Greedy Graph Algorithms: Kruskal’s Algorithm for MSTs

Announcements/Logistics

- TA hours update(s)
  - (Petros) Every other Wednesday, 4-6pm in TCL **202**
  - (Rauan) Instead of alternating Tuesdays, now every Tuesday
- Problem Set Sample Solutions
  - Compare against your answers, but often more than one way to approach a question
  - Ask plenty of questions (to TAs and classmates too!)
  - Goal is to help study and improve, but likely longer than what is necessary. I tried to be as explicit, clear, and complete as possible; may have sacrificed conciseness in the process!

More Announcements/Logistics

- Homework 4 slightly shorter than previous HWs
  - Use “extra” time to finish MCST activity, review sample solutions to previous homework
  - Monday’s activity I will include an additional “homework-like problem” on recurrences that will not be graded, but will help you to prepare for similar midterm questions
    - Opportunity to talk about concept in office hours and with TAs before midterm

Today's Plan

Kruskal's Algorithm & the Union-Find Data structure

- Review proofs from activity (or similar variants)
- (Briefly) Review Kruskal's algorithm to motivate
- (Briefly) Review Heaps
- Iterate on data structure designs to arrive at efficient Union-Find

Activity Review: MWSS are Trees

Prove. In a weighted, undirected graph $G = (V, E)$ that has strictly positive edge weights, a minimum weight spanning subgraph must always be a tree.

Proof. (By contradiction)

Suppose G has some MWSS, $S = (V, E')$, that is not a tree.

This means that the set E' connects all vertices in V, and that S contains at least one cycle. Without loss of generality, let the vertices $v_1, \ldots, v_n, v_1$ define some cycle in S.

Suppose we remove edge $e = (v_1, v_n)$ from S.

The resulting graph $S' = (V, E' - e)$ is still connected, (**Why?**) so it is still a spanning subgraph.

However, the weight of S' is less than the weight of S, since all edge weights are positive, including e. This is a contradiction, since S is a *minimum* weight spanning subgraph.

Activity Review: Cut Property

Recall. A cut is a partition of the vertices into two nonempty subsets S and $V - S$. A cut edge of a cut S is an edge with one end point in S and another in $V - S$.

Lemma (Cut Property). For any cut $S \subset V$, let $e = (u, v)$ be the *minimum* weight edge connecting any vertex in S to a vertex in $V - S$. Every minimum spanning tree must include e.

Proof. (By contradiction)

Suppose T is a spanning tree that does not contain $e = (u, v)$.

Main Idea: We will construct another spanning tree $T' = T \cup e - e'$ with weight less than T ($\Rightarrow\Leftarrow$)

Question: How to find such an edge e'?

If we replace e' with e, we get a contradiction

Activity Review: Cut Property

Proof (Cut Property). (By contradiction.)

Suppose T is a spanning tree that does not contain $e = (u, v)$.

- Adding e to T results in a unique cycle C
- Cycle C must "enter" and "leave" cut S, that is, $\exists e' = (u', v') \in C$ s.t. $u' \in S, v' \in V - S$
- $w(e') > w(e)$ (**Why?**)
- $T' = T \cup e - e'$ is a spanning tree (**Why?**)
- $w(T') < w(T)$ ($\Rightarrow\!\Leftarrow$) ∎

Kruskal's Algorithm

CS136 Review: Priority Queue

Priority Queues manage a set S of items and the following operations on S:

- **Insert.** Insert a new element into S
- **Delete.** Delete an element from S
- **Extract.** Retrieve highest priority element in S

Priorities are encoded as a 'key' value

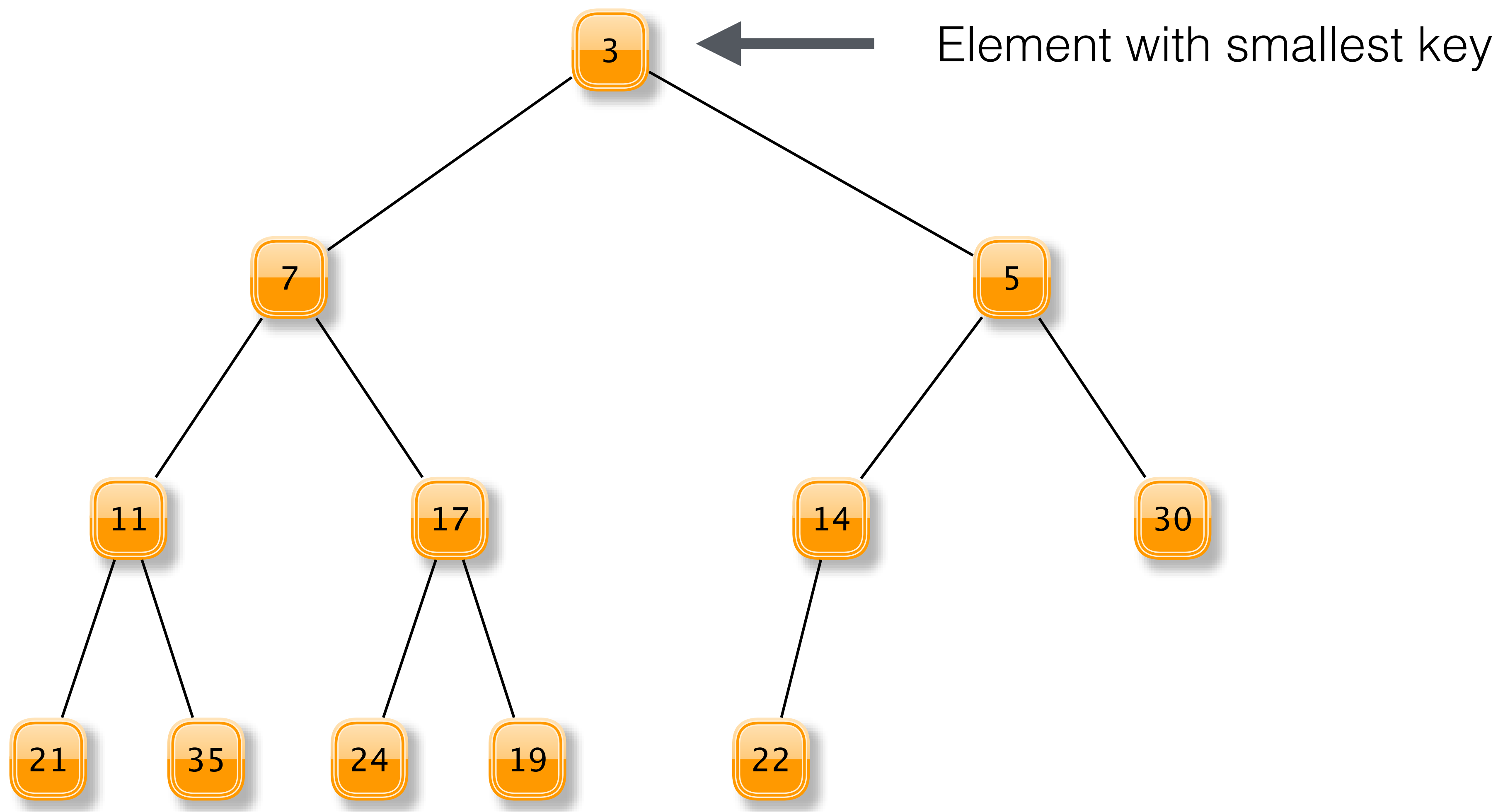
Typically: higher priority <—> lower key value (**Min**Heap)

Heap as Priority Queue. Combines tree structure with array access

- Insert and delete: $O(\log n)$ time ('tree' traversal & swaps)
- **Extract min.** Delete item with minimum key value: $O(\log n)$

Heap Example

Heap property: For every element v, at node i, the element w at i's parent satisfies key(w) $\leq$ key(v)



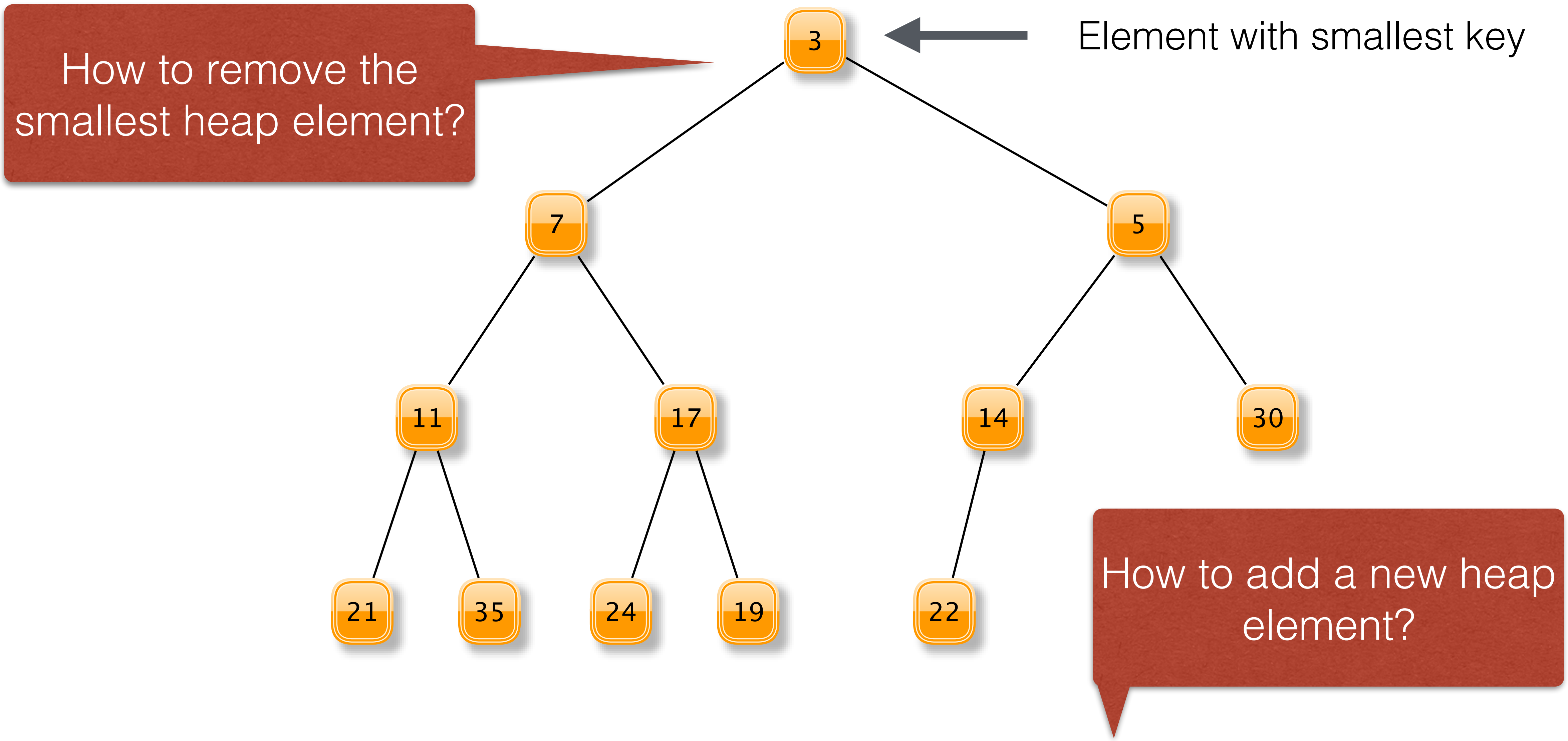
Array representation of binary tree: left child at 2i+1, right child at 2i+2

H

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	7	5	11	17	14	30	21	35	24	19	22	-	-	-	-

Heap Example

Heap property: For every element v, at node i, the element w at i's parent satisfies key$(w) \leq$ key(v)



H

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
3	7	5	11	17	14	30	21	35	24	19	22	-	-	-	-

Kruskal's Algorithm

Idea: Add the cheapest remaining edge **that does not create a cycle**.

`initialize` $T = \emptyset, H \leftarrow E$ // empty MST, all edges in heap

`while` $|T| < n - 1$:

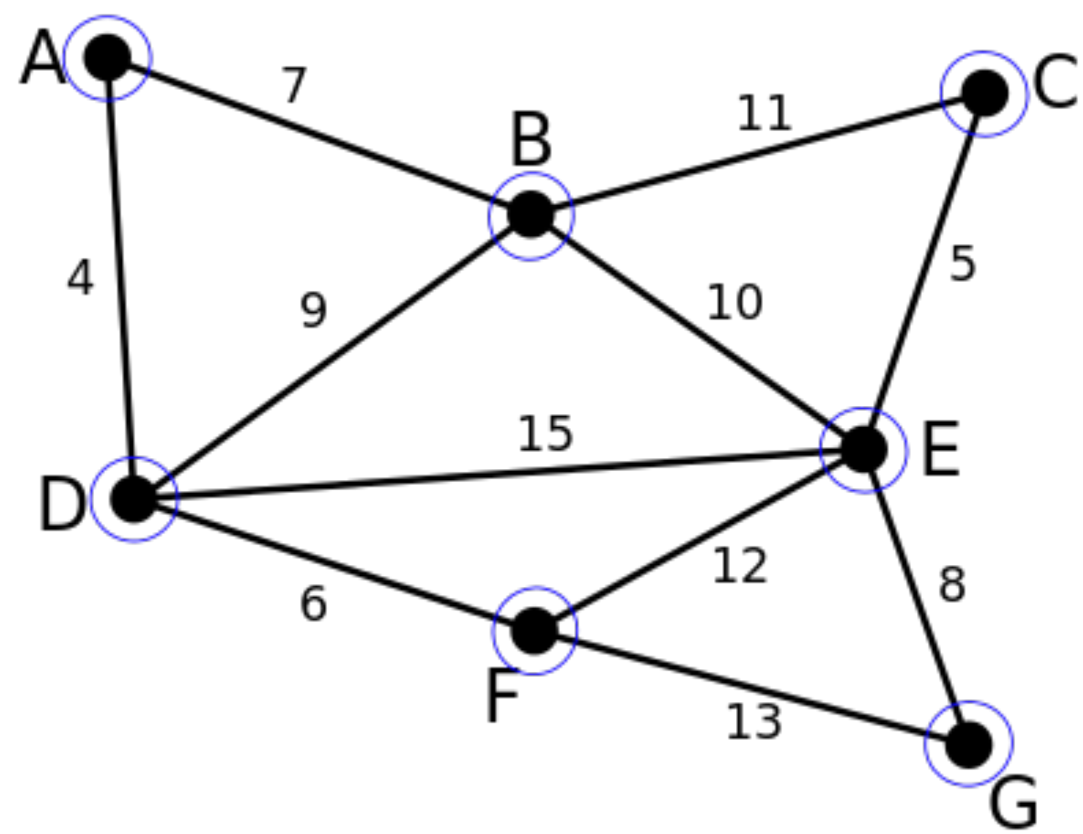
Remove cheapest edge e from H

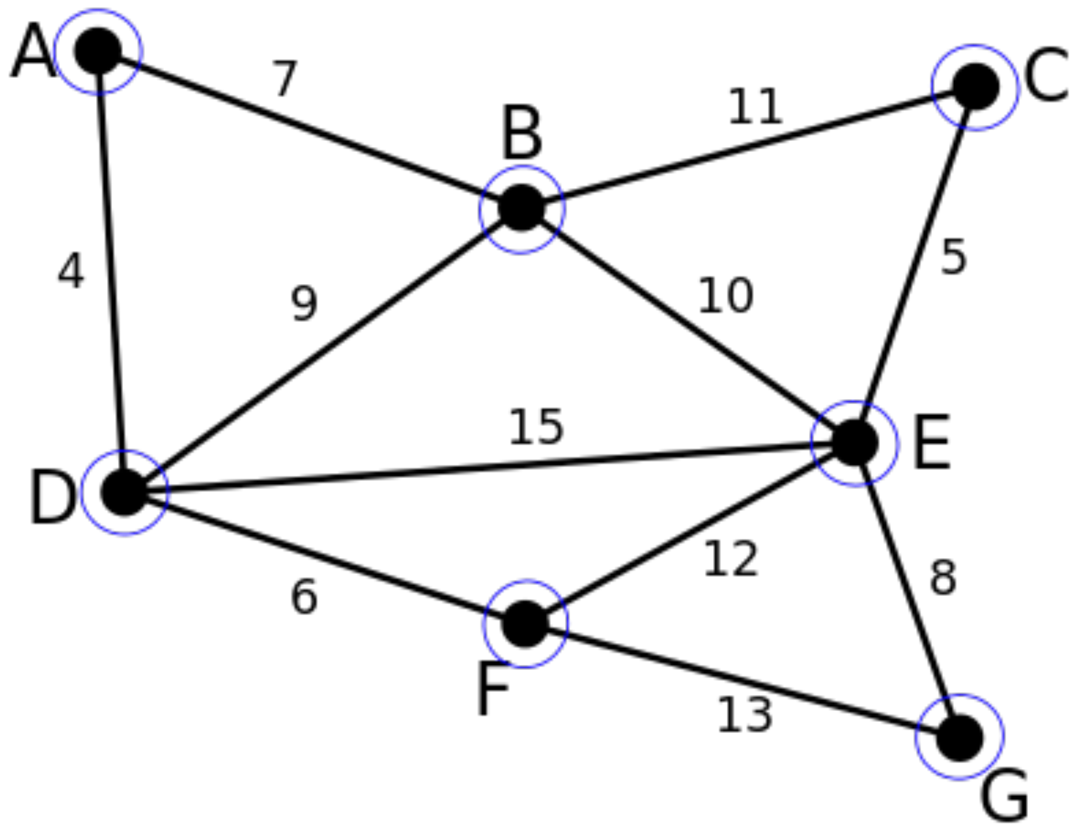
`if` adding e to T does not create a cycle:

$T \leftarrow T \cup \{e\}$

$H \leftarrow H - \{e\}$

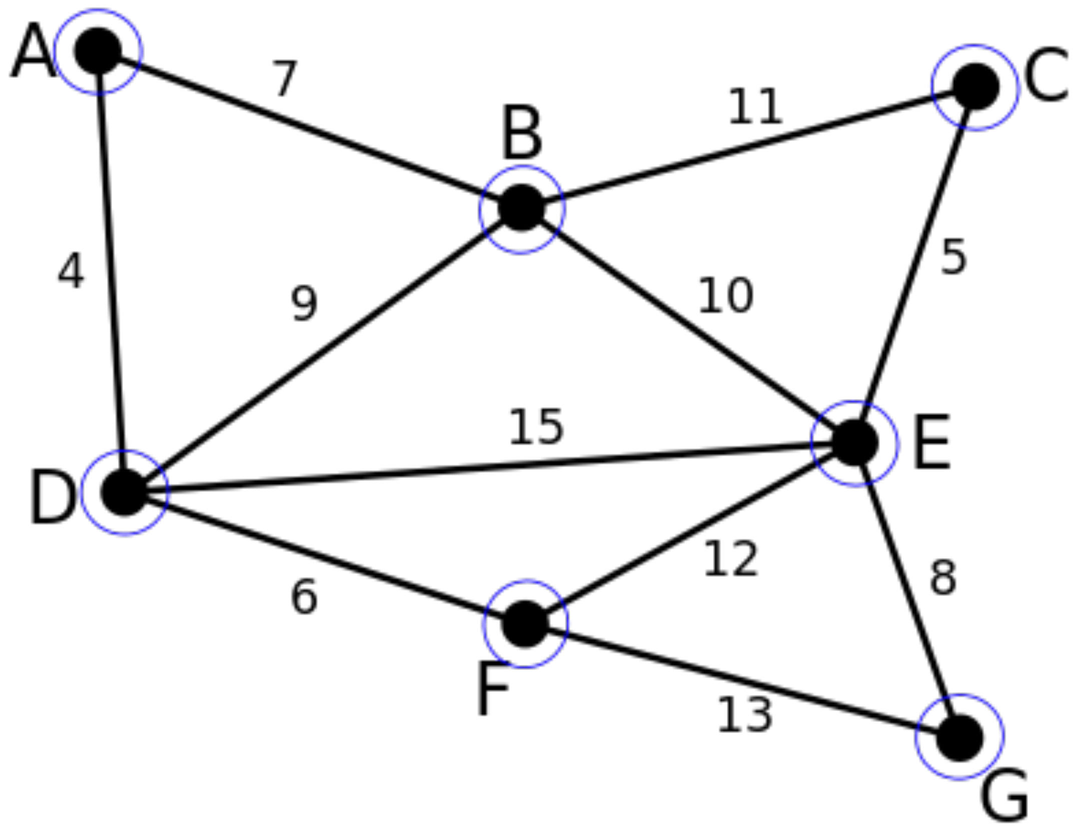
// T is now an MCST!





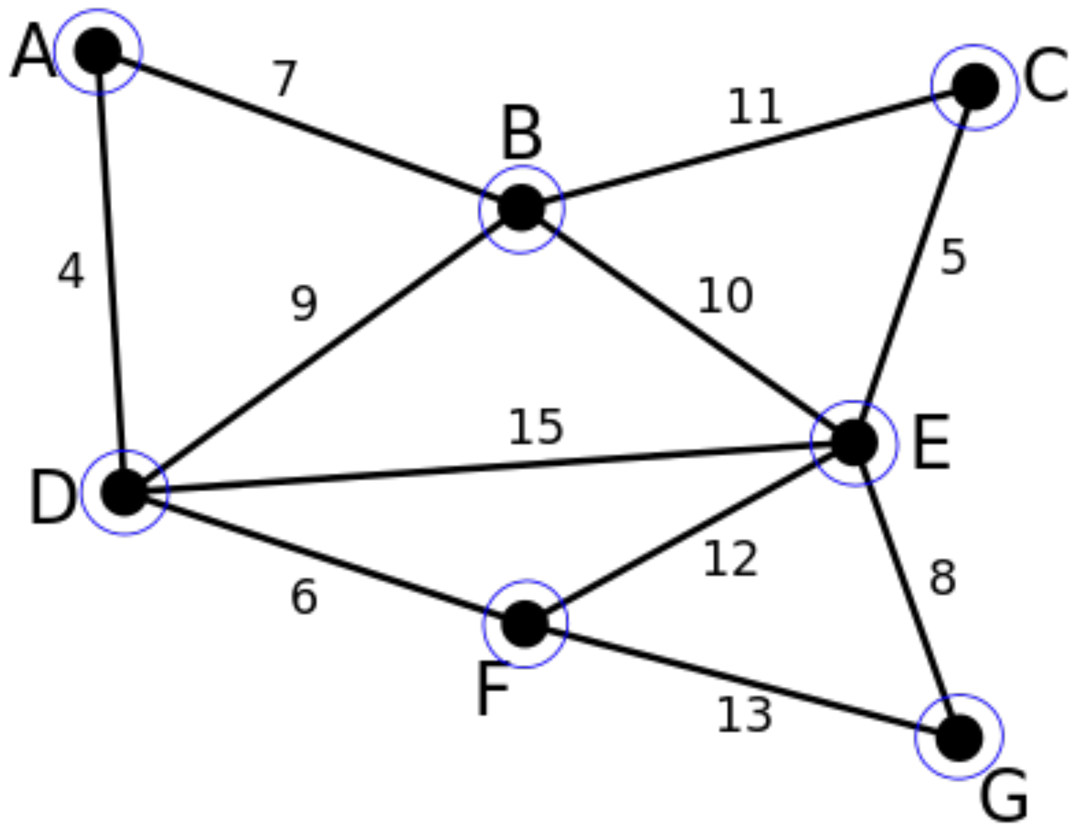
Idea: Add the cheapest remaining edge **that does not create a cycle**.

- Initialize $T = \varnothing$, $H \leftarrow E$
- While $|T| < n - 1$:
    - Remove cheapest edge e from H
    - If adding e to T does not create a cycle
        - $T \leftarrow T \cup \{e\}$
    - $H \leftarrow H - \{e\}$



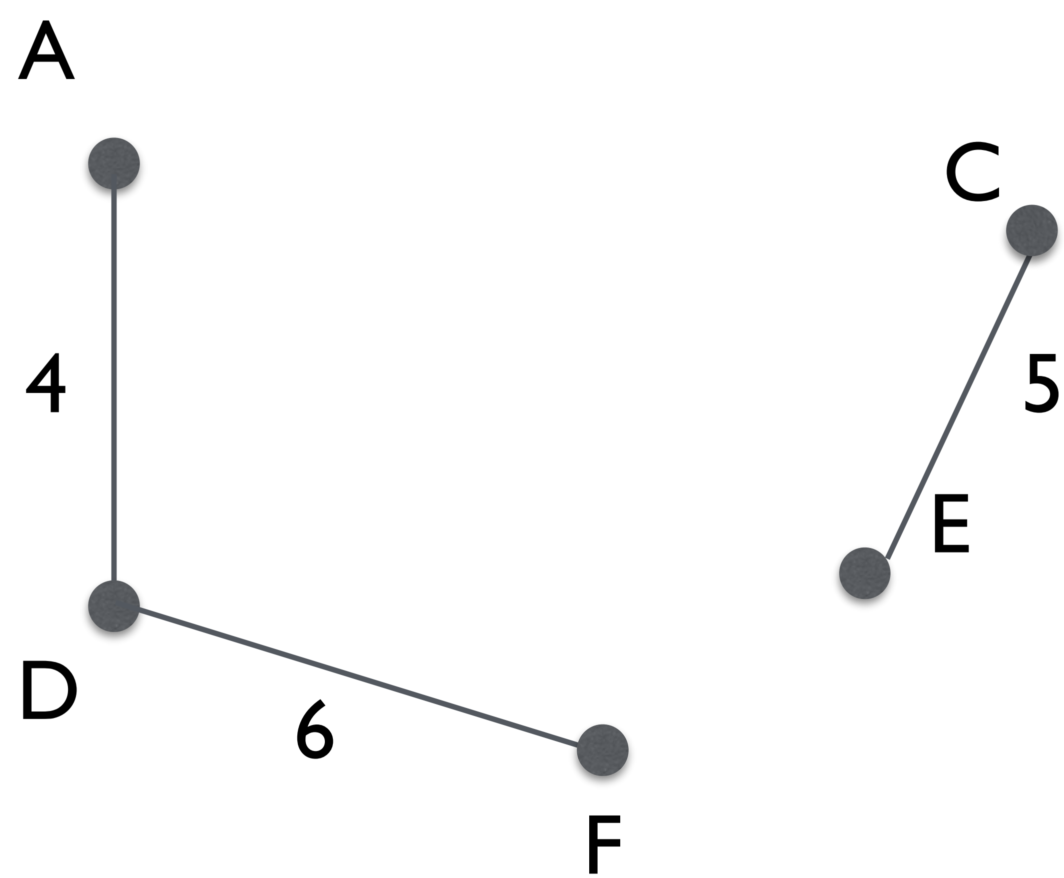
Idea: Add the cheapest remaining edge **that does not create a cycle**.

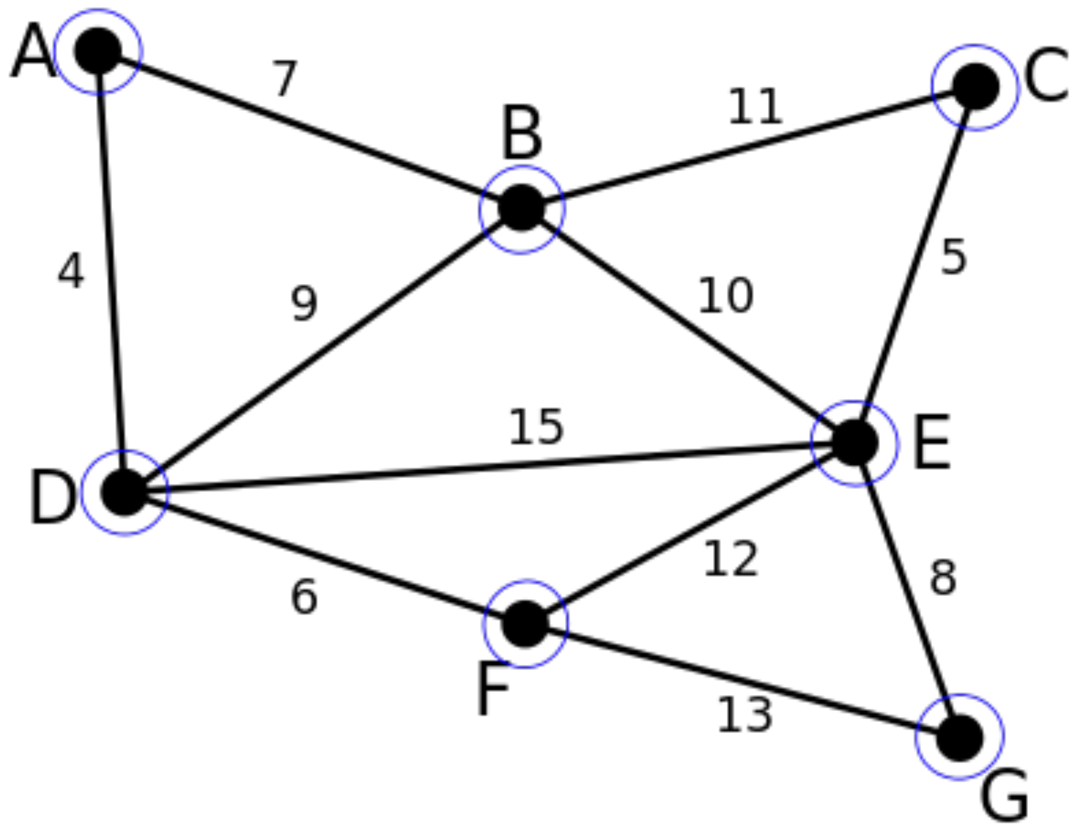
- Initialize $T = \varnothing$, $H \leftarrow E$
- While $|T| < n - 1$:
    - Remove cheapest edge e from H
    - If adding e to T does not create a cycle
        - $T \leftarrow T \cup \{e\}$
    - $H \leftarrow H - \{e\}$



Idea: Add the cheapest remaining edge **that does not create a cycle**.

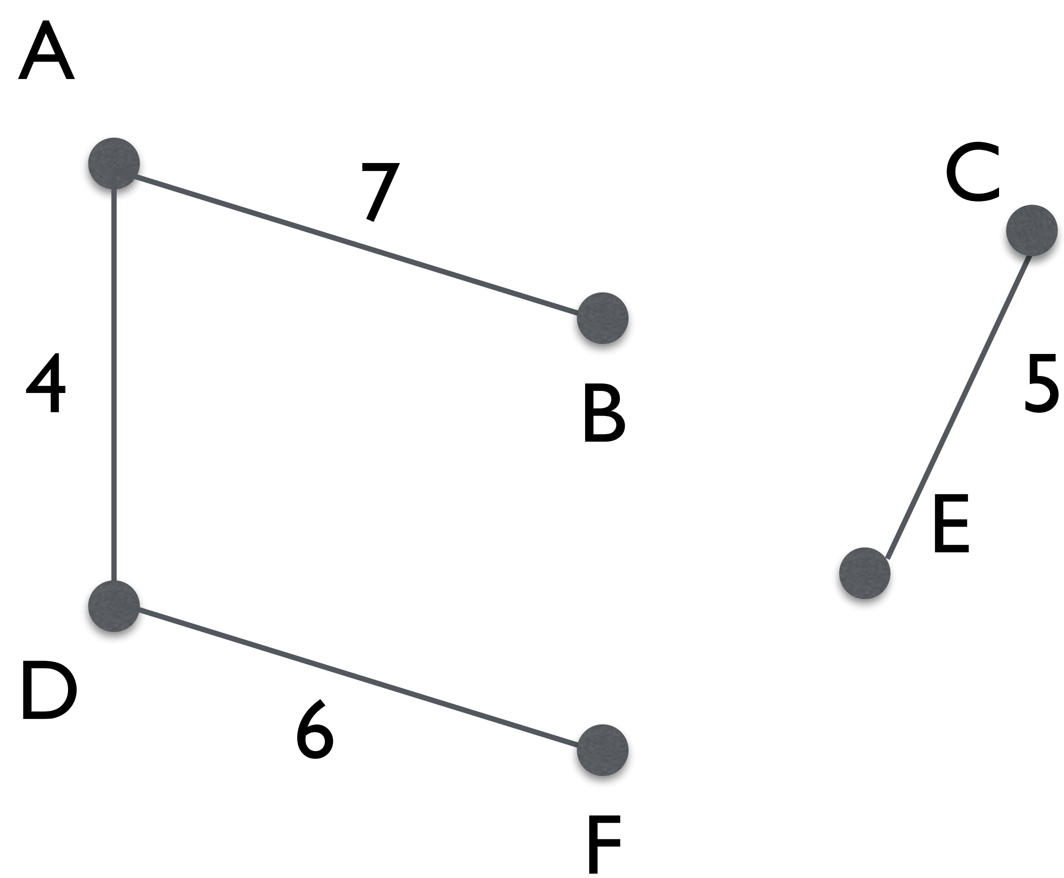
- Initialize $T = \varnothing$, $H \leftarrow E$
- While $|T| < n - 1$:
  - Remove cheapest edge e from H
  - If adding e to T does not create a cycle
    - $T \leftarrow T \cup \{e\}$
  - $H \leftarrow H - \{e\}$

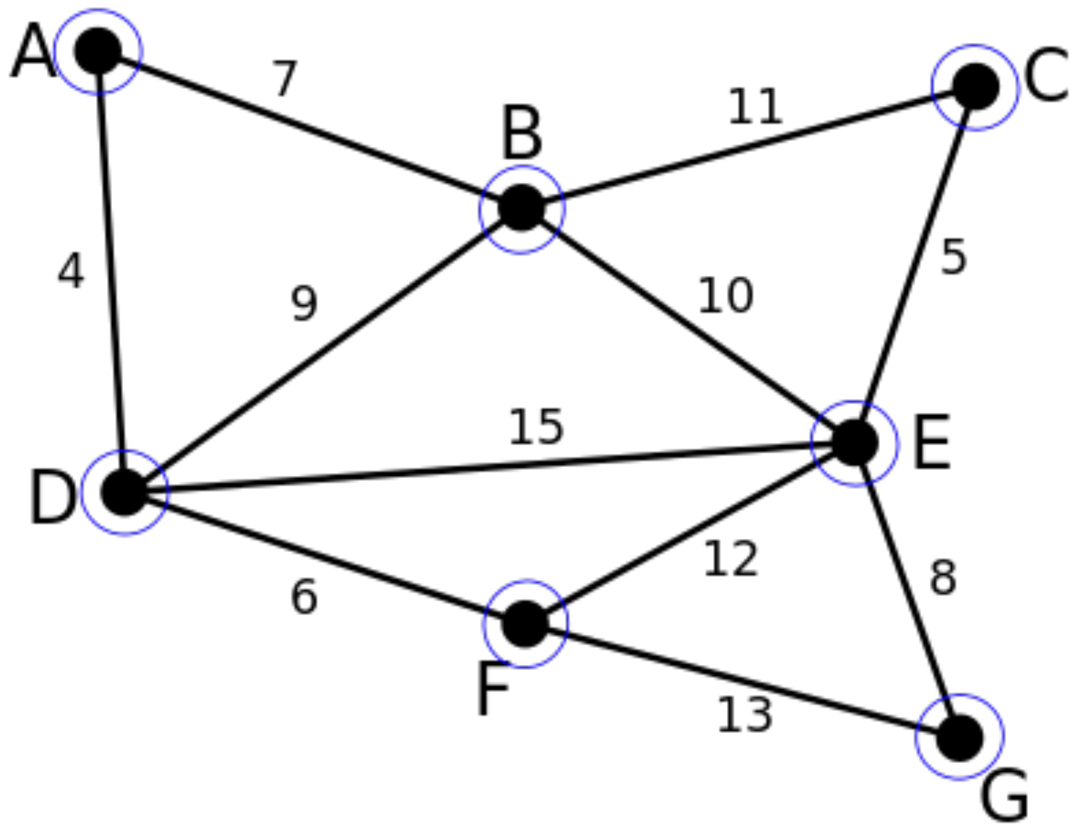


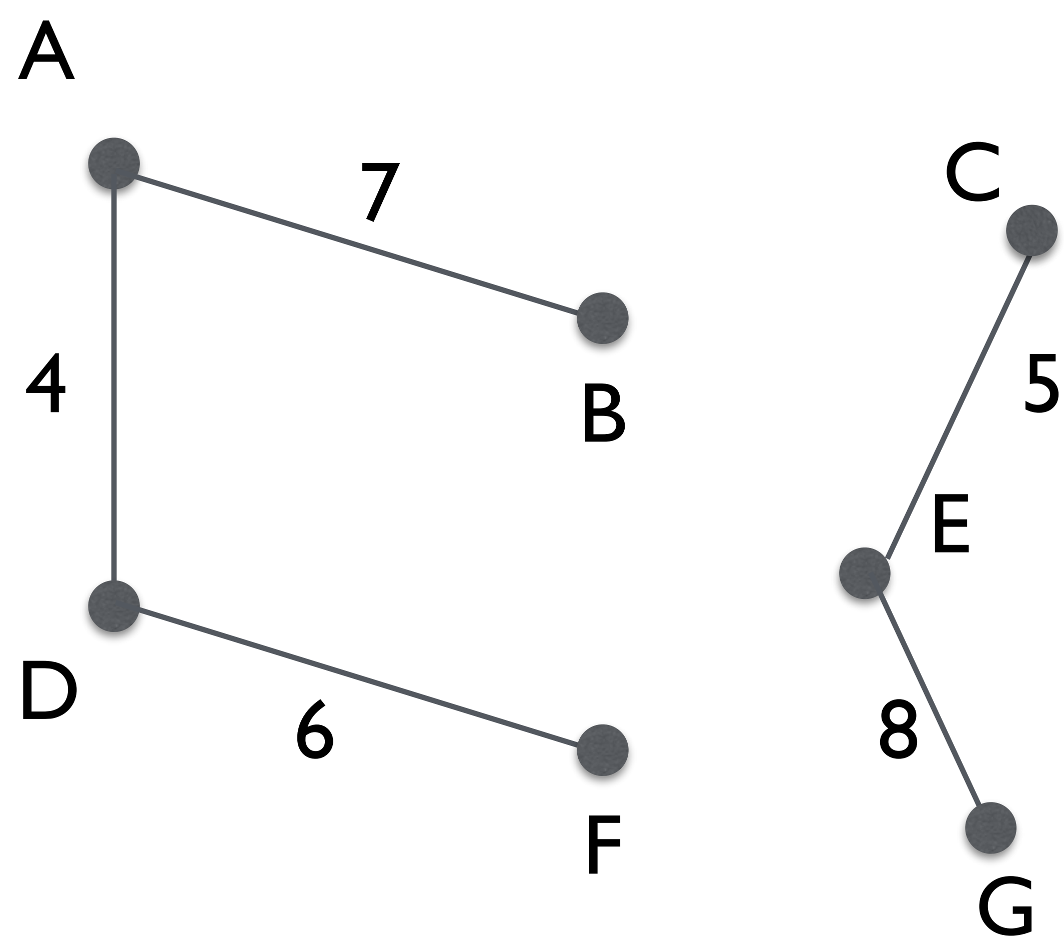


Idea: Add the cheapest remaining edge **that does not create a cycle**.

- Initialize $T = \emptyset$, $H \leftarrow E$
- While $|T| < n - 1$:
  - Remove cheapest edge e from H
  - If adding e to T does not create a cycle
    - $T \leftarrow T \cup \{e\}$
  - $H \leftarrow H - \{e\}$

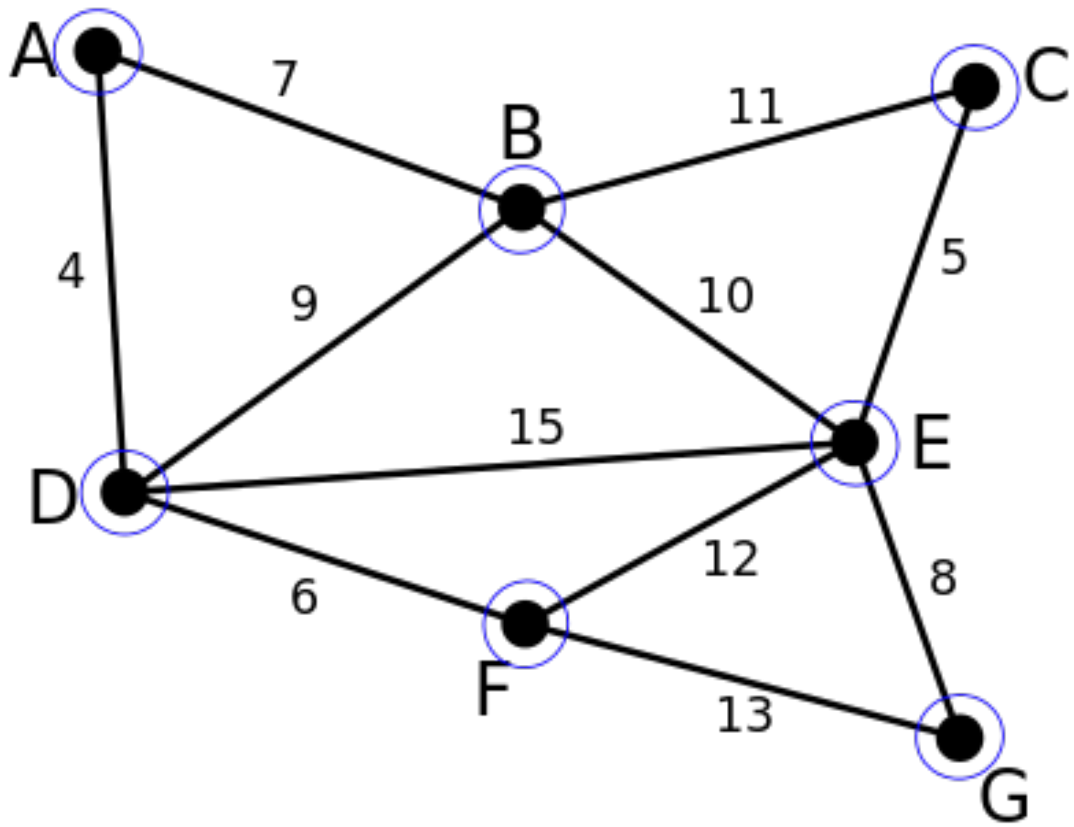




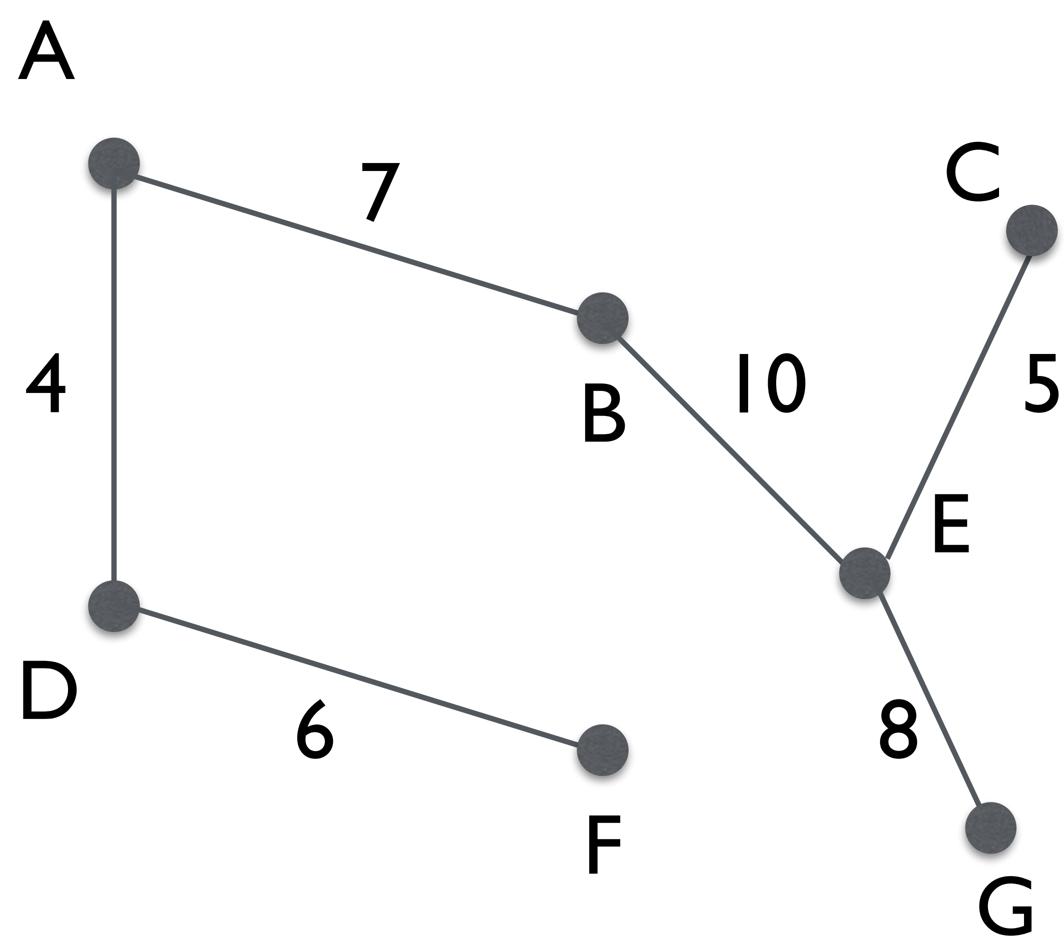


Idea: Add the cheapest remaining edge **that does not create a cycle**.

- Initialize $T = \varnothing$, $H \leftarrow E$
- While $|T| < n - 1$:
  - Remove cheapest edge e from H
  - If adding e to T does not create a cycle
    - $T \leftarrow T \cup \{e\}$
  - $H \leftarrow H - \{e\}$



Total weight: 40



Idea: Add the cheapest remaining edge **that does not create a cycle**.

- Initialize $T = \varnothing, H \leftarrow E$
- While $|T| < n - 1$:
  - Remove cheapest edge e from H
  - If adding e to T does not create a cycle
    - $T \leftarrow T \cup \{e\}$
  - $H \leftarrow H - \{e\}$

Kruskal's Analysis

- **Correctness**: Does it give us the correct MST?
- **Key Question:** Why is each edge (v, w) that we are adding safe?
  - Consider the step just before (v, w) is added
    - Let $S = \{x \in V \mid T \text{ contains a path from } v \text{ to } x\}$
    - This is a valid cut in the graph (**Why**? Can $w \in S$?)
    - If there was a cheaper cut edge for cut $(S, V - S)$ which did not form a cycle, the algorithm would have already added it; this must be the min-cost cut edge for this cut
- **Runtime**.
  - How quickly can we find the minimum remaining edge?
  - How quickly can we determine if an edge creates a cycle?

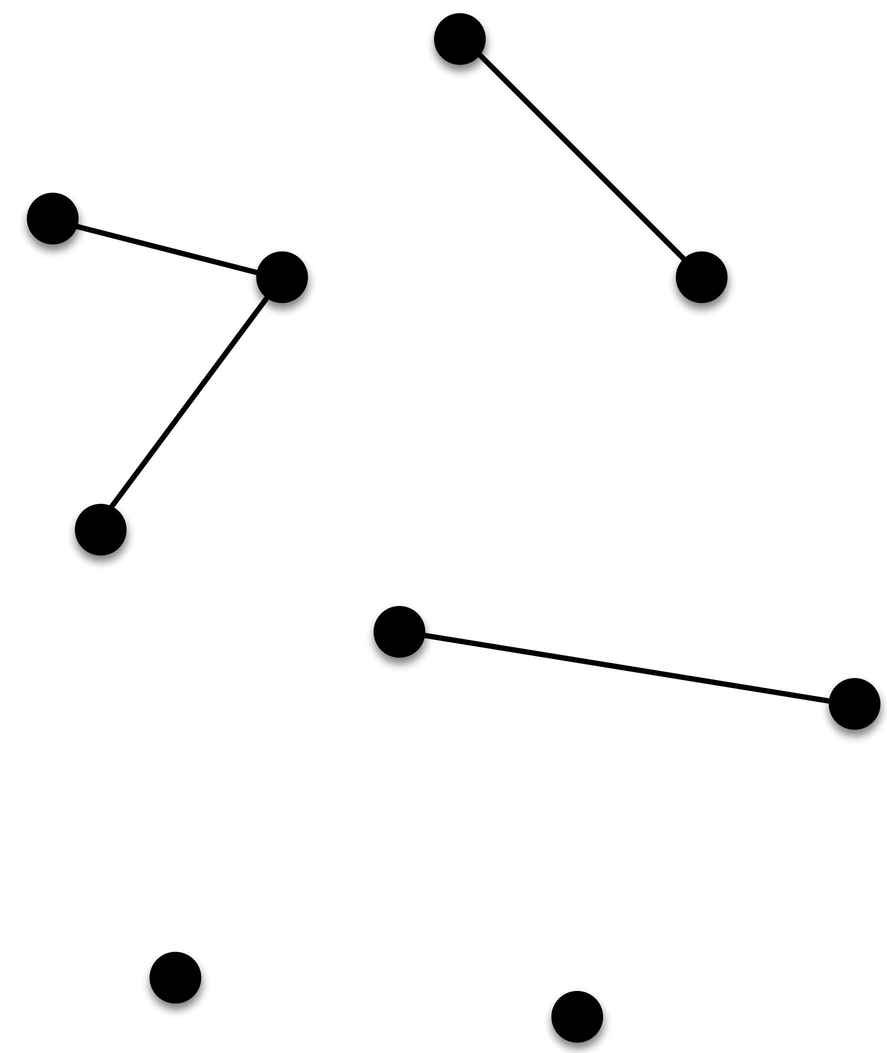
Kruskal's Implementation

What steps do we need to implement?

- Sort edges by weight (add to heap): $O(m \log m)$
  - If we do the rest efficiently, this is the dominant cost
- Determine whether $T \cup \{e\}$ contains a cycle
  - Ideas?
- Add an edge to T

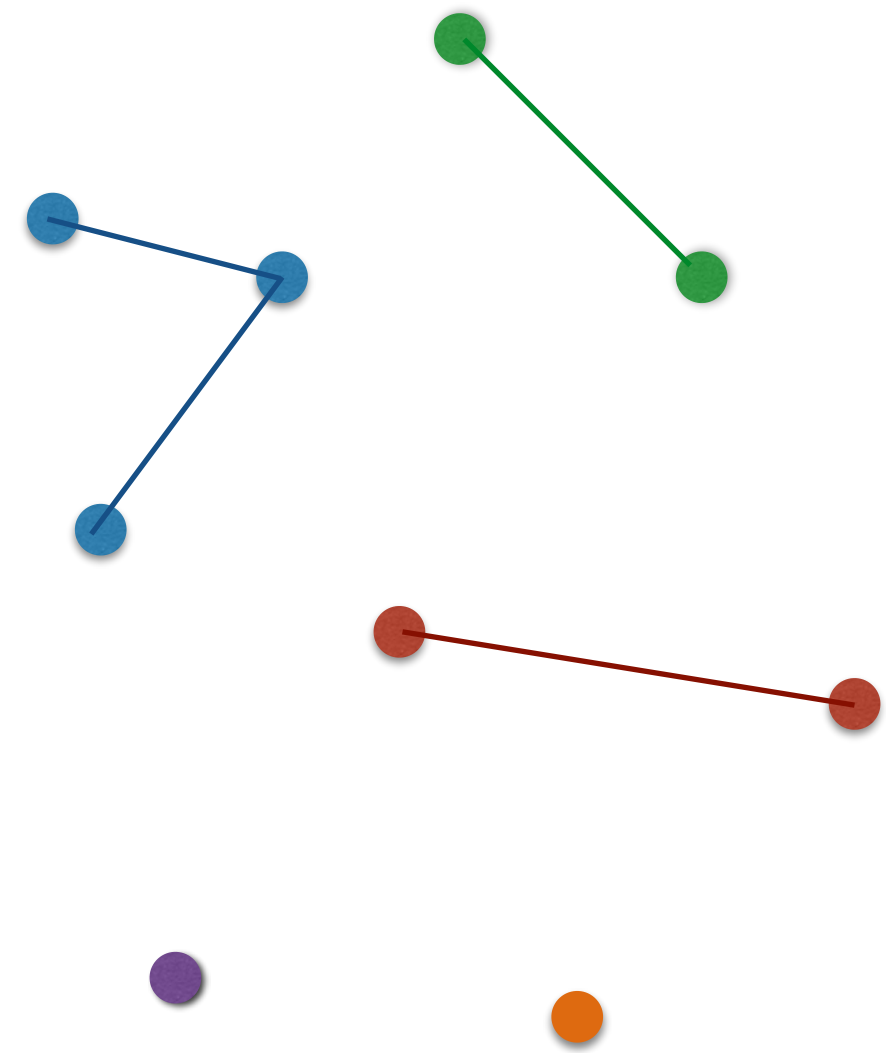
Does this edge create a cycle?

- An edge creates a cycle if it connects a subtree to another vertex in the same subtree
- What if we could label the vertices in a tree? Then we could determine if an edge creates a cycle by comparing vertex labels

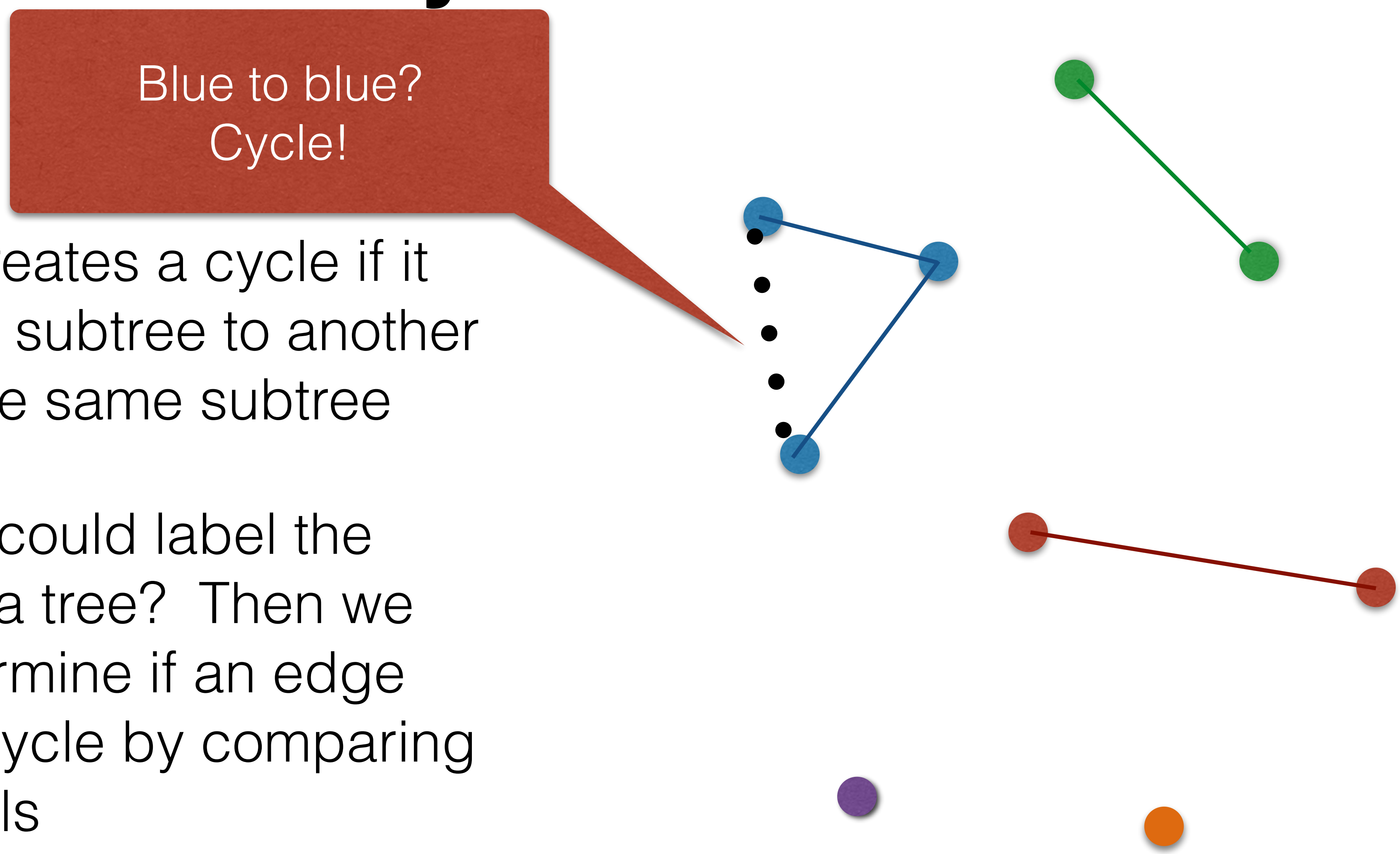


Does this edge create a cycle?

- An edge creates a cycle if it connects a subtree to another vertex in the same subtree
- What if we could label the vertices in a tree? Then we could determine if an edge creates a cycle by comparing vertex labels



Does this edge create a cycle?



- An edge creates a cycle if it connects a subtree to another vertex in the same subtree

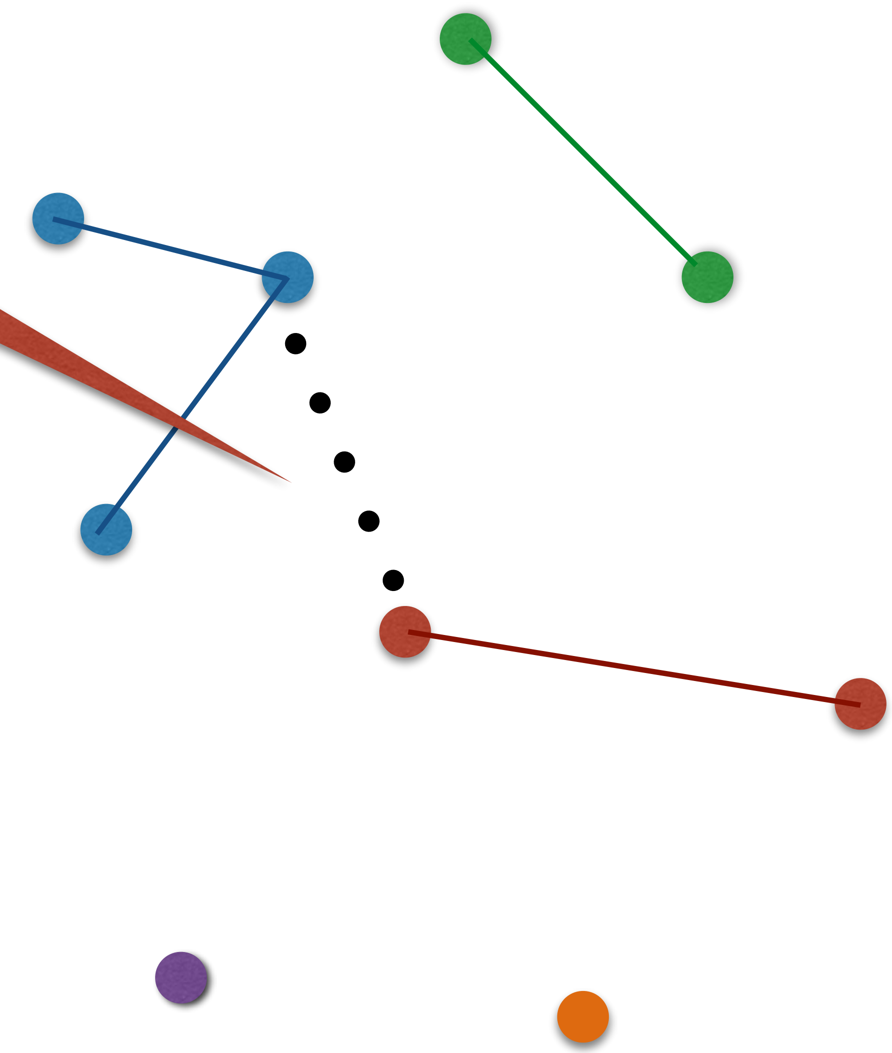
- What if we could label the vertices in a tree? Then we could determine if an edge creates a cycle by comparing vertex labels

Does this edge create a cycle?

Blue to red?
No cycle!

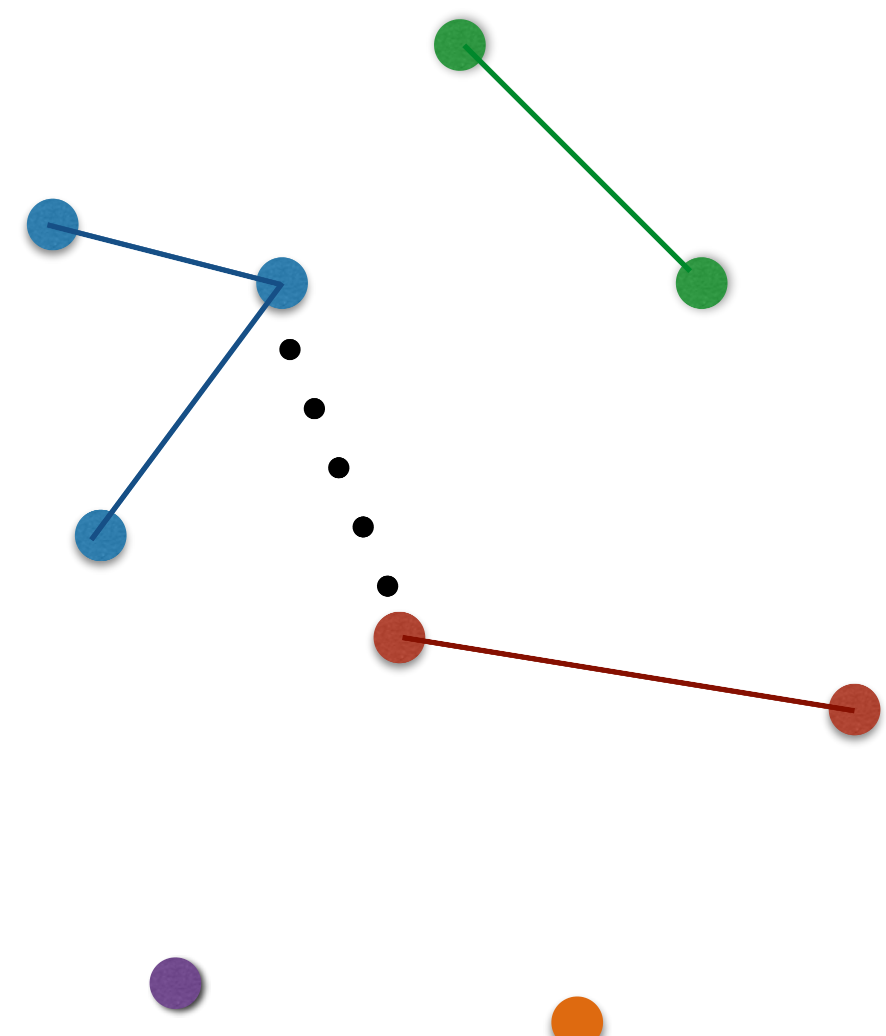
- An edge creates a cycle if it connects a subtree to another vertex in the same subtree

- What if we could label the vertices in a tree? Then we could determine if an edge creates a cycle by comparing vertex labels



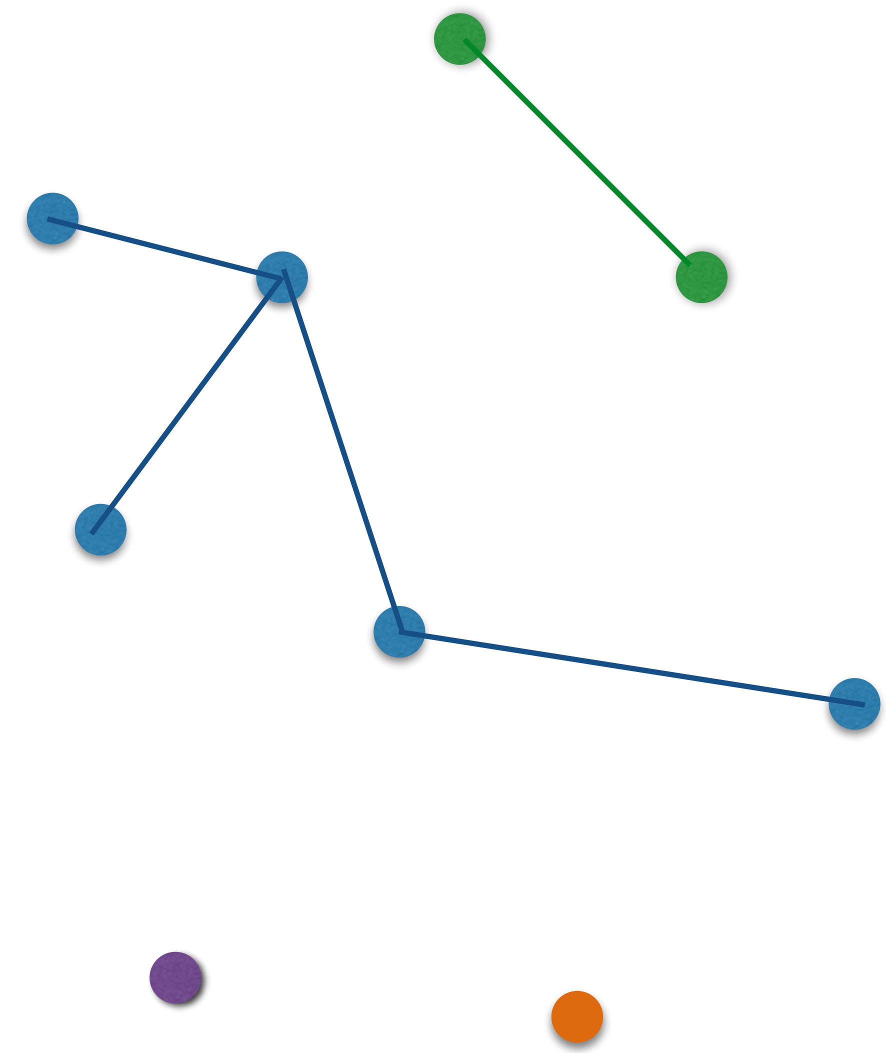
Does this edge create a cycle?

- An edge creates a cycle if it connects a subtree to another vertex in the same subtree
- What if we could label the vertices in a tree? Then we could determine if an edge creates a cycle by comparing vertex labels
- How can we update vertex labels when adding an edge?



Does this edge create a cycle?

- An edge creates a cycle if it connects a subtree to another vertex in the same subtree
- What if we could label the vertices in a tree? Then we could determine if an edge creates a cycle by comparing vertex labels
- How can we update vertex labels when adding an edge?



Ideally, what would we do?

- Start with each node as its own set
- Given a node, determine which set it’s in (i.e., a label)
- Take two sets and combine them into a single set with a single label

Union-Find Data Structure

Manages a **dynamic partition** of a set S

- Provides the following methods:
  - `MakeUnionFind()`: Initializes each vertex/set with unique label
  - `Find(x):` Return label of set containing x
  - `Union(X, Y)`: Replace sets `X, Y` with $X \cup Y$ with single label

Kruskal's Algorithm can then use

- `Find` for cycle checking
- `Union` to update after adding an edge to T

Union-Find: Any Ideas?

How can we get:

- $O(1)$ Find
- $O(n)$ Union

(Hint: we'll be maintaining labels)

Union-Find: First Attempt

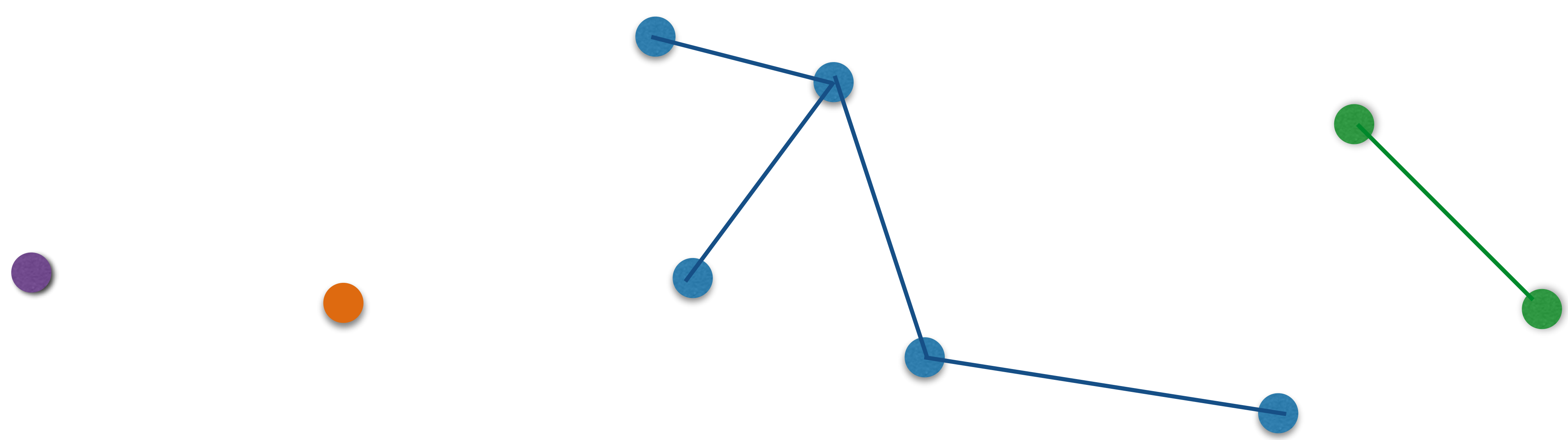
Let $S = \{1, 2, \ \dots, n\}$ be the sets.

Idea: Each item (vertex) stores the label of its set

- `MakeUnionFind():` Set $L[x] = x$ for each $x \in S$: $O(n)$
- `Find(x):` Return $L[x]$: $O(1)$
- `Union(X,Y):`
  - For each $x \in X$, update $L[x]$ to label of set Y
  - $O(n)$ in the worst case (happens when we union two large sets)

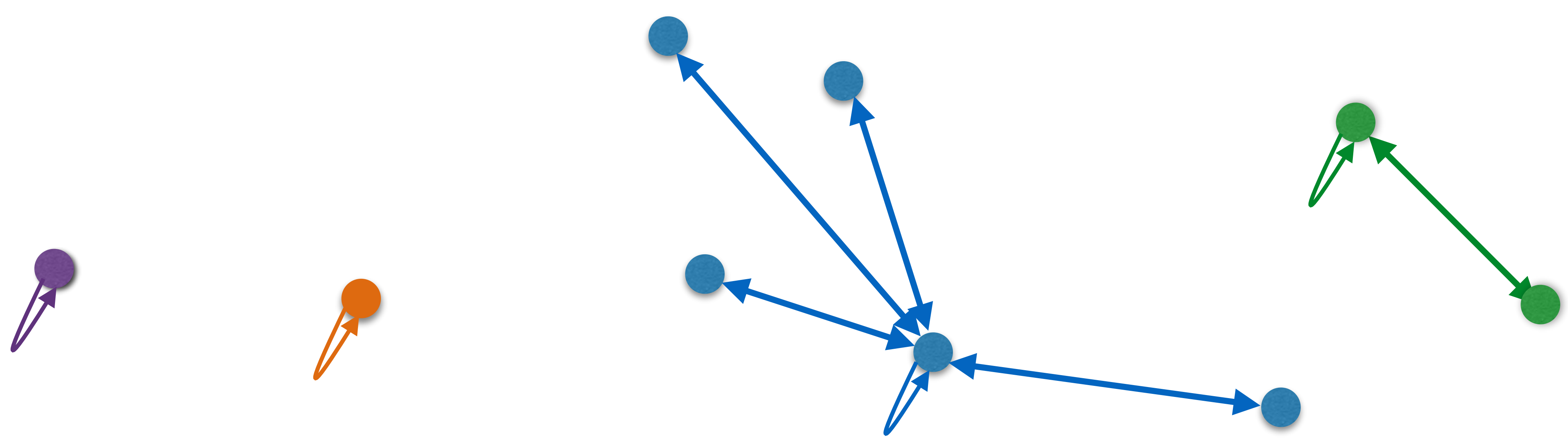
Union-Find: Improving Union

- Let's tweak that idea just a little bit and analyze it
- Think of a data structure with pointers instead of an array
- Each vertex points to a "head" node instead of a label; head points to itself



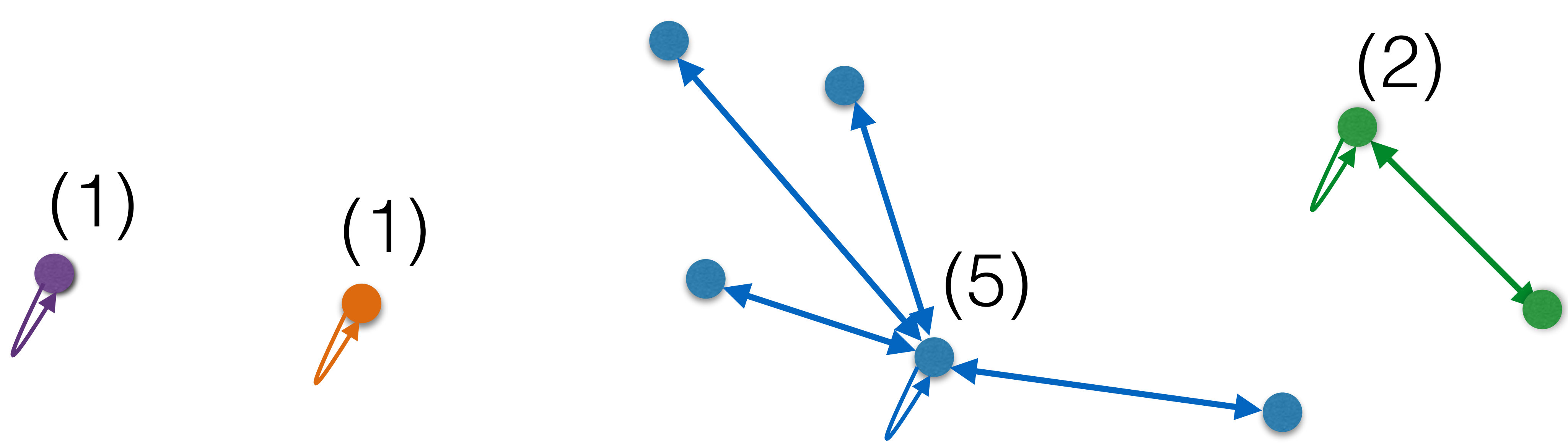
Union-Find: Improving Union

- Let’s tweak that idea just a little bit and analyze it
- Think of a data structure with pointers instead of an array
- Each vertex points to a “head” node instead of a label; head points to itself



Union-Find: Improving Union

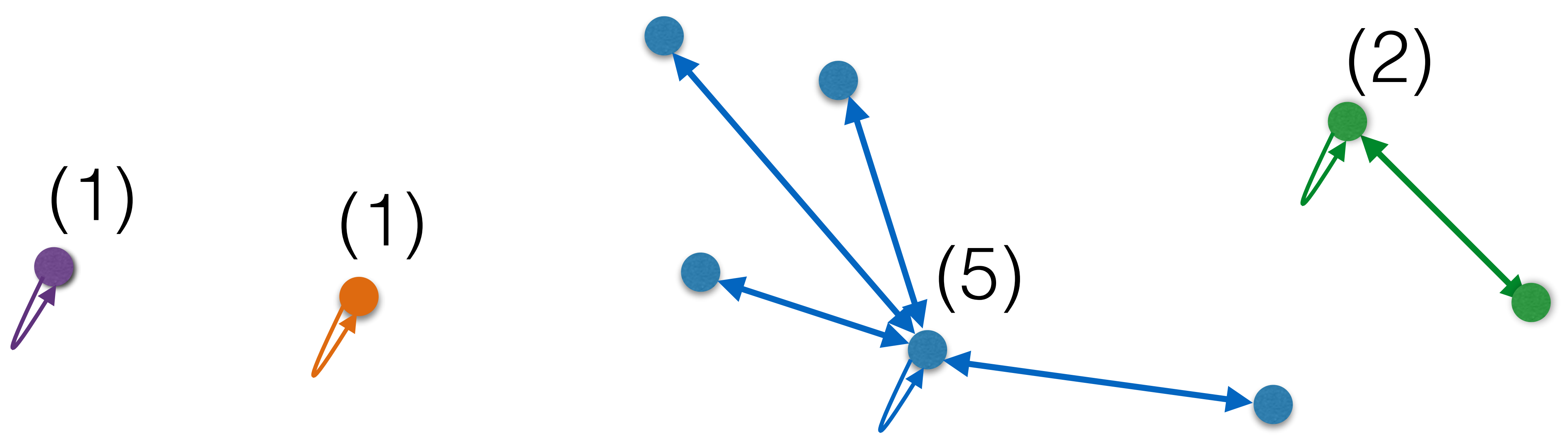
- Let’s tweak that idea just a little bit and analyze it
- Think of a data structure with pointers instead of an array
- Each vertex points to a “head” node instead of a label; head points to itself
- (Also store size of each set in the head)



Union-Find: Improving Union

Now, to do a union, what must we do?

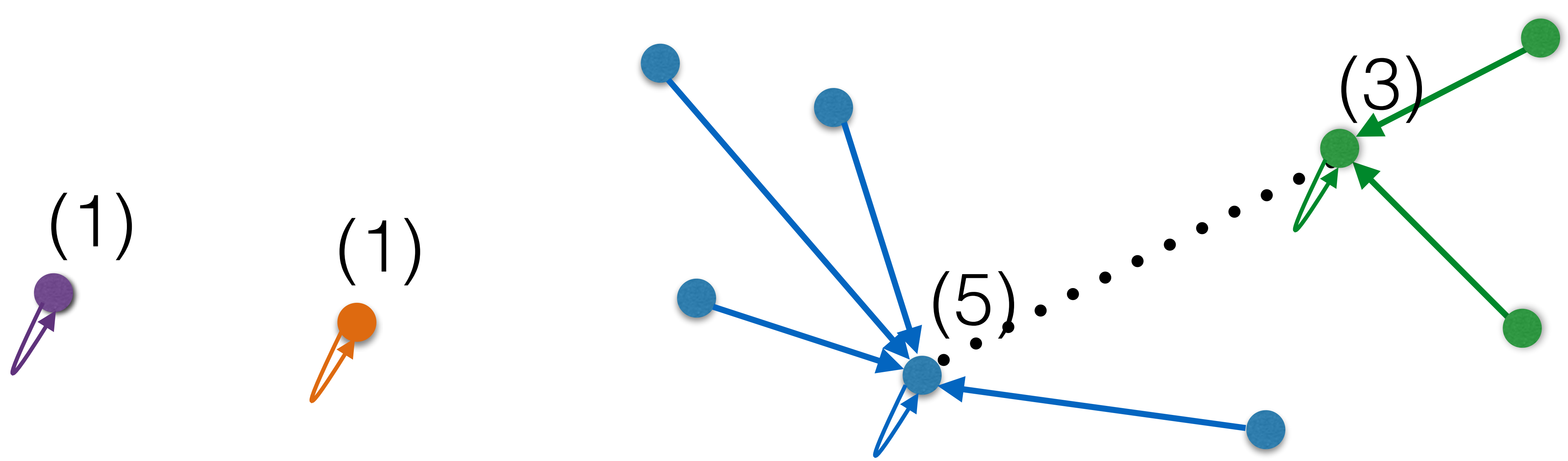
- Make every element in the smaller set point at the head of the larger set (**why?**)
- Update the size of the newly unioned set



Union-Find: Improving Union

Suppose Kruskal's identifies an edge between the blue set and the green set that we want to add. What do we do?

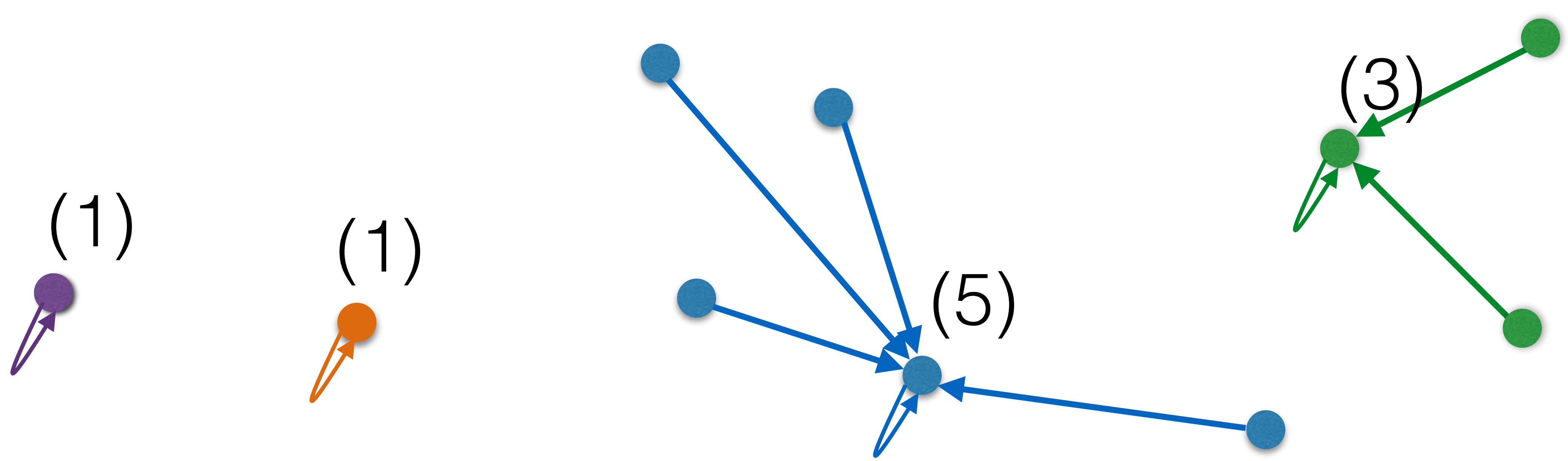
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



Union-Find: Improving Union

Suppose Kruskal's identifies an edge between the blue set and the green set that we want to add. What do we do?

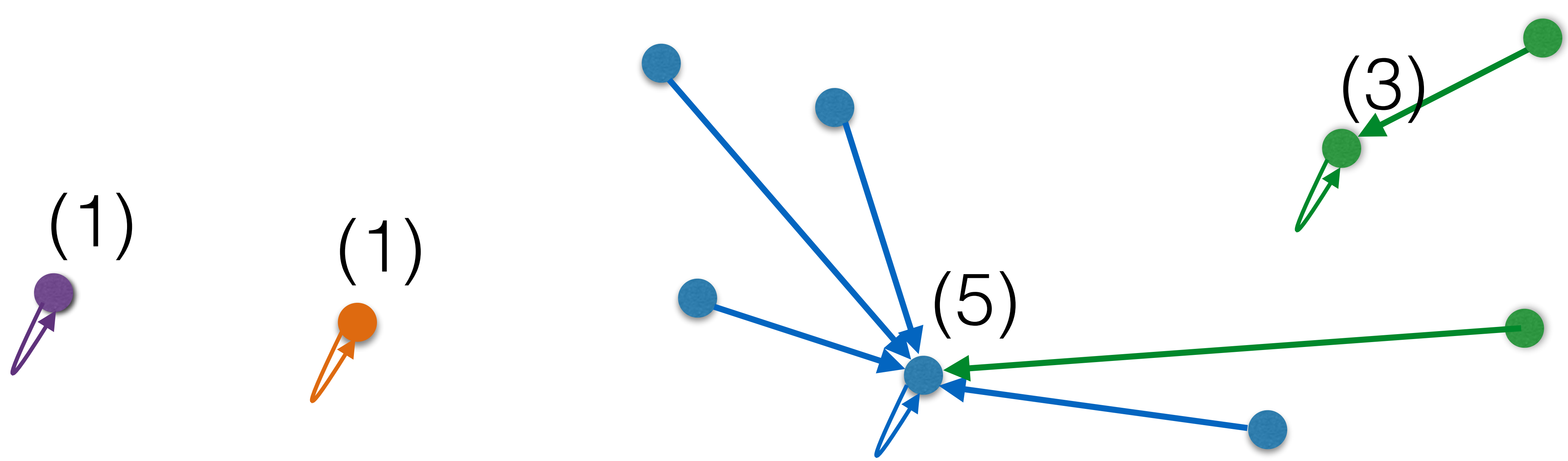
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



Union-Find: Improving Union

Suppose Kruskal's identifies an edge between the blue set and the green set that we want to add. What do we do?

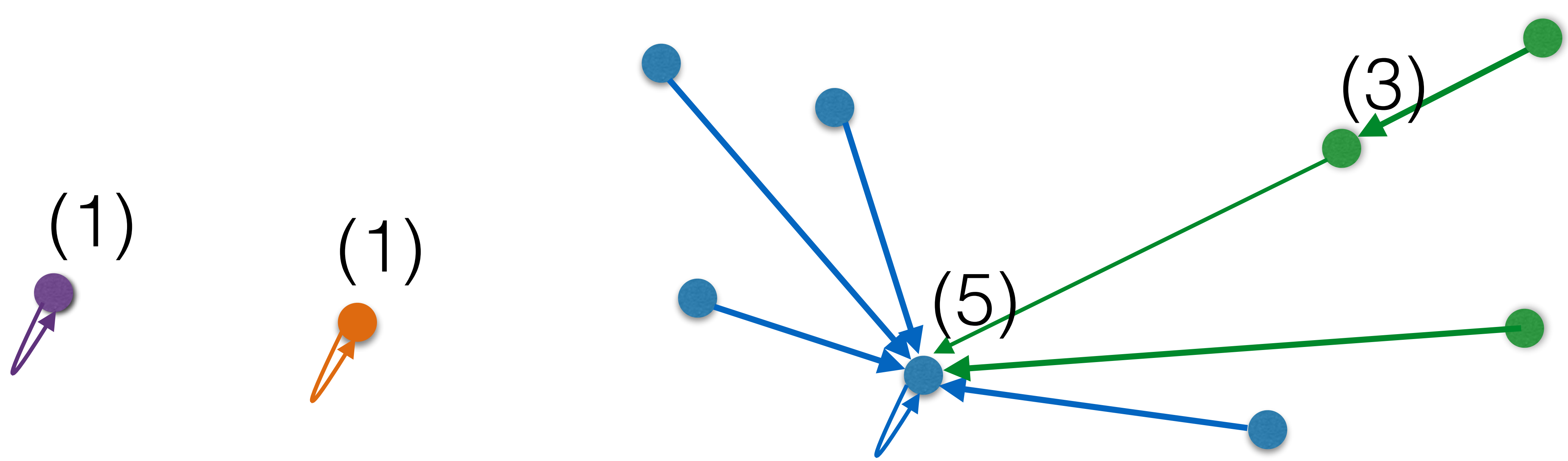
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



Union-Find: Improving Union

Suppose Kruskal's identifies an edge between the blue set and the green set that we want to add. What do we do?

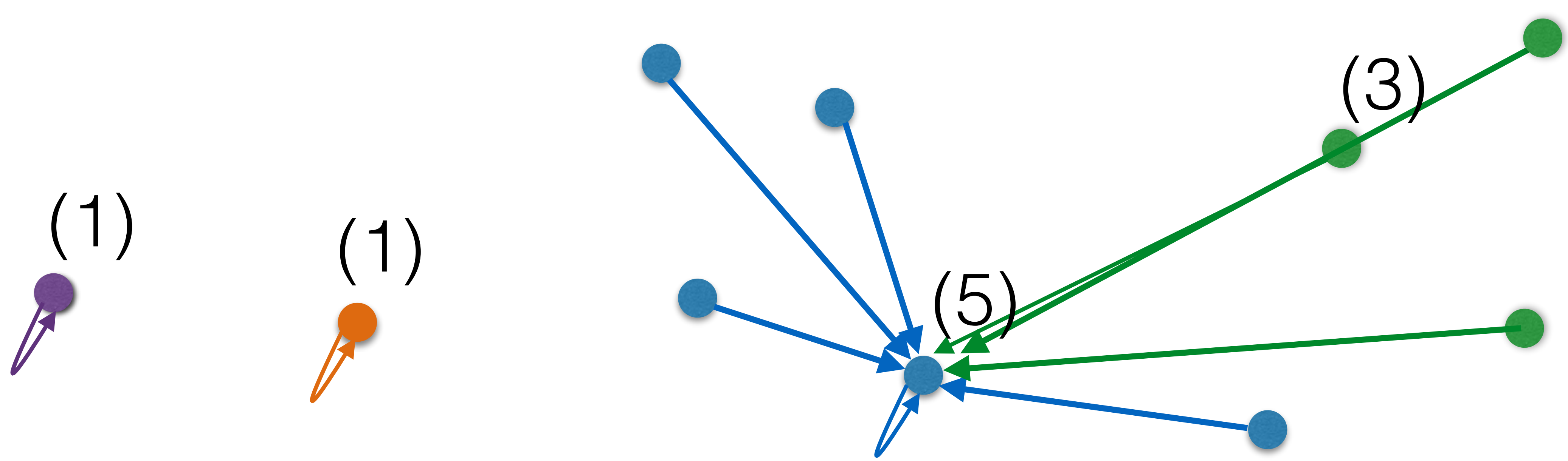
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



Union-Find: Improving Union

Suppose Kruskal's identifies an edge between the blue set and the green set that we want to add. What do we do?

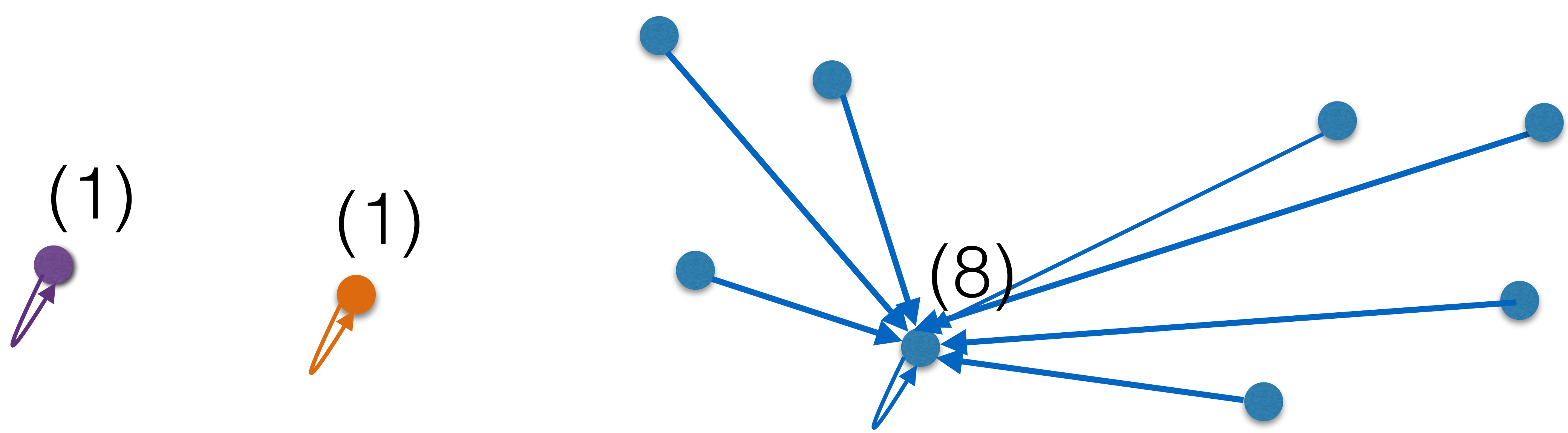
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



Union-Find: Improving Union

Suppose Kruskal's identifies an edge between the blue set and the green set that we want to add. What do we do?

- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



Union Find: Asymptotic Analysis

- Find? $O(1)$ (how?)
- Union?
  - Worst case is $O(n)$ but that's not the whole story
  - Every time we change the label ("head" pointer) of a node, the size of its set at least doubles (**Why?**)
  - Each node's head pointer only changes $O(\log n)$ times

Union Find: **Amortized** Analysis

- Starting with sets of size 1, any n Union operations will take $O(n \log n)$ time

- $O(\log n)$ amortized time for a Union operation

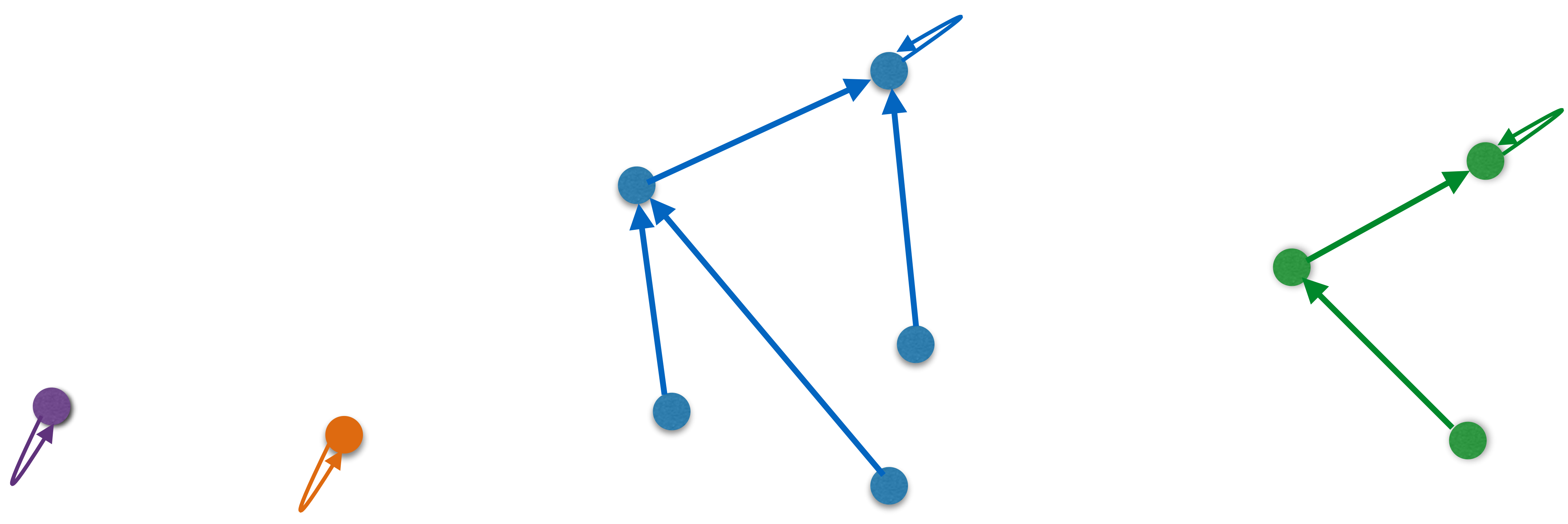
Definition. If n operations take total time $O(t \cdot n)$, then the amortized time per operation is $O(t)$.

Can We Make Union faster?

- What if, instead of
  - $O(1)$ Find, and $O(\log n)$ Union,
  - We want $O(\log n)$ Find, and $O(1)$ Union?
- Any ideas?

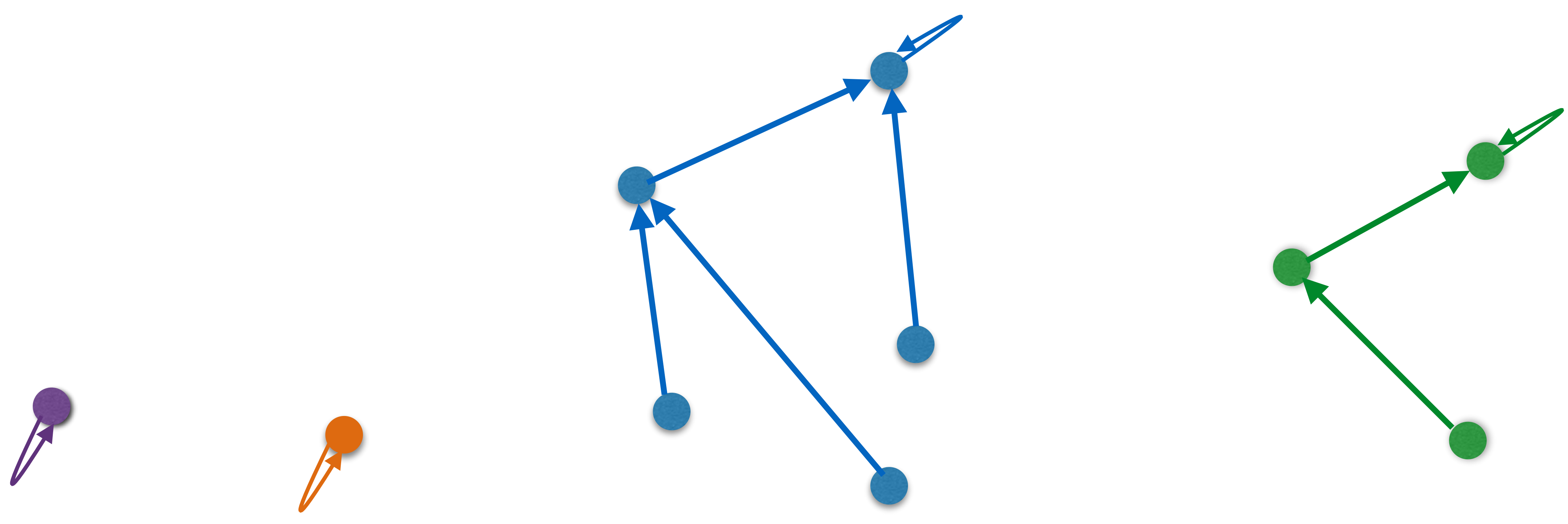
Fast Union with “Trees”

- Let’s keep a **head node** as before
- But now, instead of all nodes in a partition pointing directly to the head node, let’s have our pointers act like a tree
  - Instead of going root-to-leaf, our tree edges point up (“up tree”)



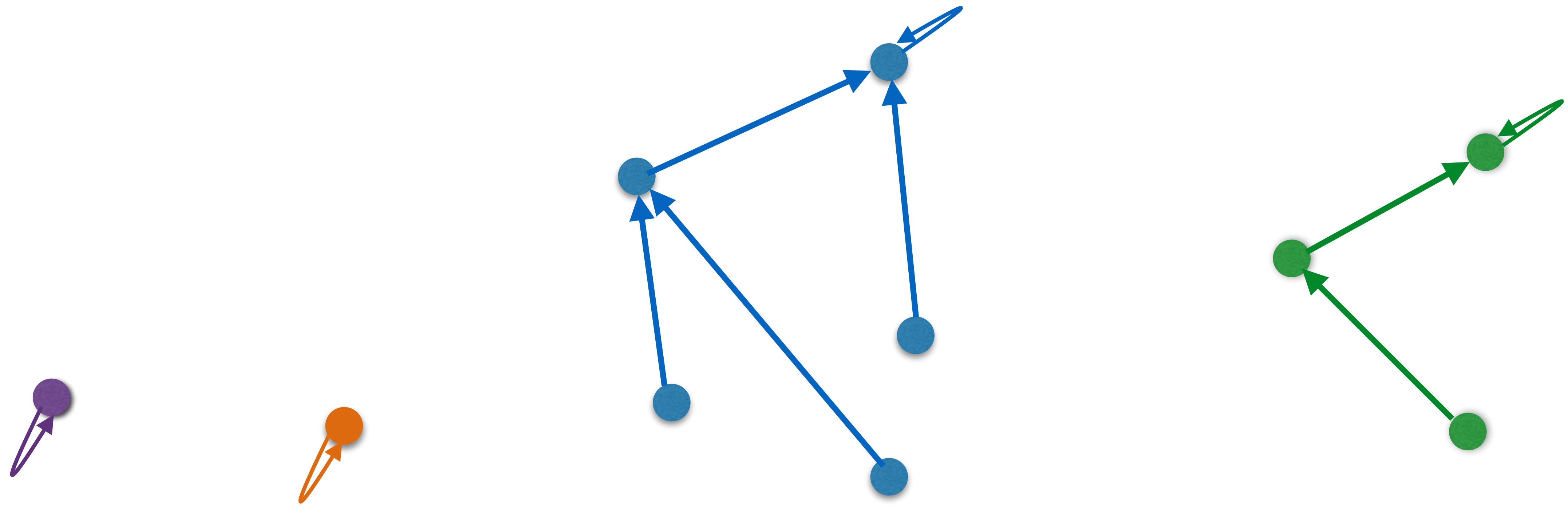
Fast Union with “Trees”

- Each partition has a single head node
- Node pointers act like a tree, but pointing up
- How can we **Find**?



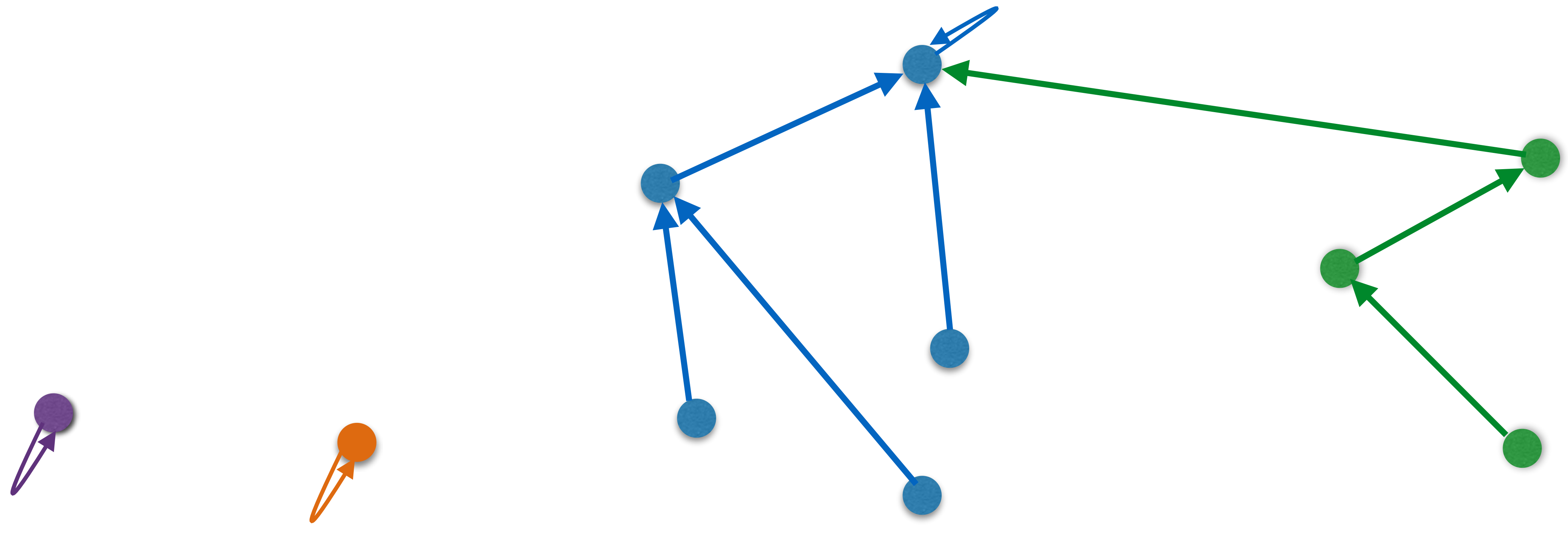
Fast Union with "Trees"

- Each partition has a single head node
- Node pointers act like a tree, but pointing up
- How can we **Union**?



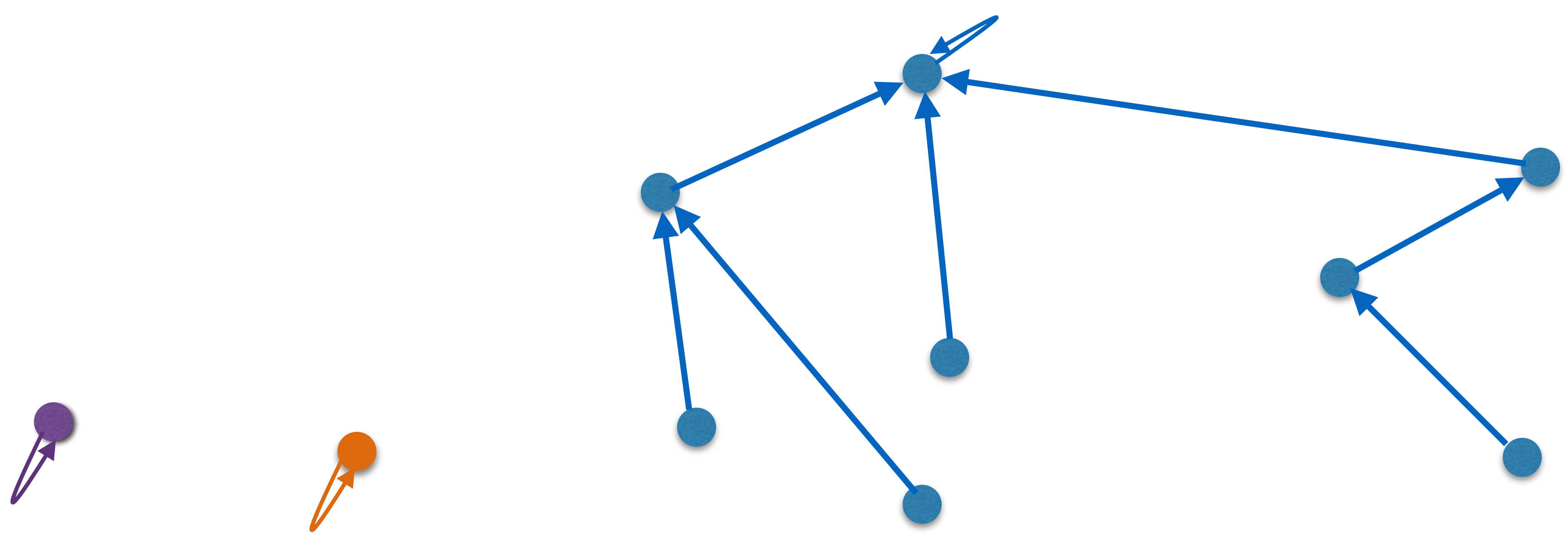
Fast Union with “Trees”

- Each partition has a single head node
- Node pointers act like a tree, but pointing up
- How can we **Union**?



Fast Union with “Trees”

- Each partition has a single head node
- Node pointers act like a tree, but pointing up
- How can we **Union**?



Fast Union with “Trees”

- How can we **Union**?
  - Keep height of each up tree
  - Up tree with smaller height points to up tree of bigger height
  - (At home) show that a set of size k is represented by an up tree of height at most $O(\log k)$

How Fast Is This?

- “Up tree” method:
  - $O(1)$ Union, $O(\log n)$ Find
- “Point-to-head” method:
  - $O(\log n)$ amortized Union, $O(1)$ Find

Class poll!

Do you think we can do better? Which of the following do you think is the case?

A. Either Union or Find take $\Omega(\log n)$

B. If you multiply Union and Find, the product of their times must be $\Omega(\log n)$

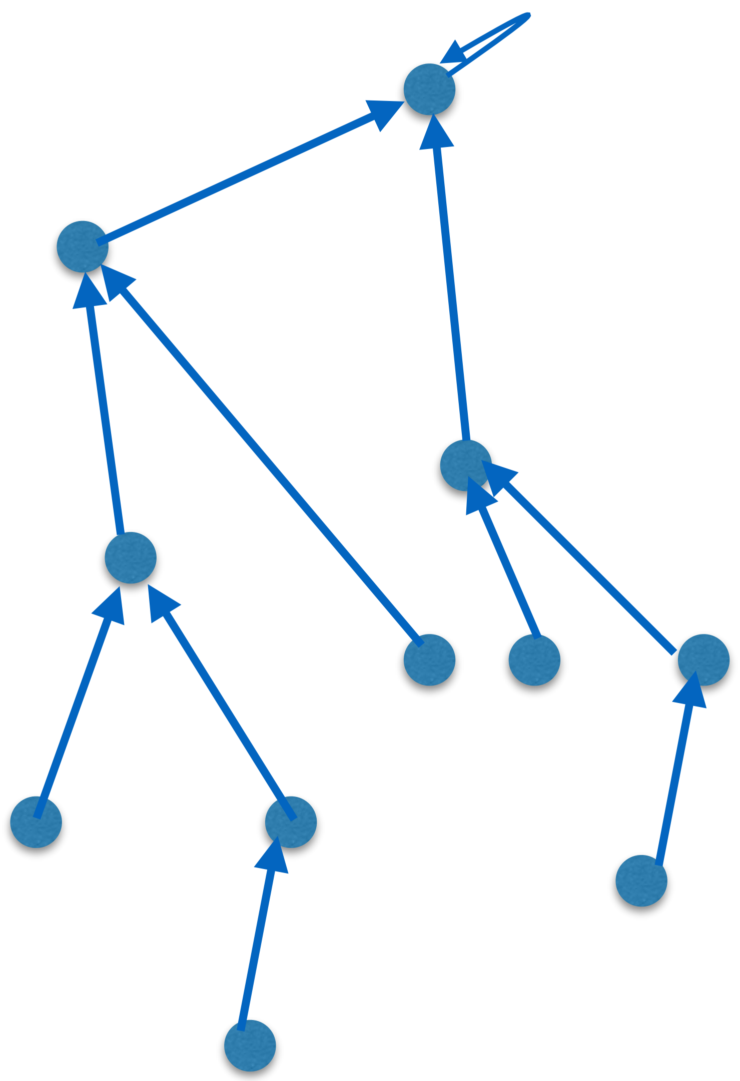
C. Both can be $O(1)$

D. Something in the middle

Let's make things work a little faster in practice

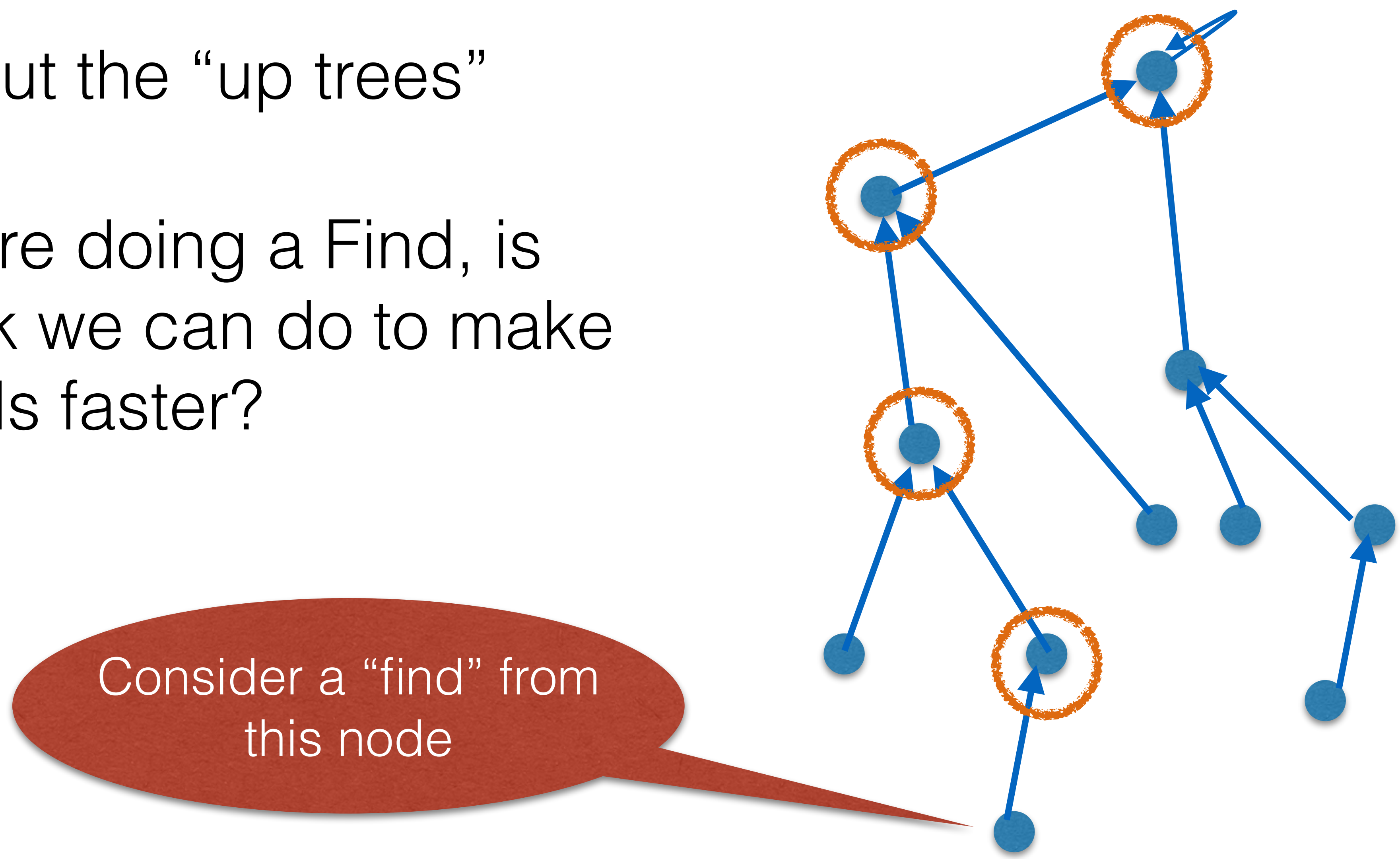
- Think about the "up trees"

- When we're doing a Find, is there work we can do to make future finds faster?



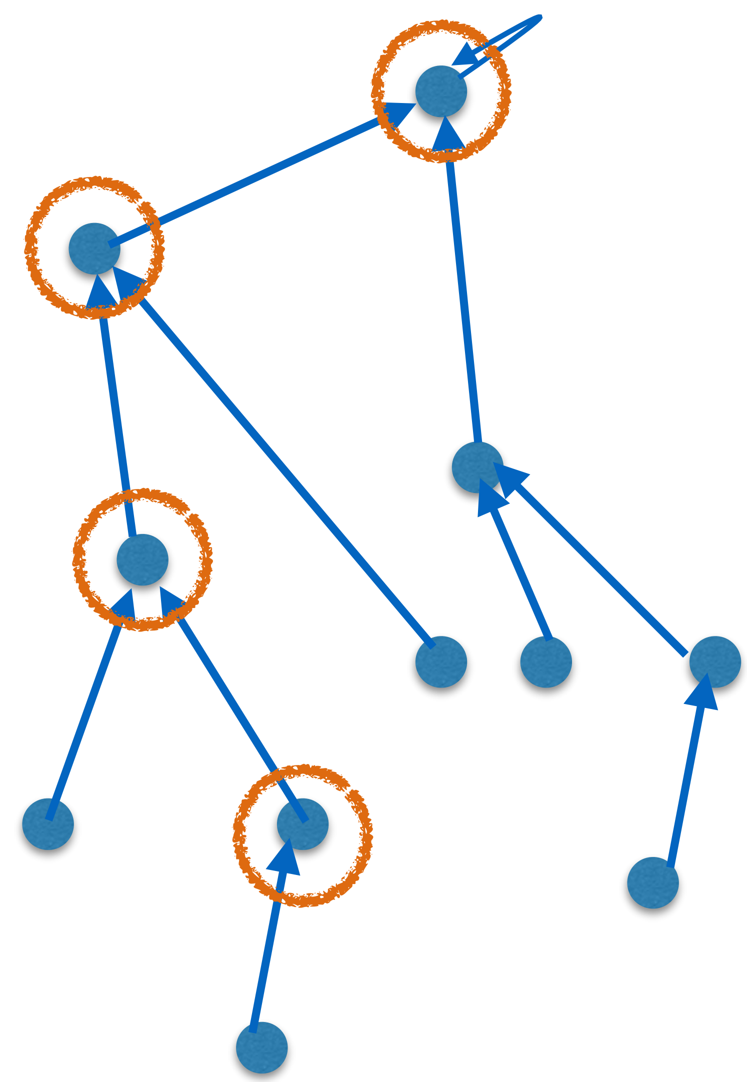
Let's make things work a little faster in practice

- Think about the "up trees"
- When we're doing a Find, is there work we can do to make future finds faster?



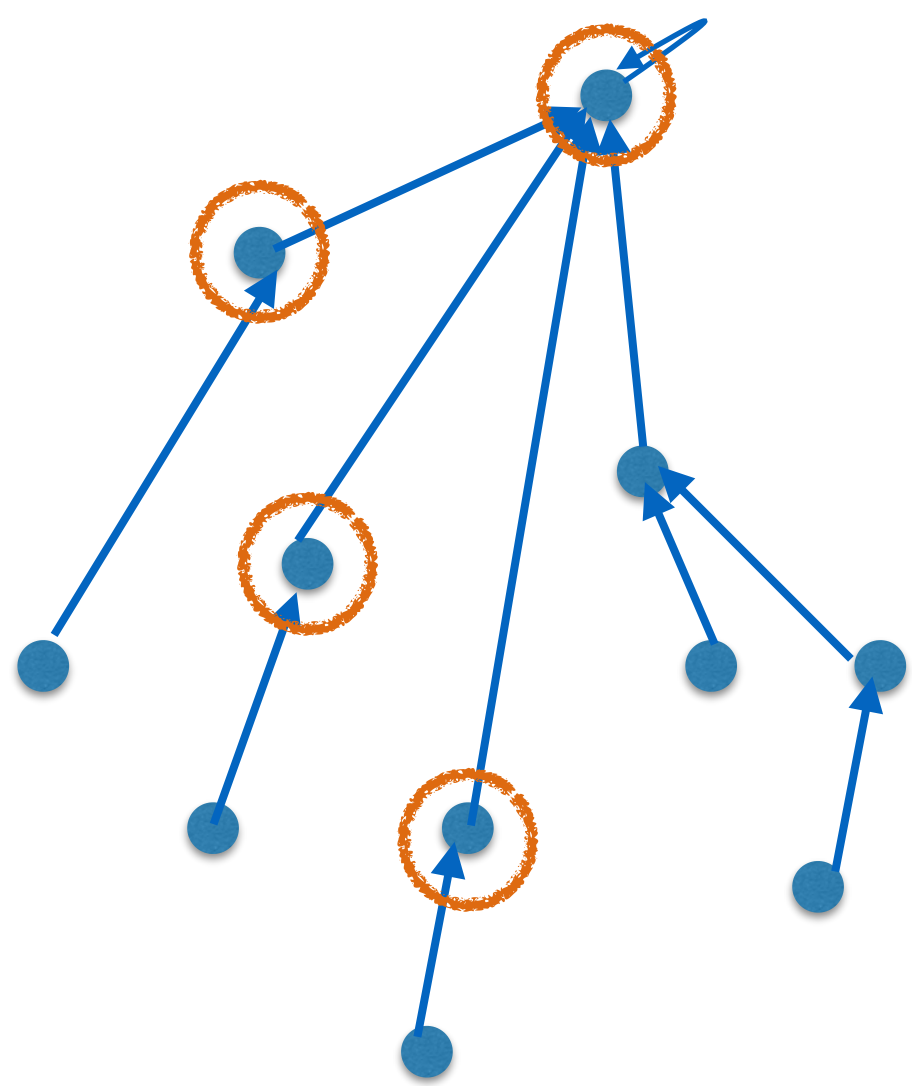
Let's make things work a little faster in practice

- When we're doing a Find, is there work we can do to make future finds faster?
- We really want all of these to point right to the head
- So…let's do that!



Let's make things work a little faster in practice

- When we're doing a Find, is there work we can do to make future finds faster?
- We really want all of these to point right to the head
- So…let's do that!
- Wait, I've broken the data structure!
    - I can't maintain "height" ?!?



Maintaining “Height”

We can’t maintain the ***exact*** height. What if we pretend we can? Just do the same bookkeeping:

- Keep a “rank”
- Always point the head of smaller rank to the head of larger rank; keep rank the same
- If both ranks are the same, point one to the other, and increment the rank

What do we get?

Every time I have an expensive Find, I get a lot of great work done for the future by shrinking the tree

- Called “path compression”

- Now I have an inaccurate “rank” instead of an actual “height”

First: did this make things worse?
Union is still $O(1)$, is Find $O(\log n)$?

- We did **not** make things worse, Find is $O(\log n)$

- Can we show that we made things better?

Surprising Result: Hopcroft Ulman'73

- Amortized complexity of union find with path compression improves significantly!
- Time complexity for n union and find operations on n elements is $O(n \log^* n)$
- $\log^* n$ is the number of times you need to apply the log function before you get to a number <= 1
- Very small! **Less than 5 for all reasonable values**

$$\log^*(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

n	1	2	$4 = 2^2$	$16 = 2^4$	$65,536 = 2^{16}$	$2^{65,536}$
$\log^*(n)$	0	1	2	3	4	5

Digging Deeper

Takeaways

- Kruskal's algorithm is a greedy algorithm to find the MST of a graph
- A heap-based priority queue can be used to efficiently yield edges in order of increasing weight
  - But cycle detection can be expensive!
- Union-Find data structure maintains dynamic partitions of vertices
  - How to detect a cycle by edge (u, v)?
  - Update connected components after adding (u, v)?
- Now we have the tools we need to implement Kruskal's algorithm!

Acknowledgments

- These slides are based on material from Shikha Singh.
- The pictures in these slides are taken from
    - Kleinberg Tardos Slides by Kevin Wayne (https://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/04GreedyAlgorithmsI.pdf)
    - Jeff Erickson's Algorithms Book (http://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf)