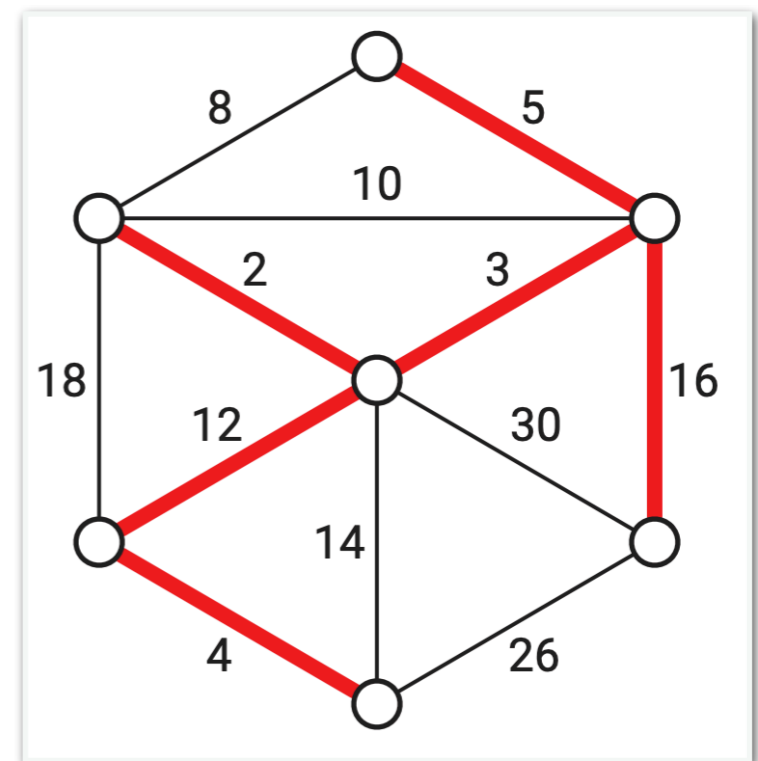# Greedy Graph Algorithms: Minimum Spanning Trees
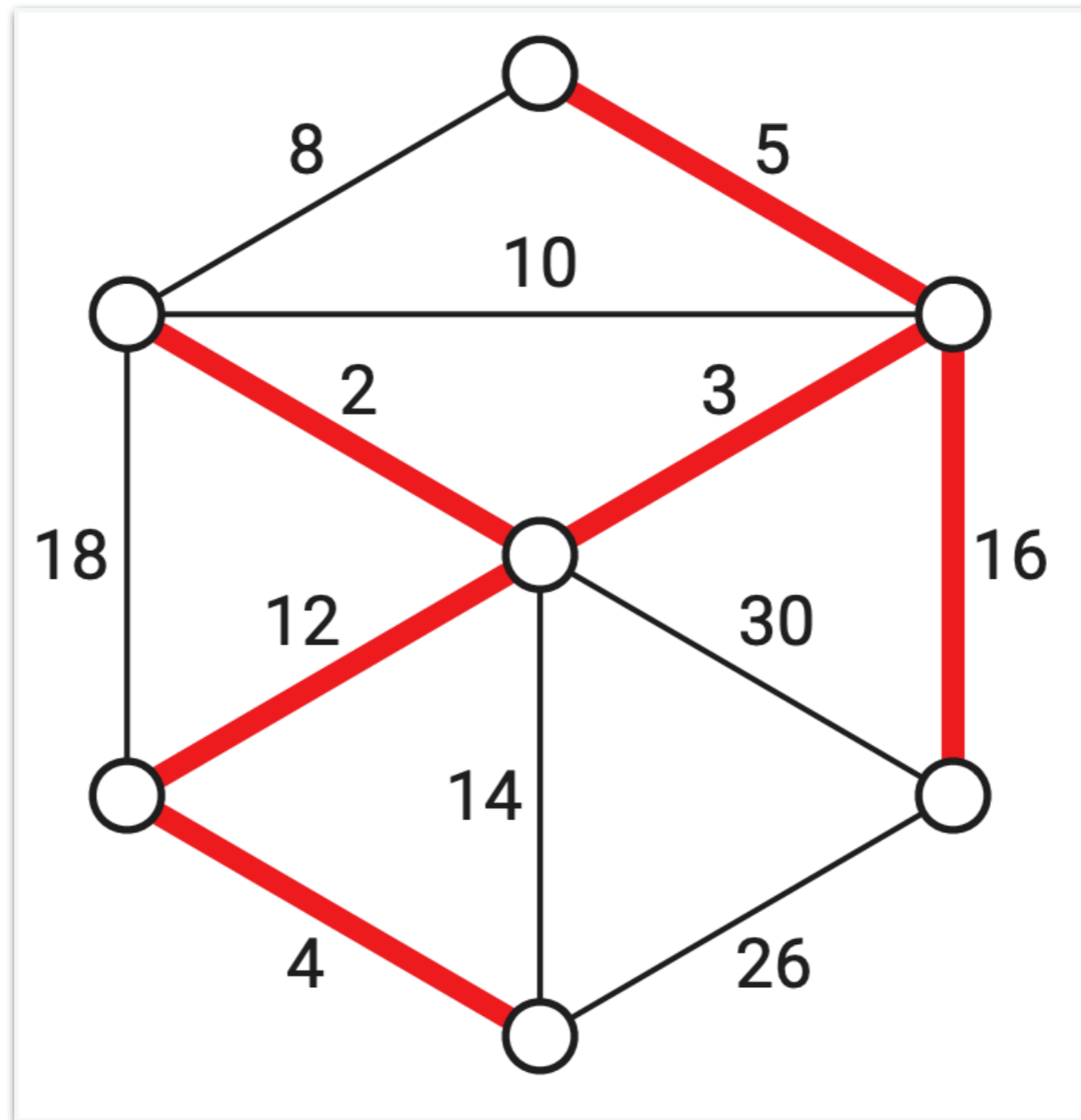
# Minimum Spanning Trees

**Problem.** Given a connected, undirected graph $G = (V, E)$ with edge costs $w_e$, output **a minimum spanning tree**, i.e., set of edges $T \subseteq E$ such that

- (a spanning tree of $G$): $T$ connects all vertices

- (has minimum weight): for any other spanning tree $T'$ of $G$, we have $\displaystyle\sum_{e \in T} w_e \leq \sum_{e \in T'} w_e$

# Minimum Cost Spanning Trees

# Cut Property: MST

**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 $V - S$, then 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 $T$ ( $\Rightarrow\!\!\Leftarrow$ )

How to find such an edge $e'$?

# Jarník's ("Prims Algorithm")

- Initialize $S = \{u\}$ for any vertex $u \in V$ and $T = \varnothing$

- 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 to implement?

  - Priority queue

  - Extract minimum and insert each in $O(\log m)$ if priority queue has at most $m$ elements

# "Prims" Implementation

- Implementation using **Binary heaps**

  - Total runtime: $O(m \log n)$ (recall that $\log m = O(\log n)$)

- If a **Fibonacci heap** is used instead of binary heap:

  - Runs in $O(m + n \log n)$

  - Support amortized $O(1)$-time inserts, $O(\log n)$ time extract min using **"amortized time"**

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

# Cycle Property

**Lemma (Cycle Property):** For any cycle C in the graph, if the weight of an edge e of C is larger than any of the individual weights of all other edges of C, then this edge cannot belong to an MST.
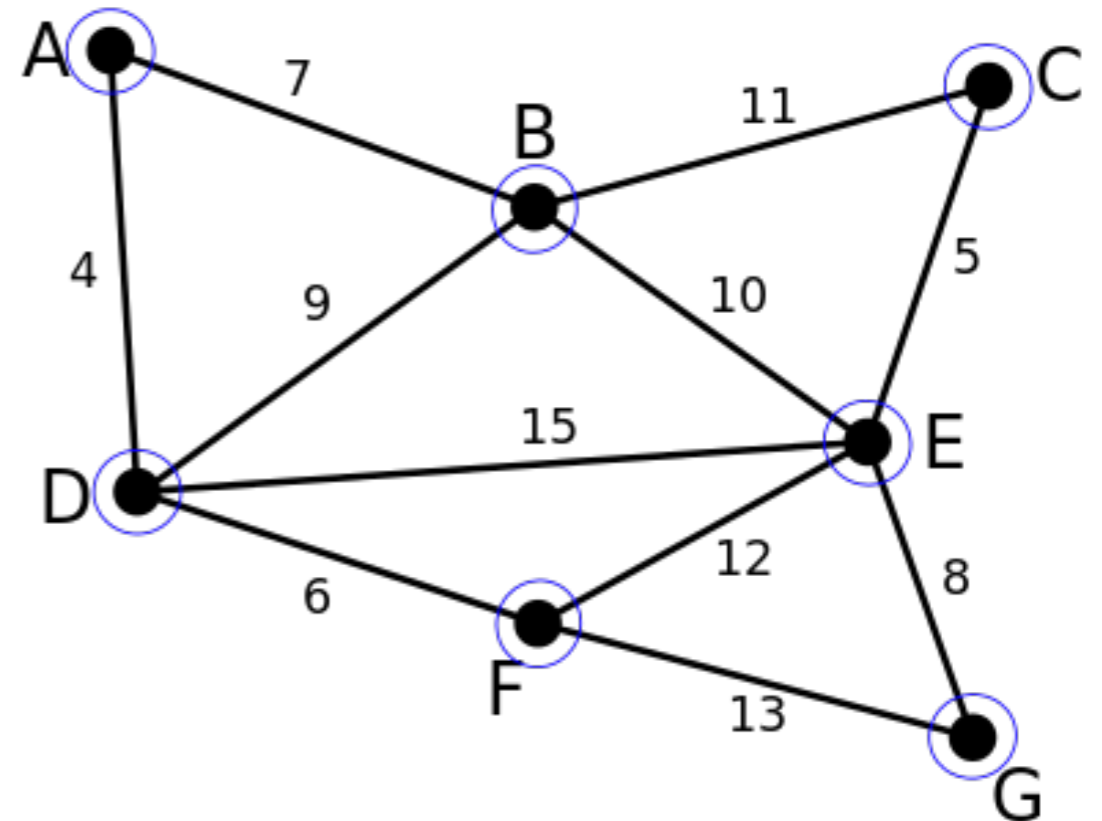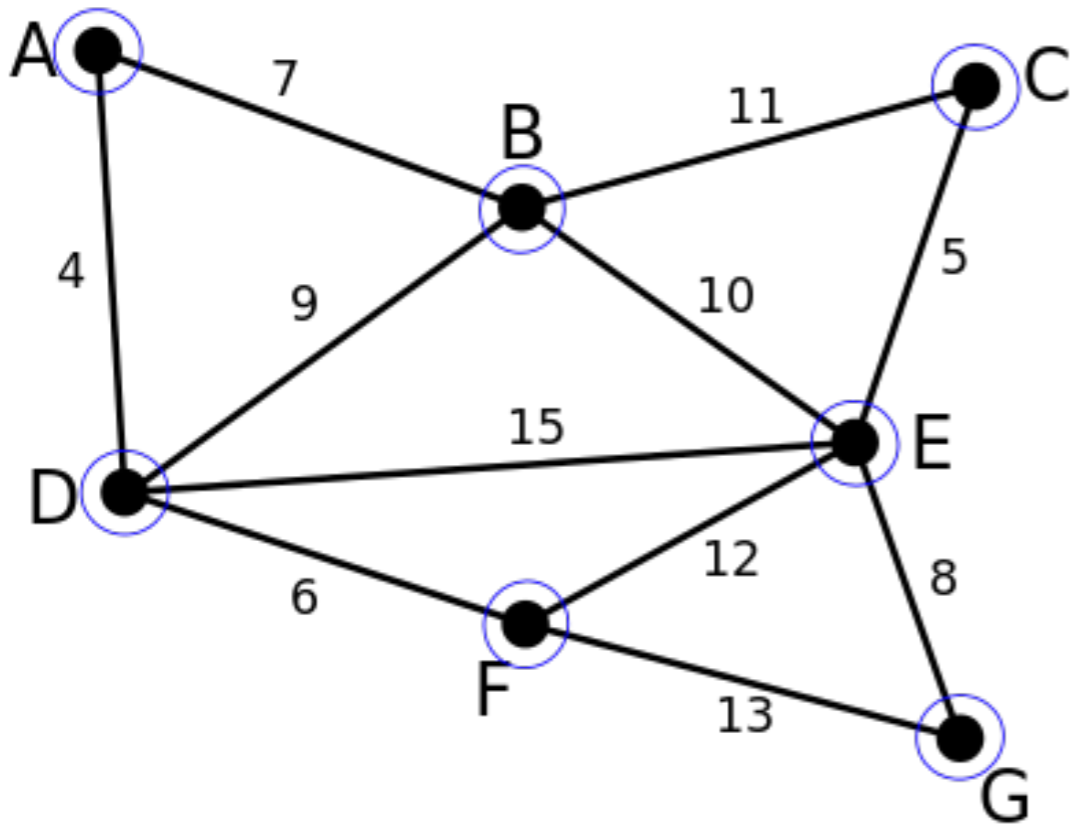
Proof on board

# Kruskal's Algorithm

# 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

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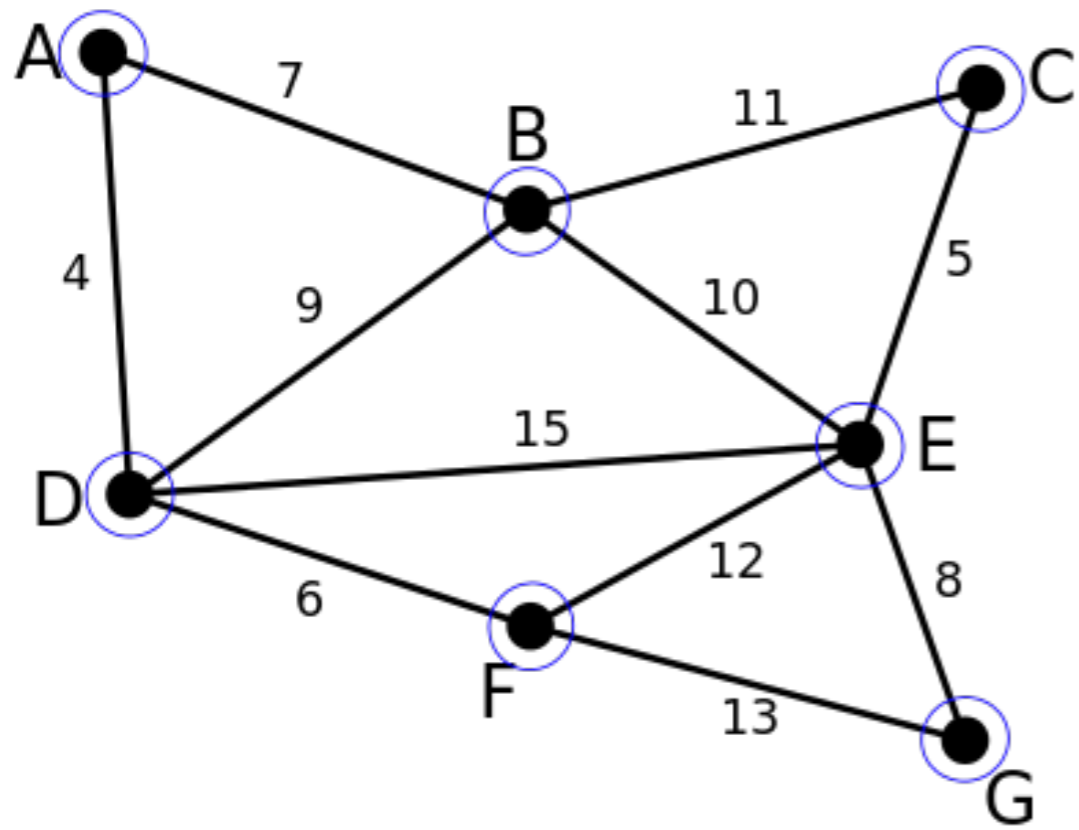
- $H \leftarrow H - \{e\}$

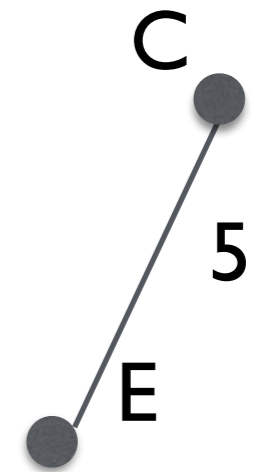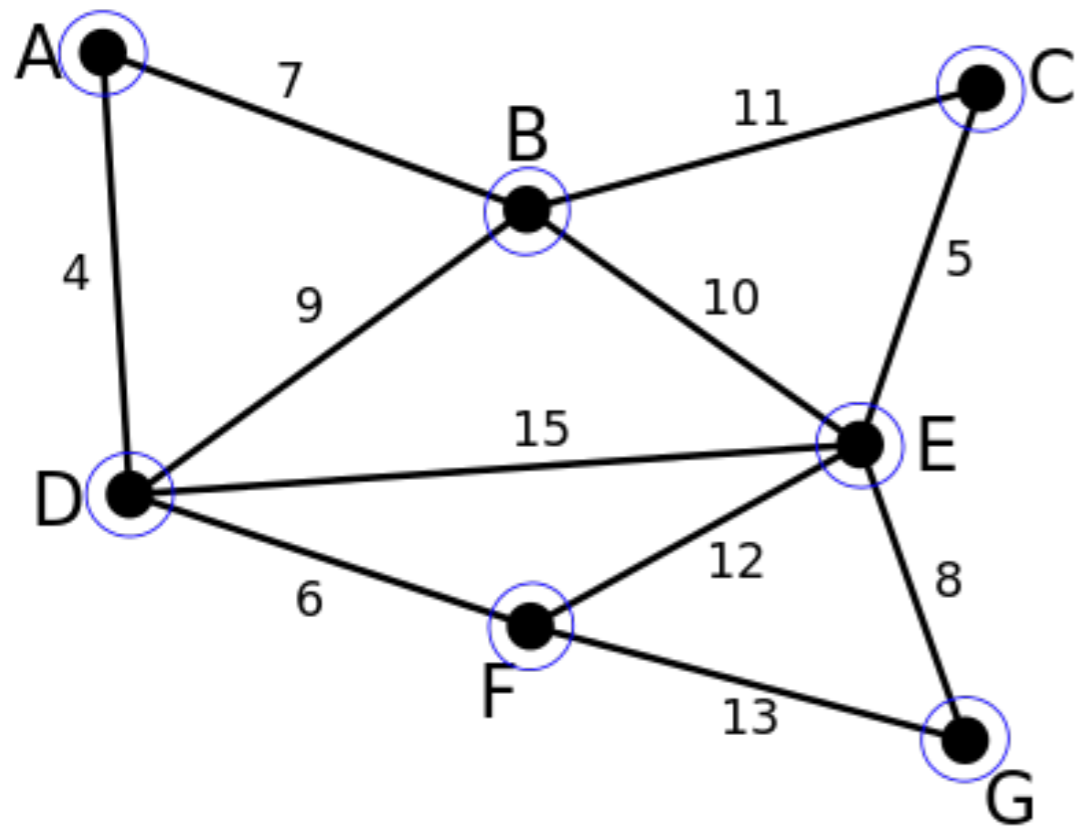**Idea:** Add the cheapest remaining edge **that does not create a cycle**.

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

A

4

D

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

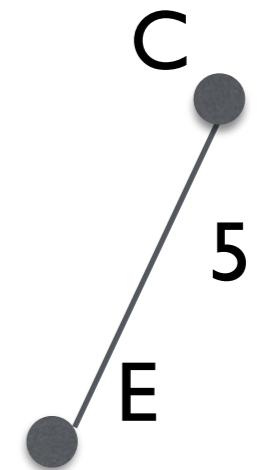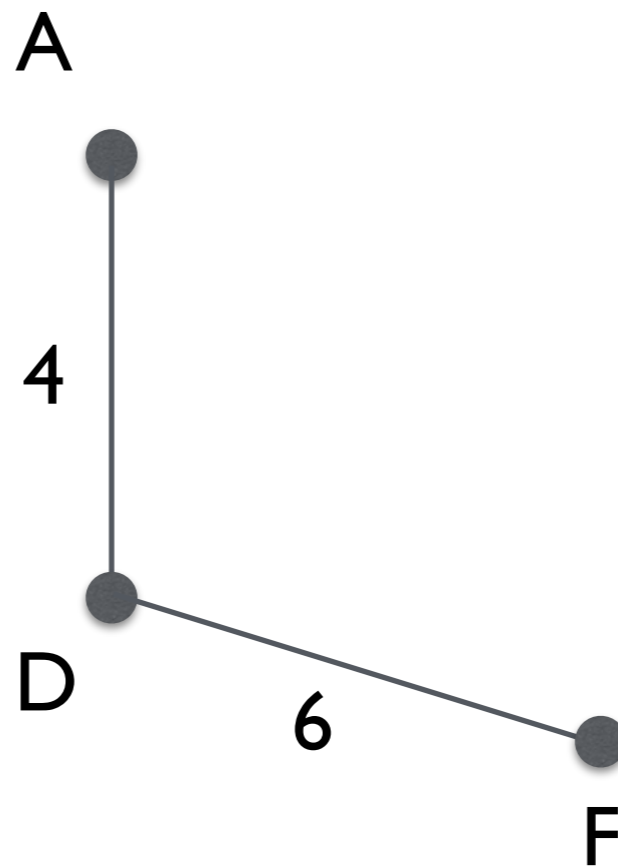**Idea:** Add the cheapest remaining edge **that does not create a cycle**.

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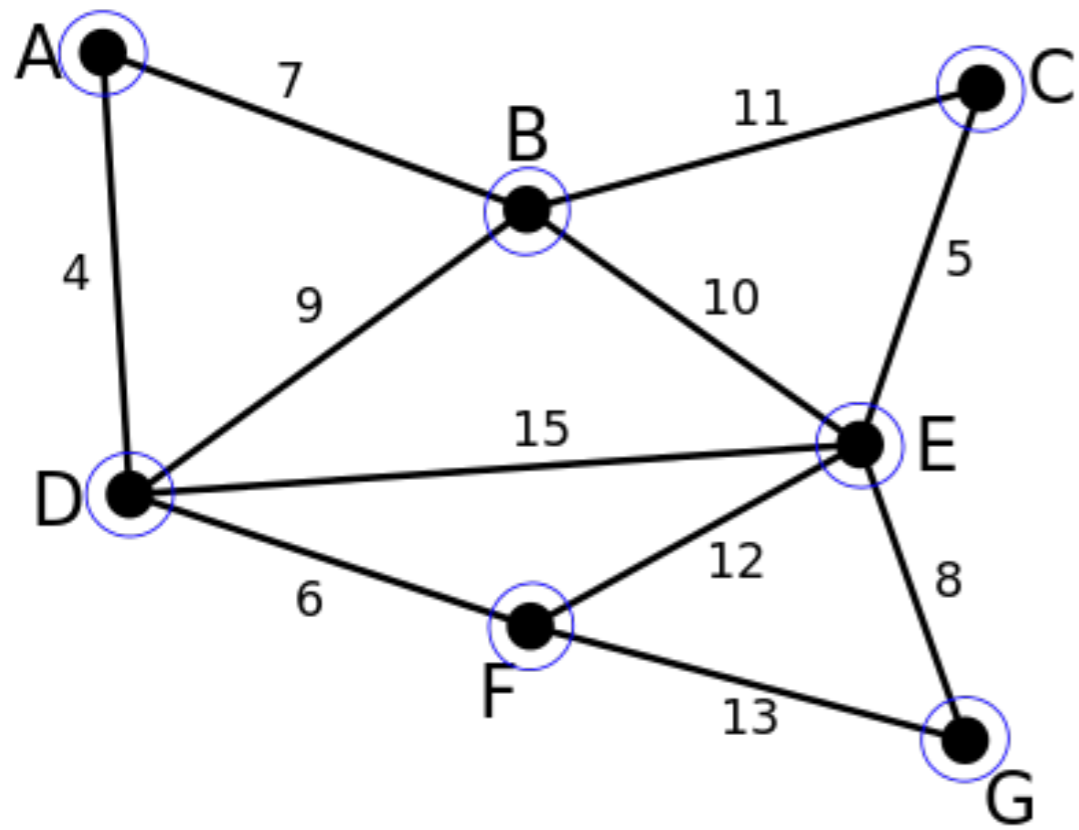        - $T \leftarrow T \cup \{e\}$
    - $H \leftarrow H - \{e\}$

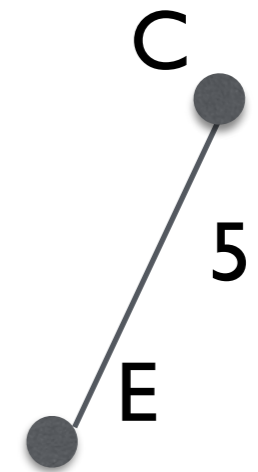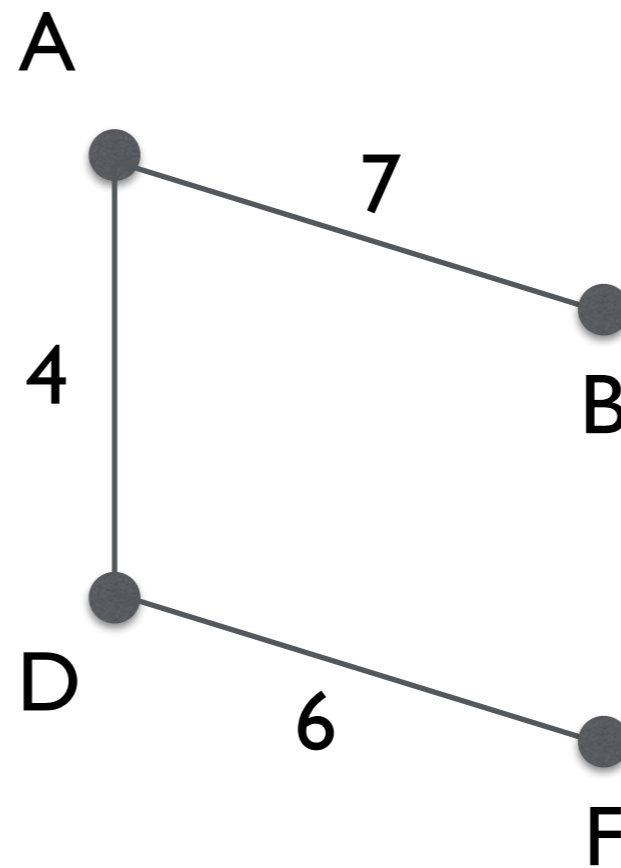**Idea:** Add the cheapest remaining edge **that does not create a cycle**.

- 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
        - $T \leftarrow T \cup \{e\}$
    - $H \leftarrow H - \{e\}$
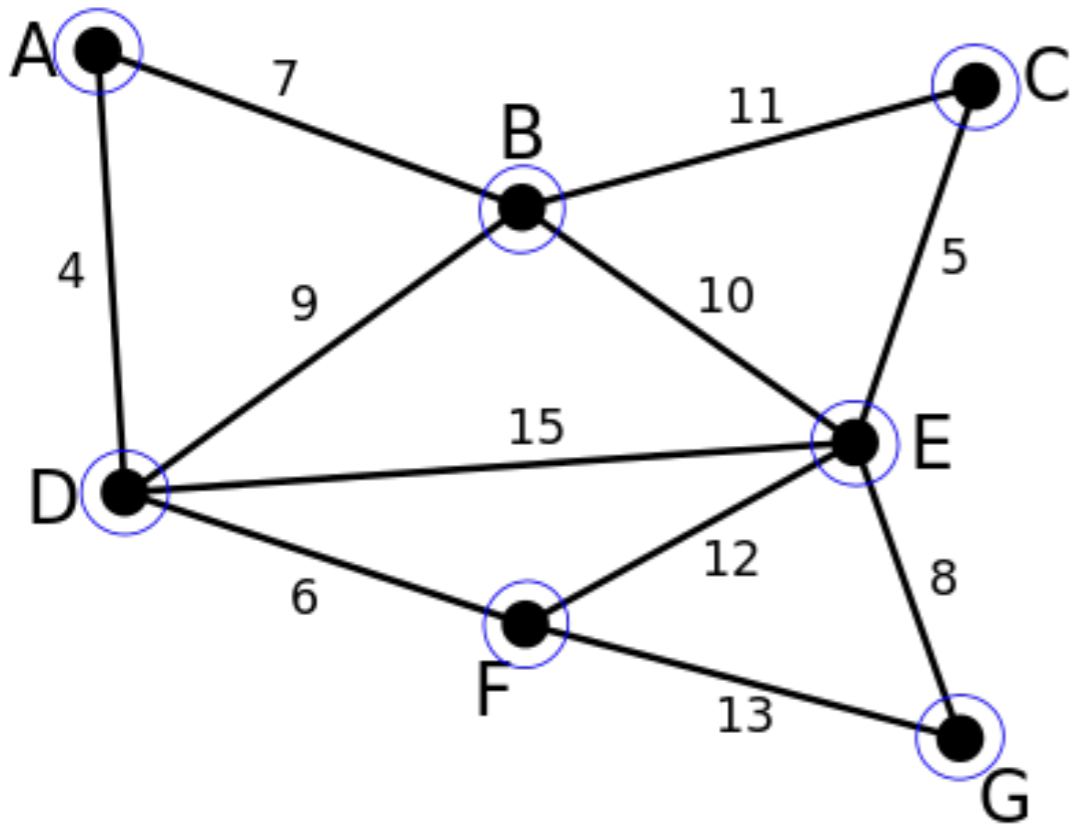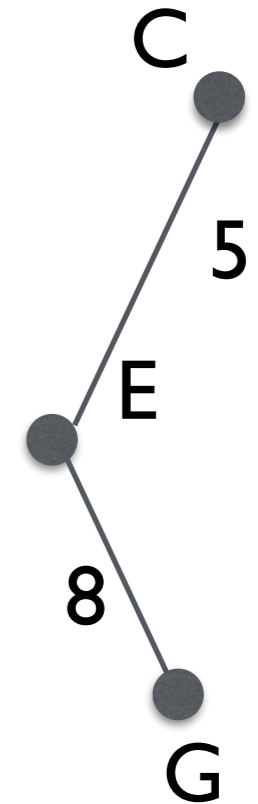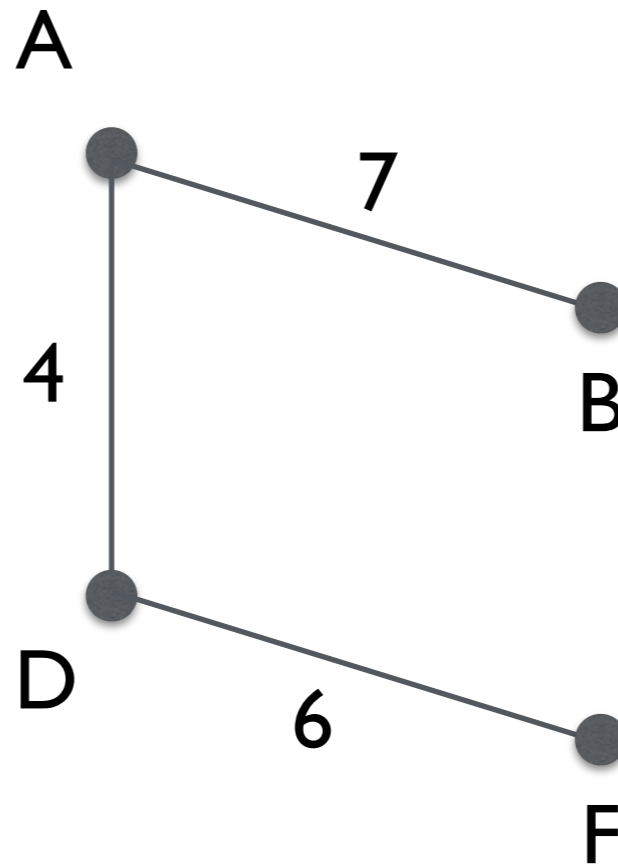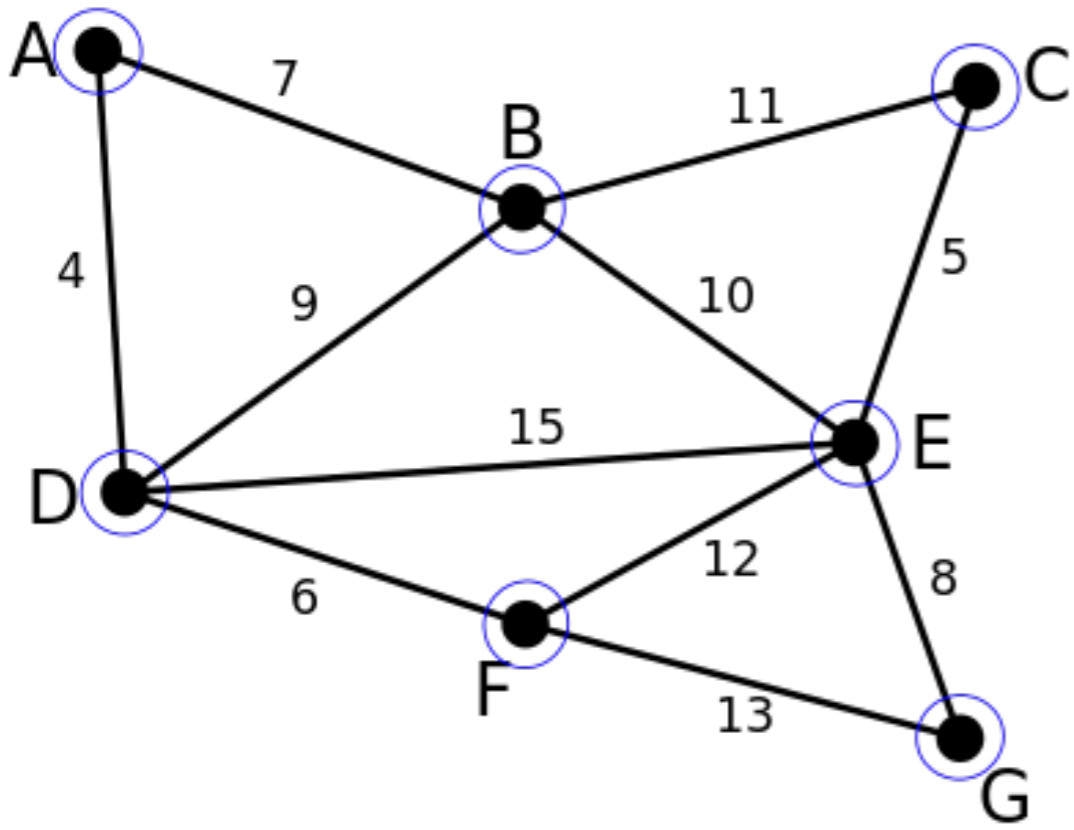
Total weight: 40

**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
    - $T \leftarrow T \cup \{e\}$
  - $H \leftarrow H - \{e\}$
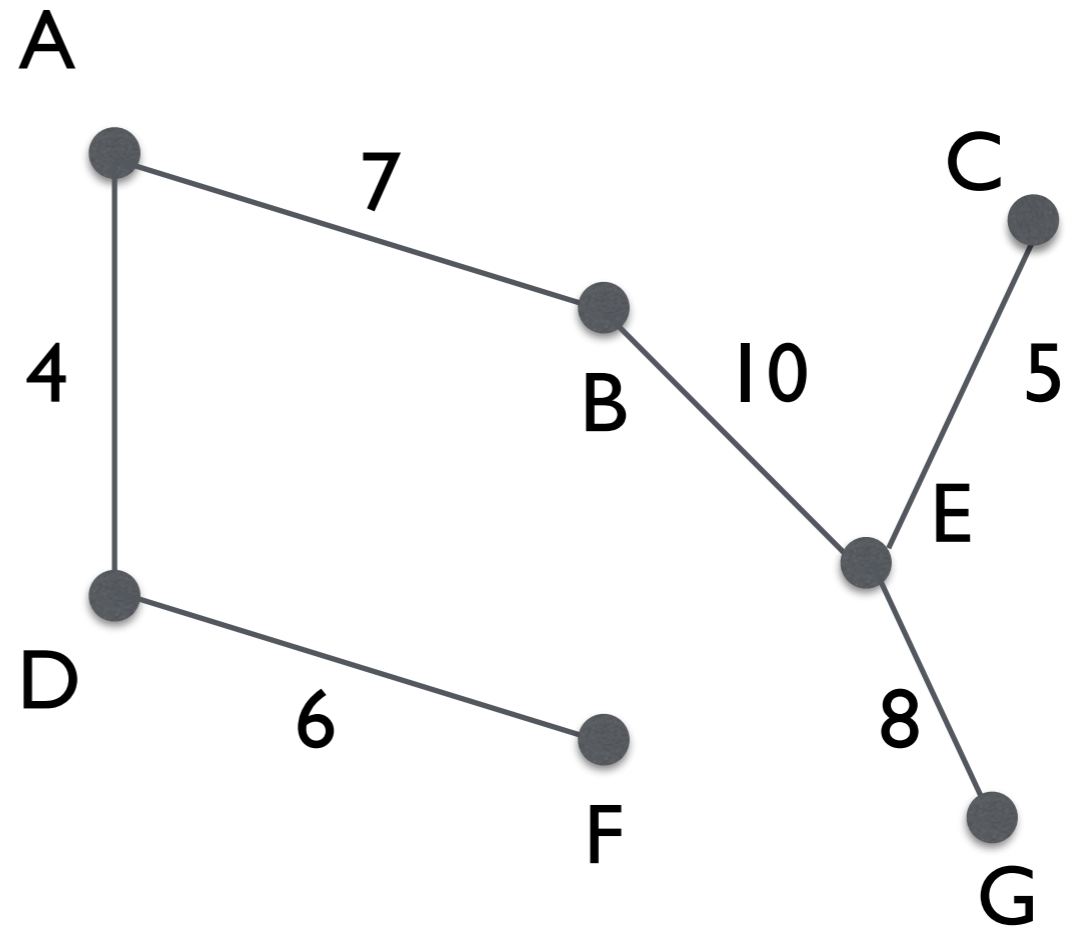
# Kruskal's Analysis

- **Correctness**: Does it give us the correct MST?

- Why is each edge $(v, w)$ that we are adding safe?

- Consider the step just before $(v, w)$ is added

  - Let $S = \{x \in V \mid v$ has a path from v 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

- Sort edges by weight: $O(m \log m)$

    - Turns out this is the dominant cost

- Determine whether $T \cup \{e\}$ contains a cycle

    - Let's talk more about how to do this efficiently

- 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
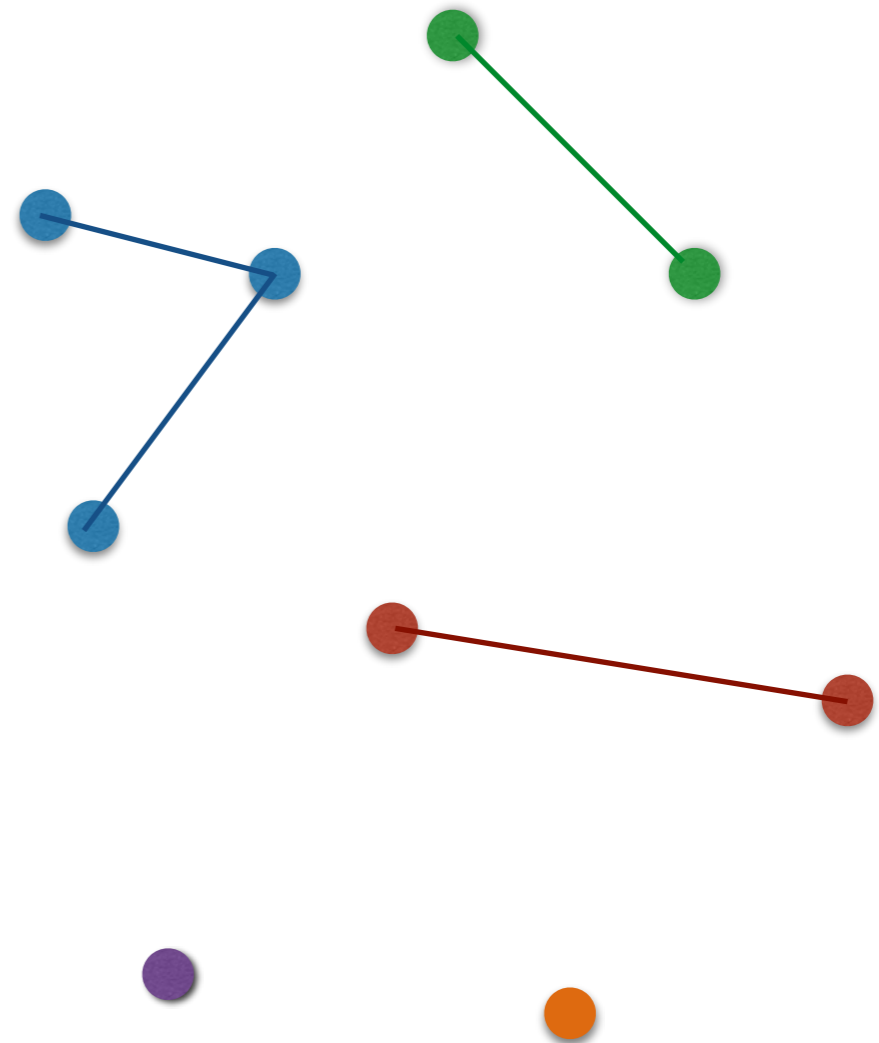
# Does this edge create a cycle?

- 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

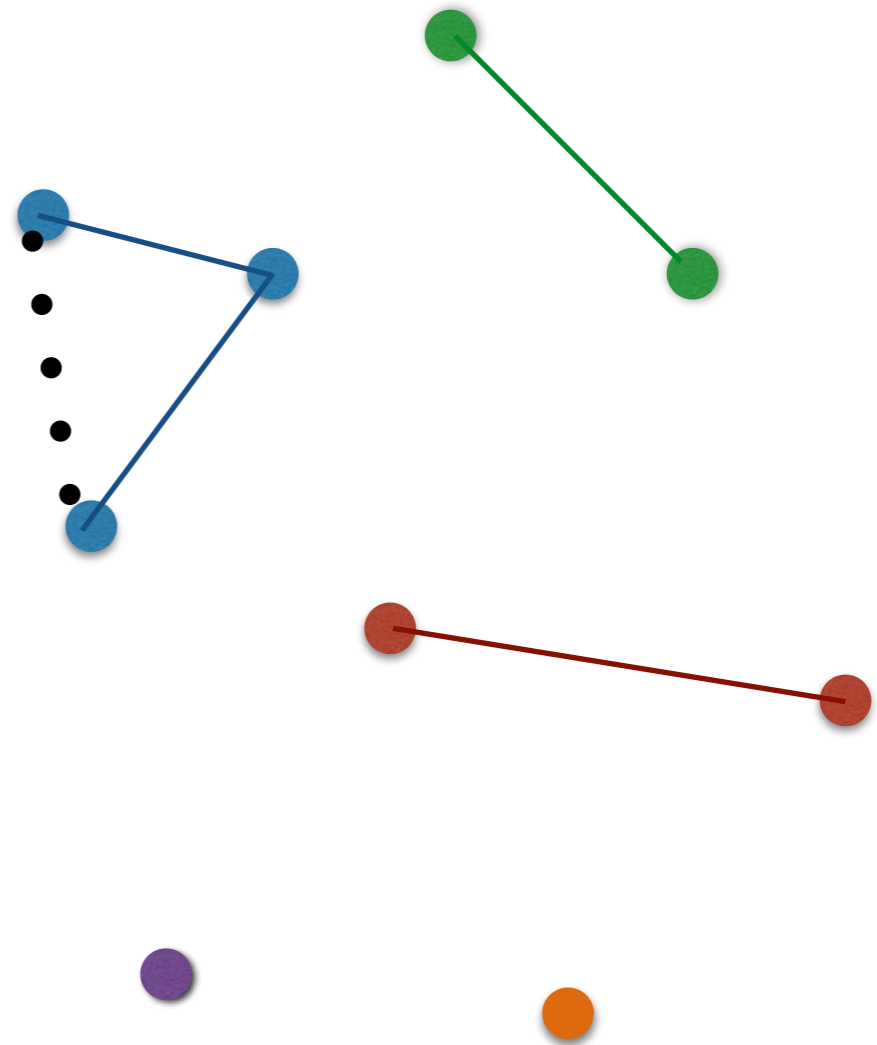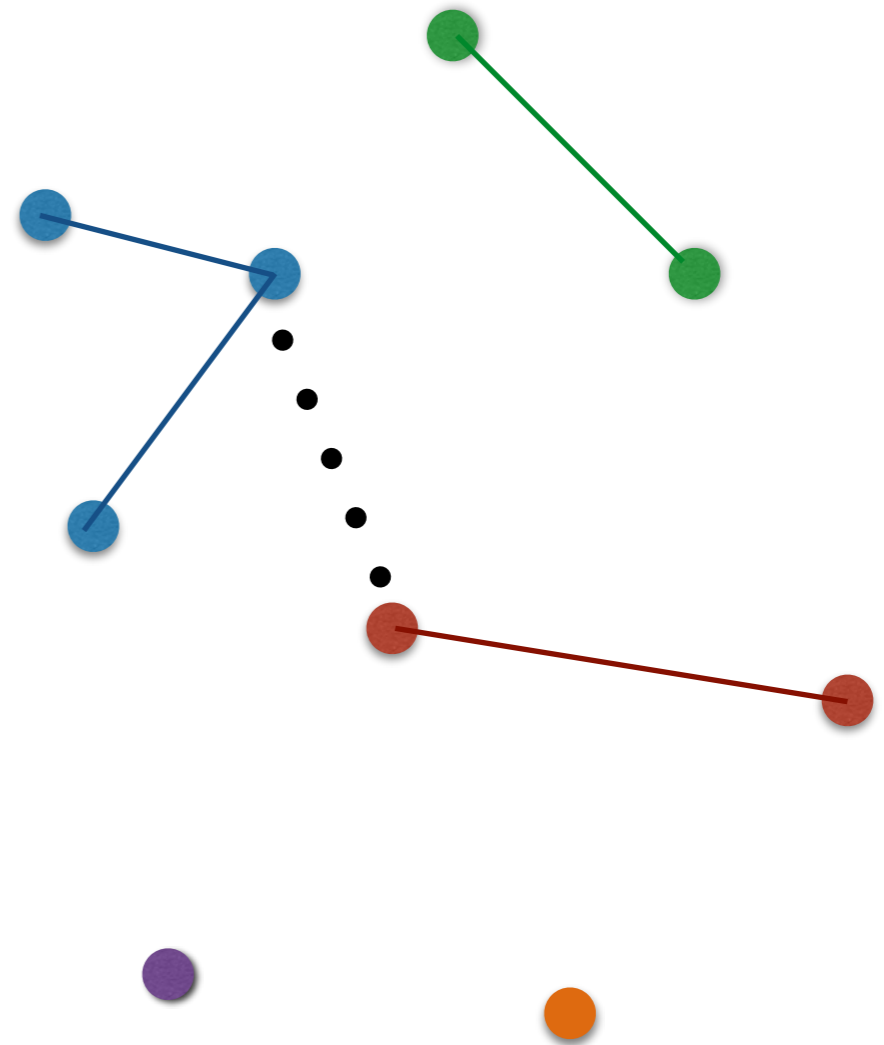- How can we update when adding an edge?

# 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?
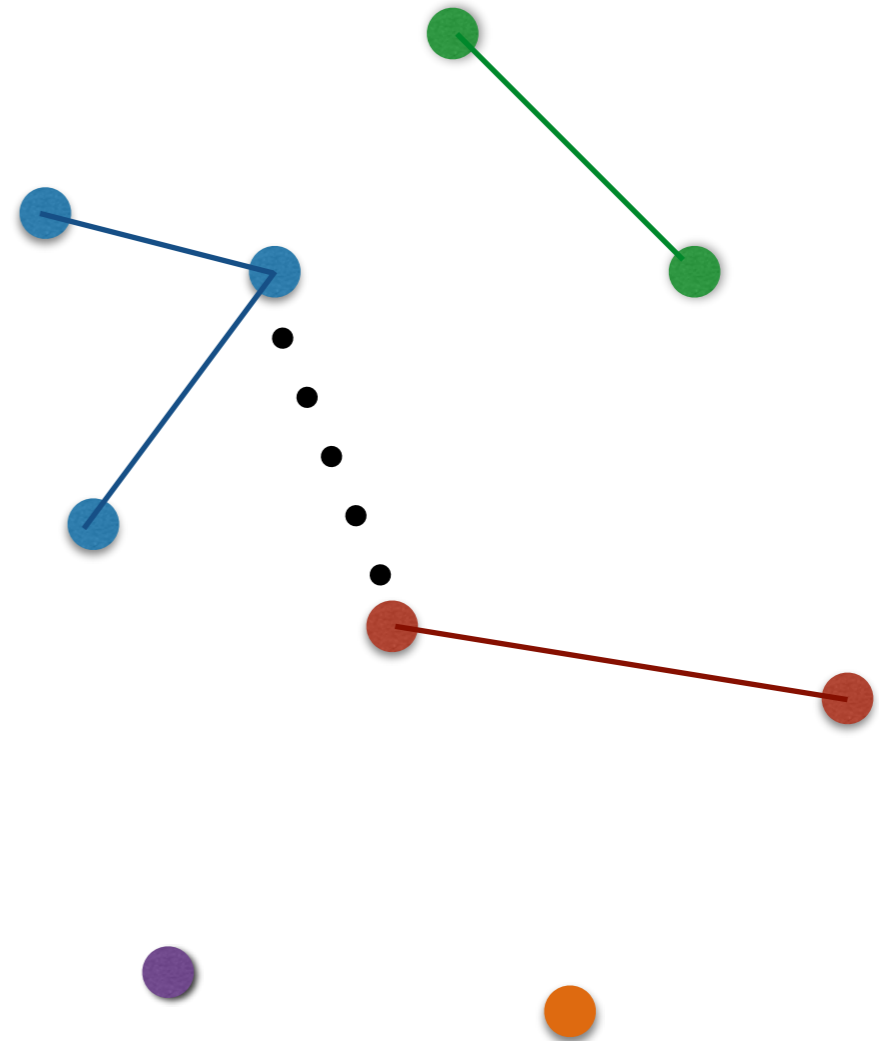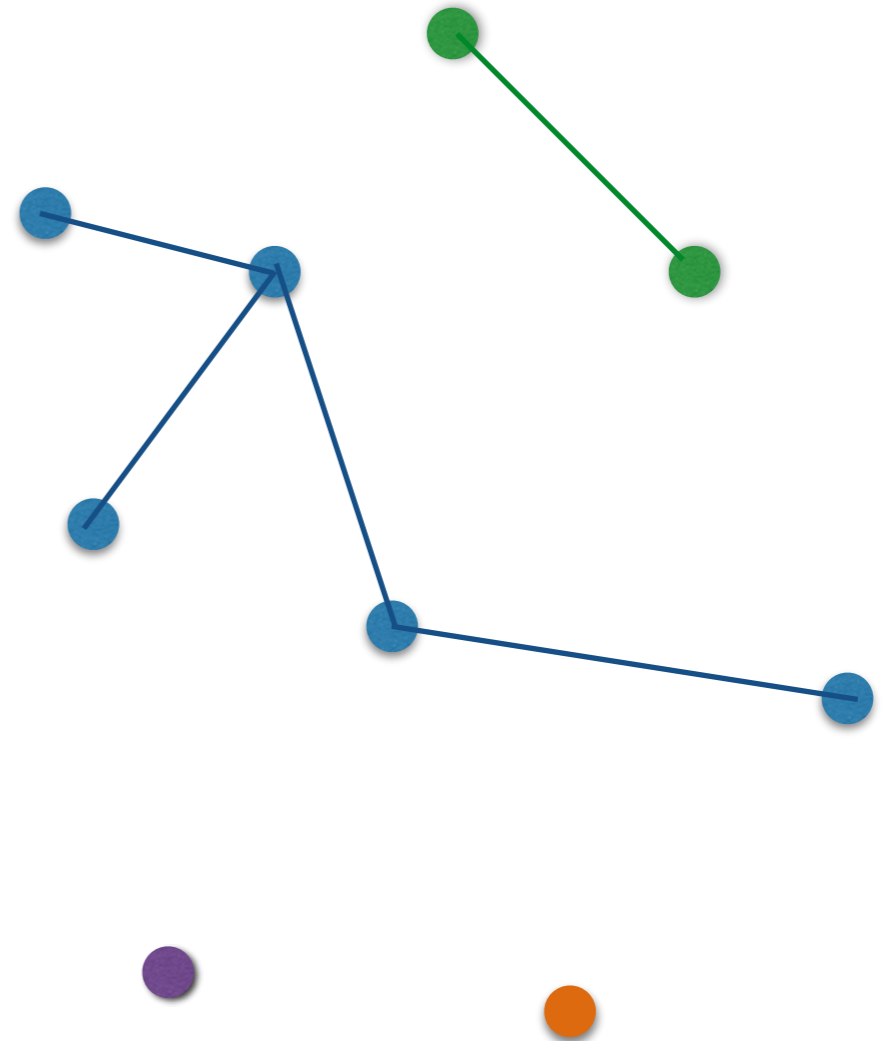
# What do we want to be able to do?

- 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

- Today we'll create a data structure for this task

# 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: First Attempt

Let $S = \{1, 2, \ldots, n\}$ be the set.

Idea: Each element stores the label of its set

- `Initialize()`: 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)

**Digging Deeper**

# Union-Find: Improving Union

- Let's perturb that idea just a little bit and analyze it a bit more carefully

- 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
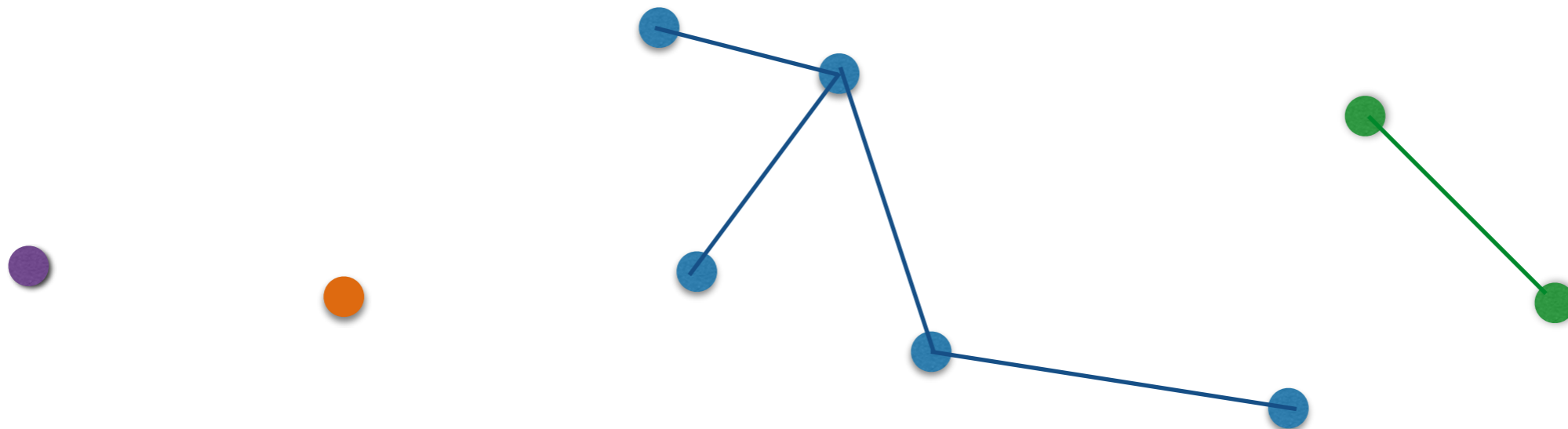
# Union-Find: Improving Union

- Let's perturb that idea just a little bit and analyze it a bit more carefully

- 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
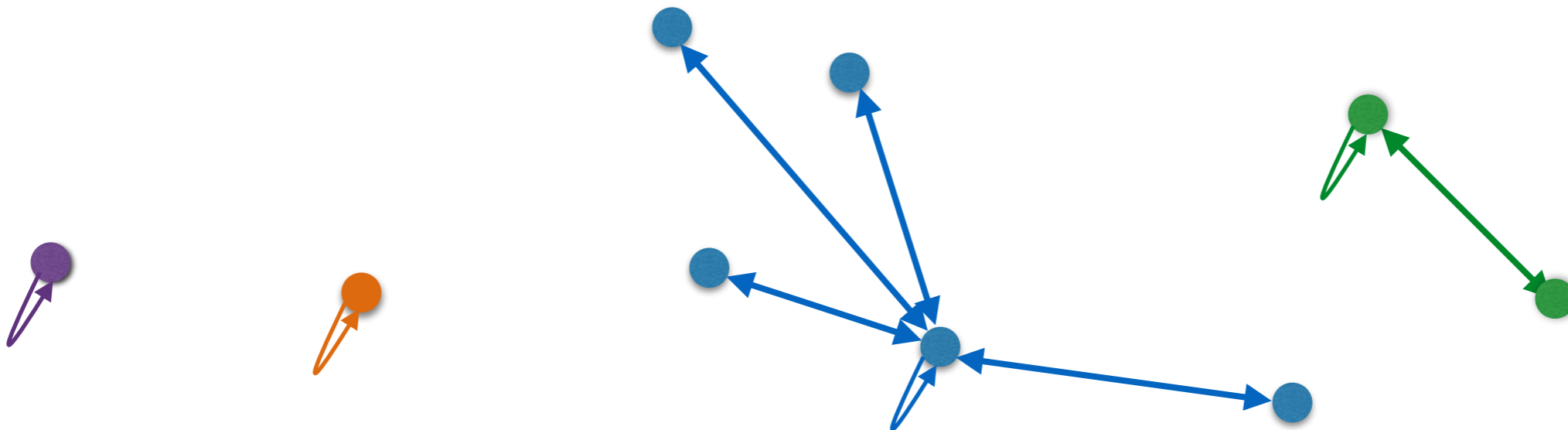
# Union-Find: Improving Union

- Let's 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

- 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!

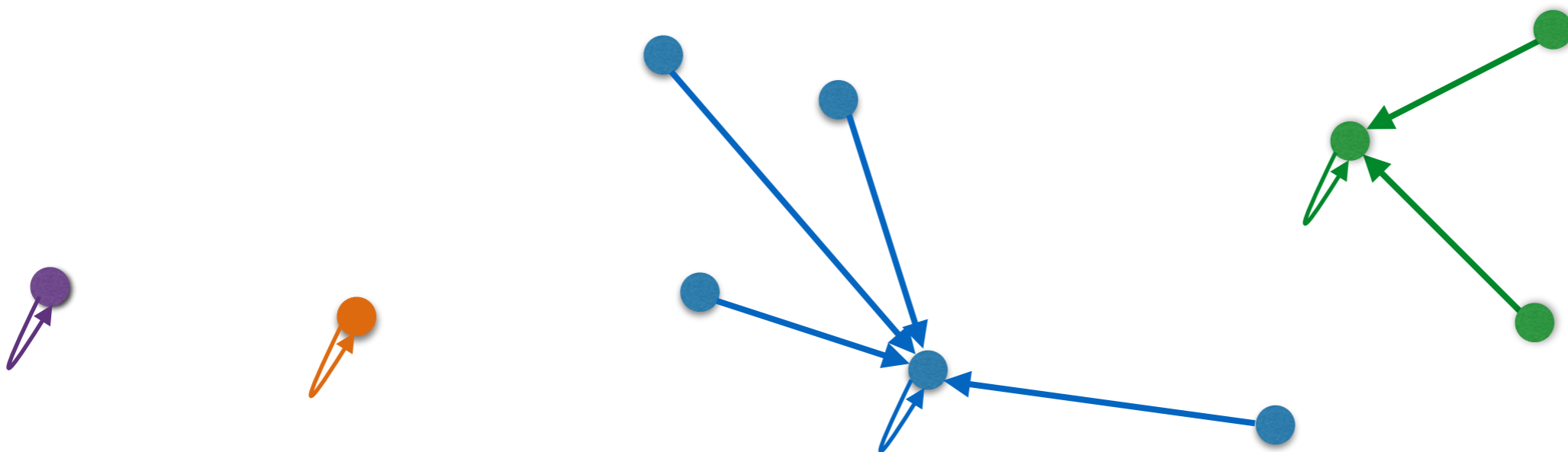- Follow back pointers from the head of the tree so we get every node
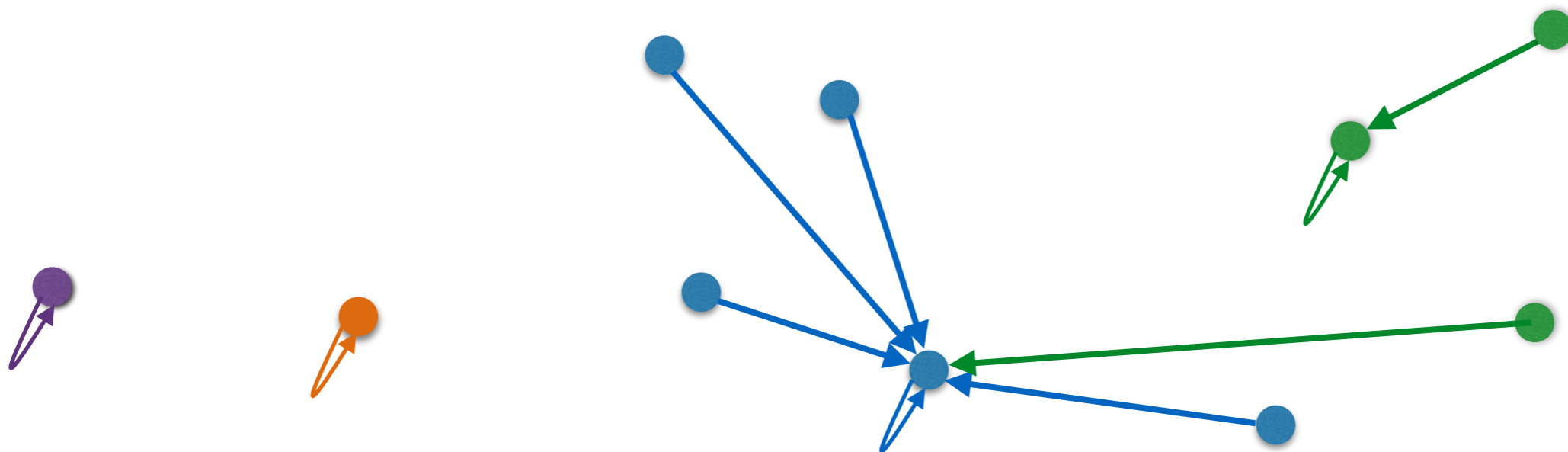
# Union-Find: Improving Union

- 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: Amortized 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: Amortized Analysis

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

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

- ("Amortized time" will not be on the midterm-but we'll see it again soon)

**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

- Now, let's have our pointers act like a tree, but pointing up ("up tree")
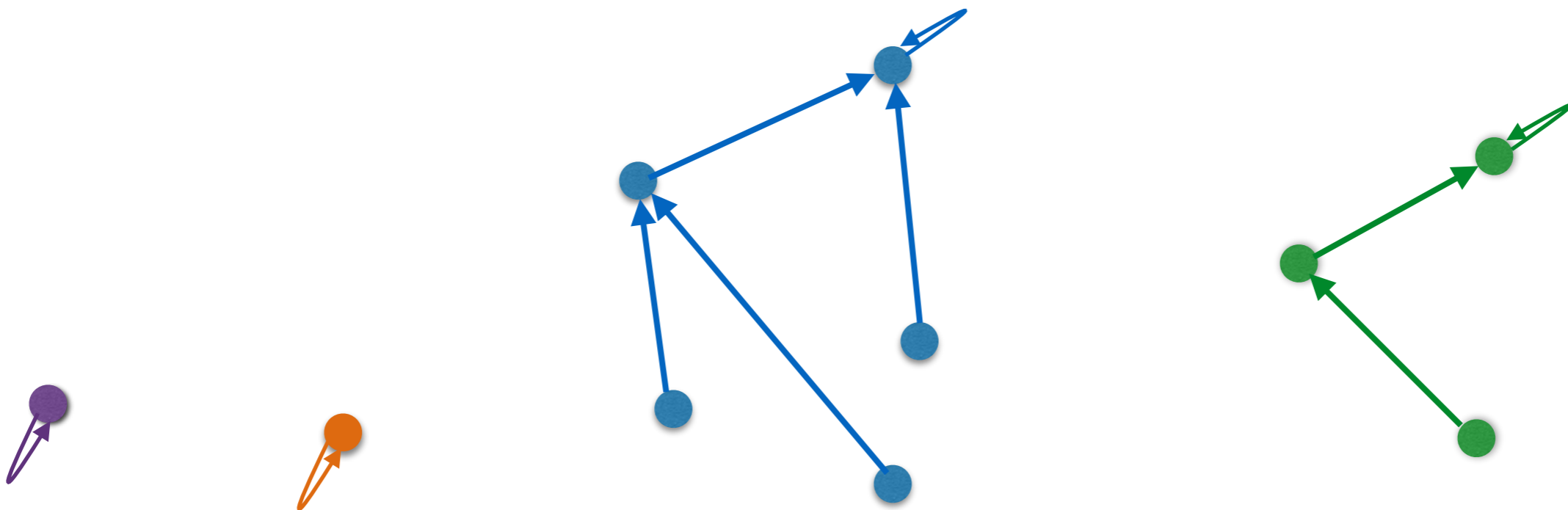
# Fast Union with "Trees"

- 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?
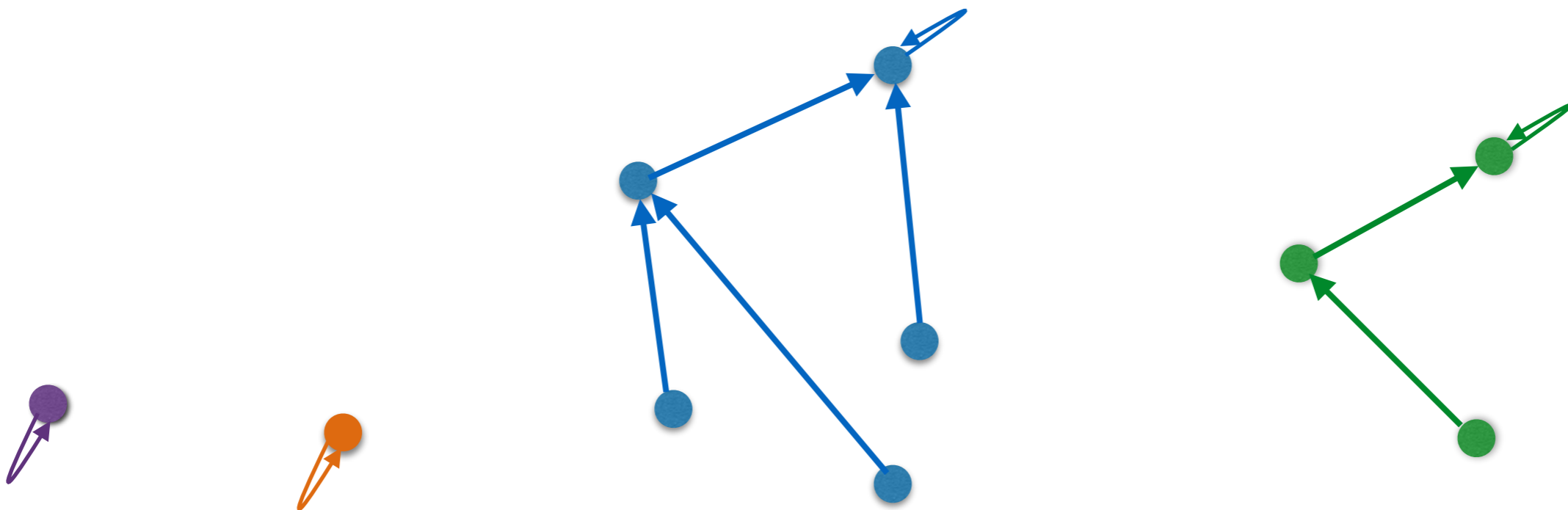
# Fast Union with "Trees"

- Let's keep a head node as before

- Now, let's have our pointers act like a tree, but pointing up
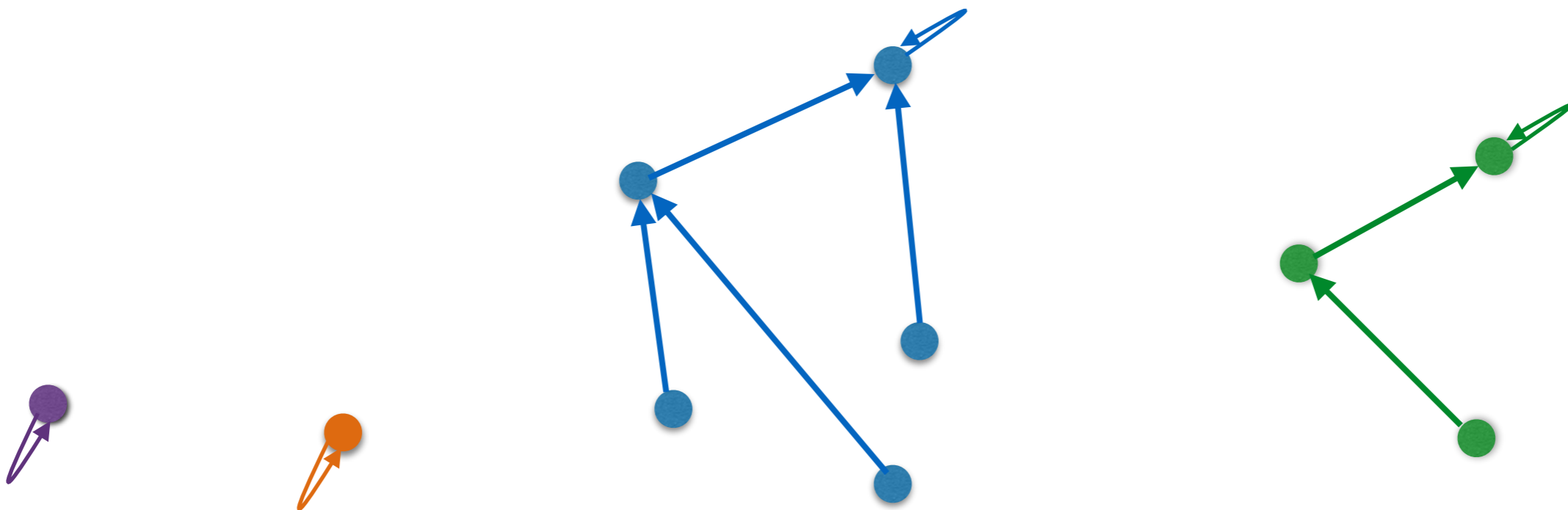
- How can we Union?

# 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
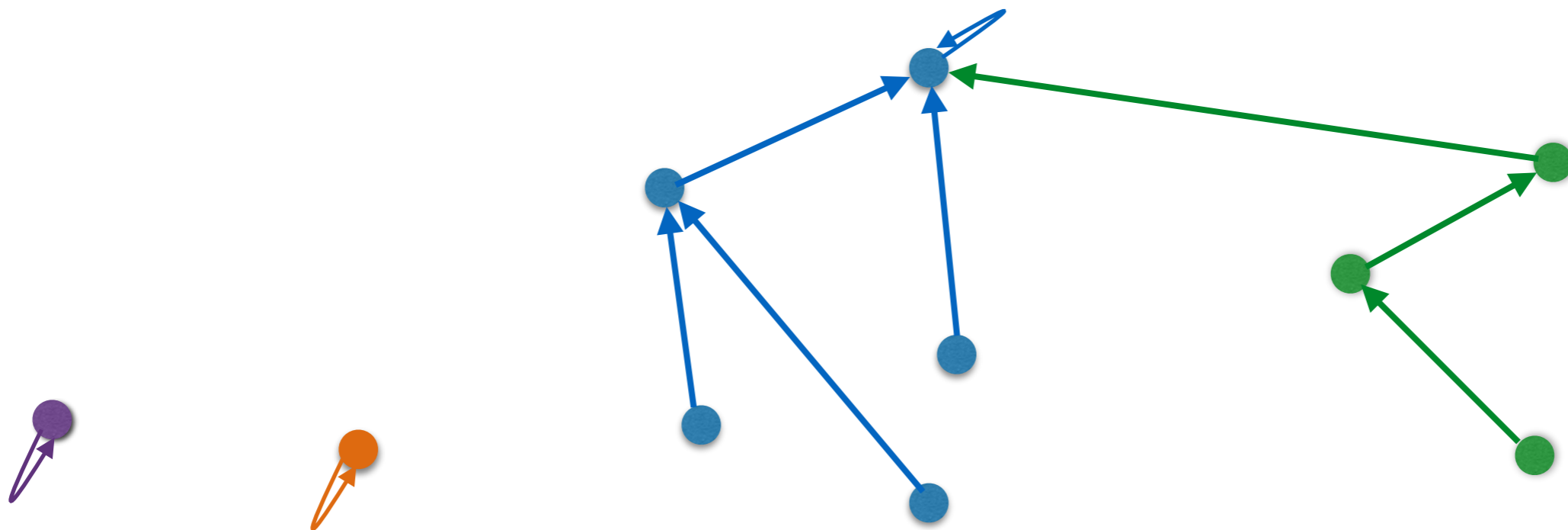
- How can we Union?

# 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
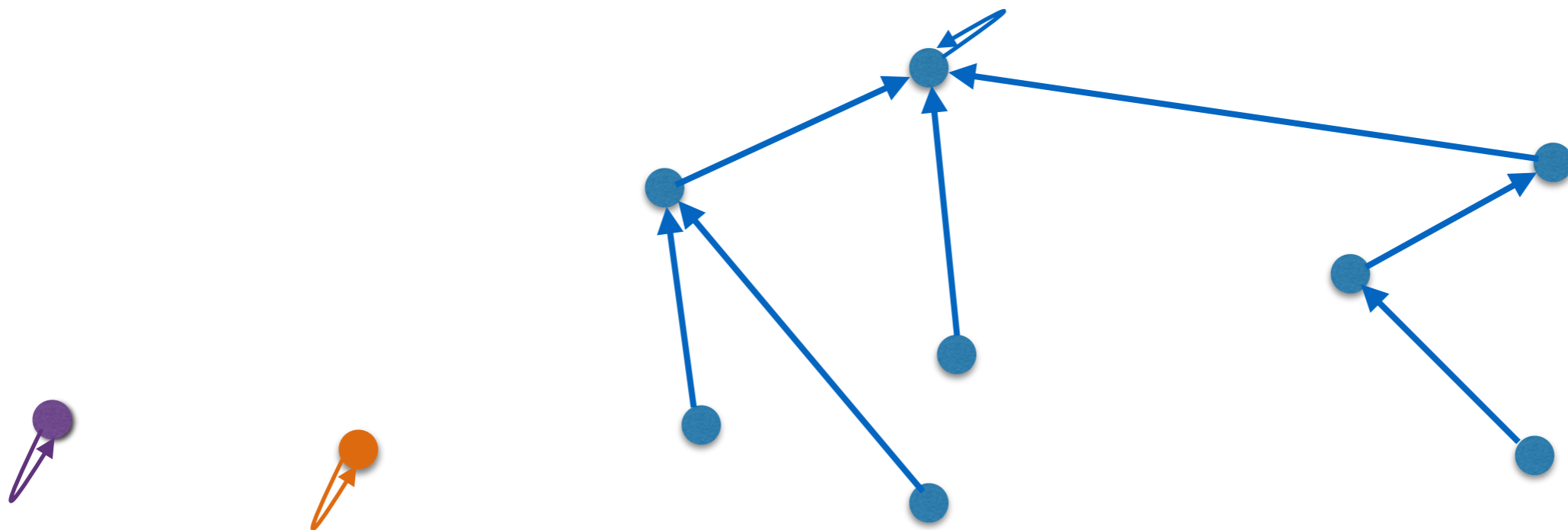
- How can we Union?

# 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 $O(\log k)$

# 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?
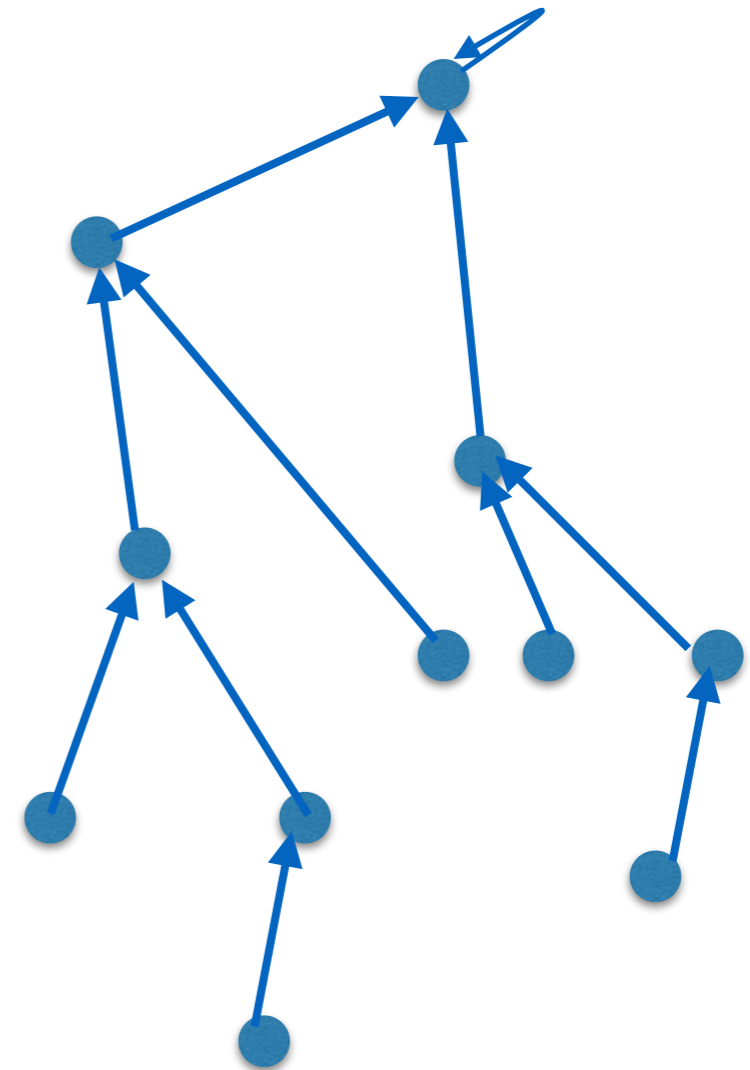
# Let's make things work a little faster in practice

- 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
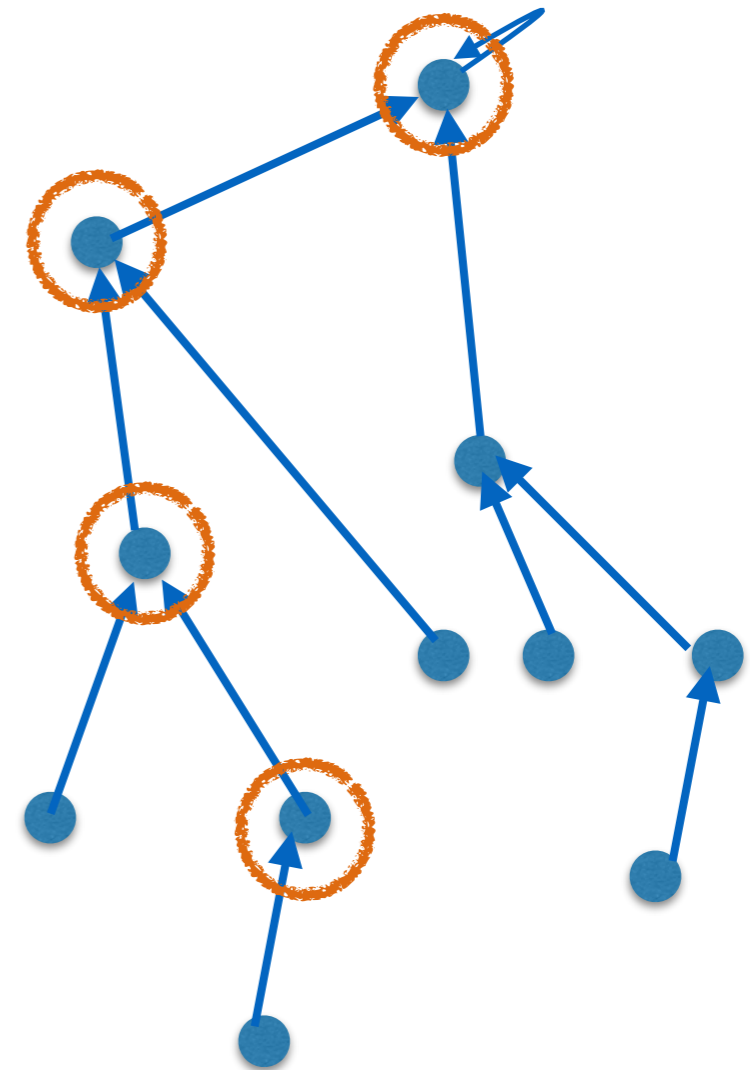
- So…let's do that!

# 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"
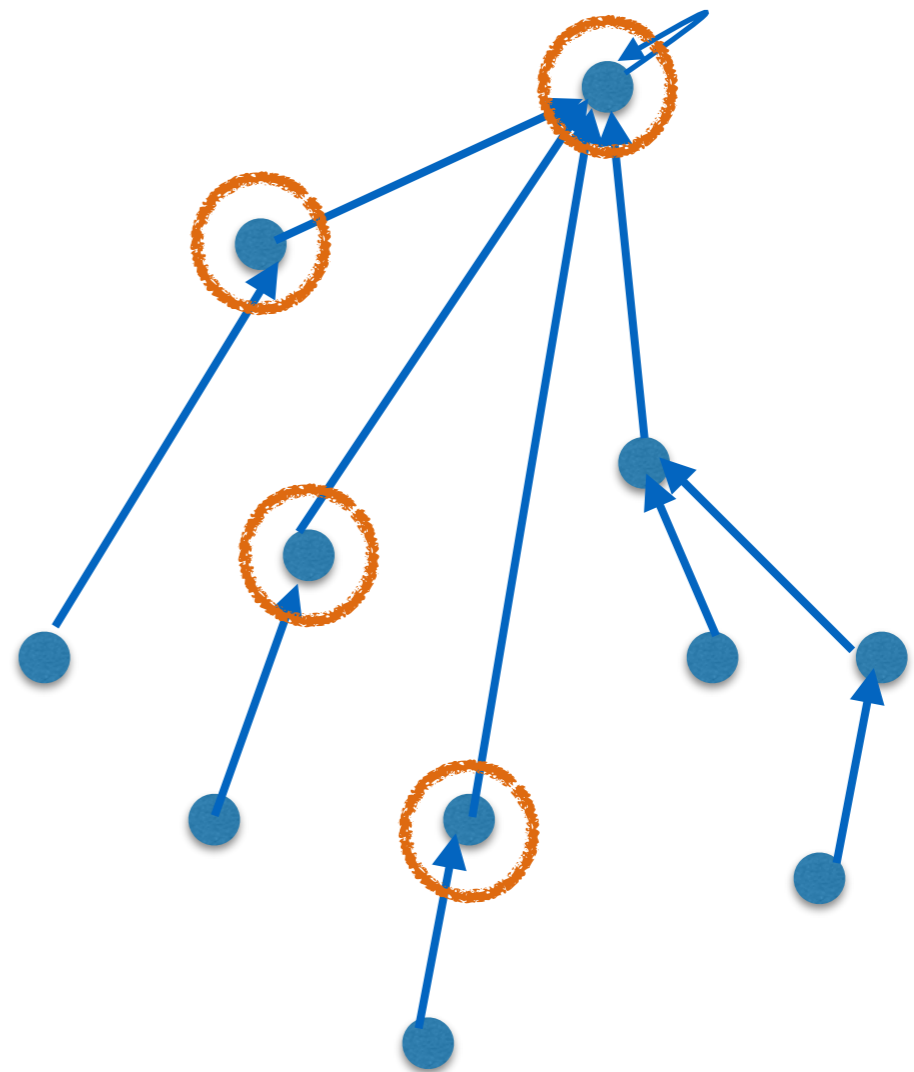
# Maintaining "Height"

- 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**

# 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) !

# 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 $n = 2^{2^{2^{2^{16}}}}$

**Digging Deeper**

# 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(!)

# Union-Find:  Applications

- 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!

**Digging Deeper**

# Back to MST

- Prim's algorithm: $O(m + n \log n)$ using a Fibonnacci tree

- Kruskal's algorithm:
  $O(m \log m + m\alpha(m)) = O(m \log m)$

- Which is better in practice?

  - Usually Kruskal's: a single sort is much better than Prim's repeated priority queue removals

- Is sorting time $\Omega(m \log m)$ required?

**Digging Deeper**

# Can we do better?

Best known algorithm by Chazelle (1999)

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

BERNARD CHAZELLE[†]

### 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 $\alpha$ 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 a
as Borůvka's work in 1926 [1, 9, 13]. In fact, MST is perhaps the oldest open p
computer science. According to Nešetřil [13], "this is a cornerstone problem of com
optimization and in a sense its cradle." Textbook algorithms run in $O(m \log n)$ time, where $n$

**Digging Deeper**

# 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 [computa-]tional model is a unit-cost random-access machine with the restriction that the only op[erations] allowed on edge weights are binary comparisons.

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

**Digging Deeper**

# 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 $O(T^*(m, n))$ where $T^*$ 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 $T^*$ are $T^*(m, n) = \Omega(m)$ and $T^*(m, n) = O(m \cdot \alpha(m$

a certain natural inverse of Ackermann's function.

Even under the assumption that $T^*$ is superlinear, we show that if the input graph

$G_{n,m}$, our algorithm runs in linear time with high probability, regardless of $n$, $m$, or th edge weights. The analysis uses a new martingale for $G_{n,m}$ similar to the edge-exp

for $G$

**Digging Deeper**

# 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** ("Prim's Algorithm", 1929)

  - Published by Jarník, independently discovered by Kruskal in 1956, by Prim in 1957

- **Kruskal's Algorithm** (1956)

  - Kruskal designed this because he found Borůvka's algorithm "unnecessarily complicated"

# Next class:
# Greedy Algorithms:
# Shortest Path

# Shortest Paths in Weighted Graph
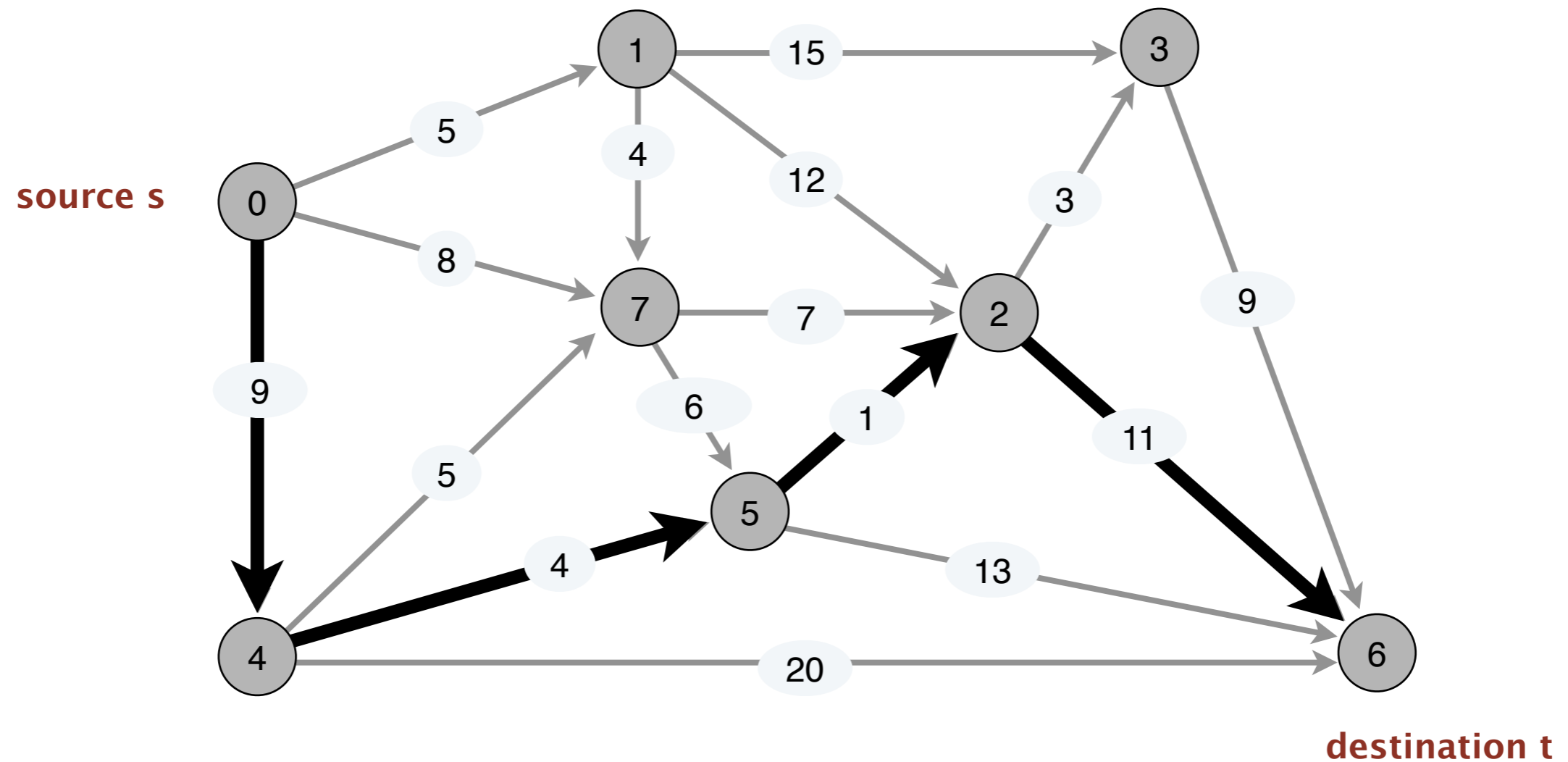

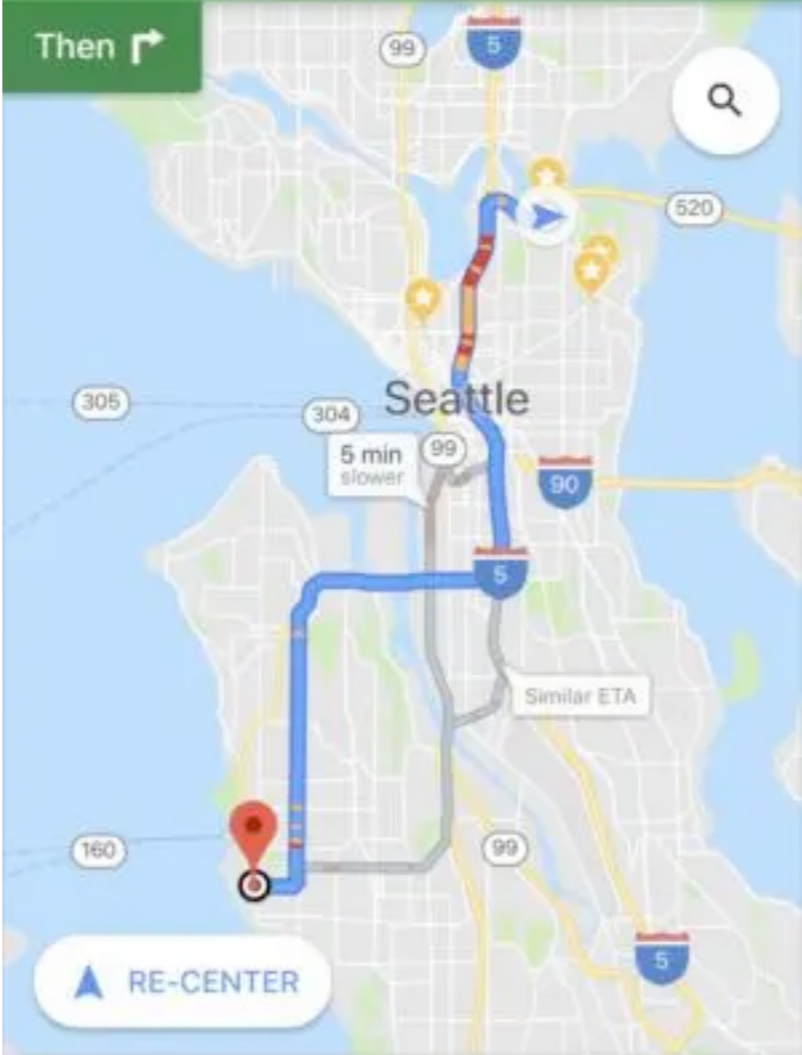
length of path = 9 + 4 + 1 + 11 = 25

# Shortest Paths in Weighted Graph

**Problem.**

Given a directed graph $G = (V, E)$ with positive edge weights: that is, each edge $e \in E$ has a positive weight $w(e)$ and vertices $s$ and $t$, find the shortest path from $s$ to $t$.

**Definition**. The shortest path from $s$ to $t$ in a weighted graph is a path $P$ from $s$ to $t$ (or a $s$-$t$ path) with minimum weight $w(P) = \sum_{e \in P} w(e)$.

# Midterm Questions?

Assignment questions (from any assignment)

Practice midterm questions

# 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)