Kruskal’s Algorithm and Union Find

Questions or Comments?

Jarník's ("Prims Algorithm")

- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
  - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
  - $T \leftarrow T \cup \{e\}$
  - $S \leftarrow S \cup \{v\}$
- **Implementation crux.** Find and add min-cost edge for the cut $(S, V - S)$ and add it to the tree in each iteration, update cut edges
- How can we prove that this finds the MST?
  - Cut property! (On board.)

Jarník's ("Prims Algorithm")

- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \emptyset$
- While $|T| \leq n - 1$:
    - Find the min-cost edge $e = (u, v)$ with one end $u \in S$ and $v \in V - S$
    - $T \leftarrow T \cup \{e\}$
    - $S \leftarrow S \cup \{v\}$
- **Implementation crux.** Find and add min-cost edge for the cut $(S, V - S)$ and add it to the tree in each iteration, update cut edges
- Running time?
    - Naive implementation may take $O(nm)$
    - Need to maintain set of edges adjacent to nodes in T and extract min-cost cut edge from it each time
    - Which data structure from CS 136 can we use?

CS136 Review: Priority Queue

Managing such a set typically involves the following operations on S

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

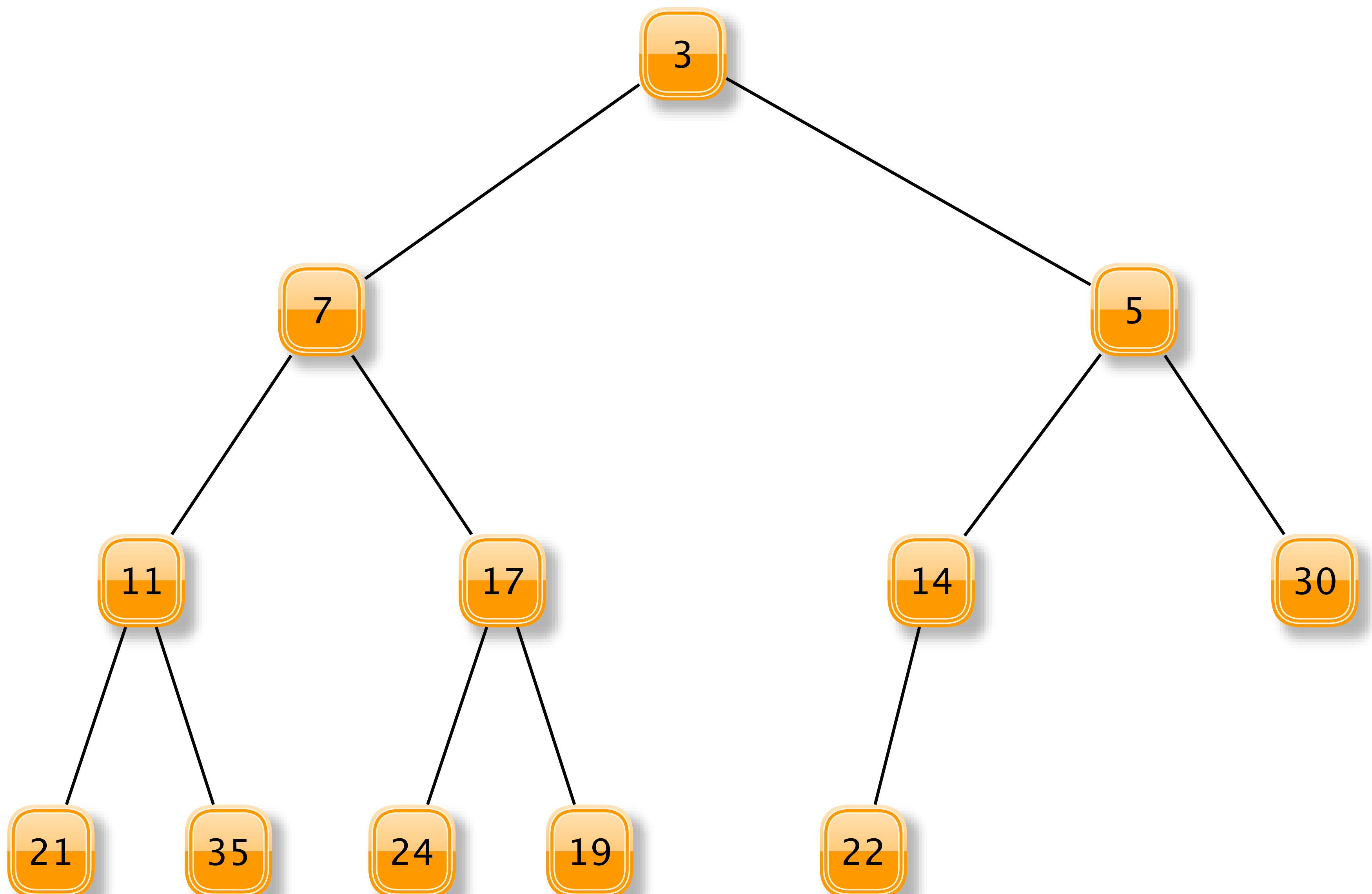
Priorities are encoded as a ‘key’ value

Typically: higher priority <—> lower key value

Heap as Priority Queue. Combines tree structure with array access

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

Heap Example



H

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

"Prims" Implementation

- Use **Binary heaps**
  - Create a priority queue initially holding all edges incident to u.
  - At each step, dequeue edges from the priority queue until we find an edge (x, y) where $x \in S$ and$y \notin S$.
  - Add (x, y) to T.
  - Add to the queue all edges incident to y whose endpoints aren't in S.
  - Each edge is enqueued and dequeued at most once
  - Total runtime: $O(m \log m)$
    - In any graph, $m = O(n^2)$
    - So $O(m \log m) = O(m \log n)$

“Prims” Implementation

- Implementation using **Binary heaps**
  - Total runtime: $O(m \log n)$
  - Can we do better?

- If a **Fibonacci heap** is used instead of binary heap:
  - Supports **amortized** $O(1)$-time inserts, $O(\log n)$ time extract min
  - Runs in $O(m + n \log n)$ total time

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

Kruskal’s Algorithm

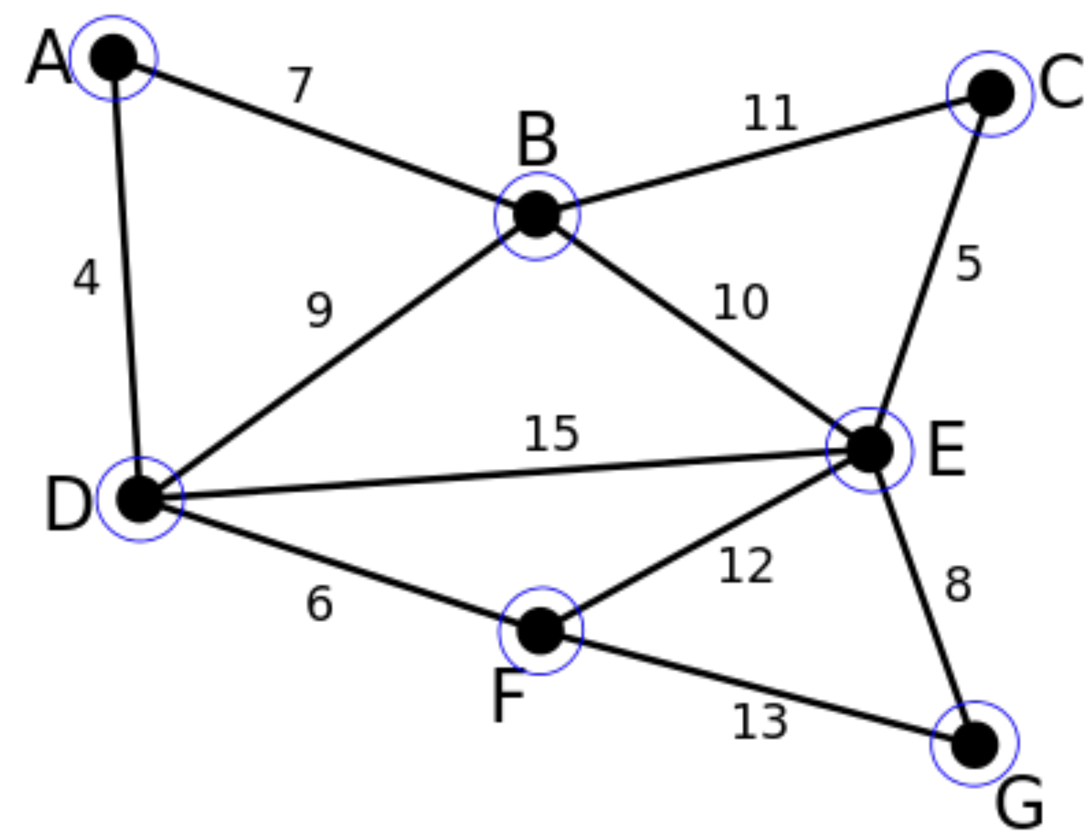
- Another MST algorithm

- Why do you think we’re looking at a second one?

Kruskal's Algorithm

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



Kruskal’s Algorithm

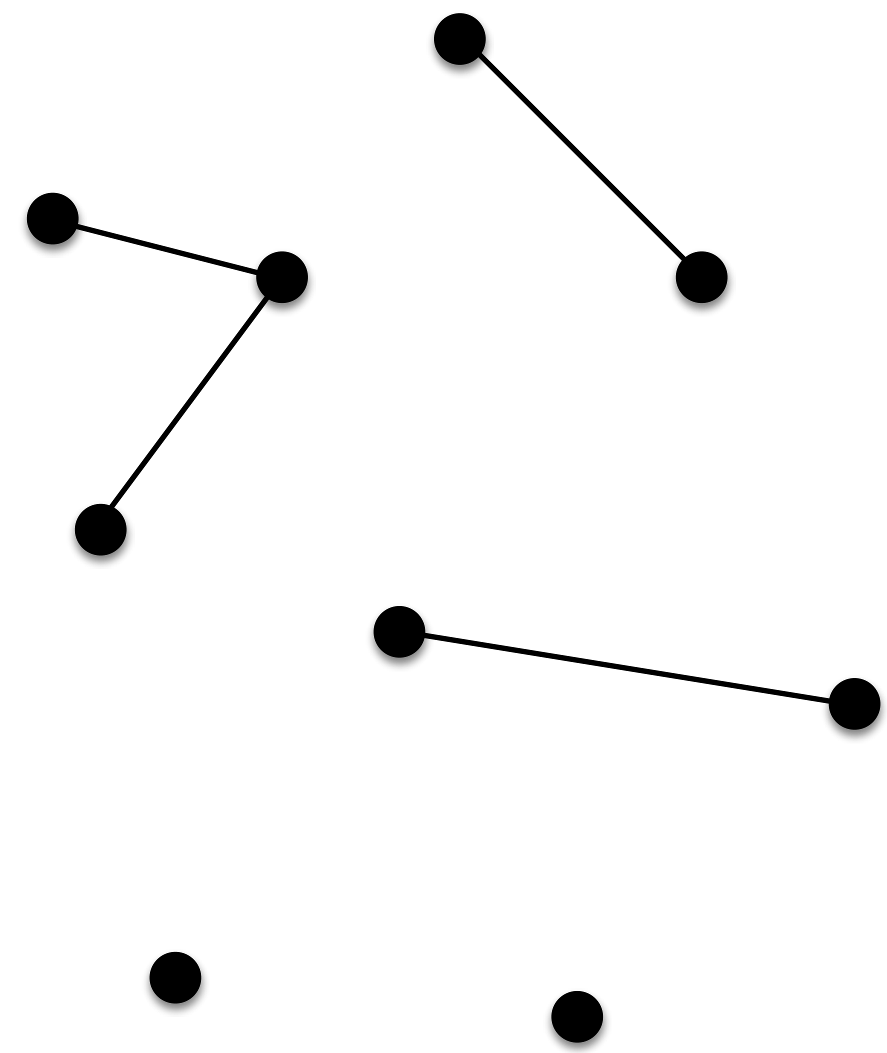
- Does it give us the correct MST?
  - Proof?
- How quickly can we find the minimum remaining edge?
- How quickly can we determine if an edge creates a cycle?

Kruskal's Implementation

- Sort edges by weight: $O(m \log m)$
  - Turns out this is the dominant cost
- Determine whether $T \cup \{e\}$ contains a cycle
  - Maintain a partition of V: components of T
  - Let $[u]$ denote component of u
  - Adding edge $e = (v, w)$ creates a cycle if and only if $[v] = [w]$
- Add an edge to T: update components

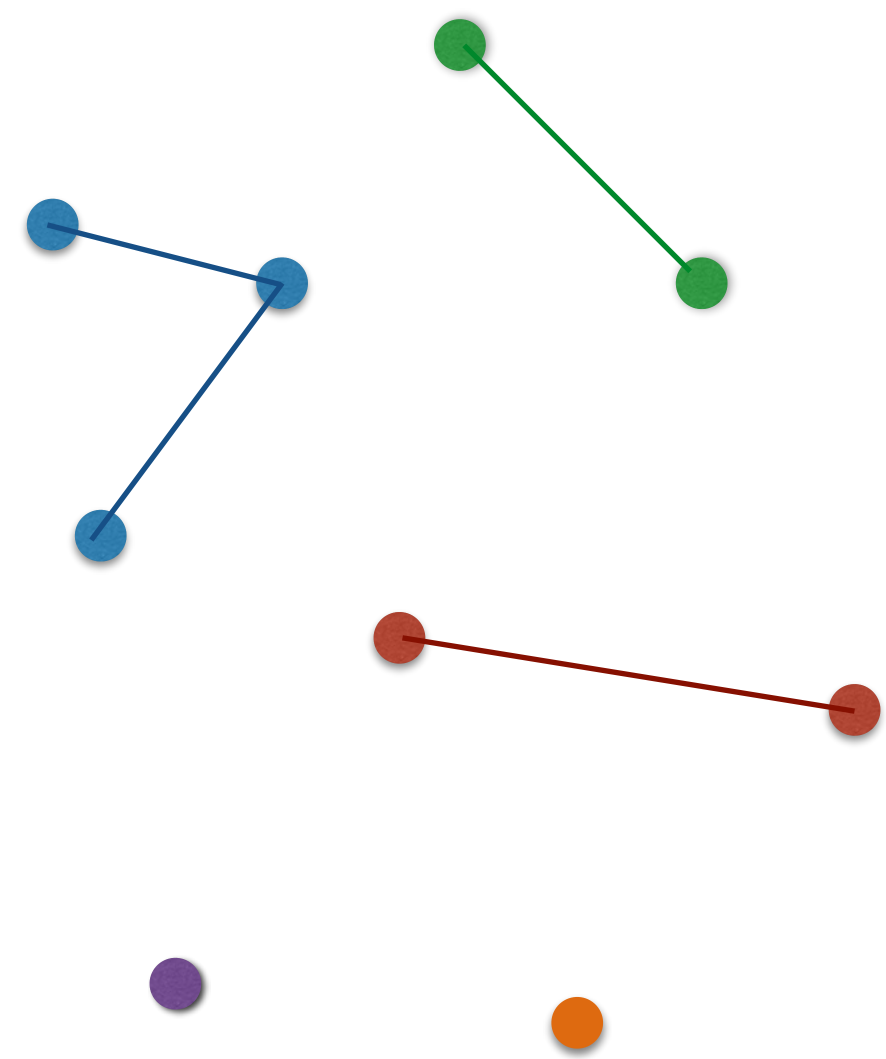
Does this edge create a cycle?

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



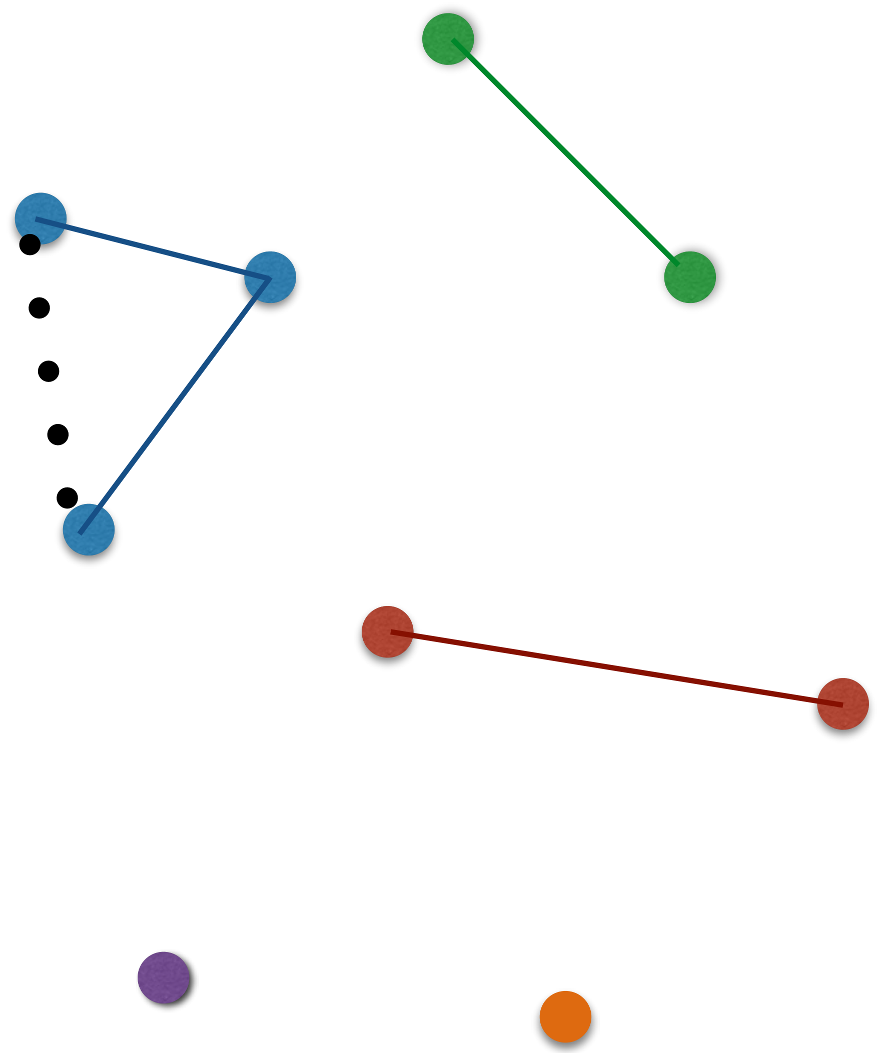
Does this edge create a cycle?

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



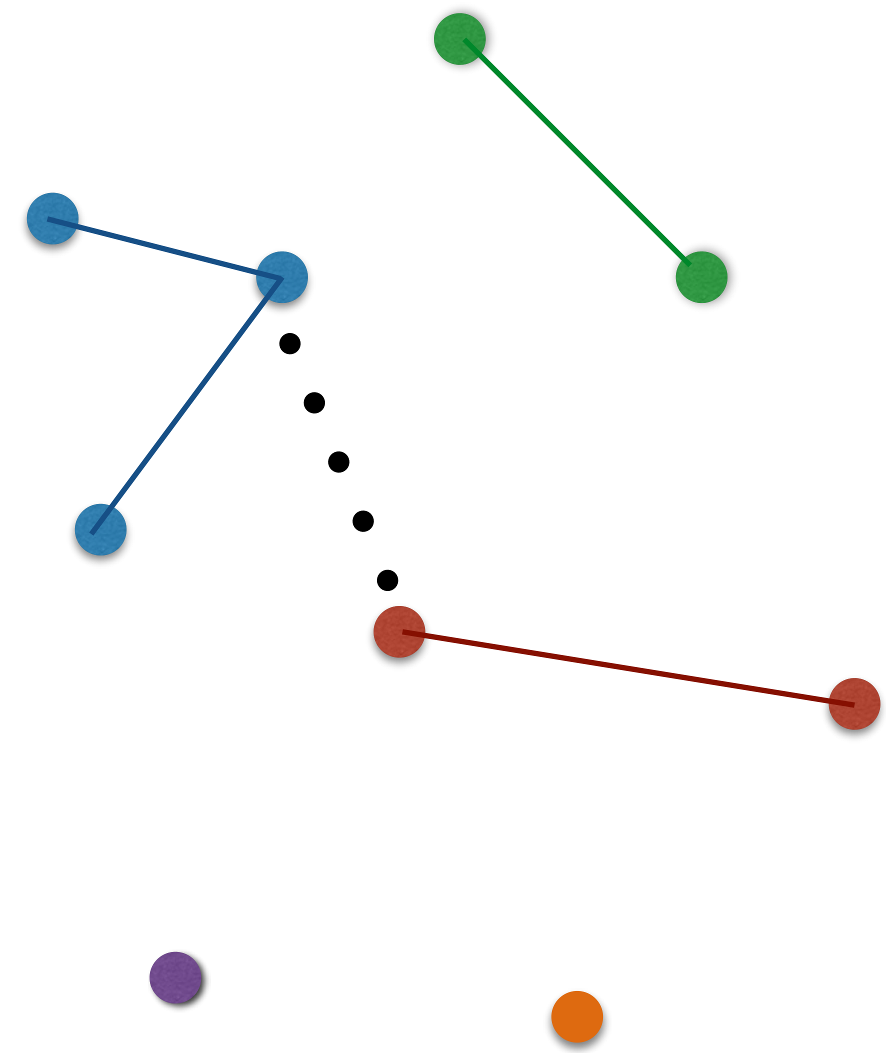
Does this edge create a cycle?

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



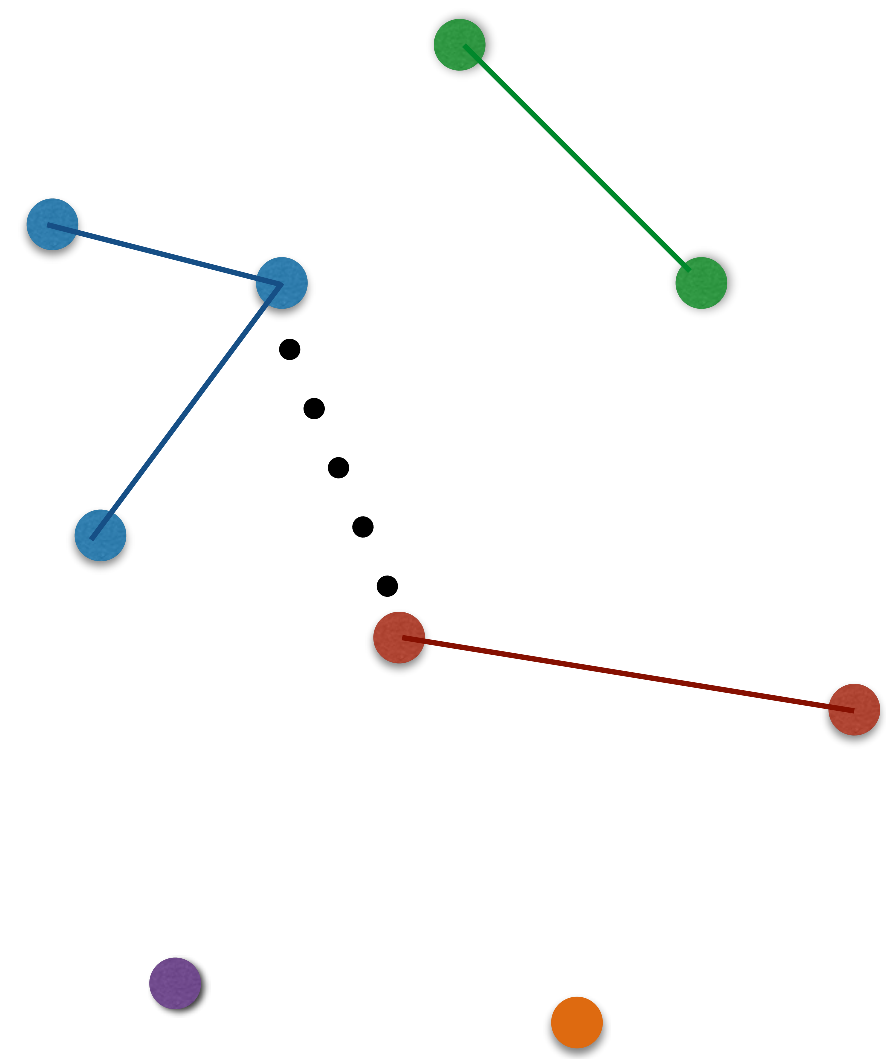
Does this edge create a cycle?

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



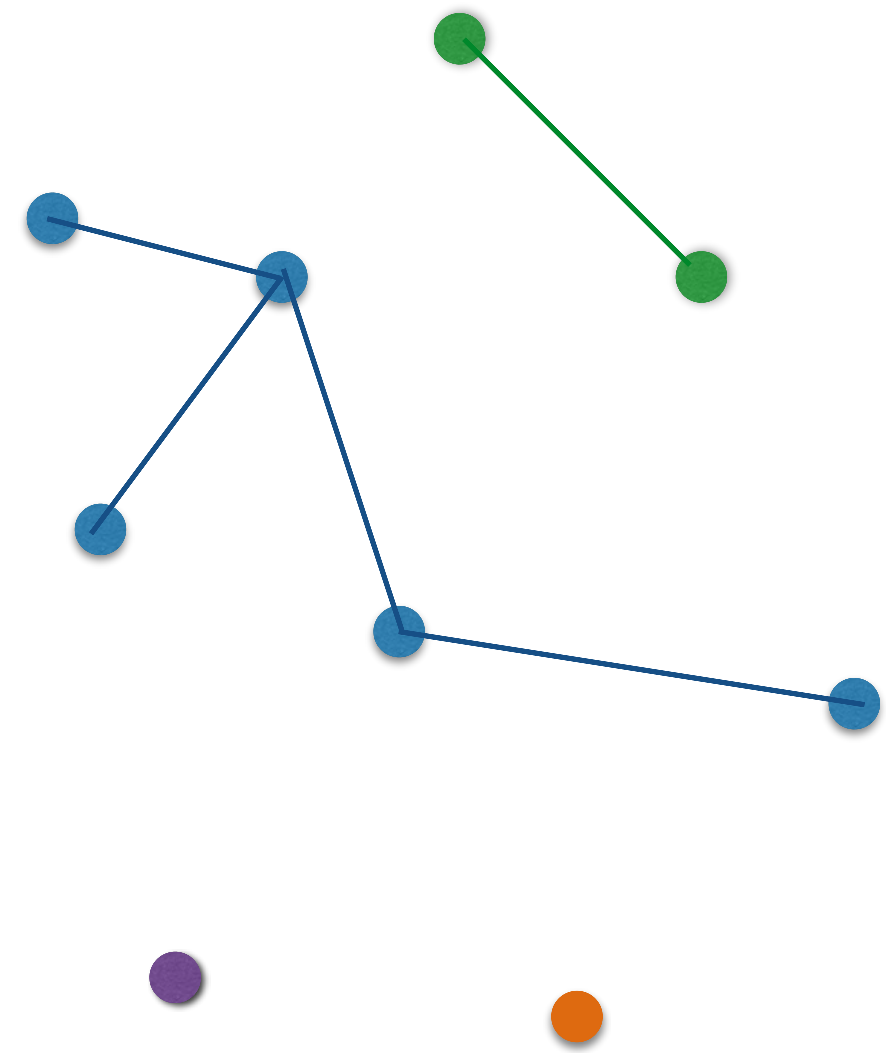
Does this edge create a cycle?

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



What do we want to be able to do?

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

Union-Find Data Structure

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

- Provides the following methods:
  - `MakeUnionFind()`: Initialize
  - `Find(x)`: Return name of set containing x
  - `Union(X, Y)`: Replace sets `X, Y` with $X \cup Y$

Kruskal's Algorithm can then use

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

Union-Find: Any Ideas?

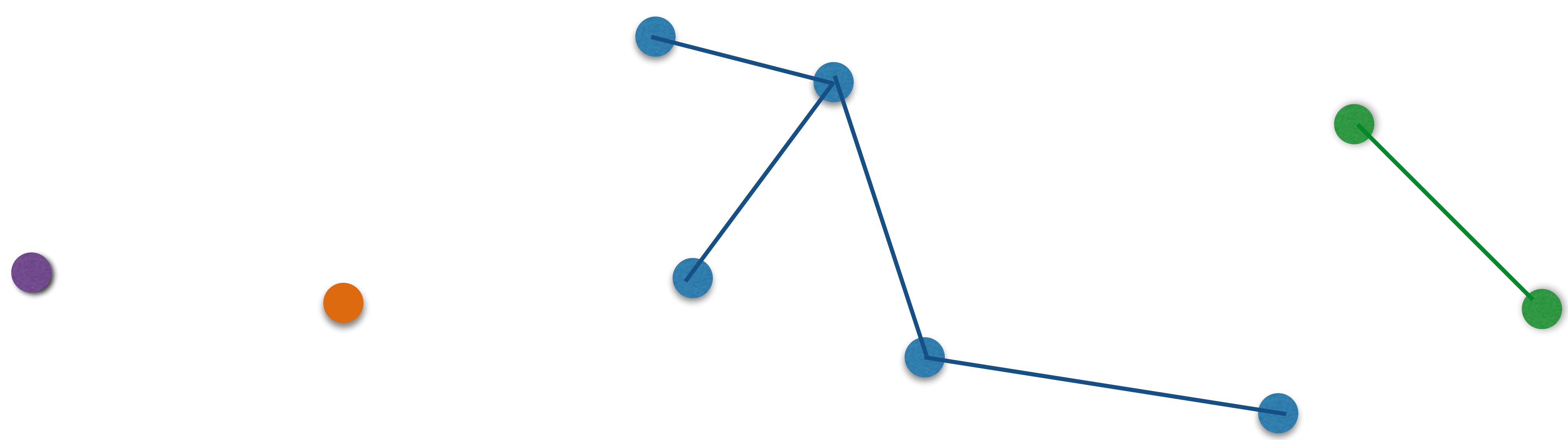
How can we get:

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

(Hint: we'll be maintaining labels)

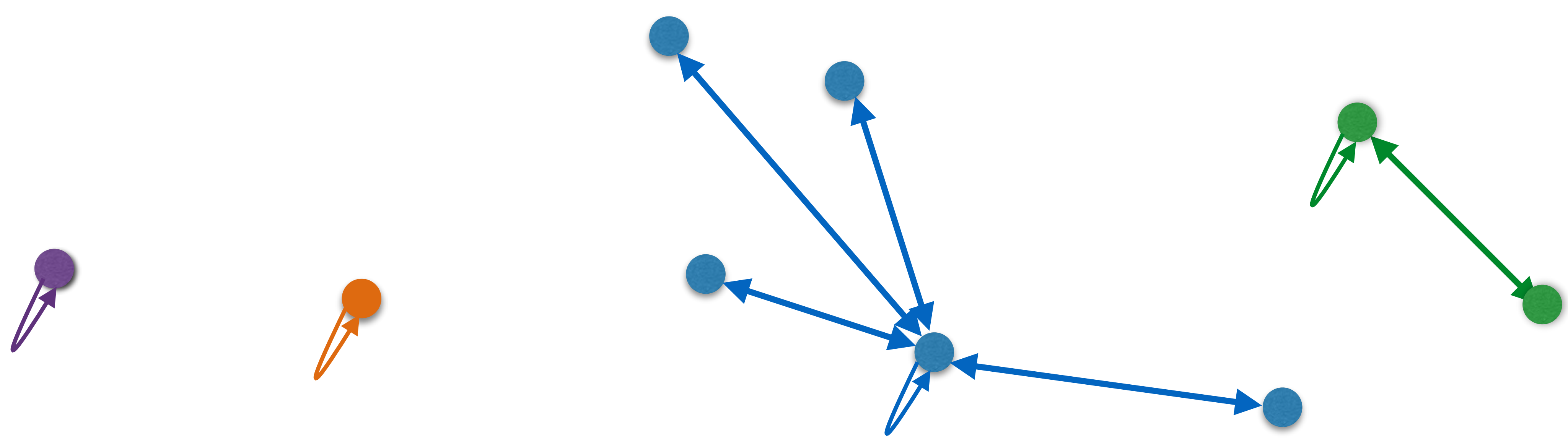
Union-Find: Improving Union

- Let's perturb that idea just a little bit and analyze it more tightly
- Each vertex points to a "head" node instead of a label; head points to itself



Union-Find: Improving Union

- Let’s perturb that idea just a little bit and analyze it more tightly (keep colors just to help)
- Each vertex points to a “head” node instead of a label; head points to itself (keep back pointers too)

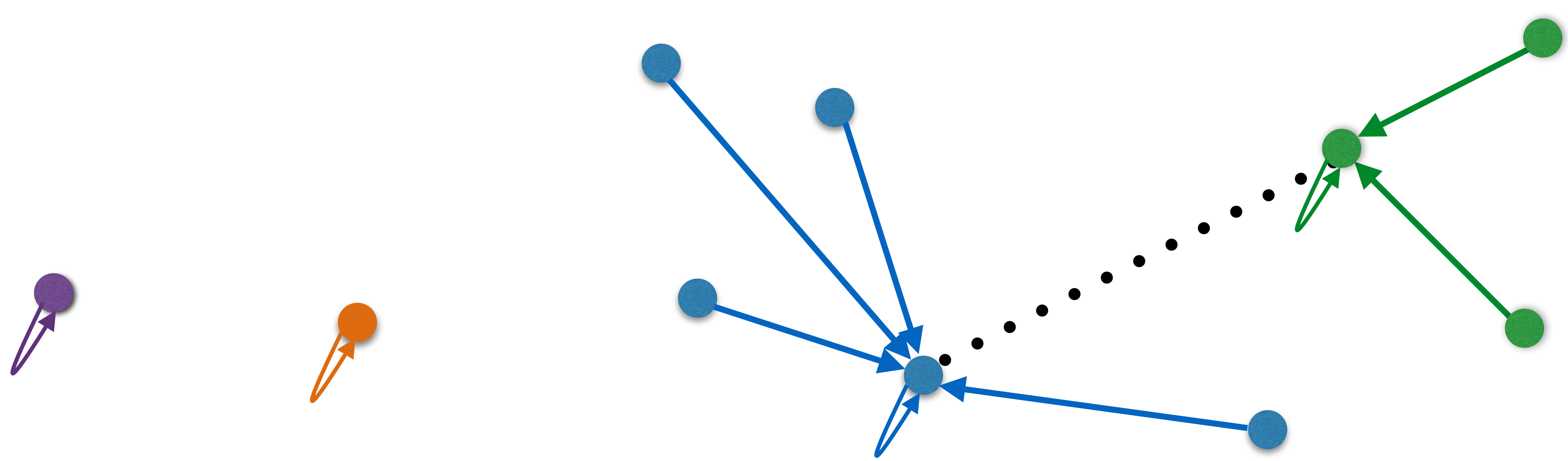


Union-Find: Improving Union

- Let’s perturb that idea just a little bit and analyze it more tightly
- Each vertex points to a “head” node instead of a label; head points to itself
- Also store size of each set in the head
    - How can we maintain that efficiently?
- Now, to do a union, make every element in the smaller set point at the head of the larger set
    - Update the size

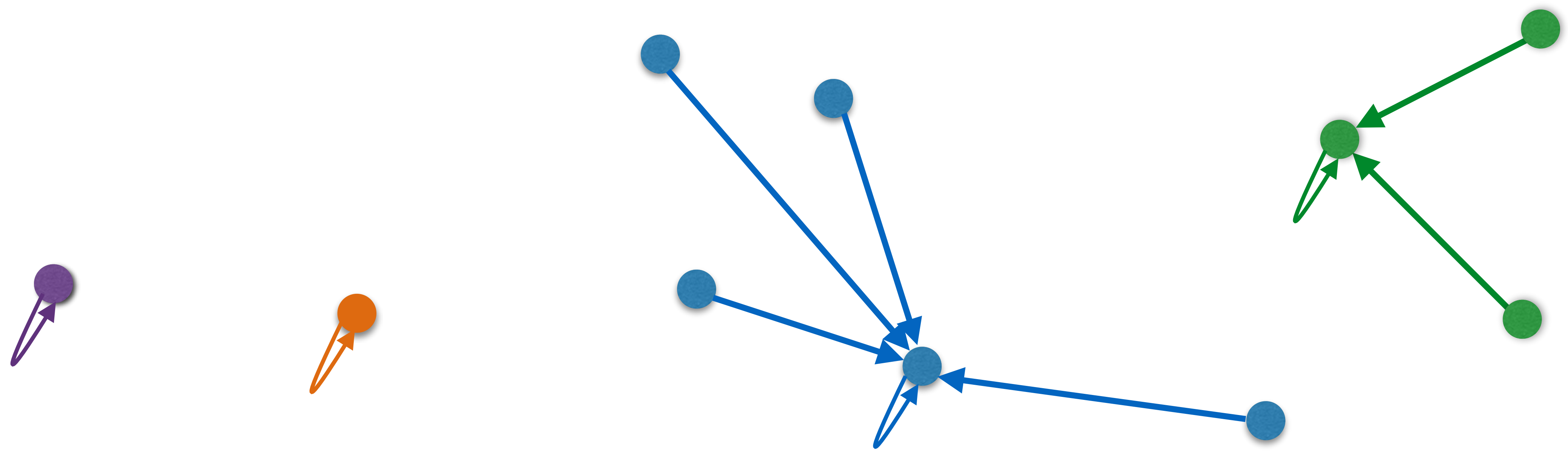
Union-Find: Improving Union

- Let's say we have an edge between the blue tree and the green tree
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



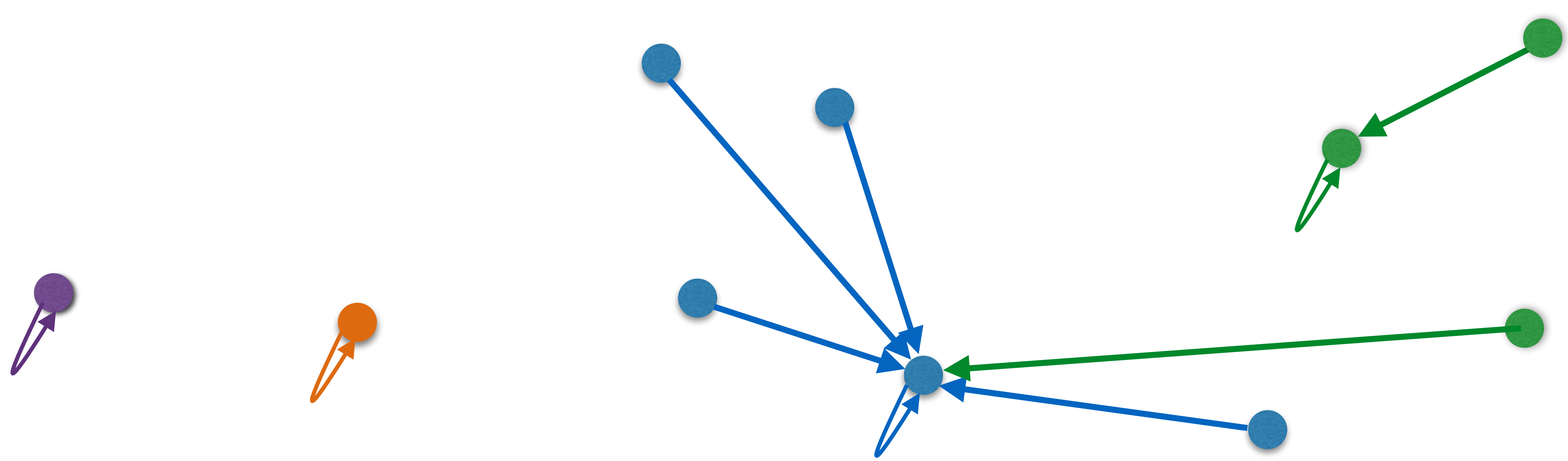
Union-Find: Improving Union

- Let’s say we have an edge between the blue tree and the green tree
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



Union-Find: Improving Union

- Let's say we have an edge between the blue tree and the green tree
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node

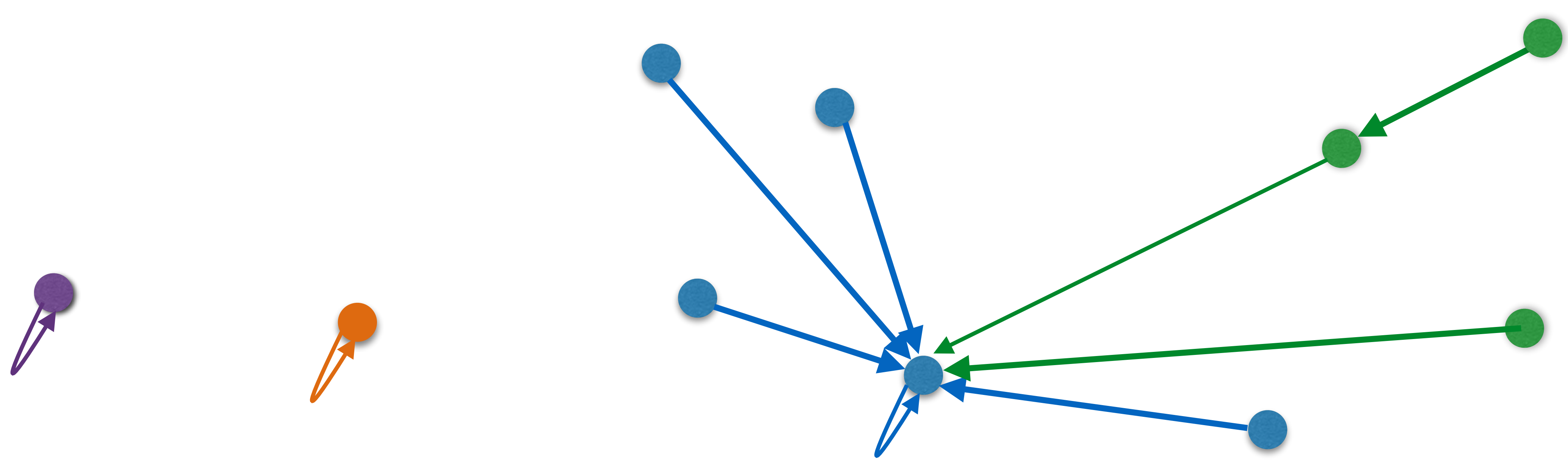


Union-Find: Improving Union

- Let's say we have an edge between the blue tree and the green tree

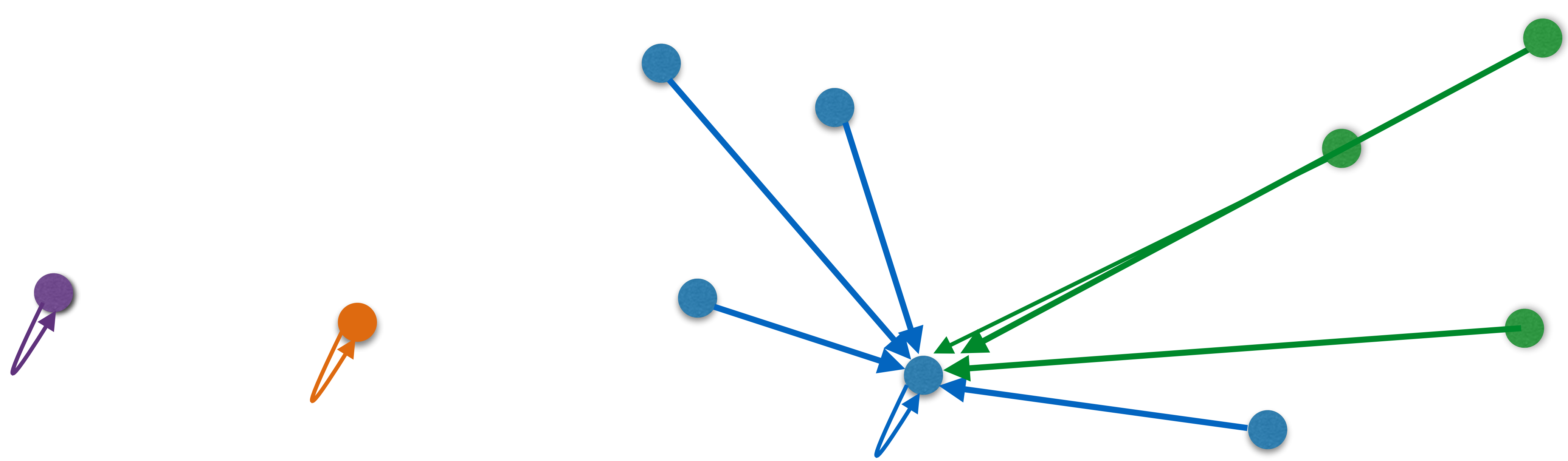
- Update the green tree!

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



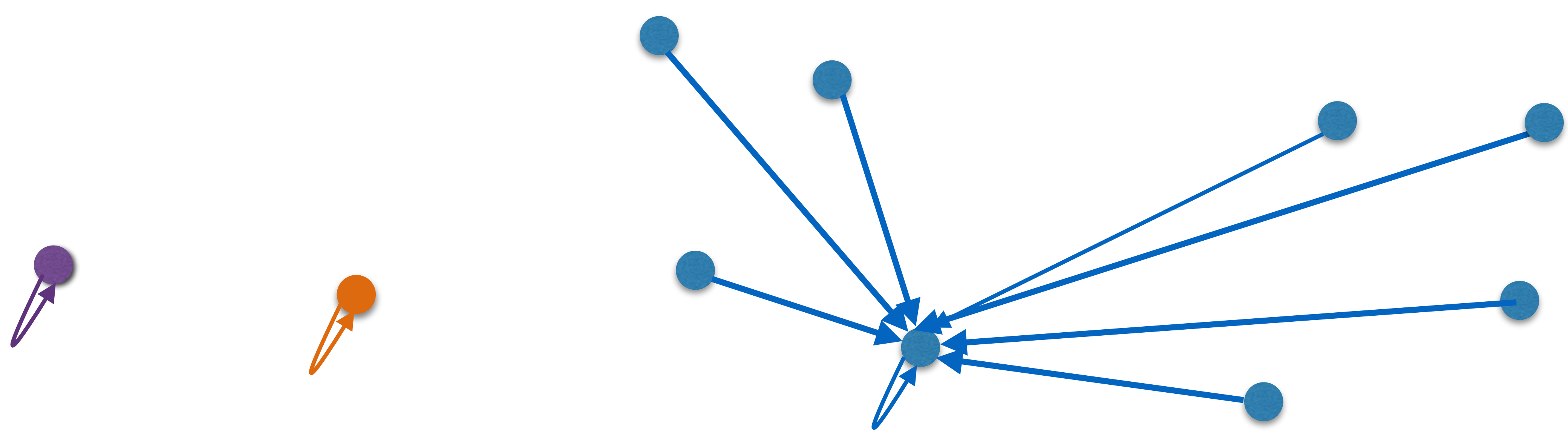
Union-Find: Improving Union

- Let’s say we have an edge between the blue tree and the green tree
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



Union-Find: Improving Union

- Let’s say we have an edge between the blue tree and the green tree
- Update the green tree!
- Follow back pointers from the head of the tree so we get every node



Union Find: Improving Union 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
  - Each node's head pointer only changes $O(\log n)$ times

Union Find: Improving Union Analysis

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

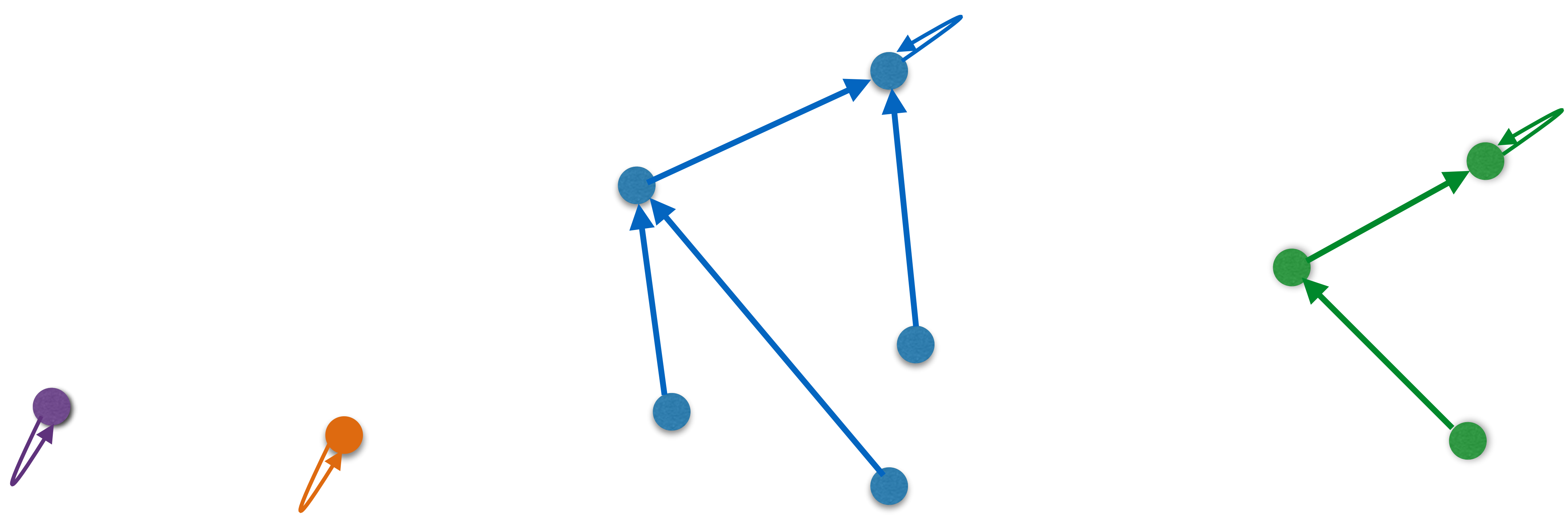
Definition. If k operations take total time $O(t \cdot k)$, then the amortized time per operation is $O(t)$.

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?

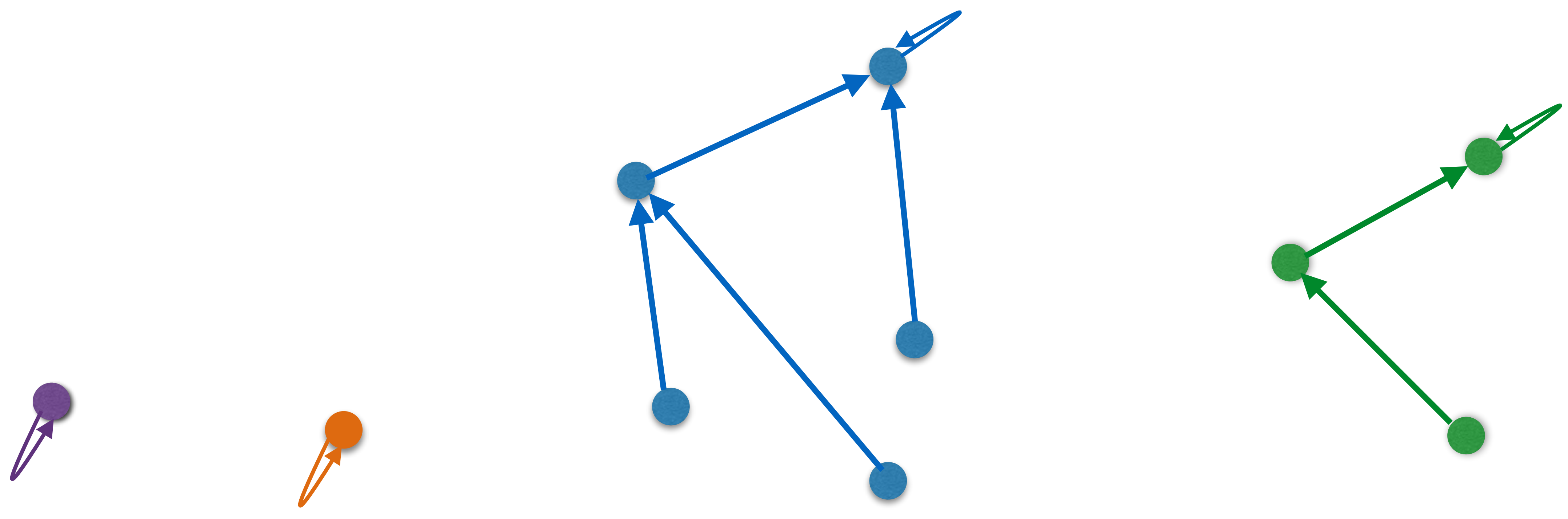
Fast Union with “Trees”

- Let’s keep a head node as before
- Now, let’s have our pointers act like a tree, but pointing up (“up tree”)



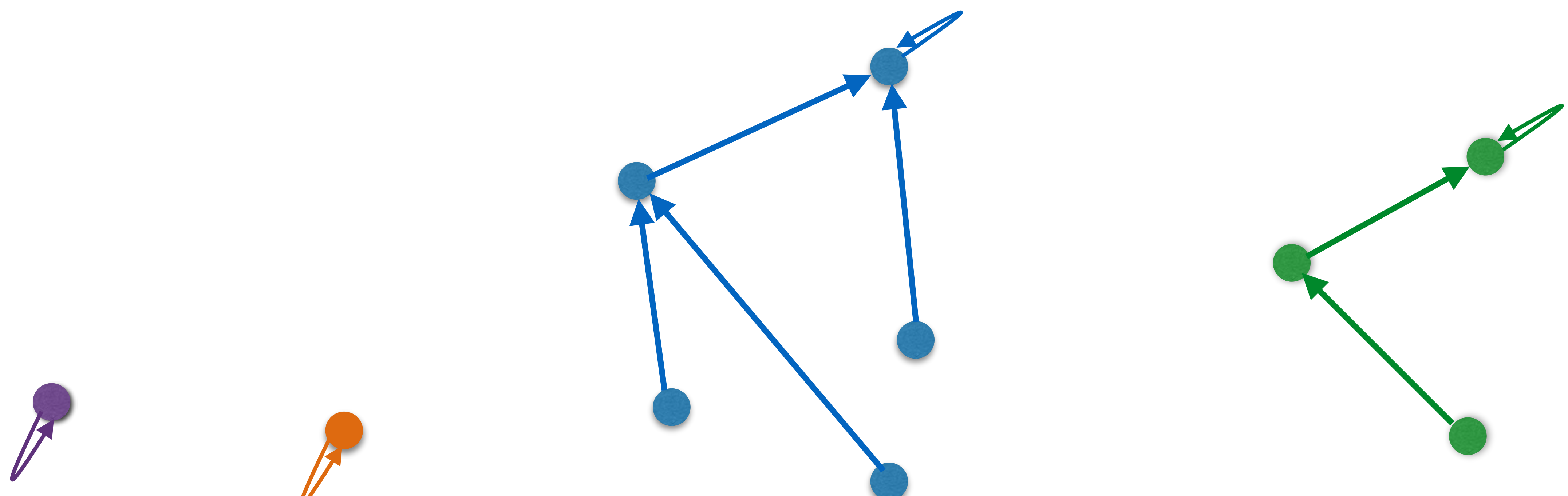
Fast Union with “Trees”

- Let’s keep a head node as before
- Now, let’s have our pointers act like a tree, but pointing up
- How can we Find?



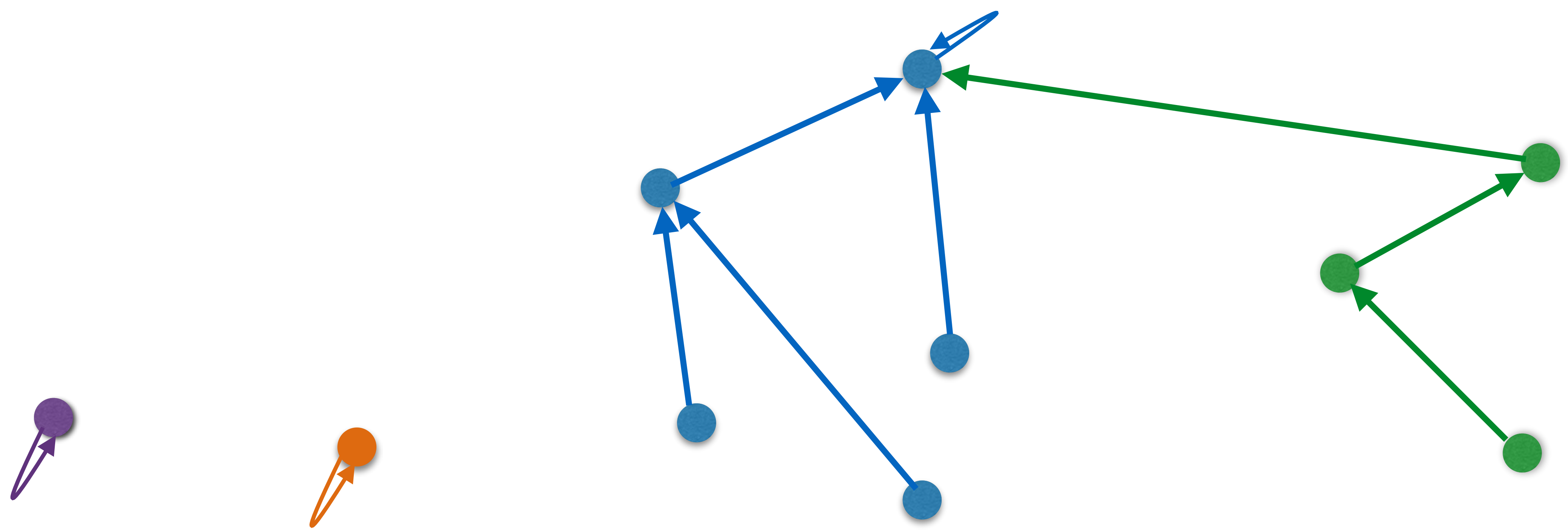
Fast Union with "Trees"

- Let's keep a head node as before
- Now, let's have our pointers act like a tree, but pointing up
- How can we Union?



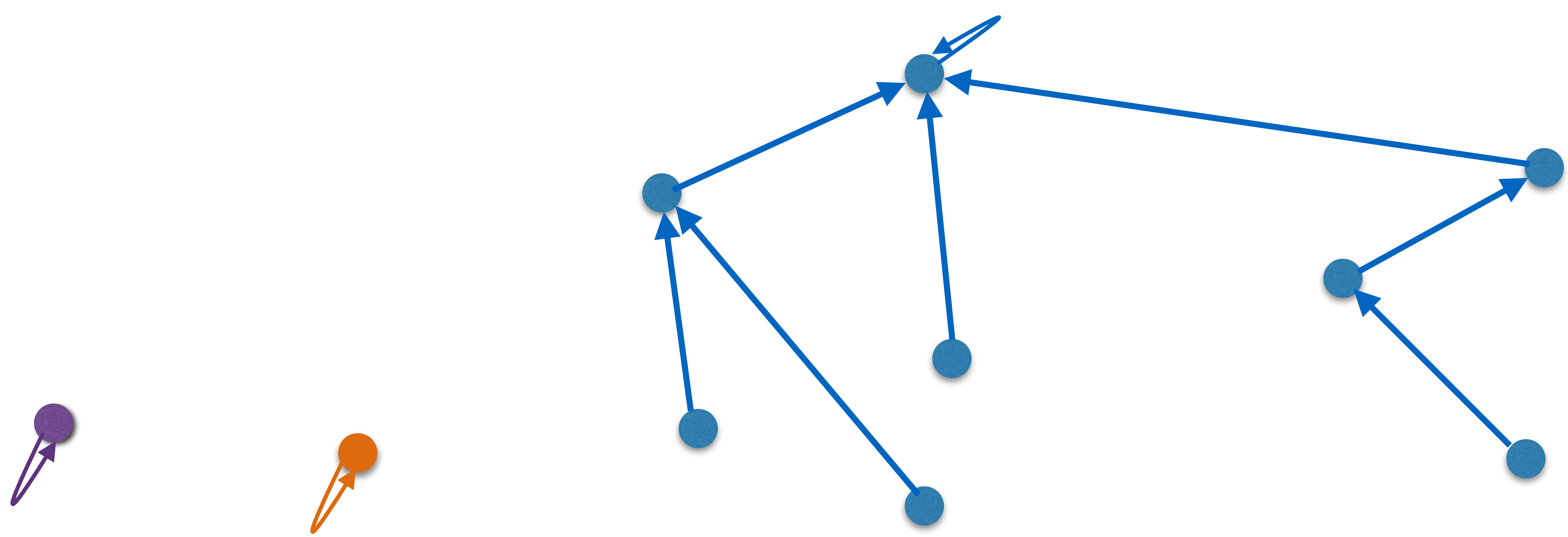
Fast Union with "Trees"

- Let's keep a head node as before
- Now, let's have our pointers act like a tree, but pointing up
- How can we Union?



Fast Union with "Trees"

- Let's keep a head node as before
- Now, let's have our pointers act like a tree, but pointing up
- How can we Union?



Fast Union with “Trees”

- Let’s keep a head node as before
- Now, let’s have our pointers act like a tree, but pointing up
- 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 $O(\log k)$

What do we get?

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

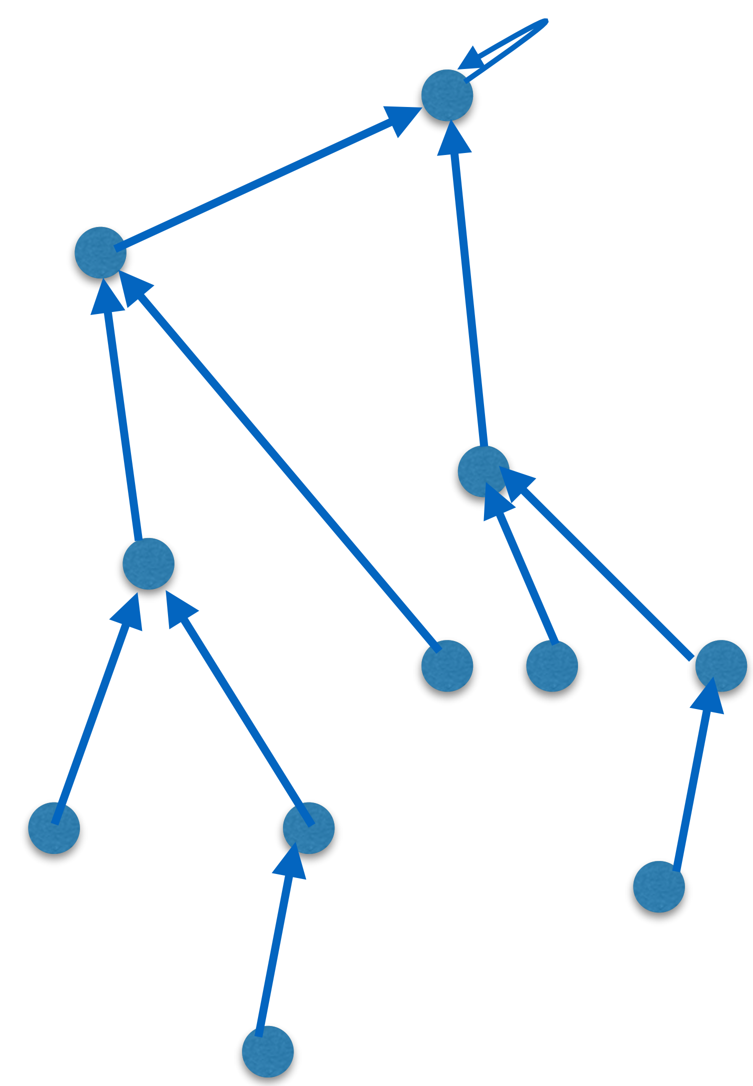
Class poll!

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

- Either Union or Find take $\Omega(\log n)$
- If you multiply Union and Find, the product of their times must be $\Omega(\log n)$
- Both can be $O(1)$
- Something in the middle

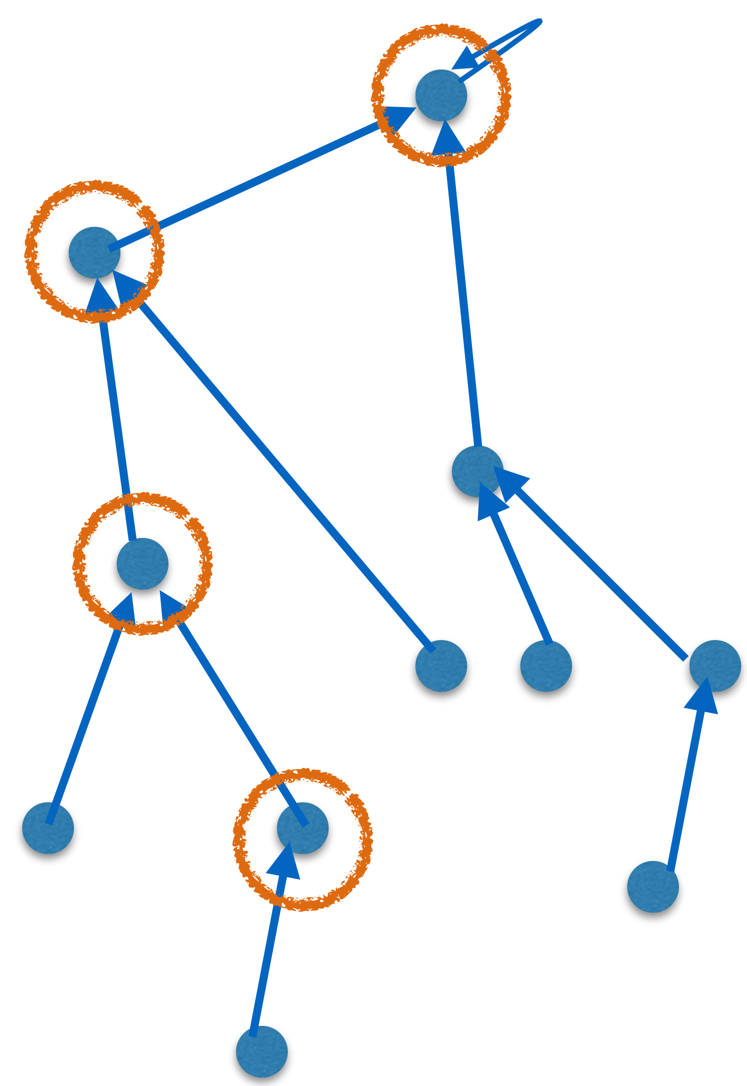
Let’s make things work a little faster in practice

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



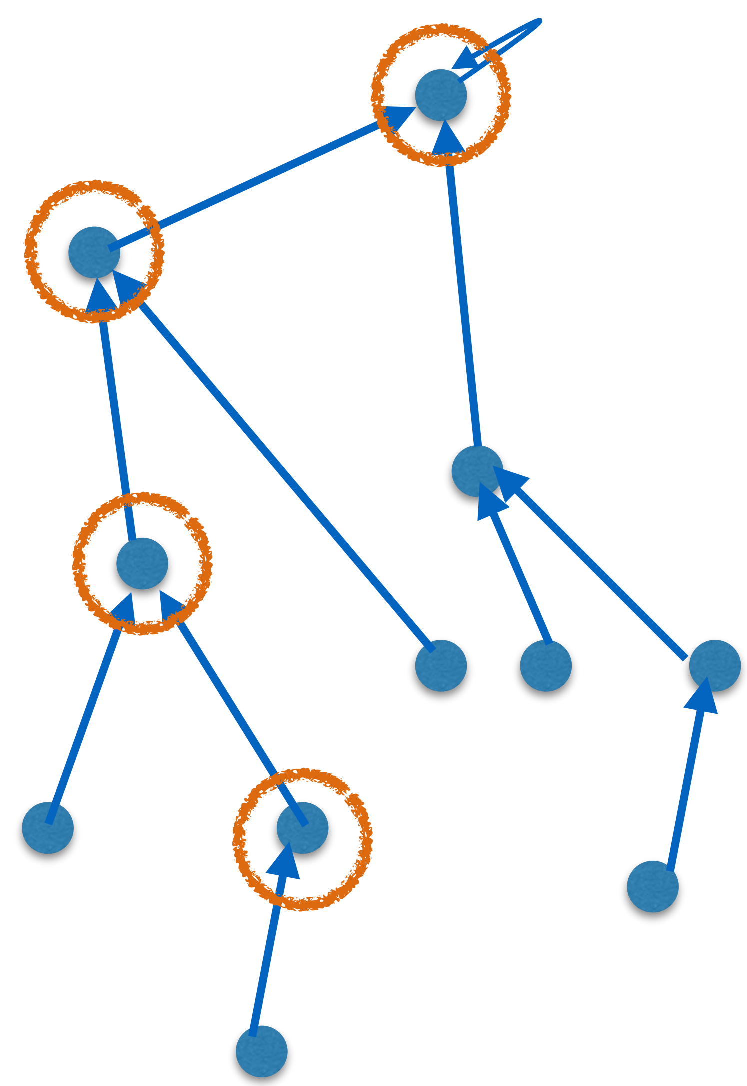
Let’s make things work a little faster in practice

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



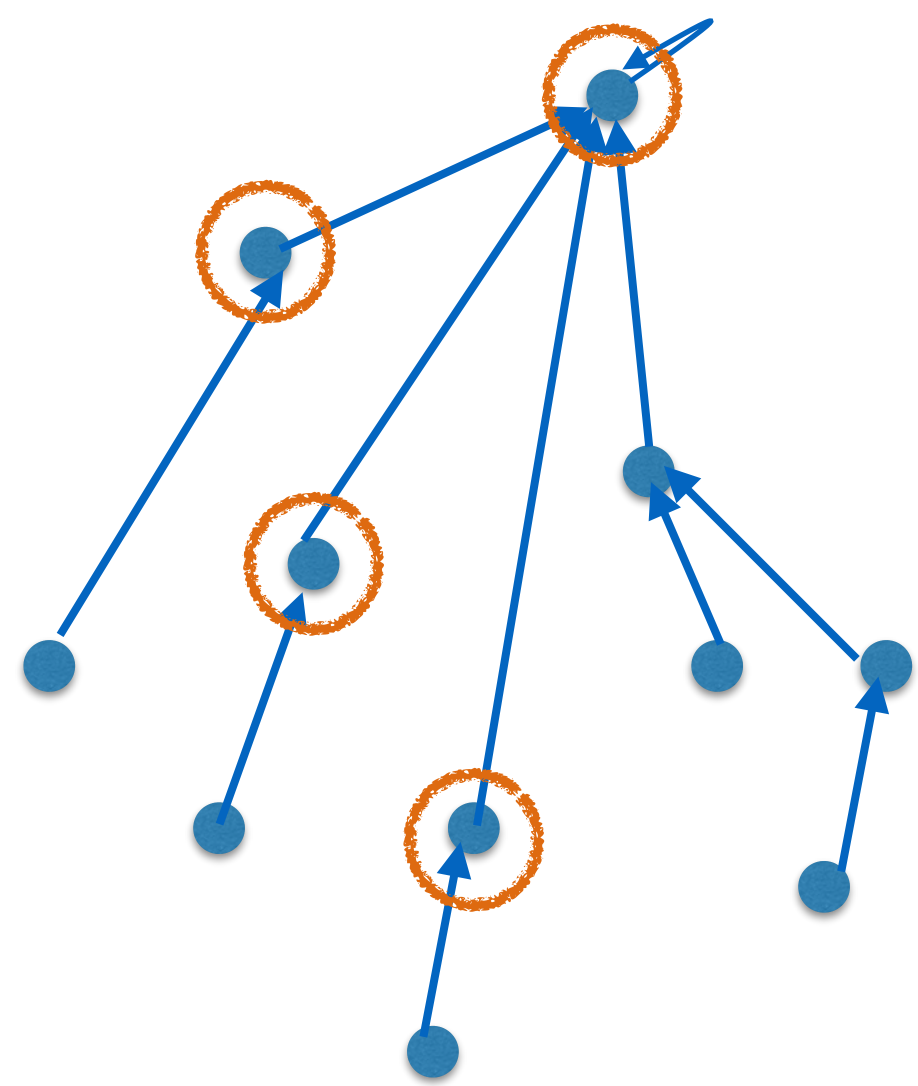
Let’s make things work a little faster in practice

- 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**

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

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

Surprising Result: Tarjan '75

- Improved bound on amortized complexity of union-find with path compression
- Time complexity for n union and find operations on n elements is $O(n\alpha(n))$, where
  - $\alpha(n)$ is extremely slow-growing, **inverse-Ackermann function**
  - Essentially a constant
- Grows much **muuchch morrree** slowly than $\log^*$
- $\alpha(n) \leq 4$ for all values in practice
- **Result.** Union and Find become (essentially) amortized constant time in practice (just short of $O(1)$ in theory) !

Digging Deeper

Inverse Ackermann

- **Inverse Ackerman:** The function $\alpha(n)$ grows much more slowly than $\log^{*^c} n$ for **any fixed c**

- With $\log^*$, you count how many times does applying $\log$ over and over gets the result to become small
- With the inverse Ackermann, essentially you count how many times does applying $\log^*$ (not log!) over and over gets the result to become small

- $\alpha(n) = \min\{k \mid \log^{\overbrace{****\cdots*}^{k}}(n) \leq 2\}$

- $\alpha(n) = 4$ for $n = 2^{2^{2^{2^{16}}}}$

Can we do better?

- OK, so that's "basically constant". Can we get constant?
- No. *Any data structure* for union find requires $\Omega(\alpha(n))$ amortized time (Fredman, Saks '89)
- So up trees with path compression are optimal(!)

Many Applications of Union-Find

- Good for applications in need of clustering
    - cities connected by roads
    - cities belonging to the same country
    - connected components of a graph
- Maintaining equivalence classes
- Maze creation!

Back to MST

- Prim's algorithm is $O(m + n \log n)$ if using a Fibonnacci tree
- Kruskal's algorithm is $O(m \log m)$
- Which is better in practice?
- Is sorting time required?

MST Algorithms History

- **Borůvka's Algorithm** (1926)
    - The Borvka / Choquet / Florek-ukaziewicz-Perkal-Steinhaus-Zubrzycki / Prim / Sollin / Brosh algorithm
    - Oldest, most-ignored MST algorithm, but actually very good
- **Jarník's Algorithm** ("Prims Algorithm", 1929)
    - Published by Jarník, independently discovered by Kruskal in 1956, by Prims in 1957
- **Kruskal's Algorithm** (1956)
    - Kruskal designed this because he found Borůvka's algorithm "unnecessarily complicated"

Can we do better?

Best known algorithm by Chazelle (1999)

A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity*

BERNARD CHAZELLE†

NECI Research Tech Report 99-099 (July 1999)
Journal of the ACM, 47(6), 2000, pp. 1028–1047.

Abstract

A deterministic algorithm for computing a minimum spanning tree of a connected graph is presented. Its running time is $O(m\alpha(m,n))$, where α is the classical functional inverse of Ackermann's function and n (resp. m) is the number of vertices (resp. edges). The algorithm is comparison-based: it uses pointers, not arrays, and it makes no numeric assumptions on the edge costs.

1 Introduction

The history of the minimum spanning tree (MST) problem is long and rich, going as far back as Borůvka's work in 1926 [1, 9, 13]. In fact, MST is perhaps the oldest open problem in computer science. According to Nešetřil [13], "this is a cornerstone problem of combinatorial optimization and in a sense its cradle." Textbook algorithms run in $O(m \log n)$ time, where n

Can we do better?

Using randomness, can get $O(n+m)$ time!

A Randomized Linear-Time Algorithm to Find Minimum Spanning Trees

DAVID R. KARGER

Stanford University, Stanford, California

PHILIP N. KLEIN

Brown University, Providence, Rhode Island

AND

ROBERT E. TARJAN

Princeton University and NEC Research Institute, Princeton, New Jersey

Abstract. We present a randomized linear-time algorithm to find a minimum spanning tree in a connected graph with edge weights. The algorithm uses random sampling in combination with a recently discovered linear-time algorithm for verifying a minimum spanning tree. Our computational model is a unit-cost random-access machine with the restriction that the only operations allowed on edge weights are binary comparisons.

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; G.2.2 [**Discrete**

Optimal MST Algorithm?

Has been discovered but don't know its running time!

An Optimal Minimum Spanning Tree Algorithm

SETH PETTIE AND VIJAYA RAMACHANDRAN

The University of Texas at Austin, Austin, Texas

Abstract. We establish that the algorithmic complexity of the minimum spanning tree problem is equal to its decision-tree complexity. Specifically, we present a deterministic algorithm to find a minimum spanning tree of a graph with n vertices and m edges that runs in time $O(\mathcal{T}^*(m, n))$ where $\mathcal{T}^*$ is the minimum number of edge-weight comparisons needed to determine the solution. The algorithm is quite simple and can be implemented on a pointer machine.

Although our time bound is optimal, the exact function describing it is not known at present. The current best bounds known for $\mathcal{T}^*$ are $\mathcal{T}^*(m, n) = \Omega(m)$ and $\mathcal{T}^*(m, n) = O(m \cdot \alpha(m, n))$, where α is a certain natural inverse of Ackermann's function.

Even under the assumption that $\mathcal{T}^*$ is superlinear, we show that if the input graph is selected from $G_{n,m}$, our algorithm runs in linear time with high probability, regardless of n, m, or the permutation of edge weights. The analysis uses a new martingale for $G_{n,m}$ similar to the edge-exposure martingale for G

Acknowledgments

- 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)