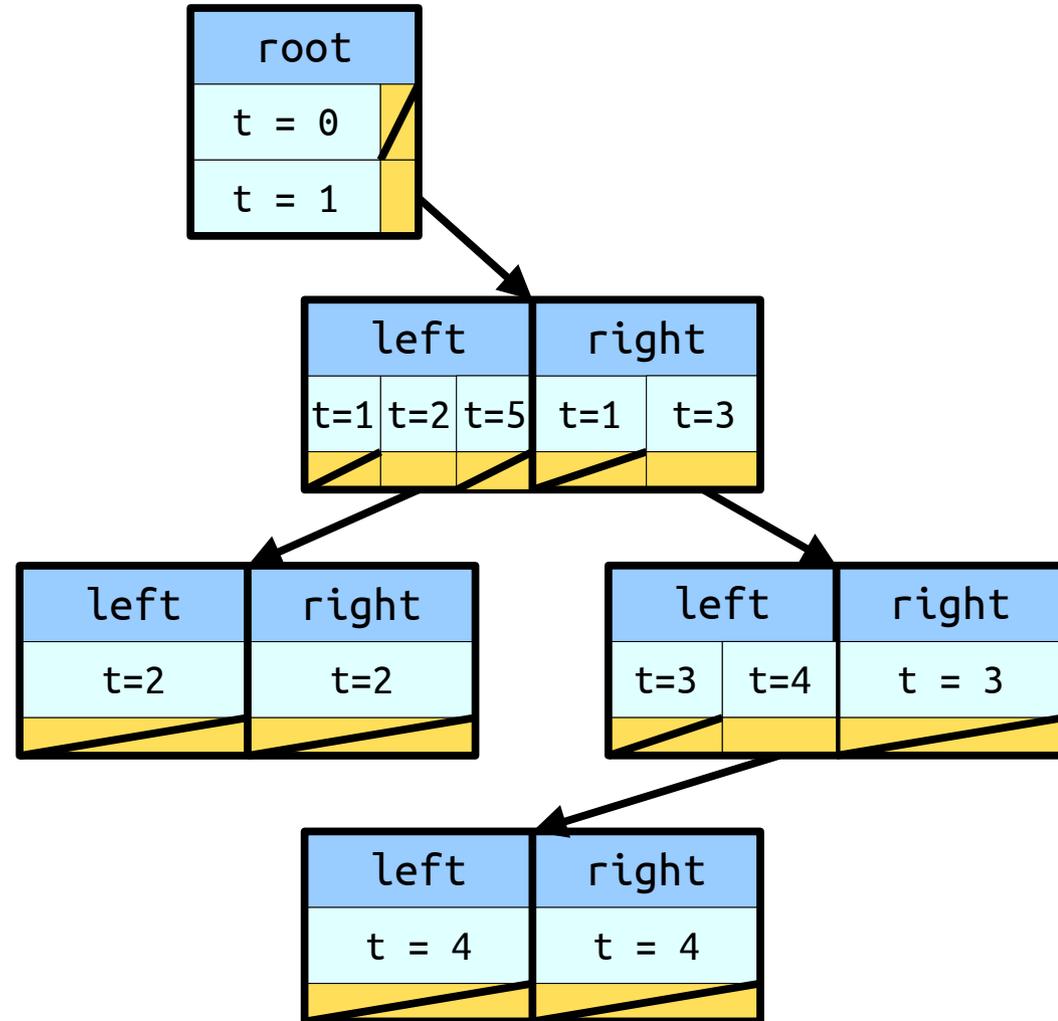
The Story So Far

- We've gotten down to linear space, but at a price – our lookups are now a bit slower than before.

- Is there some way to restore the query time?

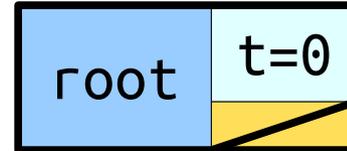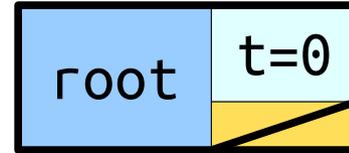|	Preprocessing Time	Query Time	Space Usage
Test All Faces	$O(n \log n)$	$O(n)$	$O(n)$
Slab Decomposition	$O(n^2)$	$O(\log n)$	$O(n^2)$
Slabs With PFRBTs	$O(n \log n)$	$O(\log n)$	$O(n \log n)$
Slabs With Journal RBTs	$O(n \log n)$	$O(\log^2 n)$	$O(n)$

Two Extremes

- With a purely functional tree, each node sees only one version of itself, but we copy lots of extra nodes on each operation.

- With a journal tree, we never copy nodes, but each node in the tree sees all versions of itself.

- ***Question:*** Is there some way to get the best of both worlds?

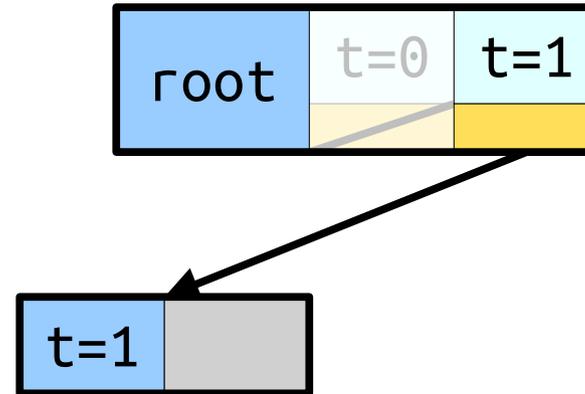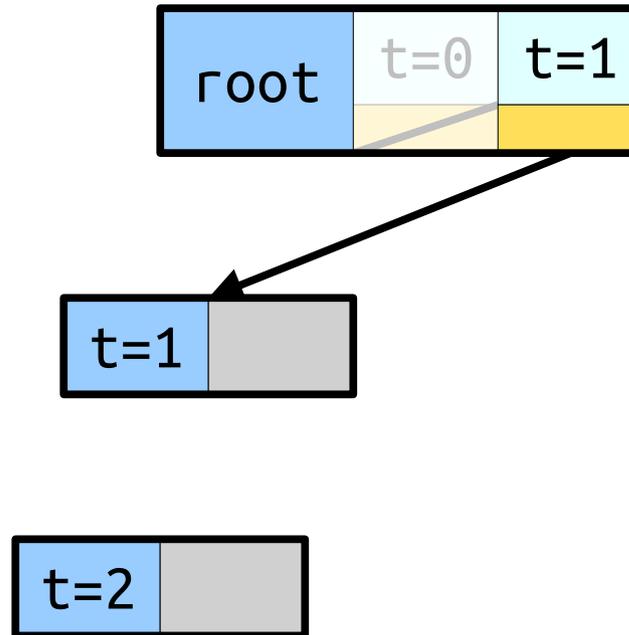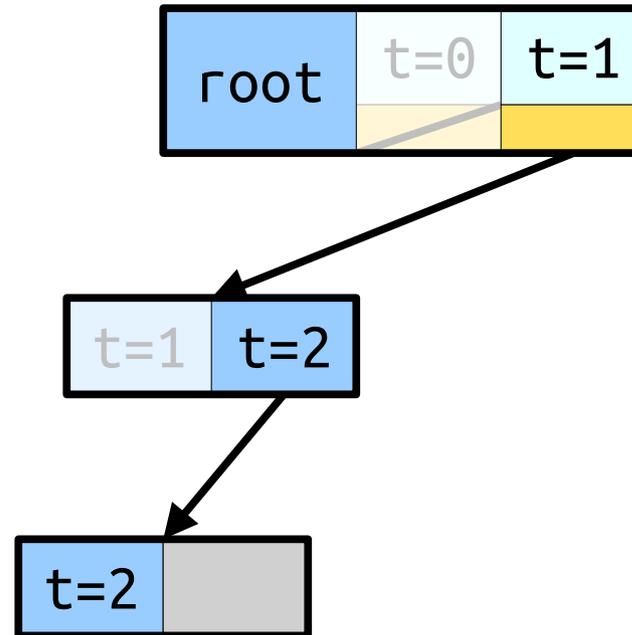|	Preprocessing Time	Query Time	Space Usage
Test All Faces	$O(n \log n)$	$O(n)$	$O(n)$
Slab Decomposition	$O(n^2)$	$O(\log n)$	$O(n^2)$
Slabs With PFRBTs	$O(n \log n)$	$O(\log n)$	$O(n \log n)$
Slabs With Journal RBTs	$O(n \log n)$	$O(\log^2 n)$	$O(n)$

Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.
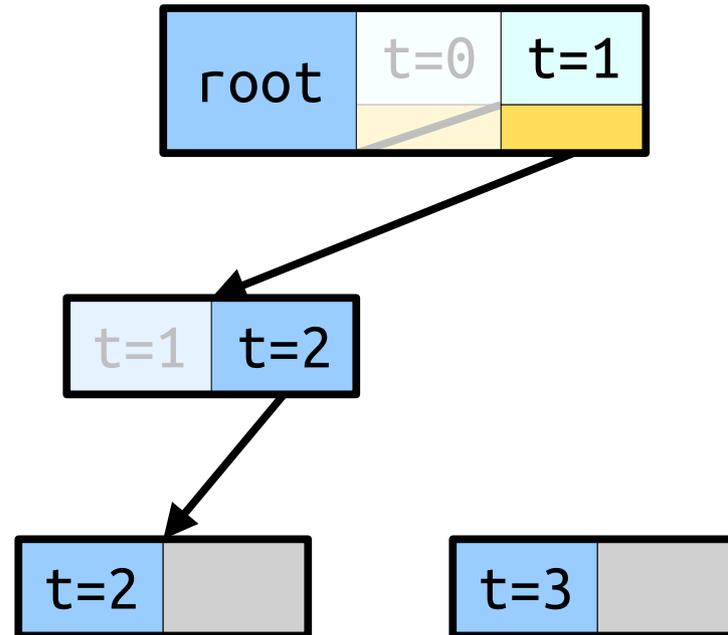
Fat Node Trees

- **_Idea:_** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

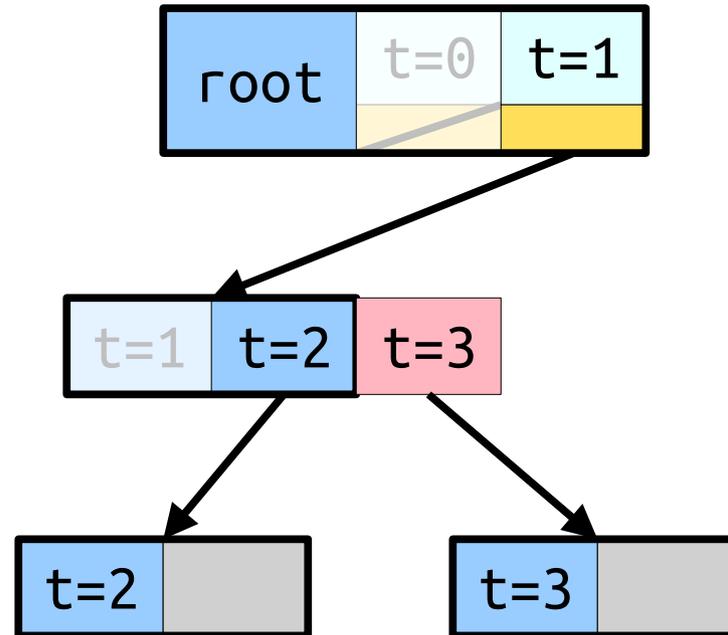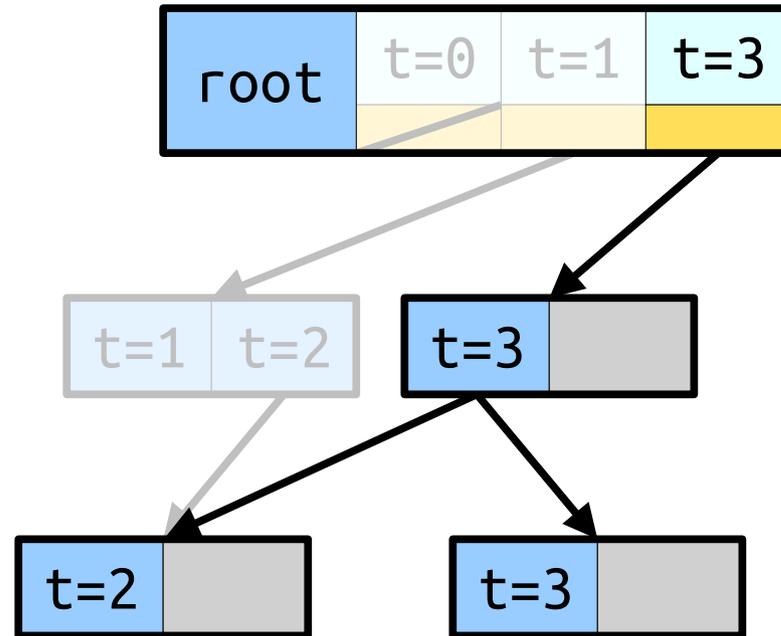# Fat Node Trees

- *Idea:* Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.
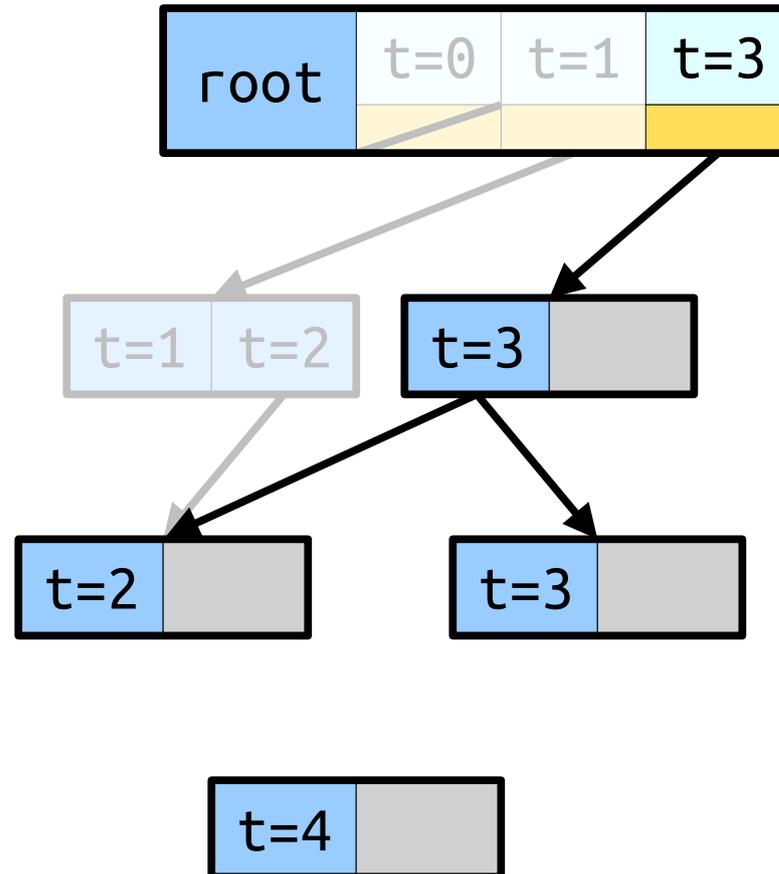
Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.
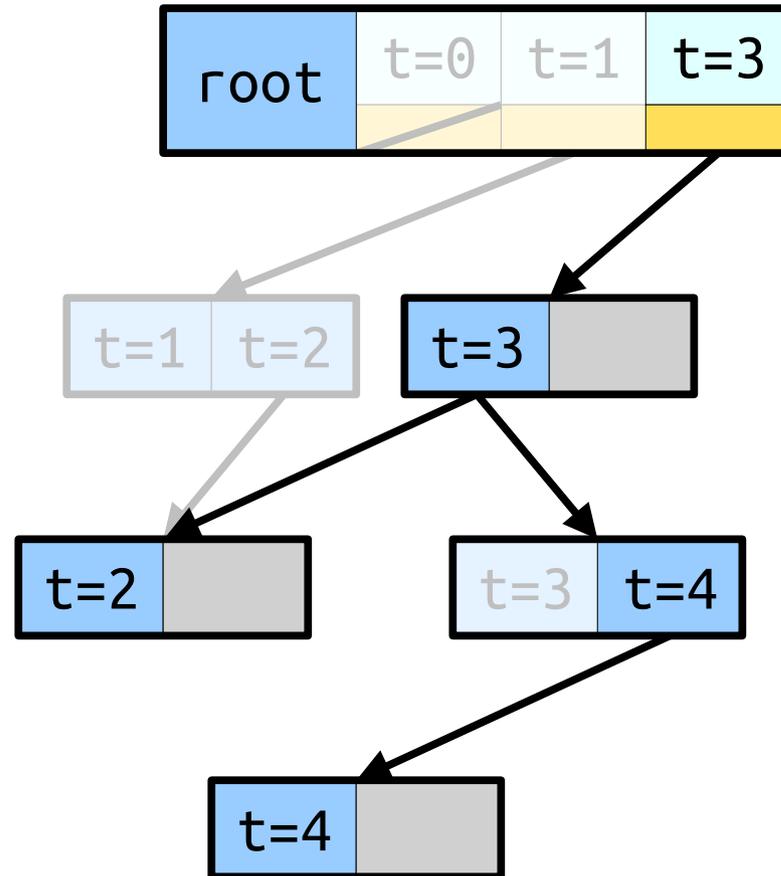
Fat Node Trees

- **_Idea:_** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

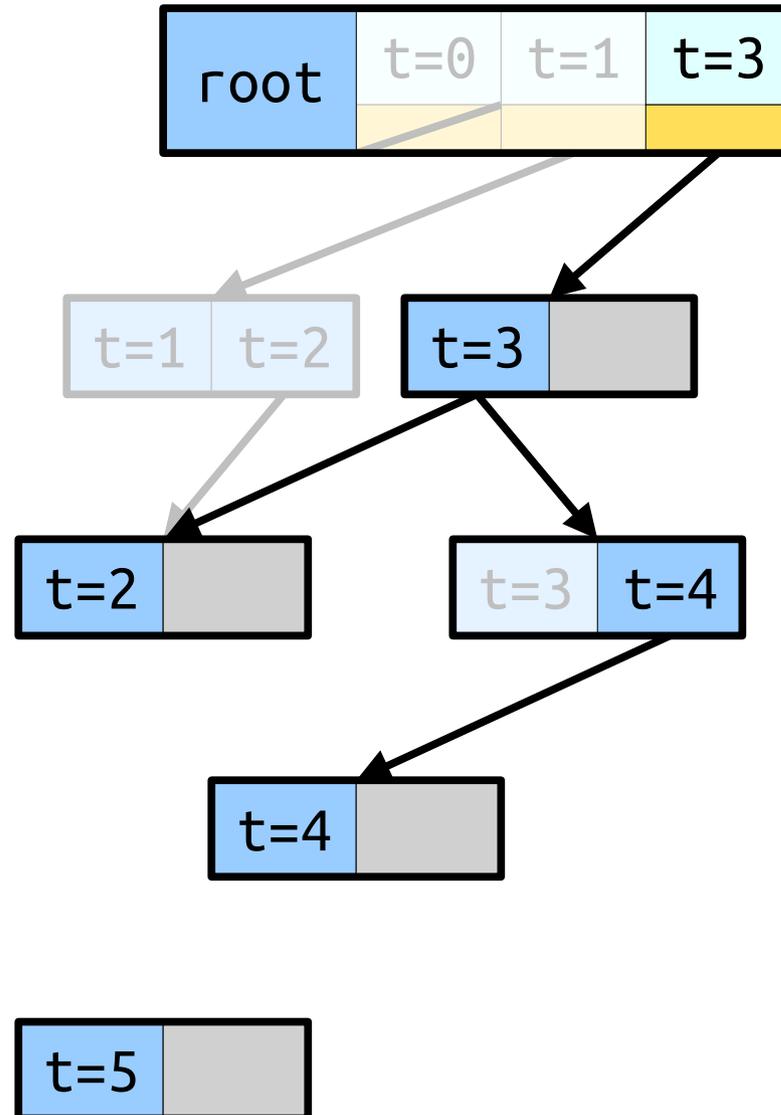# Fat Node Trees

- **_Idea:_** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

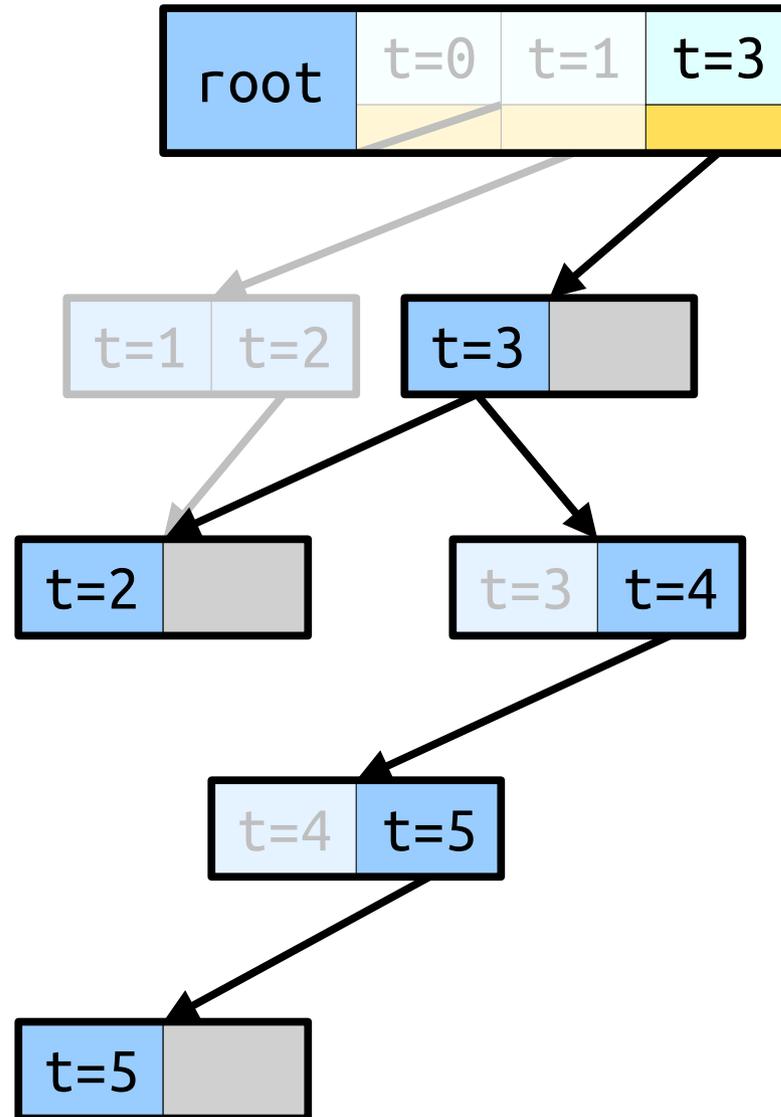# Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.
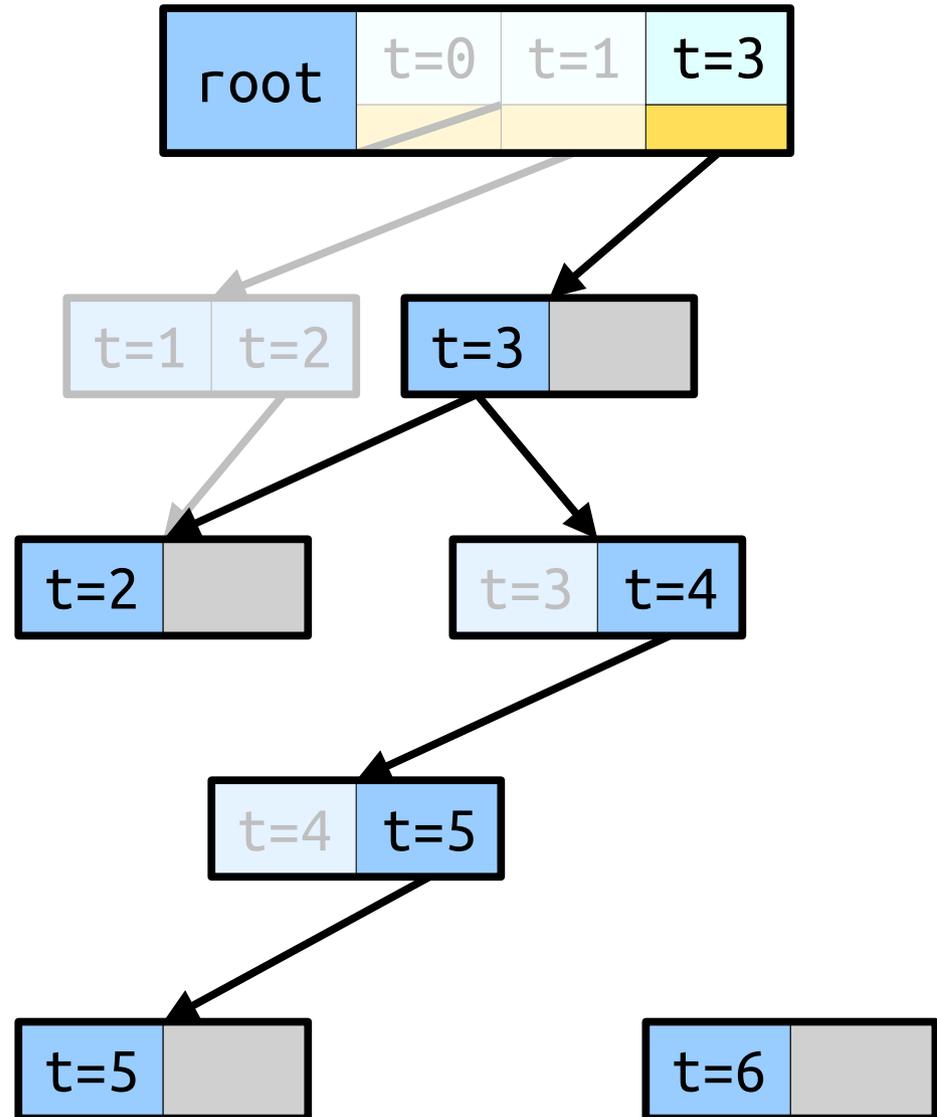
Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

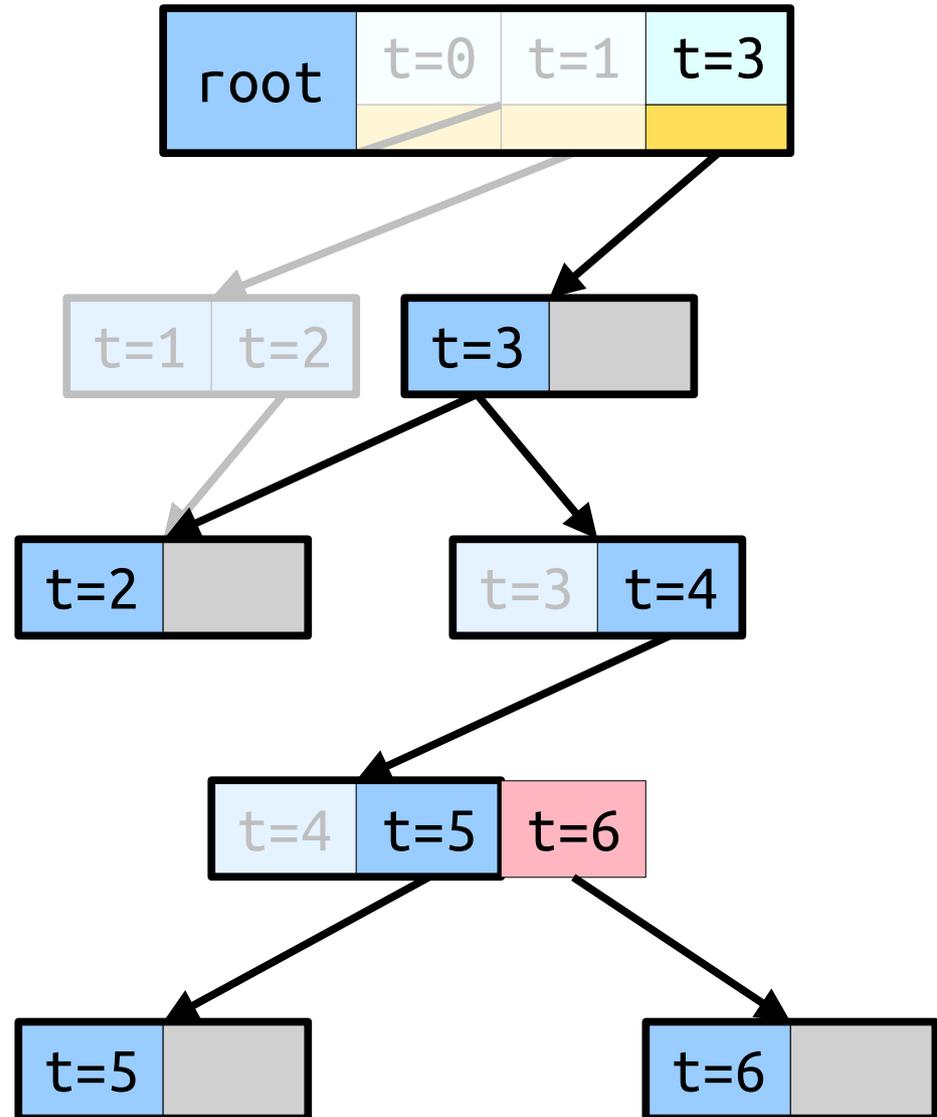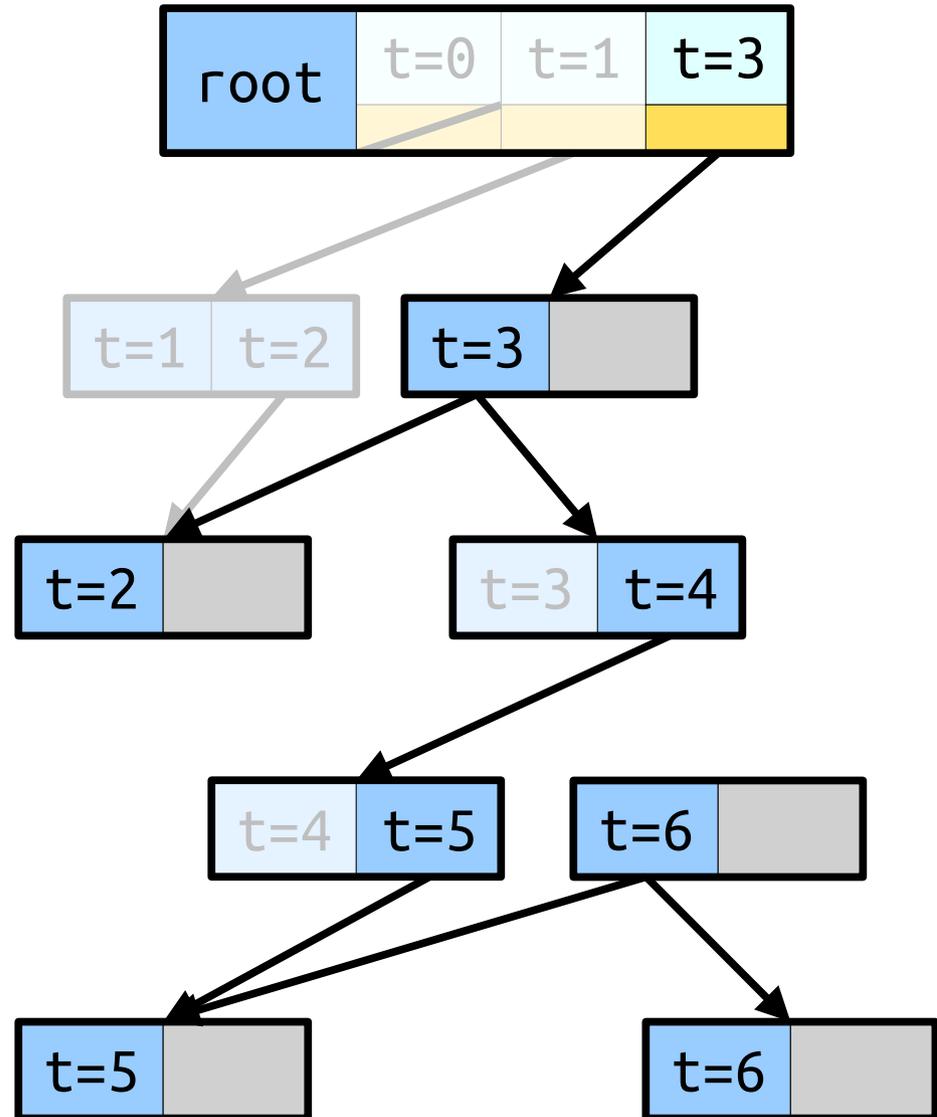# Fat Node Trees

- *Idea:* Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.
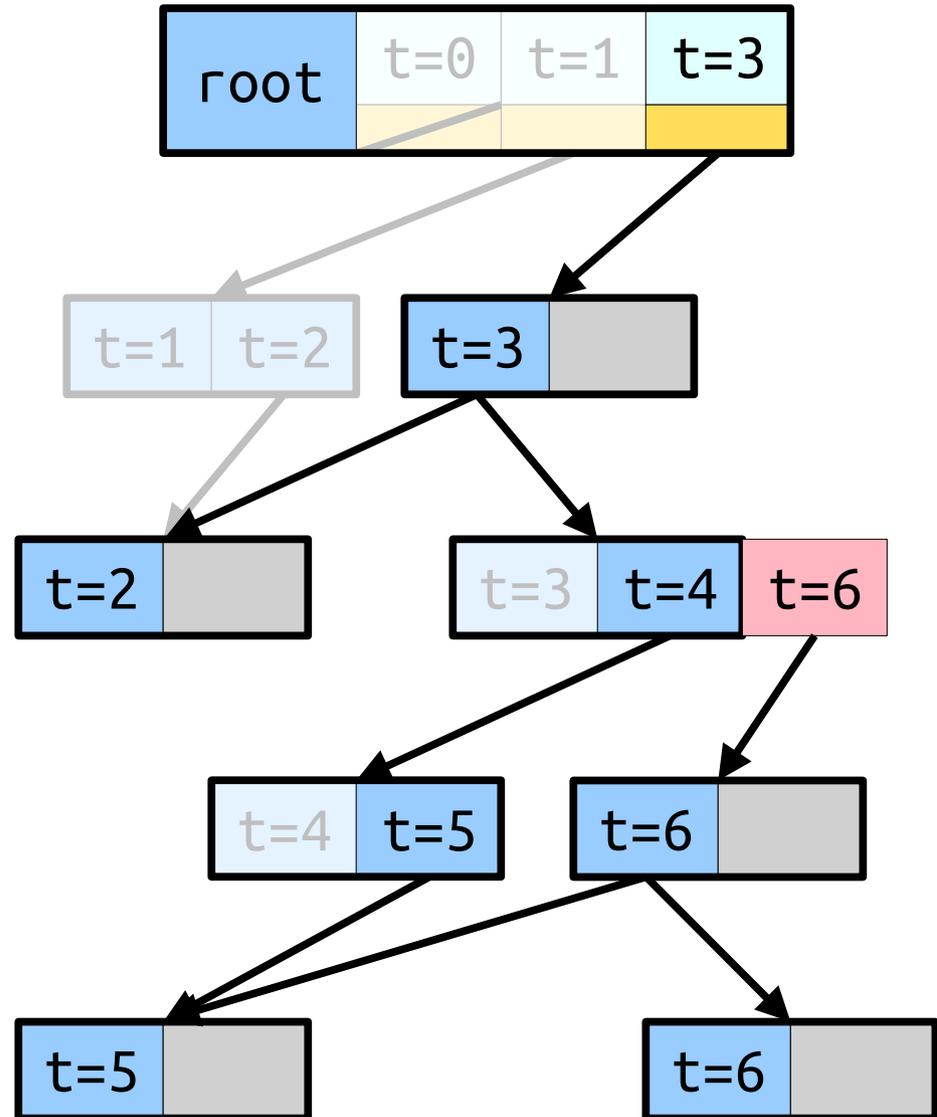
Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.
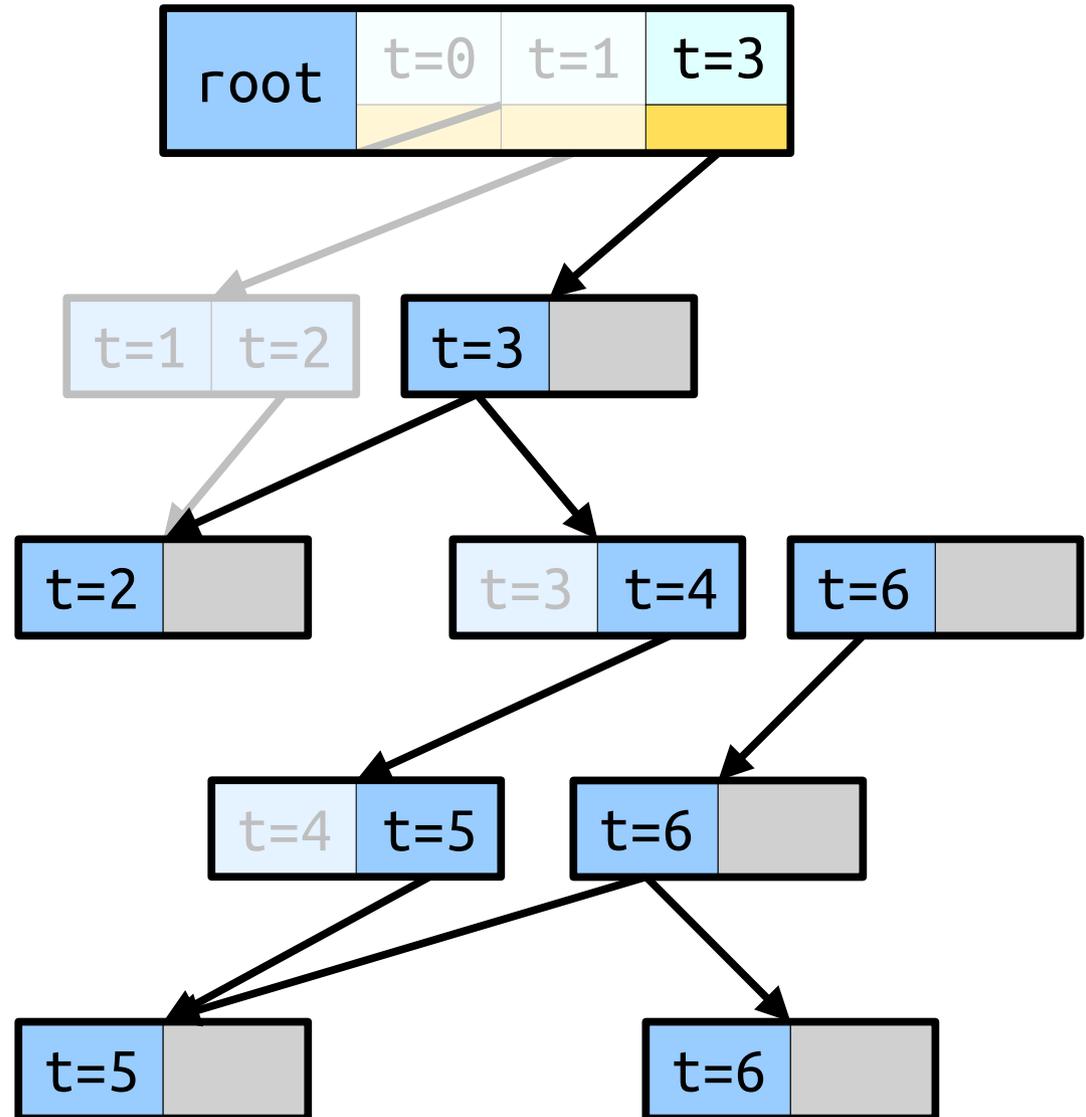
Fat Node Trees

- **_Idea:_** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

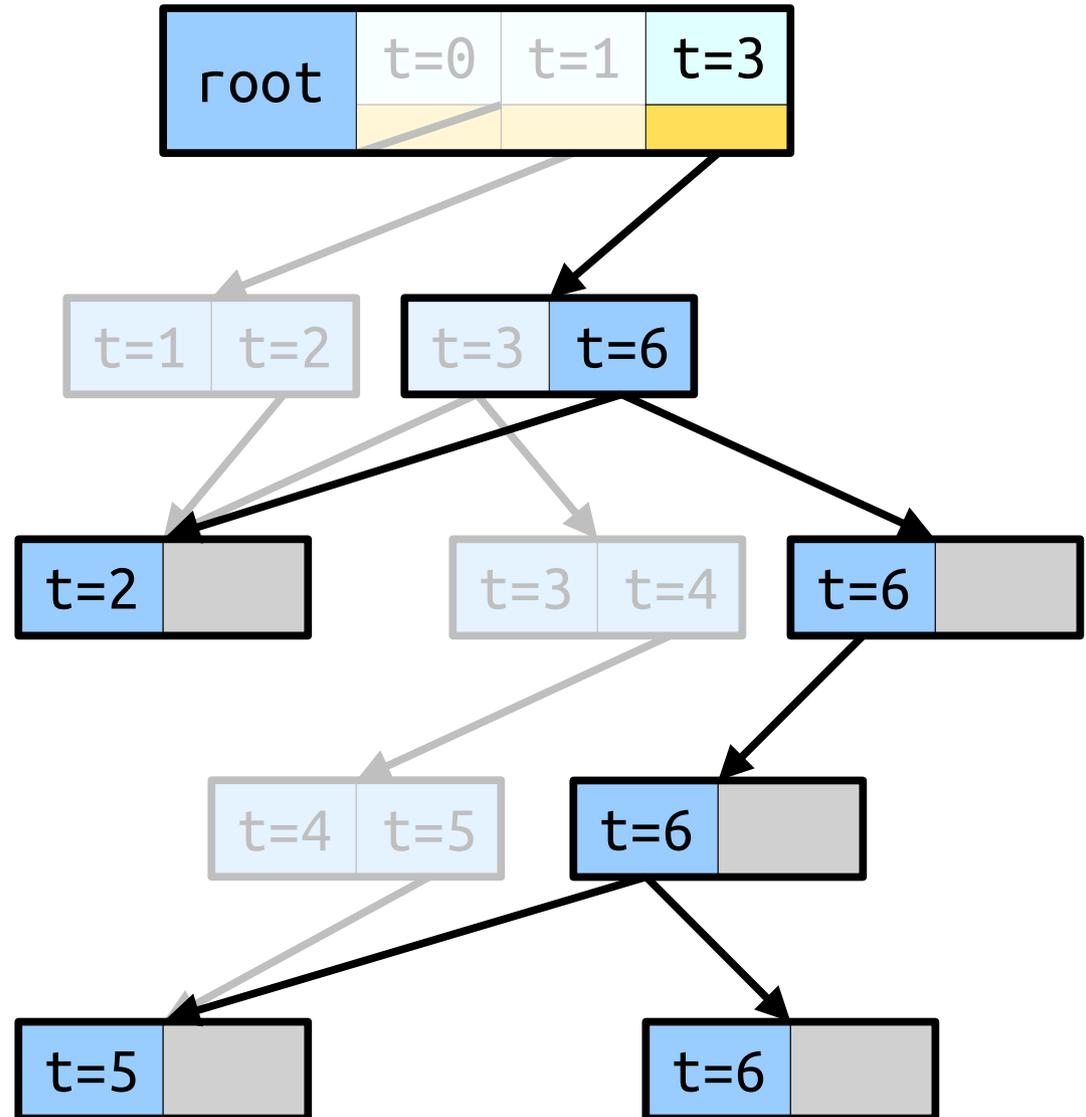# Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

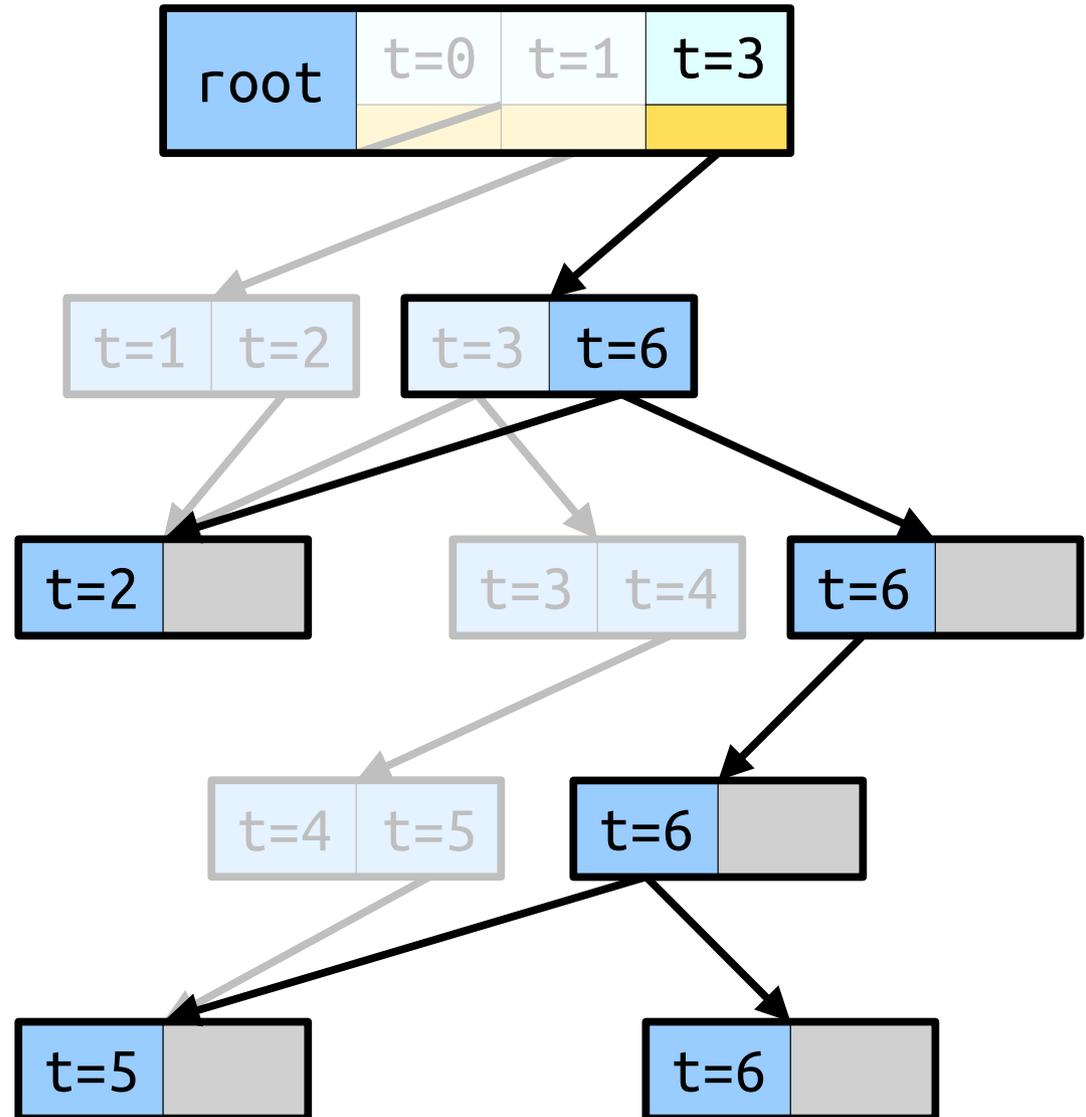# Fat Node Trees

- **_Idea:_** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

- To see the version at time t, binary search over the version number at the root, then use the normal journal idea to find the node you want.
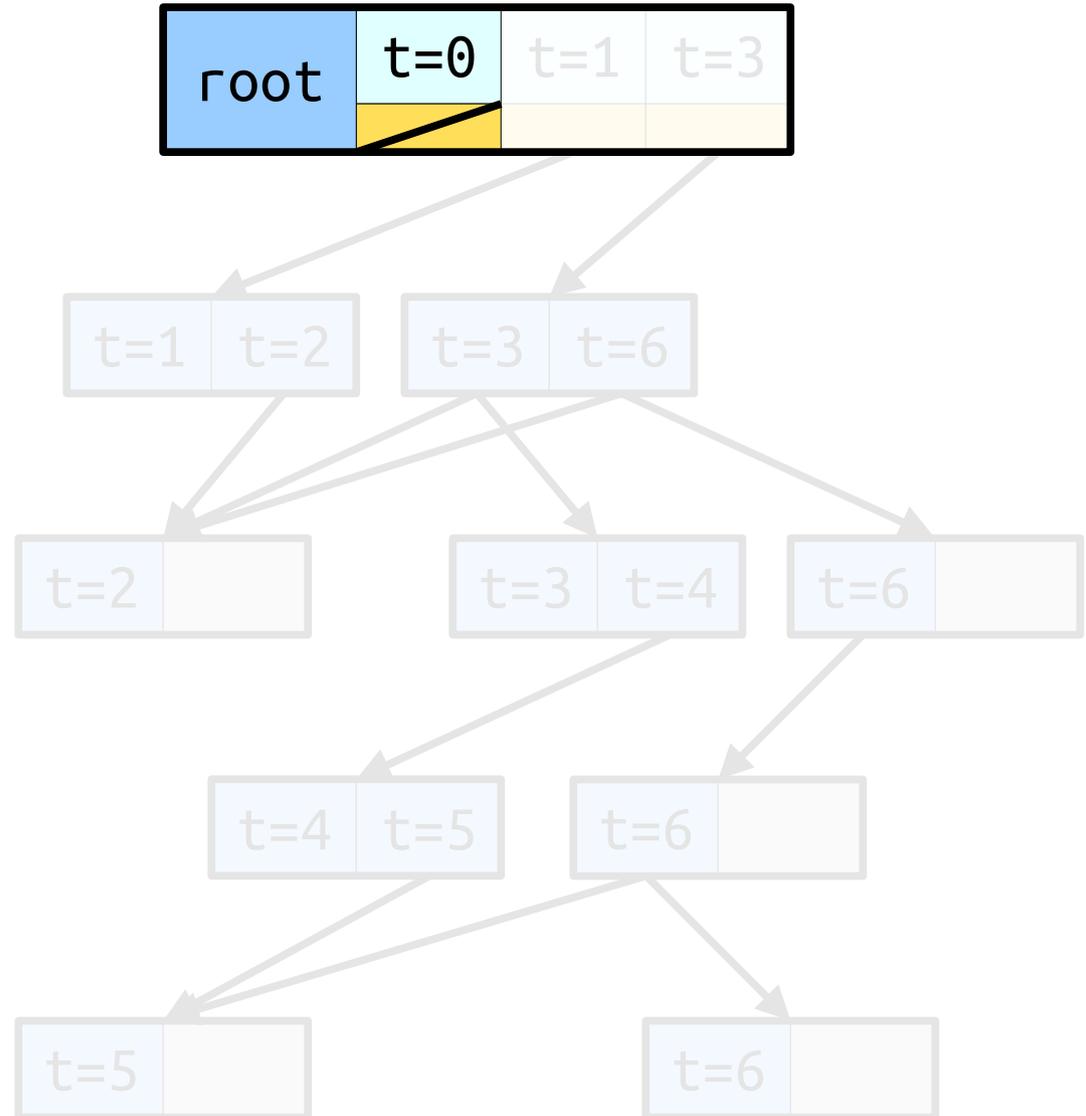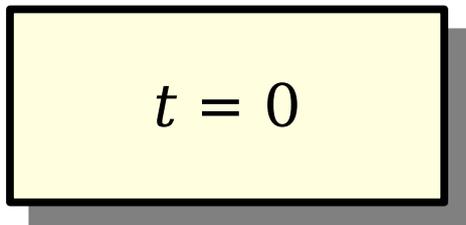
Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

- To see the version at time t, binary search over the version number at the root, then use the normal journal idea to find the node you want.
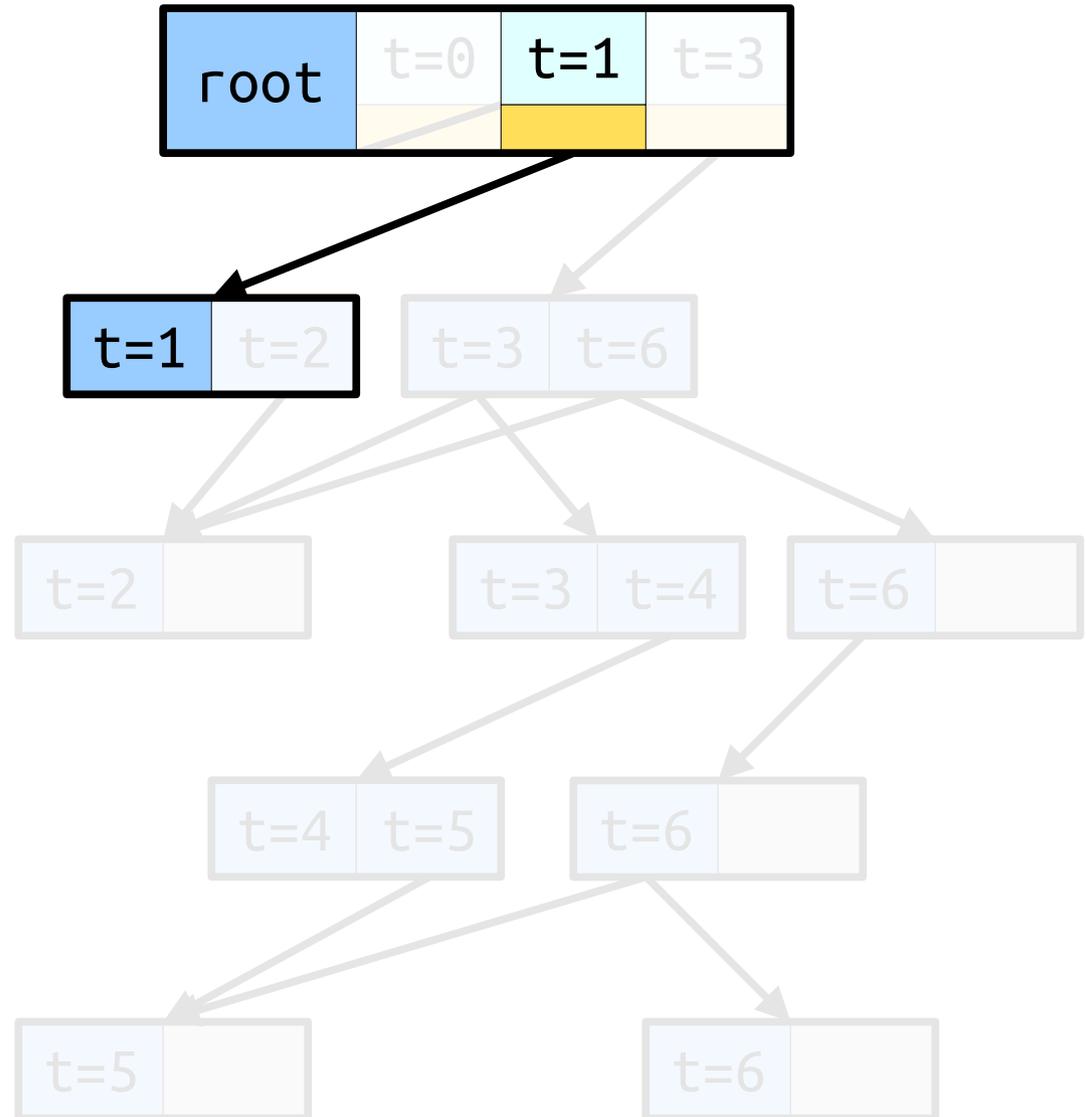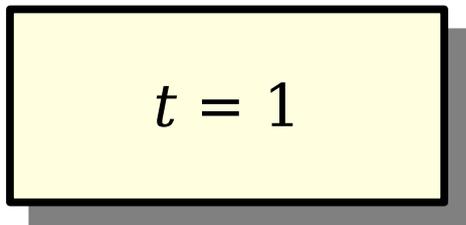
Fat Node Trees

- *Idea:* Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

- To see the version at time *t*, binary search over the version number at the root, then use the normal journal idea to find the node you want.
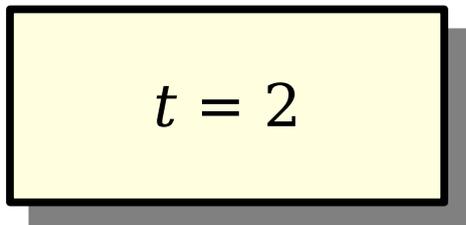
Fat Node Trees

- **Idea:** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

- To see the version at time t, binary search over the version number at the root, then use the normal journal idea to find the node you want.
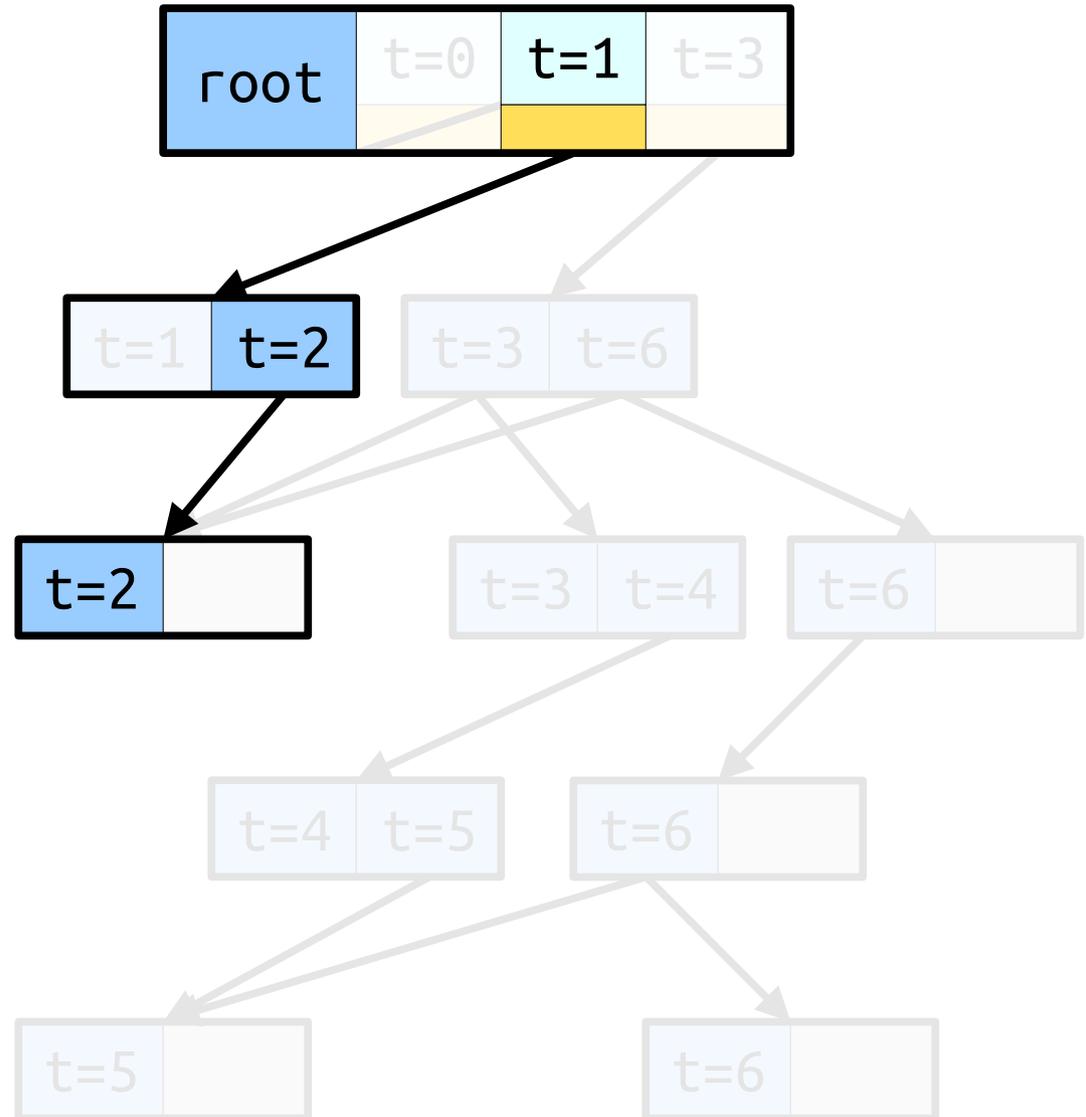
$t = 2$

Fat Node Trees

- *Idea:* Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

- To see the version at time *t*, binary search over the version number at the root, then use the normal journal idea to find the node you want.
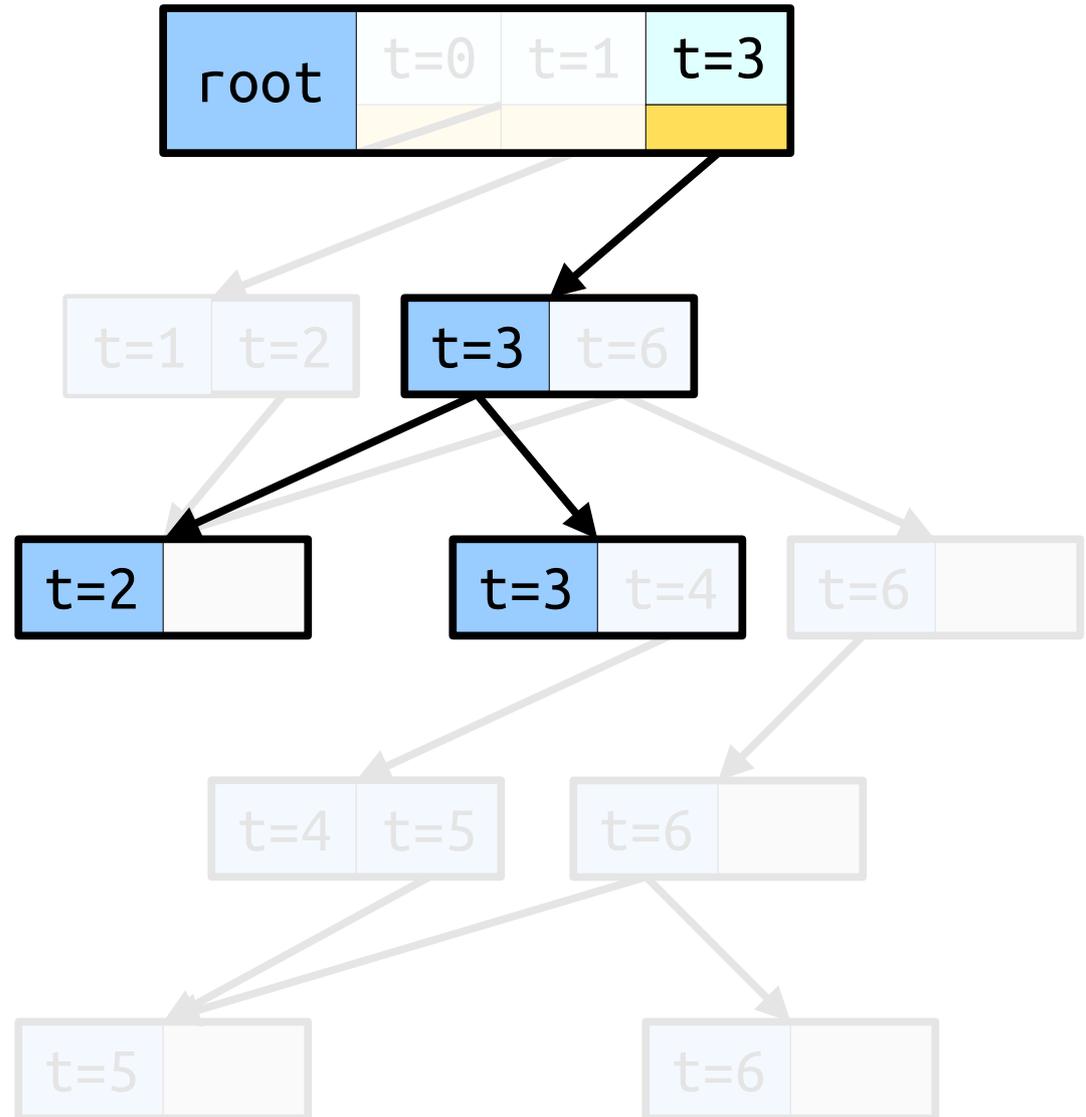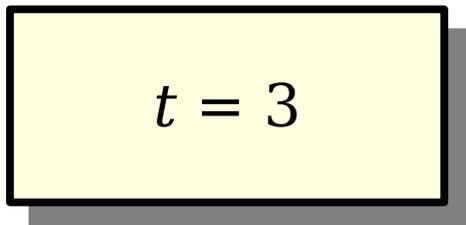

t = 3

Fat Node Trees

- **_Idea:_** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

- To see the version at time t, binary search over the version number at the root, then use the normal journal idea to find the node you want.
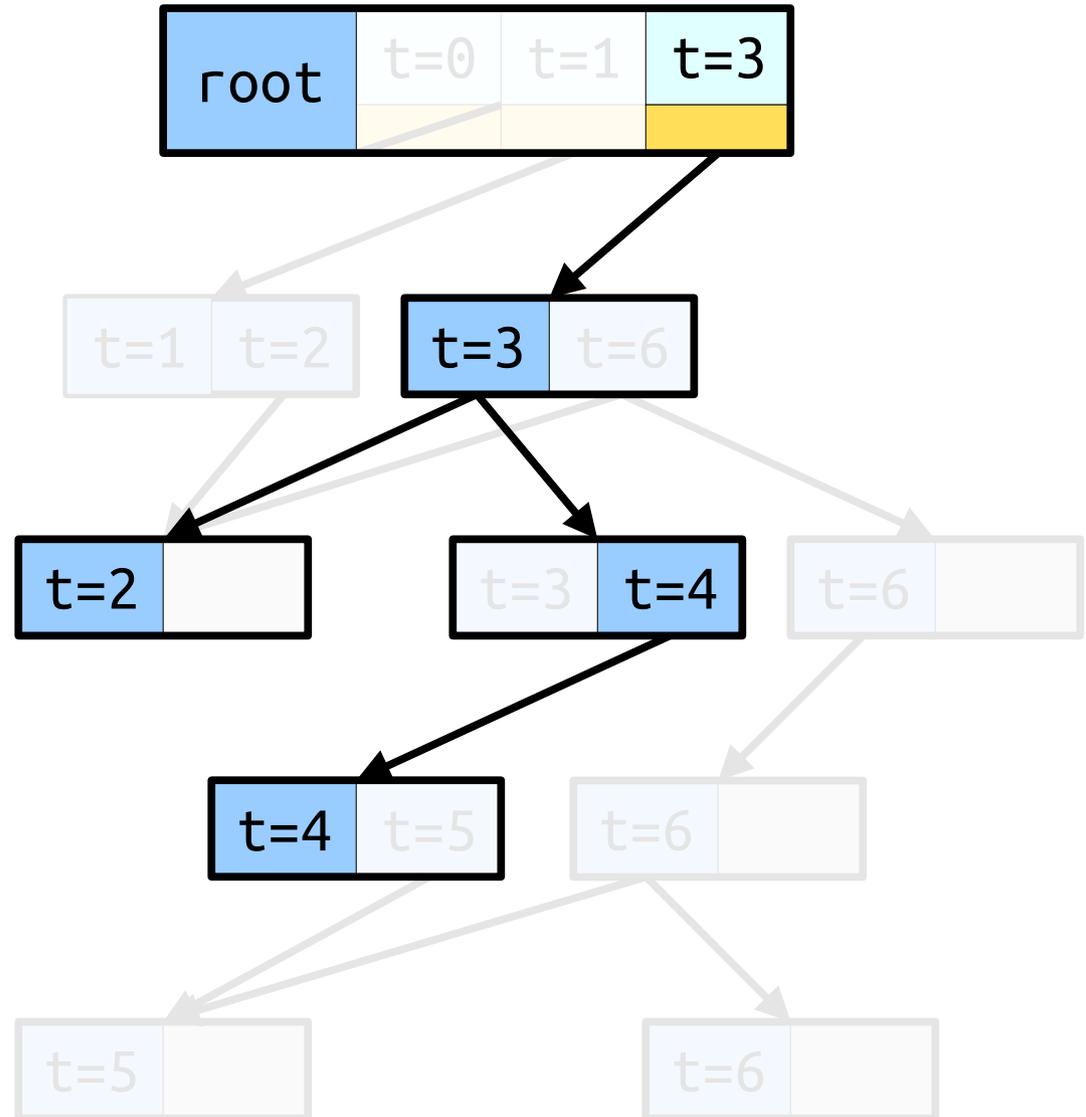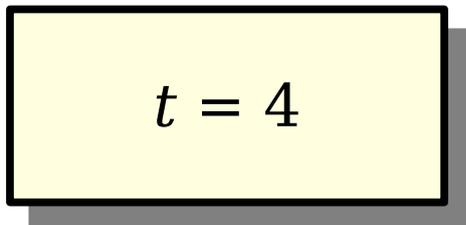
Fat Node Trees

- **_Idea:_** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

- To see the version at time t, binary search over the version number at the root, then use the normal journal idea to find the node you want.
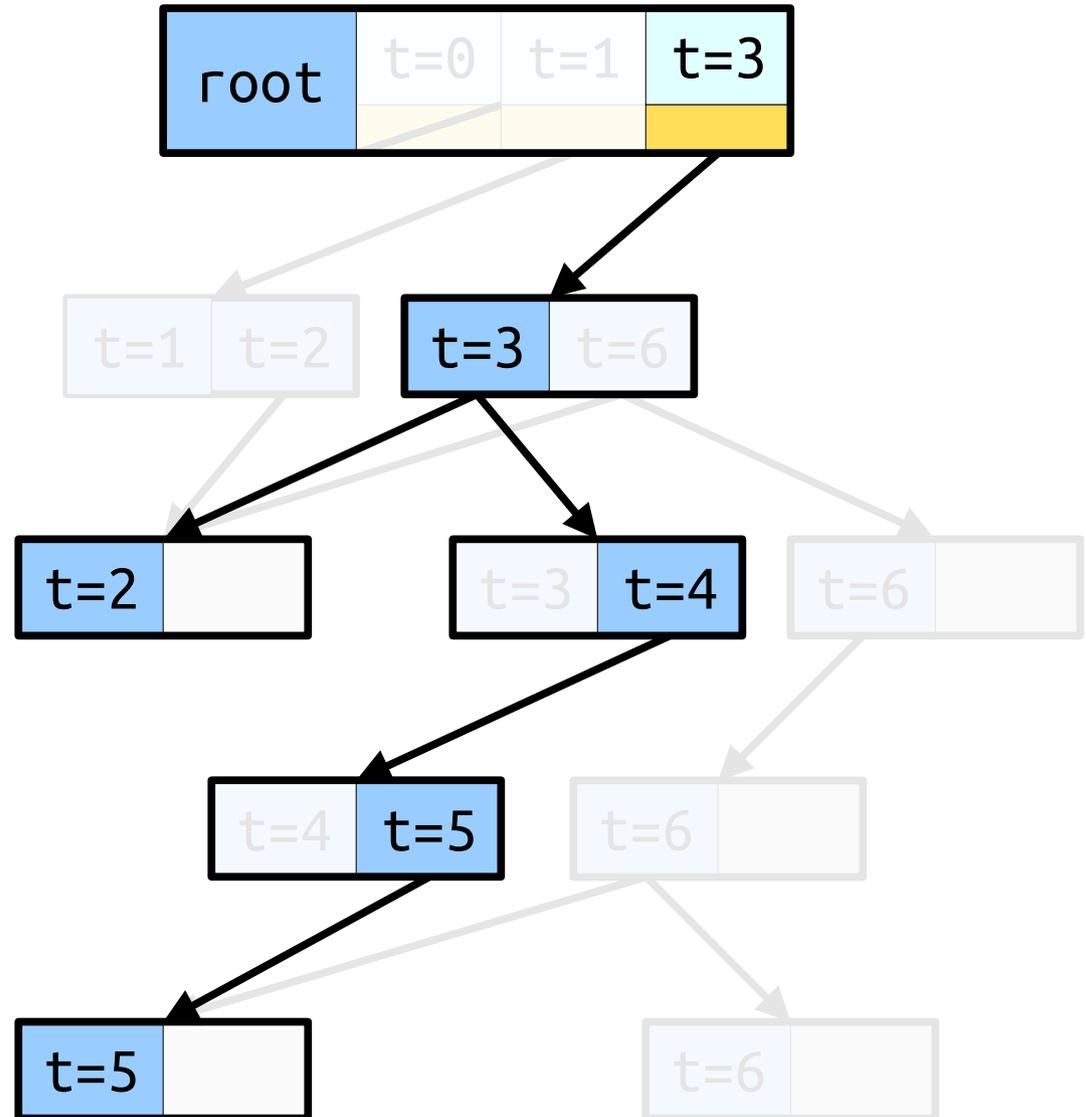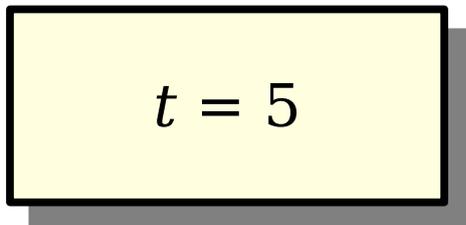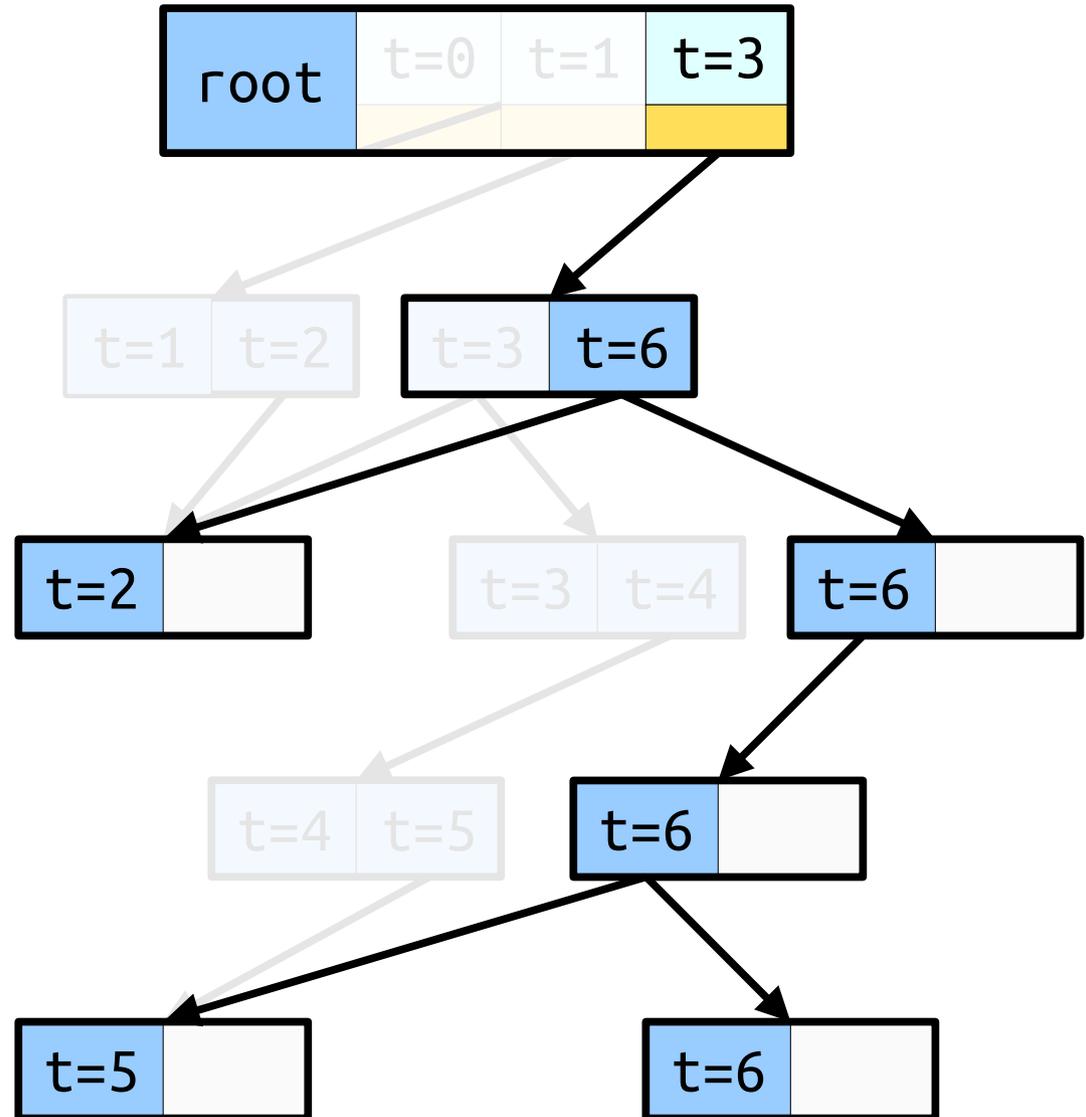
Fat Node Trees

- ***Idea:*** Use journals of size 2. If a node runs out of journal space, clone the node and (recursively) update the parent to point to the copy.

- To see the version at time *t*, binary search over the version number at the root, then use the normal journal idea to find the node you want.

Fat Node Trees
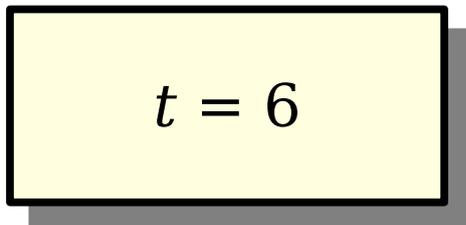
- ***Intuition:*** Use journaling to keep the space usage low, but don't let the journals get so large they slow down lookups.

- ***Intuition:*** Use path copying infrequently enough to not copy too many nodes, but enough to keep lookups fast.
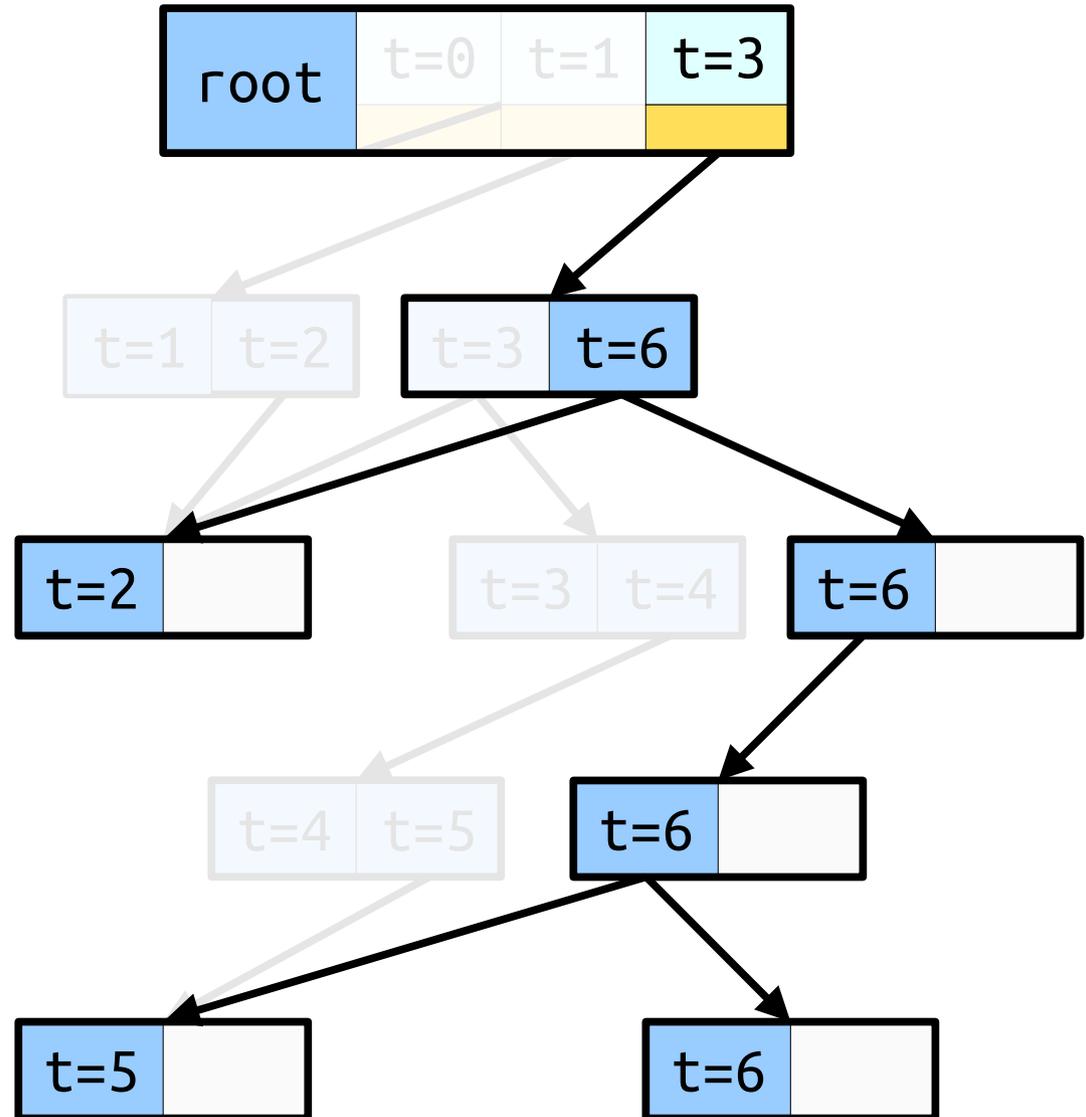
Fat Node Queries

- The cost of a lookup in a fat node tree is at most an O(1) factor slower than a regular lookup.

  - Have to potentially check one of two journal entries per node.

- Cost of a lookup in a fat node red/black tree: **O(log _n_)**.
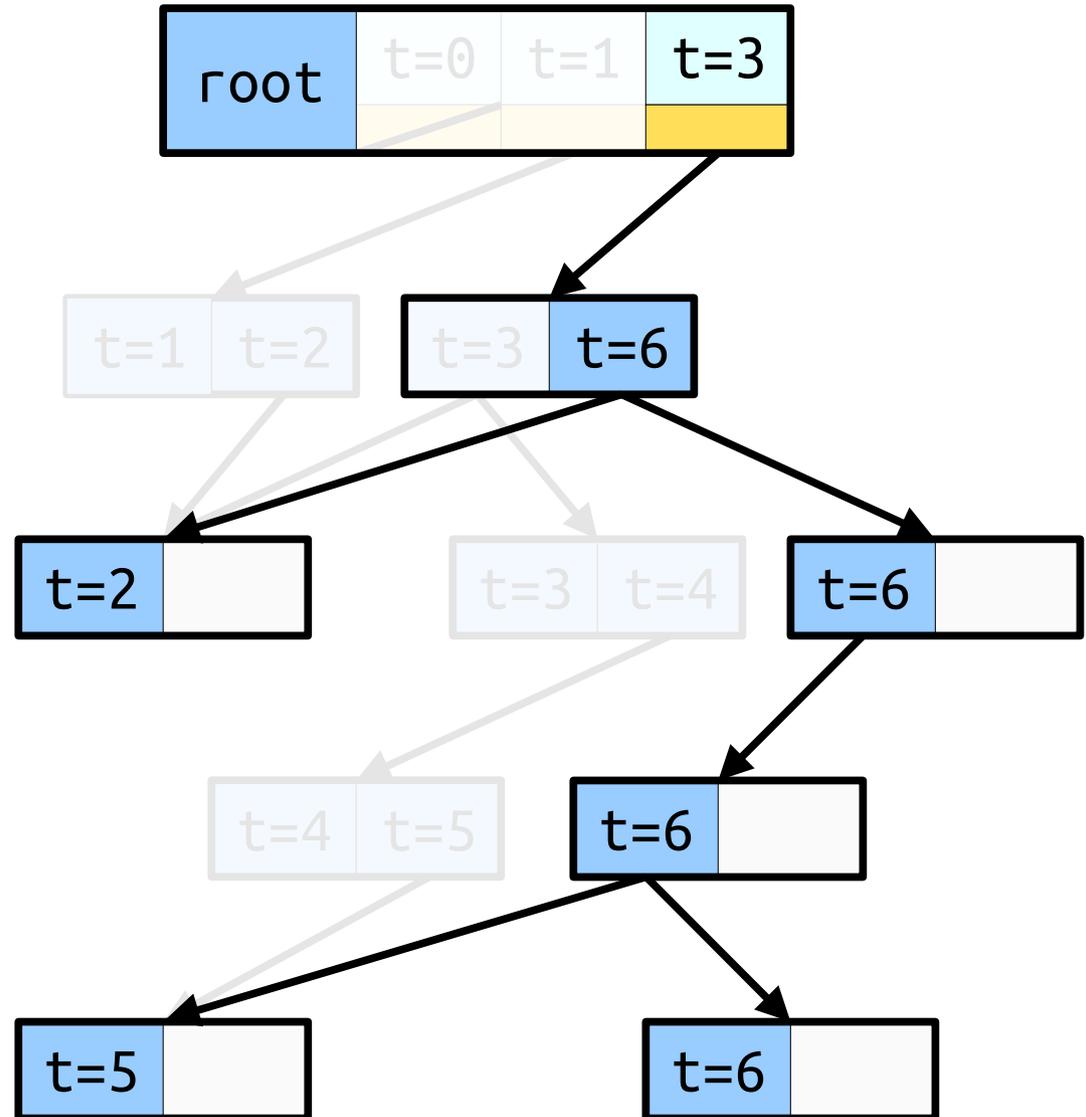
Fat Node Updates

- Most of the time, updates to a fat node red/black tree will not require any nodes to be copied.

- Every now and then, we have to copy a node one level above us.

- Even more rarely, we have to copy a node two levels above us.

- *Claim:* The amortized work to do an update in a fat node red/black tree is O(1).
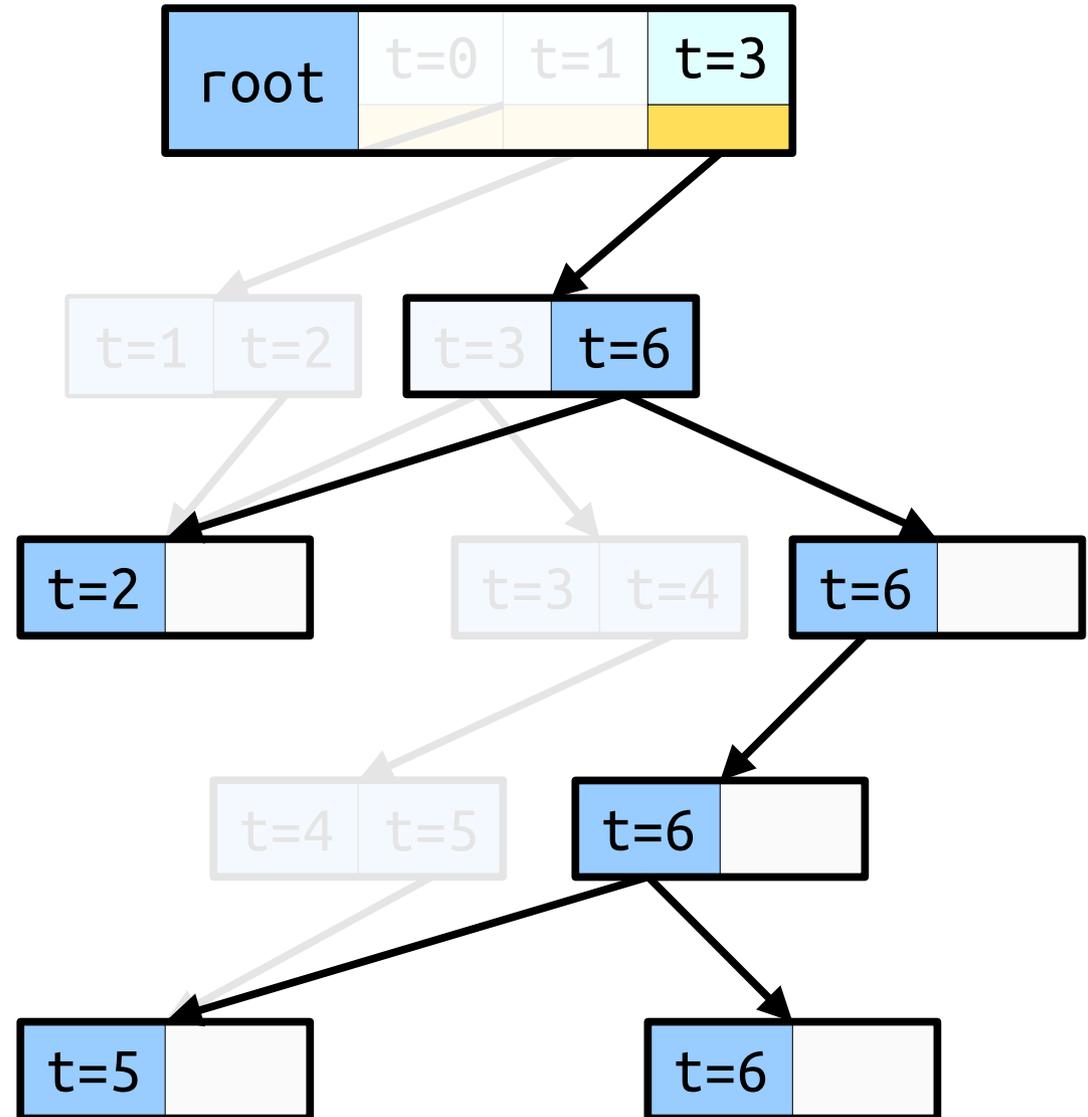
Fat Node Updates

- Define Φ to be the number of nodes with two journal entries reachable from the most recent timestamp.

- Suppose we copy k nodes during an update, for a total of $\Theta(k)$ work.

- If we copy a chain of k nodes, then
  - each of those k nodes had two journal entries before we started;
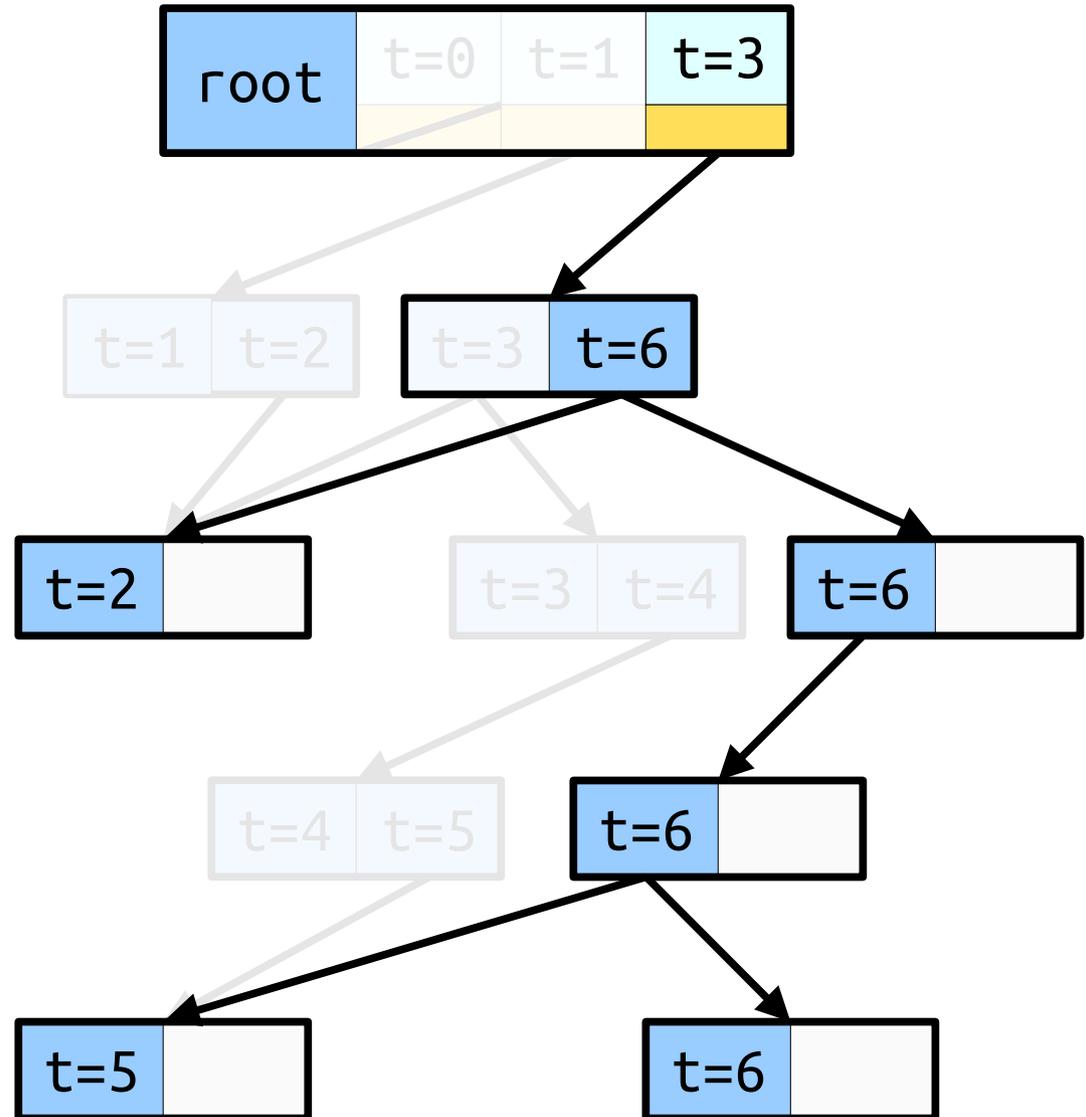 - each of those k nodes is no longer reachable from the most recent timestamp;
  - each newly created node has only one journal entry; and
  - the node above the chain has one more journal entry added.

- So $\Delta\Phi = 1 - k$, and the $-k$ cancels with the $\Theta(k)$ work for an amortized update cost of **O(1)**.

The Final Scorecard

- We've come a long way from where we started!
- Pretty much all the work went into efficiently sharing data across different versions of data structures.

	Preprocessing Time	Query Time	Space Usage
Test All Faces	$O(n \log n)$	$O(n)$	$O(n)$
Slab Decomposition	$O(n^2)$	$O(\log n)$	$O(n^2)$
Slabs With PFRBTs	$O(n \log n)$	$O(\log n)$	$O(n \log n)$
Slabs With Journal RBTs	$O(n \log n)$	$O(\log^2 n)$	$O(n)$
Slabs With Fat Node RBTs	$O(n \log n)$	$O(\log n)$	$O(n)$

More to Explore

Planar Point Location

- The ***monotone chains method*** breaks the planar subdivision apart into a series of left-to-right chains, then uses clever binary searches to meet the same time bounds as here.

- ***Kirkpatrick's algorithm*** works with fully-triangulated planar subdivisions. It repeatedly removes vertices to simplify the triangulation, giving a recursive point location algorithm.

- The ***random trapezoidal method*** breaks space apart into trapezoids in a manner similar to slabs. Using randomization, the query cost and space usage can be shown to be low.

Persistent Data Structures

- The FPRBT, journal RBT, and fat node RBT are special cases of ***persistent data structures***, where operations produce both a "before" and "after" version.

- The three methods shown here (purely functional, journaling, and fat nodes) generalize to many other data structures.

- With some more advanced techniques, you can design ***retroactive data structures***, where operations can be sent back in time.

- There are also ***confluent data structures***, where multiple timelines can be merged together.

Concluding Thoughts

Where We've Been

- What a whirlwind tour of data structures this has been!

RMQ · Red/Black Trees · B-Trees · Tree Augmentation · Count-Min Sketches · Count Sketches · HyperLogLog · Cuckoo Hashing · Two-Stack Queues · Scapegoat Trees · Tournament Heaps · Abdication Heaps · Tries · Suffix Trees · Suffix Arrays · Succinct Rank Queries · x-Fast Tries · y-Fast Tries · Word-Level Parallelism · Sardine Trees · Fusion Tress · Planar Point Location

- We've covered topics that span from the early days of computing through developments of the past few years.

Where We've Been

- Over the course of the quarter, we've seen some beautiful problem-solving strategies:

  Solve All Possible Small Problems

  Use Isometries

  Replicate to Boost Confidence

  Harness Properties of Random Graphs

  Add Wiggle Room and Defer Cleanup

  Harness Mechanical and Operational Perspectives

  Break Big Problems into Small Blocks

  Find Parallelism in Unexpected Places

  Make Every Bit Count

- These approaches have applications far beyond data structure design.

What Comes Next

- There's so much more to explore in the world of data structures!

  Querying a Graph as it Changes · Persistent Data Structures · Data Structures for Parallel Architectures · Harnessing Caches Without Knowing Their Sizes · The Quest for the Best BST · The Quest for the Best Hash Table · Finding Lower Bounds on Data Structure Performance · Data Structures for Objects in Motion · Approximate Maps and Sets · Self-Adjusting Data Structures · Data Structures for Planar Graphs · Data Structures for Finding all Points in a Region · Data Structures for Nearest-Neighbor Searching · Data Structures for Text Editors · Data Structures for Functional Programming Languages · Data Structures from Number Systems · Data Structures for High-Performance Databases · ...

- There's easily enough material here for a CS166 sequel. Let me know if you'd be interested in that!

What Comes Next

- Resources for the Future:
  - Erik Demaine's Advanced Data Structures course at MIT.
  - Jeff Erickson's Advanced Data Structures course at UIUC.
  - David Eppstein's Data Structures course at UC Irvine.
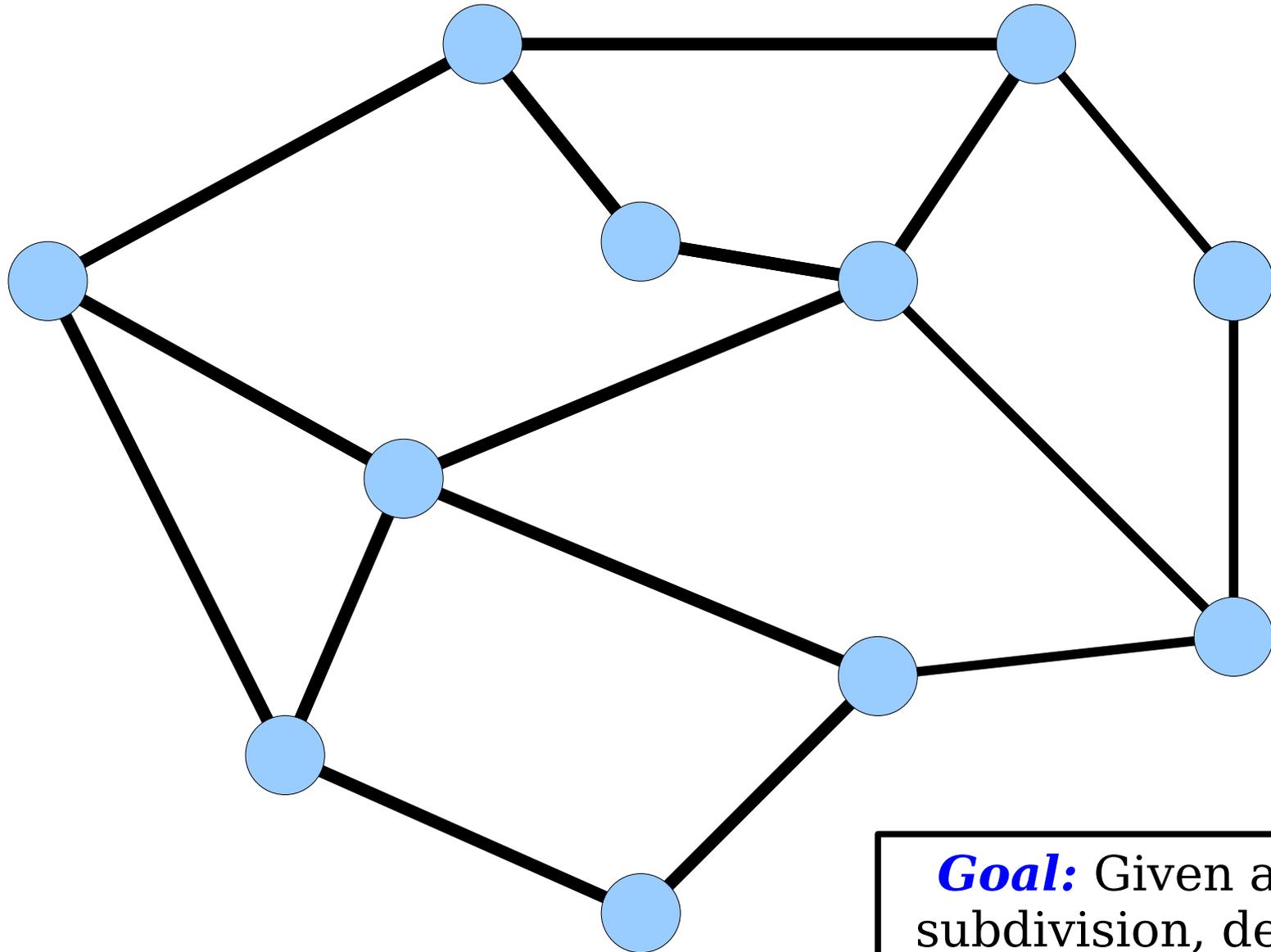- Conferences to Watch:

  STOC   SODA   FOCS

- And you can always ping me directly!

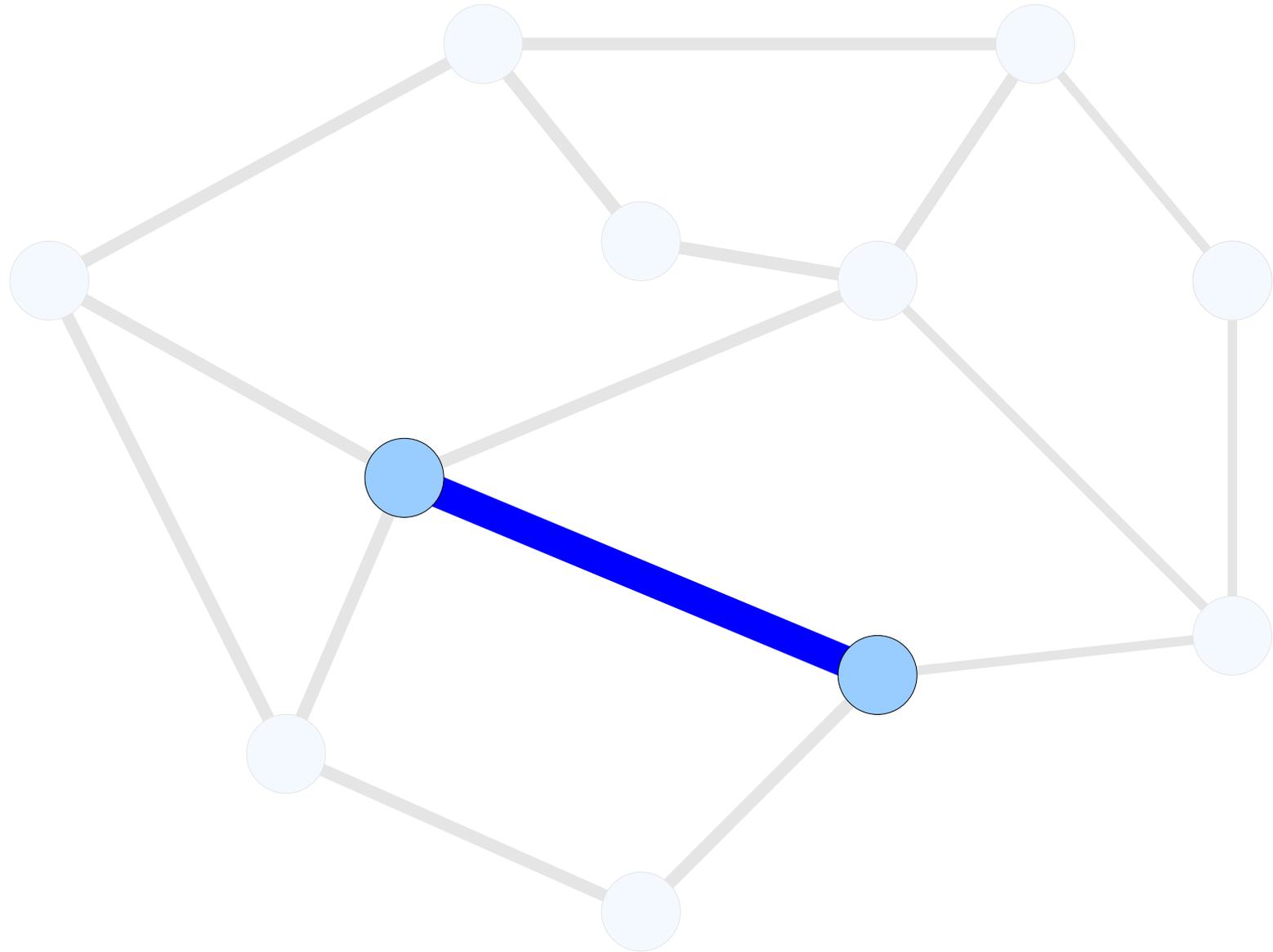Keep asking if we can improve on our existing approaches.
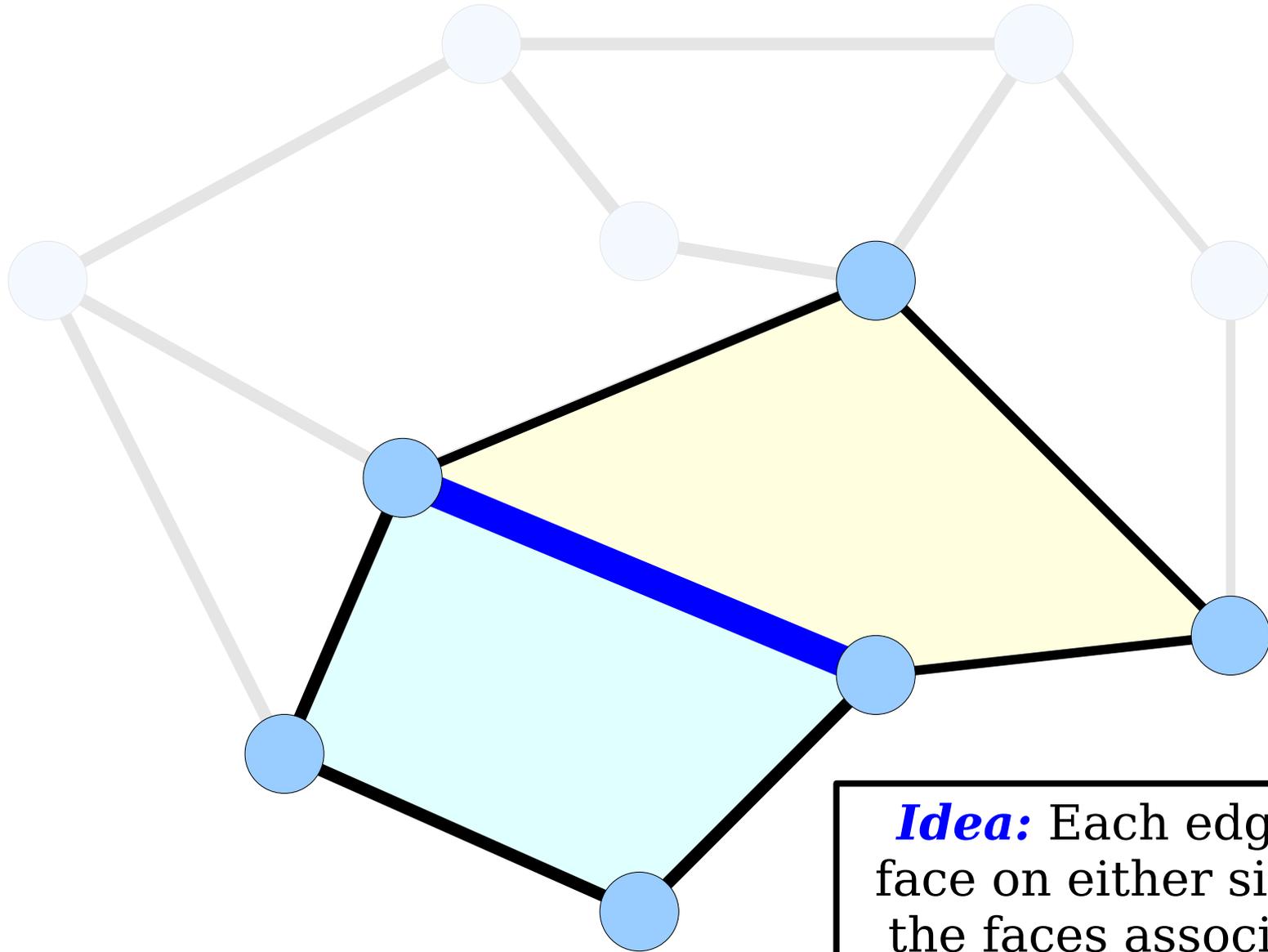
Stay curious and maintain your sense of wonder.

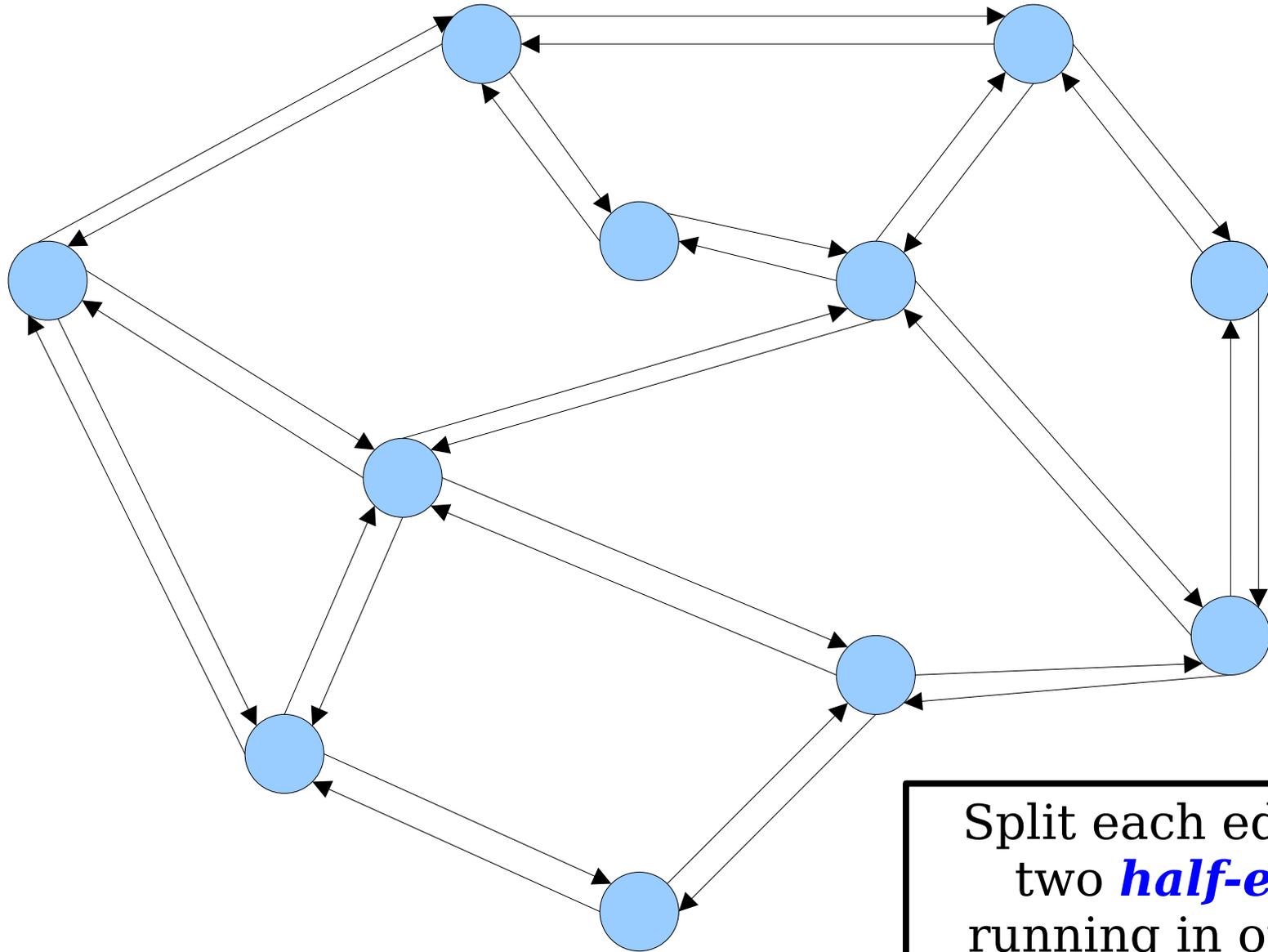Keep in touch! Best of luck going forward.

Appendix: Associating Faces to Edges

Goal: Given a planar subdivision, determine its faces.

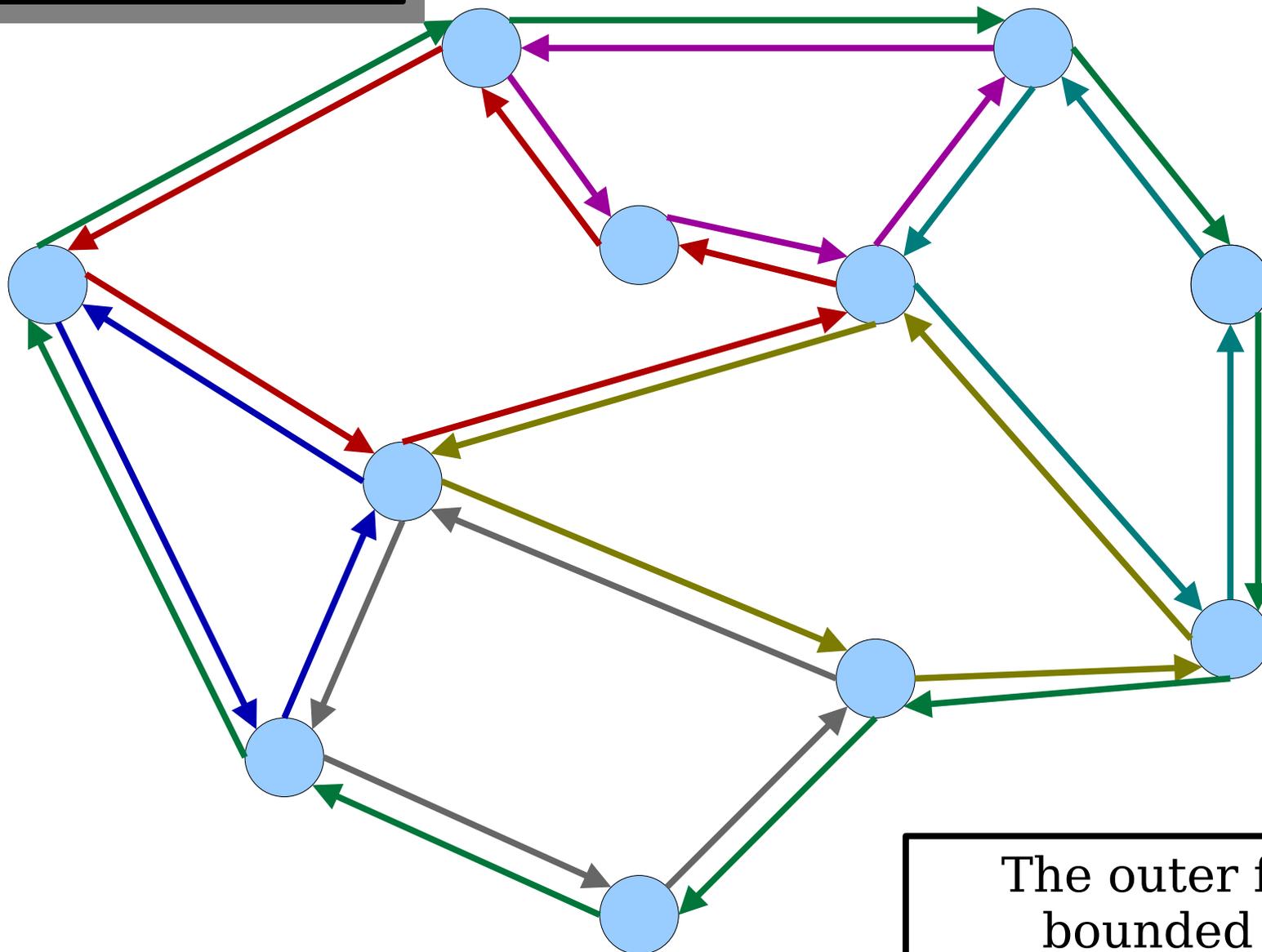**Idea:** Each edge has a face on either side. Find the faces associated on each side of each edge.

Split each edge into
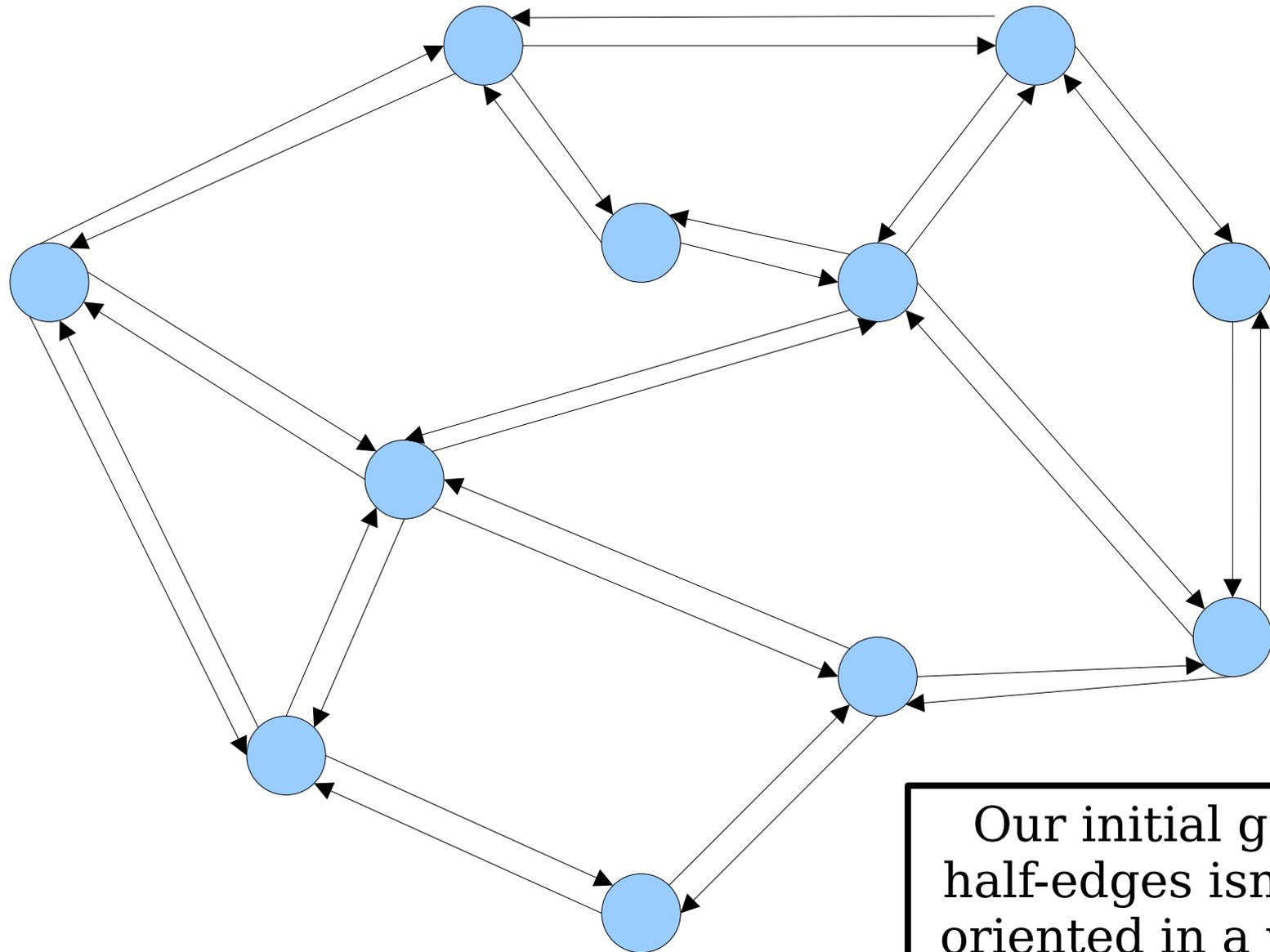two ***half-edges***
running in opposite
directions.
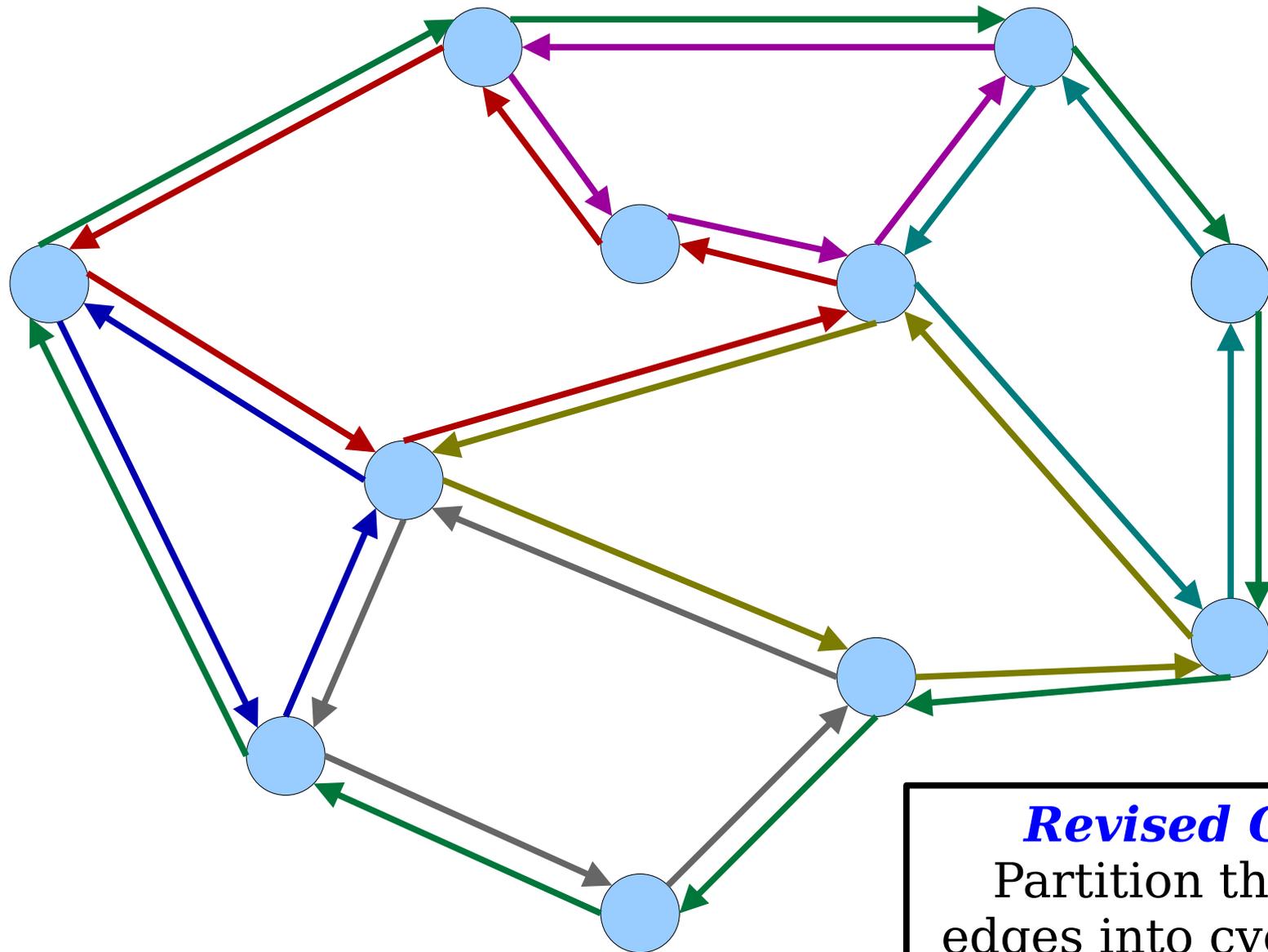
Revised Goal:
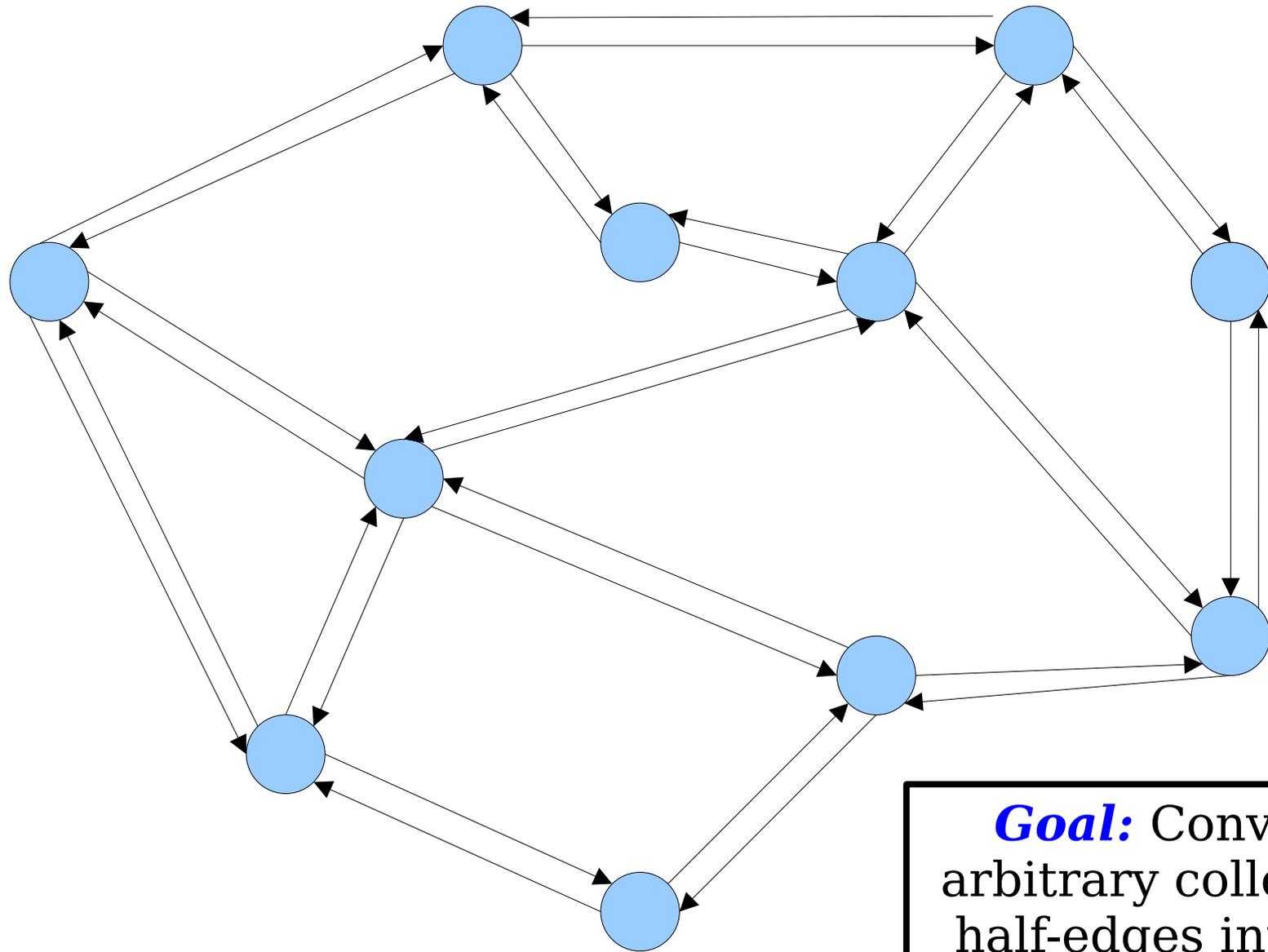Partition the half-edges into cycles that define each face.

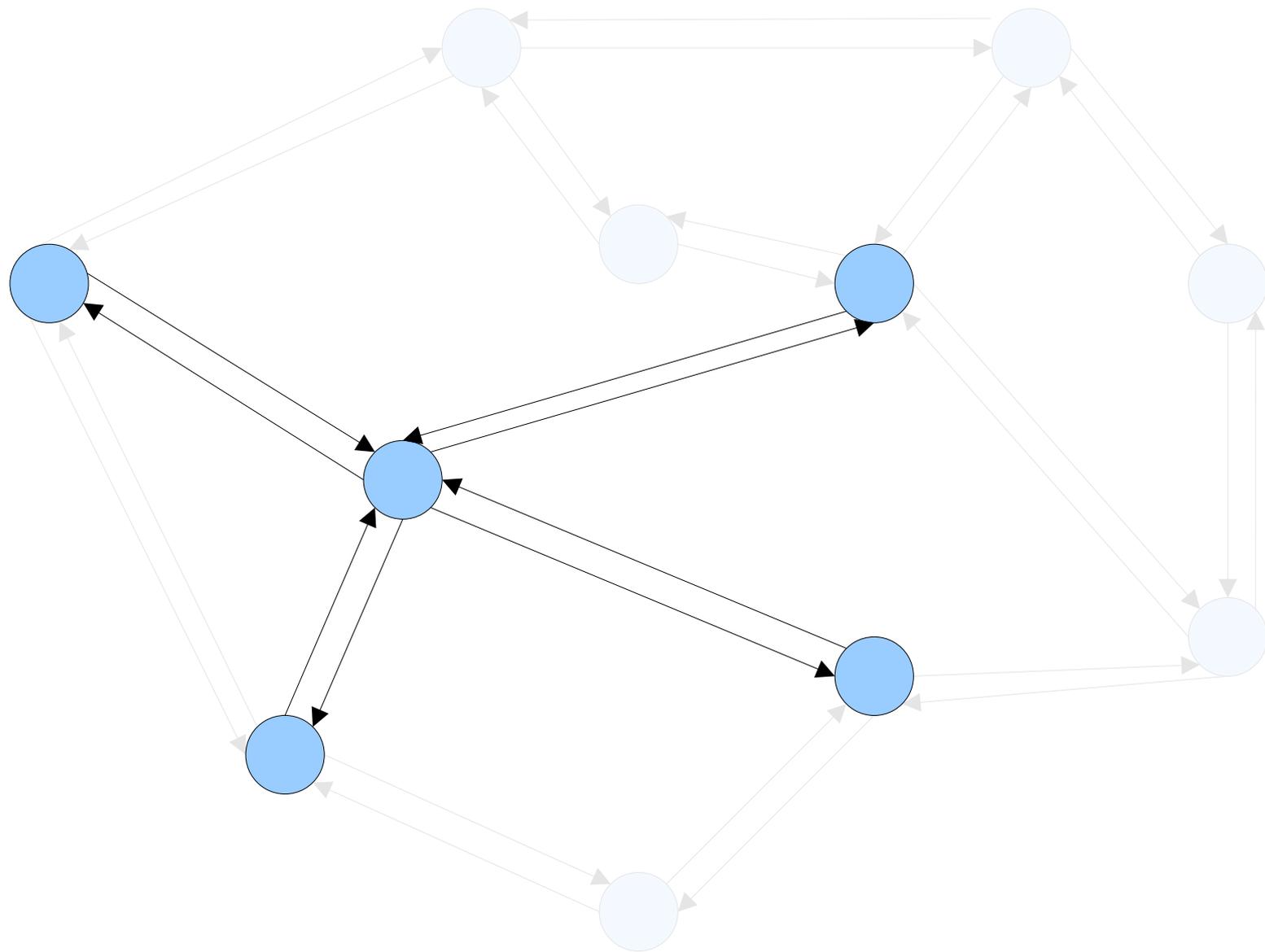Internal faces are bounded by anticlockwise loops.
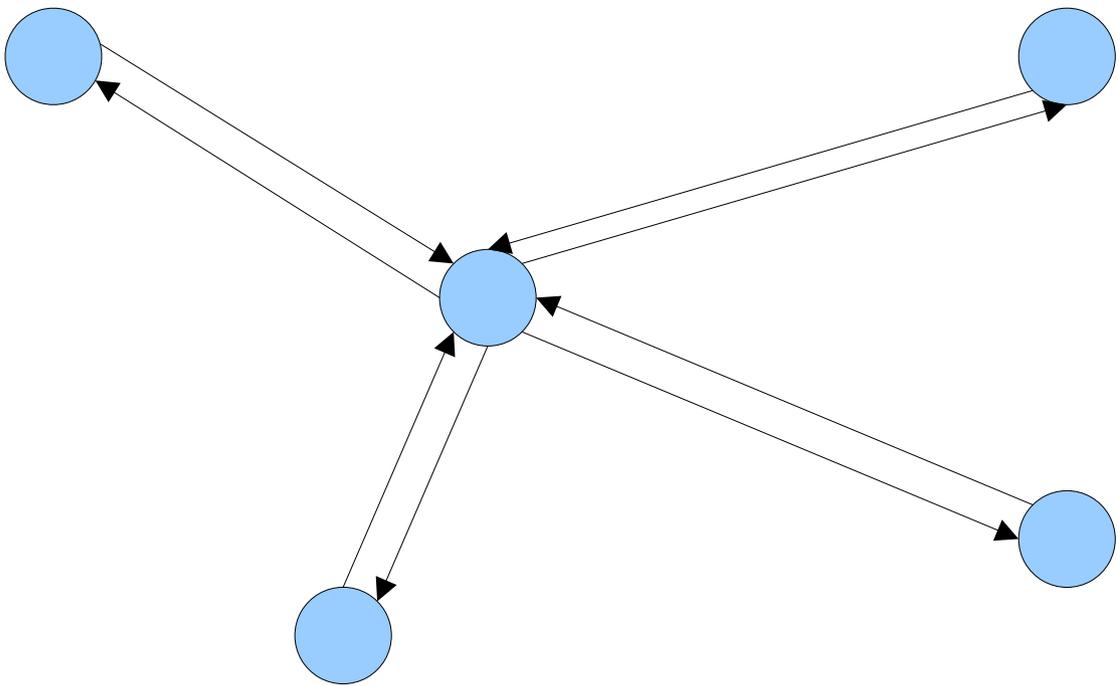
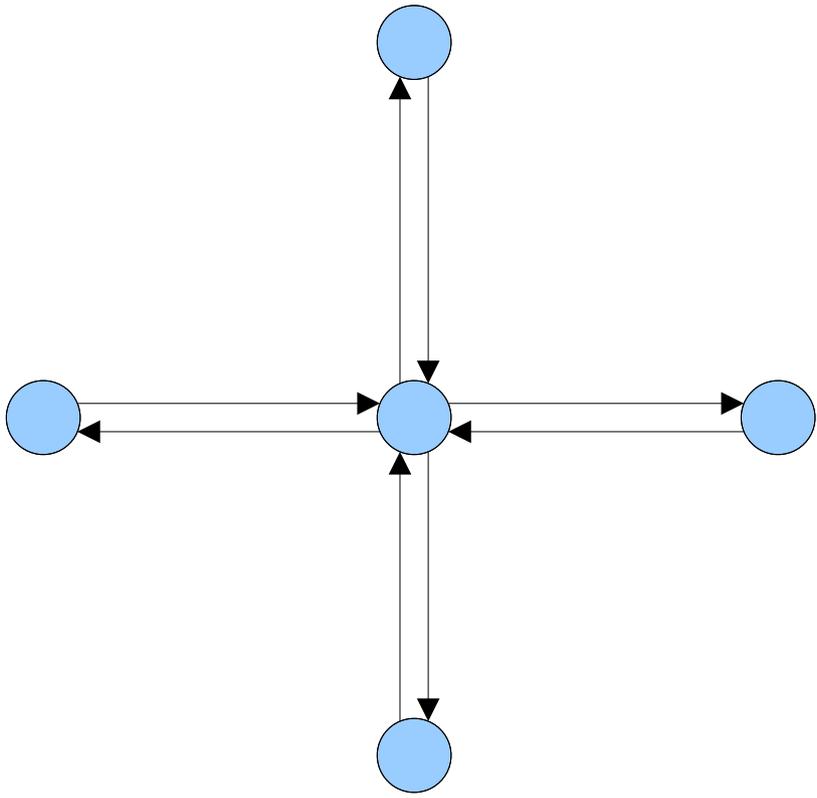The outer face is bounded by a clockwise loop.

Our initial group of half-edges isn't nicely oriented in a way that defines the faces.
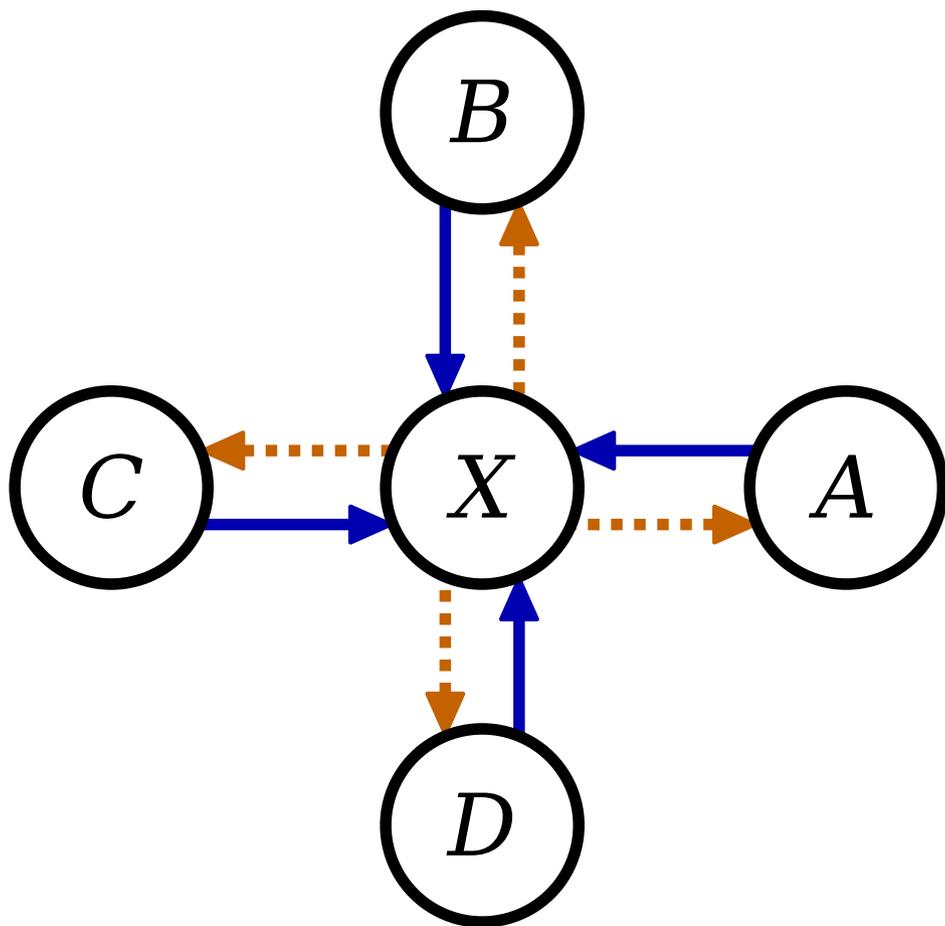
Goal: Convert an arbitrary collection of half-edges into loops bounding faces.
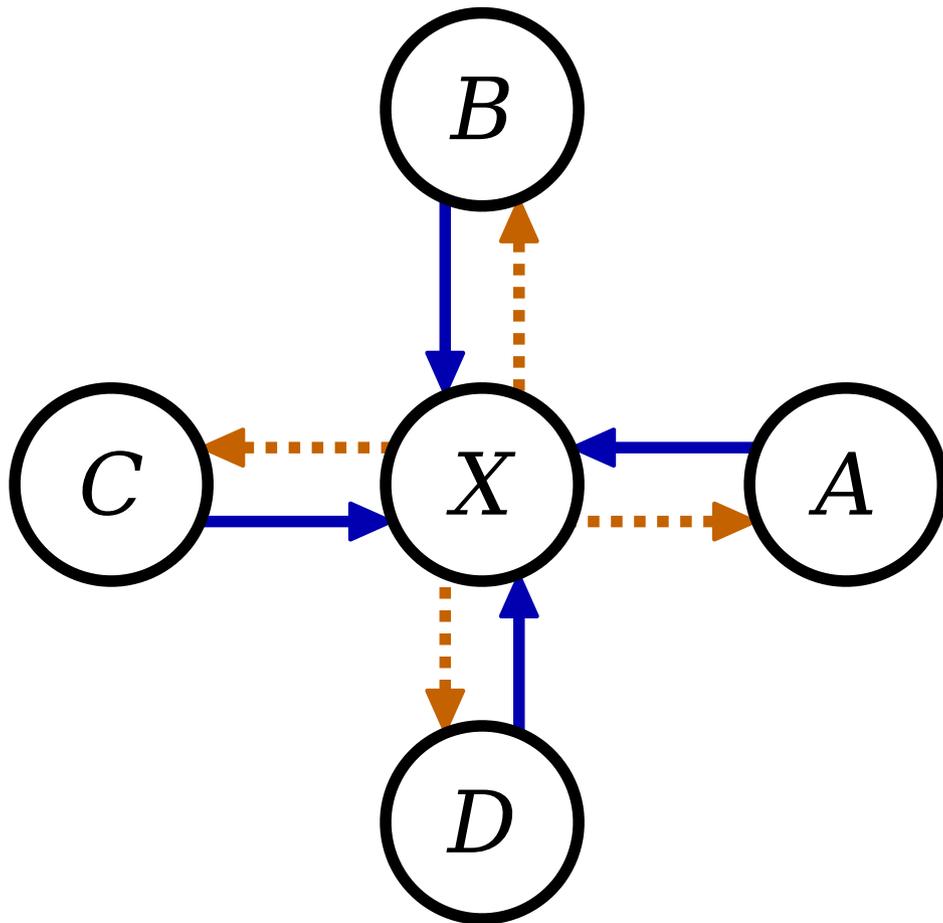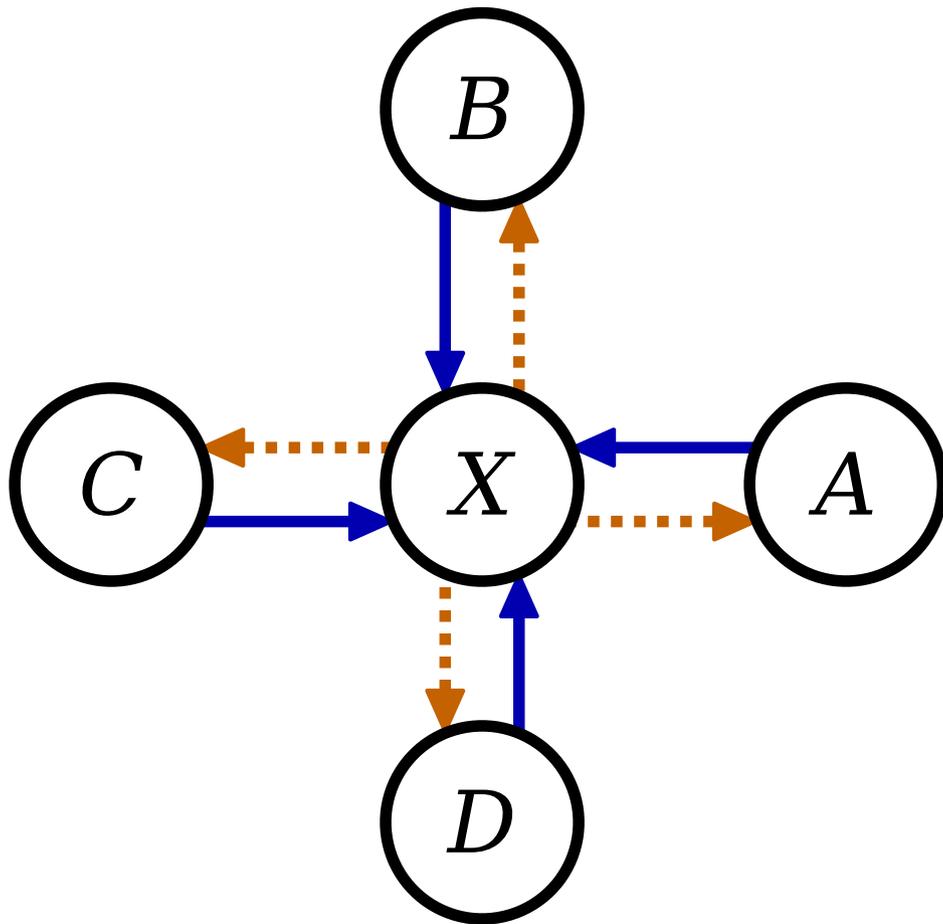
Connecting Half-Edges

- Pick at a **blue** half-edge entering node *X*.

- Imagine you enter node *X* via that half-edge.

- Which *orange* half-edge leaving *X* should you follow?

- *Answer:* The one pointing at the next neighbor, moving counterclockwise around *X*.

Connecting Half-Edges

- For each node v, sort the nodes around it in anticlockwise order.

- For each neighbor u, chain the half-edge (v, u) to be followed by the half-edge (u, x), where x is the next node after v, ordered counterclockwise.

- Time: O($e \log e$) work per node, where e is its number of neighbors. Summing over all nodes gives a runtime of **O($n \log n$)**.