

# Minimum Spanning Trees

CSE 332 Summer 2020

**Instructor:** Richard Jiang

## Teaching Assistants:

Hamsa Shankar Kristin Li Winston Jodjana

Maggie Jiang Hans Zhang Michael Duan

Jeffery Tian Annie Mao

*Lecture Q&A: [pollev.com/332summer](https://pollev.com/332summer)*

*Lecture clarifications: [tinyurl.com/332-08-17A](https://tinyurl.com/332-08-17A)*

# Announcements

- ❖ Welcome to the last week of 332 🙄🙄🙄
- ❖ Important Due Dates this Week:
  - Monday, 11:59pm: Ex 14 due
  - Tuesday, 11:59pm: P3 due – Thursday 11:59pm : With 2 late days
  - Thursday, 12:00am: Quiz 5 out
  - Friday, 11:59pm: Ex 15 due
  - Saturday, 3:00am: Quiz 5 due
- ❖ Mock interviews running from Tuesday to Thursday
  - No normal section on Thursday

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-17A](https://tinyurl.com/332-08-17A)*

# Lecture Outline

## ❖ Minimum Spanning Tree

- Prim's Algorithm
- Kruskal's Algorithm

## ❖ Disjoint Sets ADT

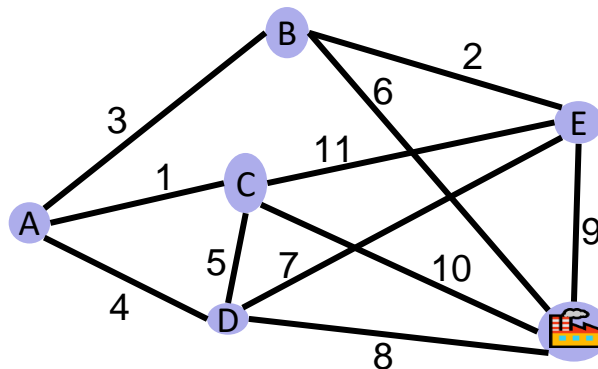
## ❖ Up-Tree Data Structure

- Representation
- Optimization: Weighted Union
- Optimization: Path Compression

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-08-17A](http://tinyurl.com/332-08-17A)*

# Problem Statement

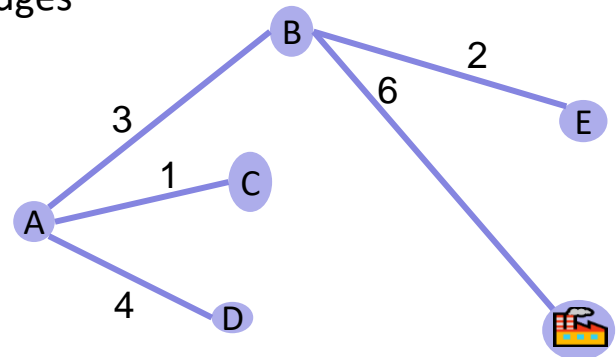
- ❖ Your friend at the electric company needs to connect all these cities to the power plant
- ❖ She knows the cost to lay wires between any pair of cities and wants the cheapest way to ensure electricity gets to every city



- ❖ Assume:
  - The graph is connected and undirected
  - *(In general, edge weights can be negative; just not in this example)*

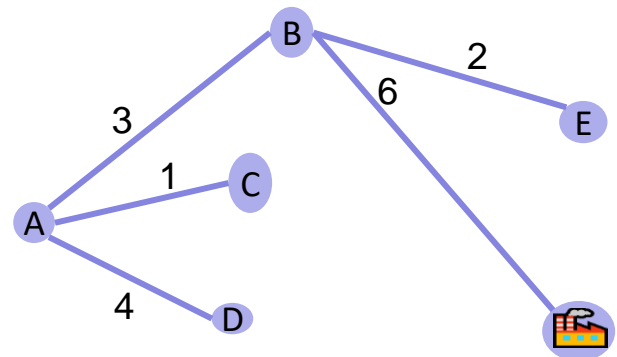
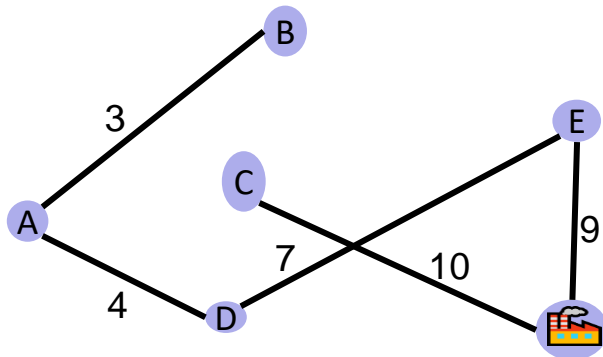
# Solution Statement

- ❖ We need a set of edges such that:
  - Every vertex touches at least one edge (“the edges **span** the graph”)
  - The graph using just those edges is **connected**
  - The total weight of these edges is **minimized**
- ❖ *Claim:* The set of edges we pick never forms a cycle. Why?
  - $V-1$  edges is the exact number of edges to connect all vertices
  - Taking away 1 edge breaks connectiveness
  - Adding 1 edge makes a cycle



# Solution Statement (v2)

- ❖ We need a ~~set of edges such that~~ Minimum Spanning Tree:
  - Every vertex touches at least one edge (“the edges **span** the graph”)
  - The graph using just those edges is **connected**
  - The total weight of these edges is **minimized**



# Minimum Spanning Trees

- ❖ Given an undirected graph  $G = (V, E)$ , a minimum spanning tree is a graph  $G' = (V, E')$  such that:
  - $E'$  is a subset of  $E$
  - $|E'| = |V| - 1$
  - $G'$  is connected
  - $\sum_{(u,v) \in E'} c_{uv}$  is minimal

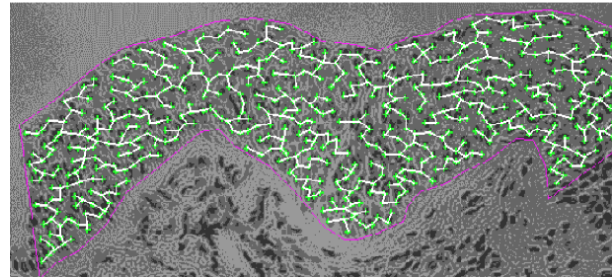
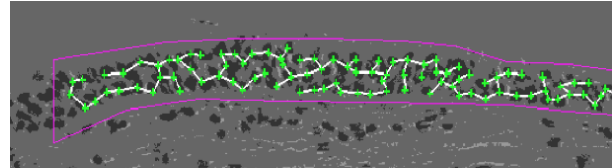
# Applications of MSTs

- ❖ Handwriting recognition
  - <http://dSPACE.mit.edu/bitstream/handle/1721.1/16727/43551593-MIT.pdf;sequence=2>



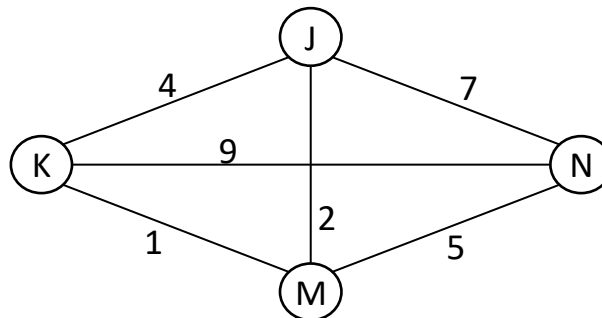
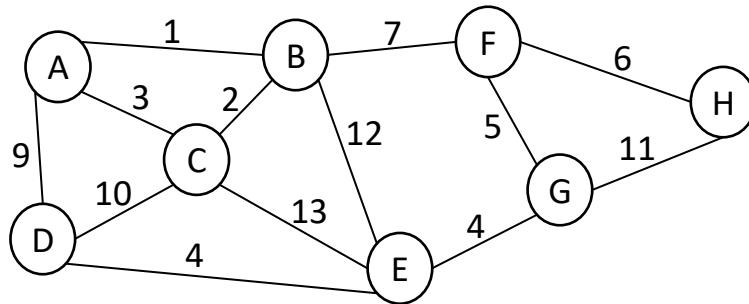
Figure 4-3: A typical minimum spanning tree

- ❖ Medical imaging
  - e.g. arrangement of nuclei in cancer cells

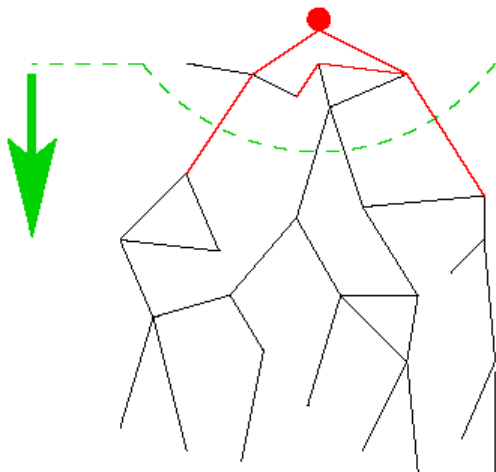


For more, see: <http://www.ics.uci.edu/~eppstein/gina/mst.html>

# Student Activity: Find the MSTs

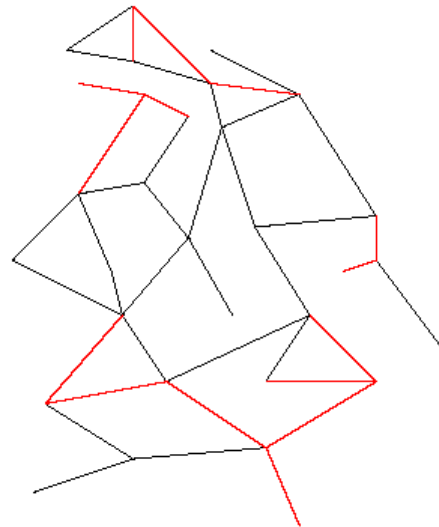


# MST Algorithms: Two Different Approaches



## **Prim's Algorithm**

Almost identical to Dijkstra's  
Start with one node, grow greedily



## **Kruskal's Algorithm**

Completely different!  
Start with a *forest* of MSTs, union them together  
(Need a new data structure for this)

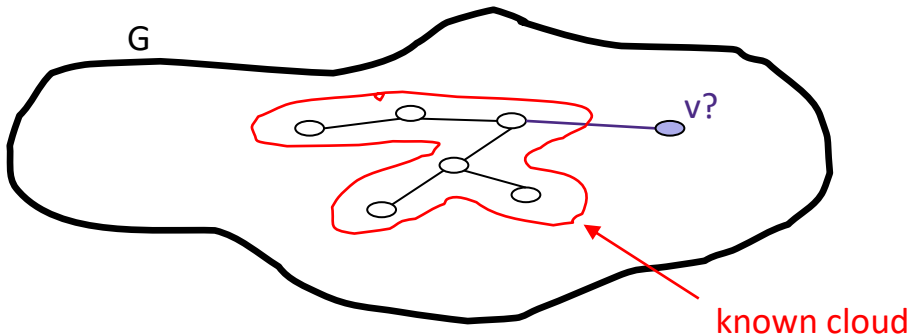
# Lecture Outline

- ❖ Minimum Spanning Tree
  - **Prim's Algorithm**
  - Kruskal's Algorithm
- ❖ Disjoint Sets ADT
- ❖ Up-Tree Data Structure
  - Representation
  - Optimization: Weighted Union
  - Optimization: Path Compression

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-17A](https://tinyurl.com/332-08-17A)*

# Prim's Algorithm\*\*

- ❖ *Intuition*: a vertex-based greedy algorithm
  - Builds MST by greedily adding vertices
- ❖ *Summary*: Grow a single tree by picking a vertex from the fringe that has the smallest cost
  - Unlike Dijkstra's, cost is the *edge weight* into the known set



\*\* This algorithm was developed in 1930 by Votěch Jarník, then independently rediscovered by Robert Prim in 1957 and then Dijkstra in 1959. It's also known as Jarník's, Prim-Jarník, or DJP

# Prim's Algorithm: Pseudocode

```
prims(Graph g) {
  foreach vertex v in g:
    v.distance =  $\infty$ 
  start = g.getSomeArbitraryVertex()
  start.distance = 0

  mst = {}
  heap = buildHeap(g.vertices - {start})
  foreach vertex v in start.neighbors():
    v.distance = g.weight(start, v)
    v.previous = start
    heap.decreaseKey(v, v.distance)

  while (! heap.empty()):
    v = heap.deleteMin()
    mst.addEdge(v, v.previous)
    foreach edge (v, u) in g:
      d1 = v.distance
      d2 = u.distance
      if (d1 < d2):
        u.previous = v
}
```

*Remember our 5-step pattern for a graph traversal?*

# Prim's Algorithm vs. Dijkstra's Algorithm (1 of 2)

- ❖ Dijkstra's picks an unknown vertex with smallest *distance to the source*
  - ie, path weights
- ❖ Prim's picks an unknown vertex with smallest *distance to the known set*
  - i.e., edge weights
- ❖ Some differences in the initialization, but otherwise identical

# Prim's Algorithm: Pseudocode

```
prims(Graph g) {
  foreach vertex v in g:
    v.distance =  $\infty$ 
  start = g.getSomeArbitraryVertex()
  start.distance = 0

  mst = {}
  heap = buildHeap(g.vertices - {start})
  foreach vertex v in start.neighbors():
    v.distance = g.weight(start, v)
    v.previous = start
    heap.decreaseKey(v, v.distance)

  while (! heap.empty()):
    v = heap.deleteMin()
    mst.addEdge(v, v.previous)
    foreach edge (v, u) in g:
      d1 = v.distance
      d2 = u.distance
      if (d1 < d2):

        u.previous = v
}
```

```
dijkstra(Graph g, Vertex start) {
  foreach vertex v in g:
    v.distance =  $\infty$ 

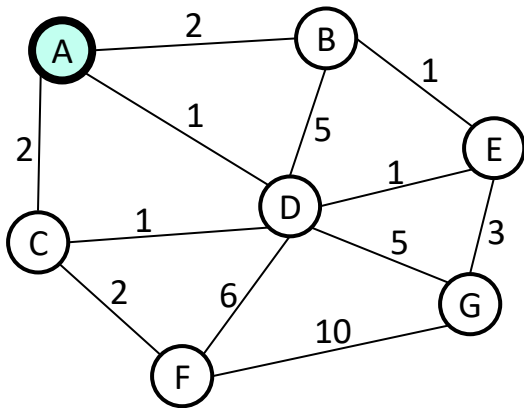
  start.distance = 0

  heap = buildHeap(g.vertices)

  while (! heap.empty()):
    v = heap.deleteMin()

    foreach edge (v, u) in g:
      d1 = v.dist + g.weight(v, u)
      d2 = u.dist
      if (d1 < d2):
        heap.decreaseKey(u, d1)
        u.previous = v
}
```

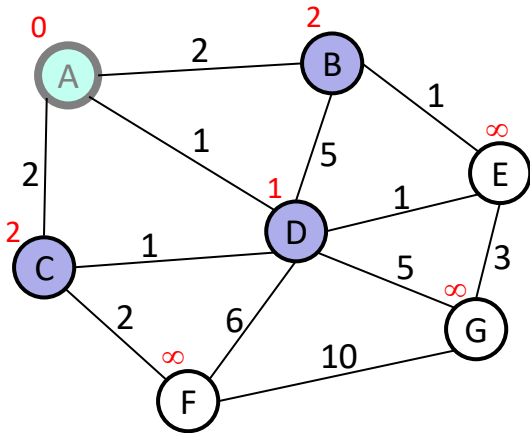
# Prim's Algorithm: Example



Order Added to Known Set:

Vertex	Known?	Distance	Previous
A		$\infty$	
B		$\infty$	
C		$\infty$	
D		$\infty$	
E		$\infty$	
F		$\infty$	
G		$\infty$	

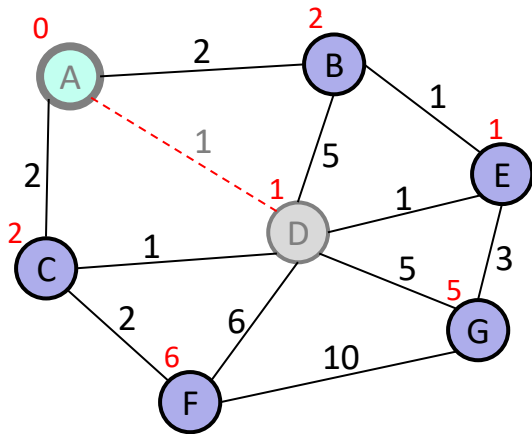
# Prim's Algorithm: Example



Order Added to Known Set:  
A

Vertex	Known?	Distance	Previous
A	Y	0	\
B		2	A
C		2	A
D		1	A
E		$\infty$	
F		$\infty$	
G		$\infty$	

# Prim's Algorithm: Example

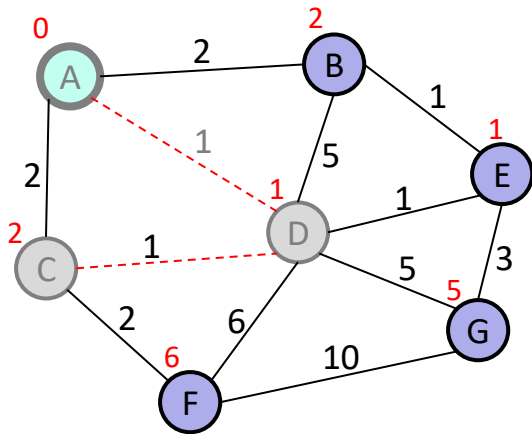


Order Added to Known Set:

A, D

Vertex	Known?	Distance	Previous
A	Y	0	\
B		2	A
C		1	D
D	Y	1	A
E		1	D
F		6	D
G		5	D

# Prim's Algorithm: Example

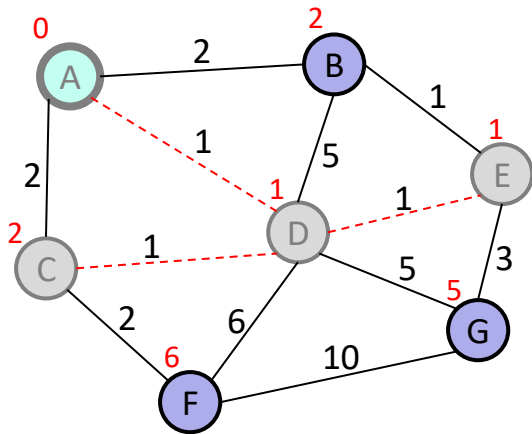


Order Added to Known Set:

A, D, C

Vertex	Known?	Distance	Previous
A	Y	0	\
B		2	A
C	Y	1	D
D	Y	1	A
E		1	D
F		2	C
G		5	D

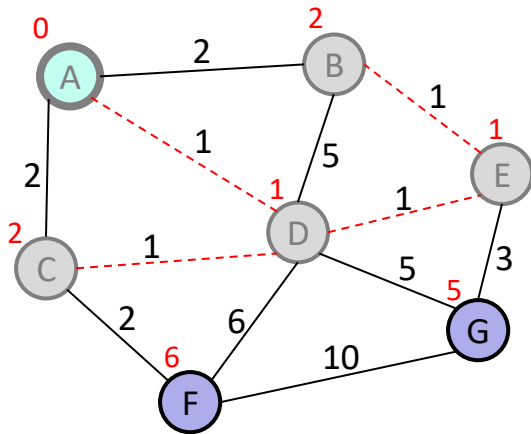
# Prim's Algorithm: Example



Order Added to Known Set:  
A, D, C, E

Vertex	Known?	Distance	Previous
A	Y	0	\
B		1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

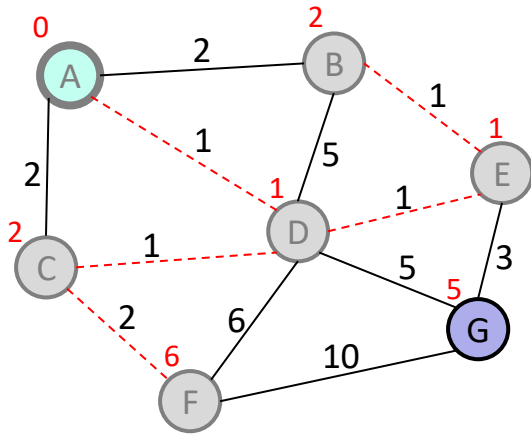
# Prim's Algorithm: Example



Order Added to Known Set:  
A, D, C, E, B

Vertex	Known?	Distance	Previous
A	Y	0	\
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F		2	C
G		3	E

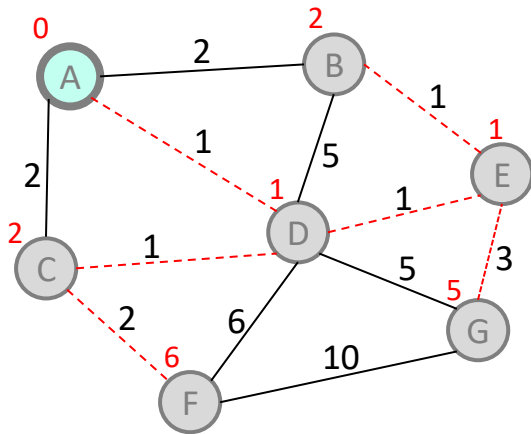
# Prim's Algorithm: Example



Order Added to Known Set:  
A, D, C, E, B, F

Vertex	Known?	Distance	Previous
A	Y	0	\
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G		3	E

# Prim's Algorithm: Example



Order Added to Known Set:

A, D, C, E, B, F

🐾🐾🐾 All Done!!! 🐾🐾🐾

Total Cost: 9

Vertex	Known?	Distance	Previous
A	Y	0	\
B	Y	1	E
C	Y	1	D
D	Y	1	A
E	Y	1	D
F	Y	2	C
G	Y	3	E

# Prim's Algorithm: Demos and Visualizations

## ❖ Dijkstra's Visualization

- <https://www.youtube.com/watch?v=1oiQ0hrVwJk>
- Dijkstra's proceeds radially from its source, because it chooses edges by *path length from source*

## ❖ Prim's Visualization

- <https://www.youtube.com/watch?v=6uq0cQZOyoY>
- Prim's jumps around the MST-under-construction (the fringe), because it chooses edges by *edge weight* (there's no source)

## ❖ Demo:

- [https://docs.google.com/presentation/d/1GPizbySYM5UhnXSXKvbqV4UhPCvrt750MiqPPgU-eCY/present?ueb=true&slide=id.g9a60b2f52\\_0\\_205](https://docs.google.com/presentation/d/1GPizbySYM5UhnXSXKvbqV4UhPCvrt750MiqPPgU-eCY/present?ueb=true&slide=id.g9a60b2f52_0_205)

# Prim's Algorithm: Analysis

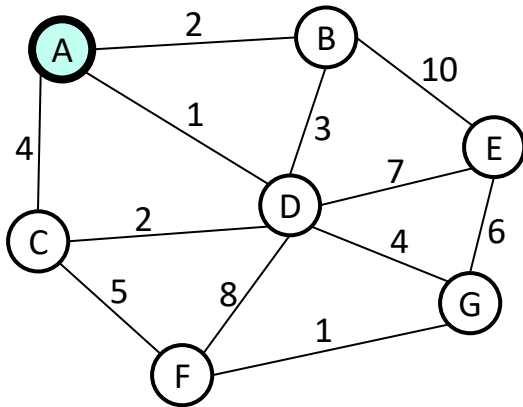
## ❖ Correctness:

- A bit tricky to prove, but intuitively similar to Dijkstra
- Might return to this time permitting (unlikely)

## ❖ Run-time:

- Same as Dijkstra's!  $O(|E|\log|V| + |V|\log|V|)$  using a priority queue
- But since  $E \in O(|V|^2)$ , can also state as  $O(|E|\log|V|)$

# Prim's Algorithm: Student Activity



Order Added to Known Set:

Vertex	Known?	Distance	Previous
A		$\infty$	
B		$\infty$	
C		$\infty$	
D		$\infty$	
E		$\infty$	
F		$\infty$	
G		$\infty$	

# Lecture Outline

- ❖ Minimum Spanning Tree
  - Prim's Algorithm
  - **Kruskal's Algorithm**
- ❖ Disjoint Sets ADT
- ❖ Up-Tree Data Structure
  - Representation
  - Optimization: Weighted Union
  - Optimization: Path Compression

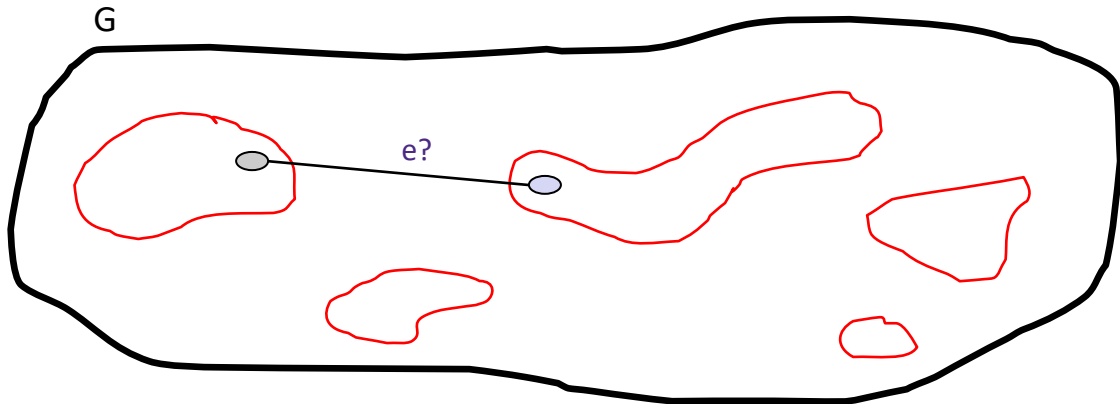
*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-17A](https://tinyurl.com/332-08-17A)*

# Kruskal's Algorithm: A Different Approach

- ❖ Prim's thinks vertex by vertex
  - Eg, add the closest vertex to the currently reachable set
- ❖ What if you think edge by edge instead?
  - Eg, start from the lightest edge; add it if it connects new things to each other (don't add it if it would create a cycle)

# Kruskal's Algorithm

- ❖ *Intuition*: an edge-based greedy algorithm
  - Builds MST by greedily adding edges
- ❖ *Summary*: Start with a **forest** of MSTs, and successively connect them by adding edges; do not create a cycle



# Kruskal's Algorithm: Pseudo-pseudocode

```
kruskals(Graph g) {  
    mst = {}  
    forests = buildForests(g.vertices)  
    edges = buildHeap(g.edges)  
  
    while (forests.numForests() != 1):  
        e = edges.deleteMin()  
        u_id = forests.getForestId(e.u)  
        v_id = forests.getForestId(e.v)  
        if (u_id != v_id):  
            mst.addEdge(e)  
            forests.connect(e.u, e.v)  
}
```

*Does this fit our 5-step pattern for a graph traversal?*

*What data structure is this?!?!?*

## Aside: Disjoint Sets ADT (1 of 2)

**Disjoint Sets ADT.** A collection of elements and sets of those elements.

- An element can only belong to a single set.
- Each set is identified by a unique id.
- Sets can be combined/connected/ unioned.

- ❖ The Disjoint Sets ADT has two operations:
  - `find(e)`: gets the id of the element's set
  - `union(e1, e2)`: combines the set containing `e1` with the set containing `e2`
- ❖ Example: ability to travel to drive to a country
  - `union(france, germany)`
  - `union(spain, france)`
  - `find(spain) == find(germany)?`
  - `union(england, france)`

## Aside: Disjoint Sets ADT (2 of 2)

- ❖ The Disjoint Sets ADT has two operations:
  - `find(e)`: gets the id of the element's set
  - `union(e1, e2)`: combines the set containing `e1` with the set containing `e2`
  
- ❖ Applications include percolation theory (computational chemistry) and .... Kruskal's algorithm
  
- ❖ Simplifying assumptions
  - We can map elements to indices quickly
  - We know all the items in advance; they're all disconnected initially
  
- ❖ In the this/next lecture, we'll see:
  - We can do `union()` in constant time
  - We can get `find()` to be **amortized** constant time
    - Worst case  $O(\log n)$  for an individual find operation

# Kruskal's Algorithm: Pseudocode

```

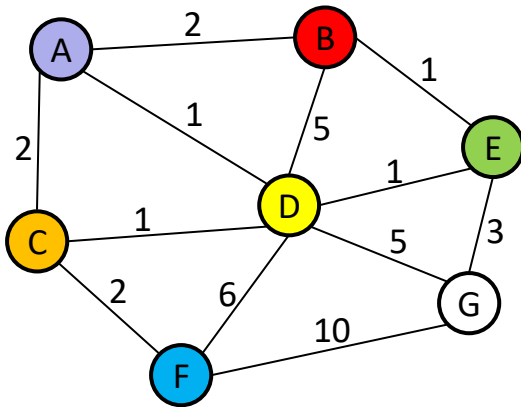
kruskals(Graph g) {
  mst = {}
  forests = buildDisjointSets(g.vertices)
  numforests = g.vertices
  edges = buildHeap(g.edges)

  while (numforests > 1):
    e = edges.deleteMin()
    u_id = forests.find(e.u)
    v_id = forests.find(e.v)
    if (u_id != v_id):
      mst.addEdge(e)
      forests.union(e.u, e.v)
      numforests--
}

```

*Runtime:*  $|E|(\log|E| + 2\log|V| + 1) + |V|(1 + 1 + 1) \in O(|E|\log|V| + |V|\log|V|)$   
*However, since we know  $E \in O(|V|^2)$ , runtime  $\in O(|E|\log|V|)$*

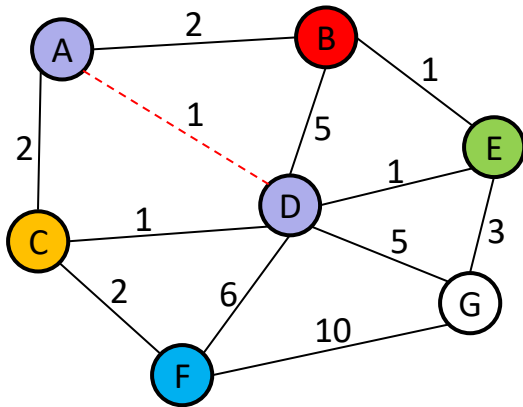
# Kruskal's Algorithm: Example



MST:

Weight	Edges
1	(A,D), (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

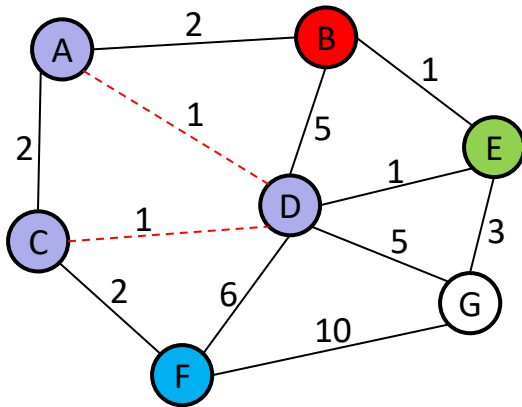
# Kruskal's Algorithm: Example



MST:  
(A, D)

Weight	Edges
1	(A,D), (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

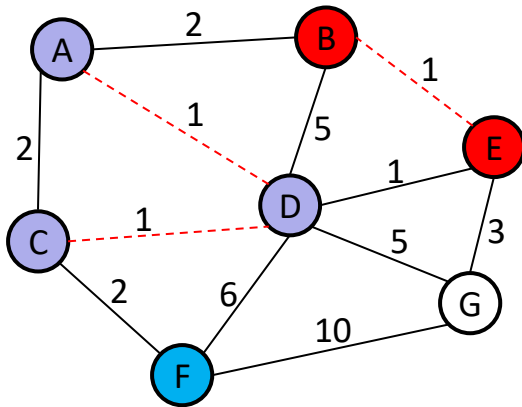
# Kruskal's Algorithm: Example



MST:  
(A, D), (C, D)

Weight	Edges
1	(A,D), (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

# Kruskal's Algorithm: Example

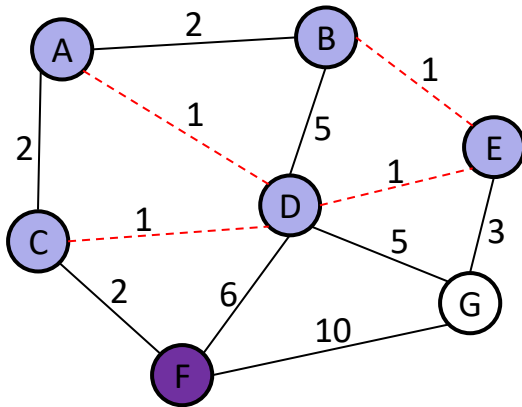


MST:

(A, D), (C, D), (B, E)

Weight	Edges
1	$(A,D), (C,D), (B,E), (D,E)$
2	$(A,B), (C,F), (A,C)$
3	$(E,G)$
5	$(D,G), (B,D)$
6	$(D,F)$
10	$(F,G)$

# Kruskal's Algorithm: Example

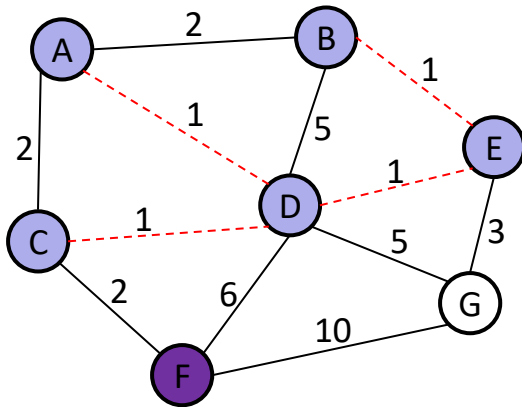


MST:

(A, D), (C, D), (B, E), (D, E)

Weight	Edges
1	(A,D), (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

# Kruskal's Algorithm: Example

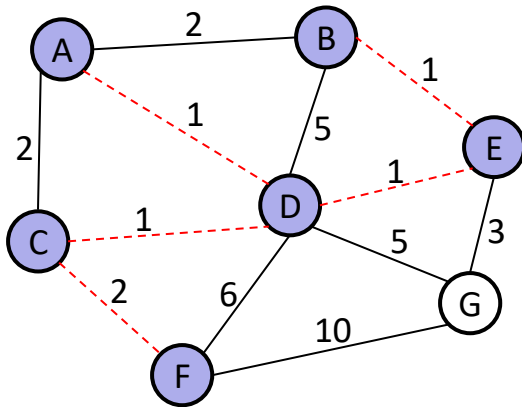


MST:

(A, D), (C, D), (B, E), (D, E)

Weight	Edges
1	(A,D), (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

# Kruskal's Algorithm: Example

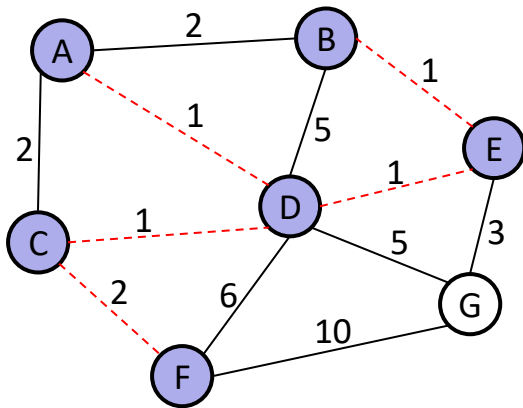


MST:

(A, D), (C, D), (B, E), (D, E), (C, F)

Weight	Edges
1	(A,D), (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

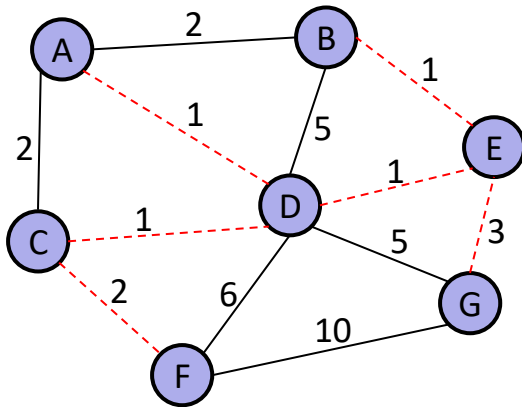
# Kruskal's Algorithm: Example



MST:  
 (A, D), (C, D), (B, E), (D, E), (C, F)

Weight	Edges
1	(A,D), (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

# Kruskal's Algorithm: Example



  An MST!!!  

Total Cost: 9

**MST:**  
 (A, D), (C, D), (B, E), (D, E), (C, F), (E, G)

Weight	Edges
1	(A,D), (C,D), (B,E), (D,E)
2	(A,B), (C,F), (A,C)
3	(E,G)
5	(D,G), (B,D)
6	(D,F)
10	(F,G)

# Kruskal's Algorithm: Demos and Visualizations

## ❖ Prim's Visualization

- <https://www.youtube.com/watch?v=6uq0cQZOyoY>
- Prim's jumps around the fringe, adding edges by edge weight

## ❖ Kruskal's Visualization:

- <https://www.youtube.com/watch?v=ggLyKfBTABo>
- Kruskal's jumps around the graph – not just the fringe – because it chooses edges by edge weight independent of the “tree under construction”

## ❖ Conceptual demo:

- [https://docs.google.com/presentation/d/1RhRSYs9Jbc335P24p7vR-6PLXZUI-1EmeDtqieL9ad8/present?ueb=true&slide=id.g375bbf9ace\\_0\\_645](https://docs.google.com/presentation/d/1RhRSYs9Jbc335P24p7vR-6PLXZUI-1EmeDtqieL9ad8/present?ueb=true&slide=id.g375bbf9ace_0_645)

# Kruskal's Algorithm: Correctness

- ❖ Kruskal's algorithm is clever, simple, and efficient
  - But does it generate a minimum spanning tree?
- ❖ *First*: prove it is a spanning tree
  - Use **Contradiction!**
  - *Intuition*: Original graph was connected; we kept edges that didn't create a cycle
- ❖ *Second*: there is no spanning tree with lower total cost
  - A more complex proof that uses **Induction** and **Contradiction**
  - Take CSE 421!

# Summary

- ❖ Minimum Spanning Trees are a subset of the edges in an undirected connected graph
- ❖ Prim's looks a lot like the vertex-based graph traversals we've seen so far, except it uses *edge weight* instead of *path weight*
  - And since edge weights don't change during the algorithm's execution, we don't need a `decreaseKey()` operation
- ❖ Kruskal's is an edge-based graph traversal (which we haven't seen so far), but still uses *edge weight* to choose edges
  - Doesn't need `decreaseKey()` for the same reason
  - Needs an auxiliary ADT – the Disjoint Sets ADT – to speed up execution

# Lecture Outline

- ❖ Minimum Spanning Tree
  - Prim's Algorithm
  - Kruskal's Algorithm
- ❖ **Disjoint Sets ADT**
- ❖ Up-Tree Data Structure
  - Representation
  - Optimization: Weighted Union
  - Optimization: Path Compression

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-08-17A](http://tinyurl.com/332-08-17A)*

# Implementing the Disjoint Sets ADT (1 of 2)

❖ If we have  $n$  elements:

- What is the total cost of  $m$  find()s +  $\leq n-1$  union()s?

*Can we have  $>n$  union()s?*

❖ Goal:  $O(m+n)$

- i.e.  $O(1)$  amortized
- Can get  $O(1)$  worst-case union()
- Would be nice if we could get  $O(1)$  worst-case find(), but...
- *Known result*: both find() and union() can't have worst-case  $O(1)$

# Implementing the Disjoint Sets ADT (2 of 2)

## ❖ *Observation:*

- Trees let us find many elements given a single root

## ❖ *Idea:*

- If we reverse the pointers (ie, point up from child to parent), we can find a single root from many elements

## ❖ *Decision:*

- One up-tree for each set
- The ID of the set is (hash of) the tree root

# Lecture Outline

- ❖ Minimum Spanning Tree
  - Prim's Algorithm
  - Kruskal's Algorithm
- ❖ Disjoint Sets ADT
- ❖ Up-Tree Data Structure
  - **Representation**
  - Optimization: Weighted Union
  - Optimization: Path Compression

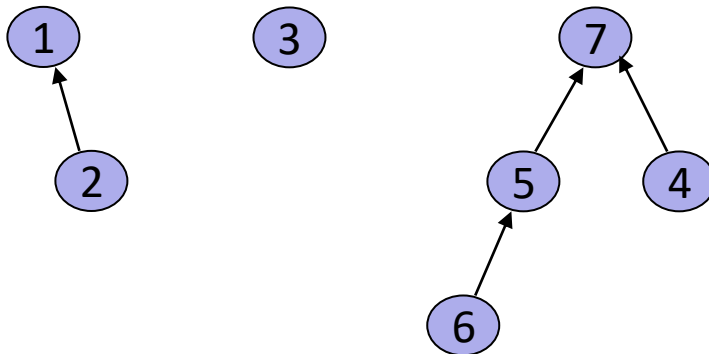
*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-17A](https://tinyurl.com/332-08-17A)*

# Up-Tree Data Structure for Disjoint Sets ADT

❖ Initial State:



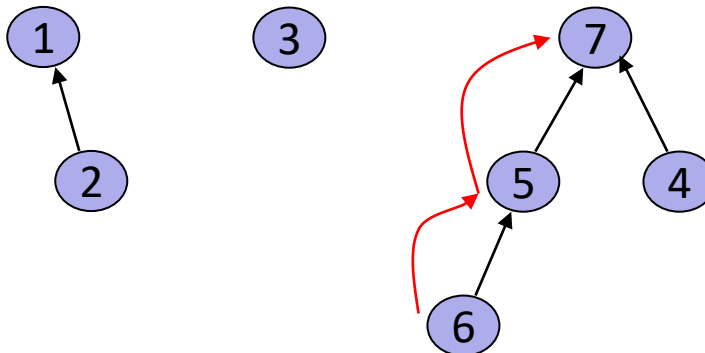
❖ After several union():s:



❖ Roots are the IDs for each set

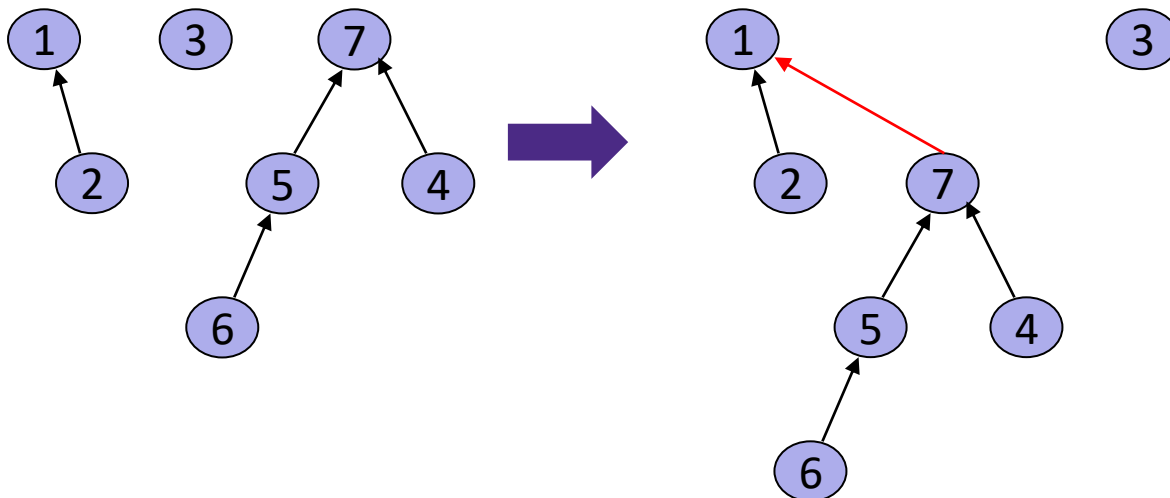
# Up-Tree Find

- ❖  $\text{find}(x)$ : follow  $x$  to the root and return the root
  - Eg:  $\text{find}(6) = 7$



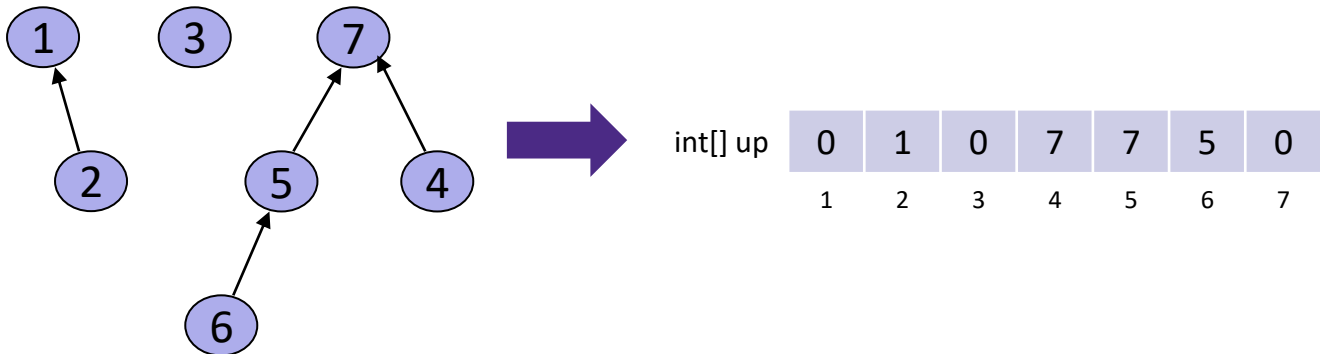
# Up-Tree Union

- ❖  $\text{union}(x)$ : assuming  $x$  and  $y$  are roots, point  $y$  to  $x$ 
  - If  $x$  or  $y$  are not roots, can require caller to call  $\text{find}()$  first
  - Eg:  $\text{union}(1, 7)$



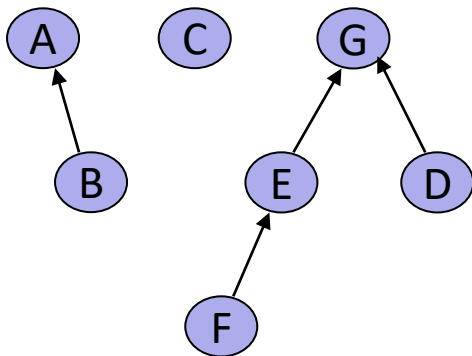
# Up-Tree Representation (1 of 2)

- ❖ Up-Trees can be represented as an array of indices
  - $\text{up}[x] = 0$  means  $x$  is a root
  - Note: array is 1-indexed



# Up-Tree Representation (2 of 2)

- ❖ Up-Trees can be represented as an array of indices
  - Can contain non-integer values using a hash table to map to indices



A	1
B	2
C	3
D	4
E	5
F	6
G	7

int[] up	0	1	0	7	7	5	0
	1	2	3	4	5	6	7

# Up-Tree Implementation

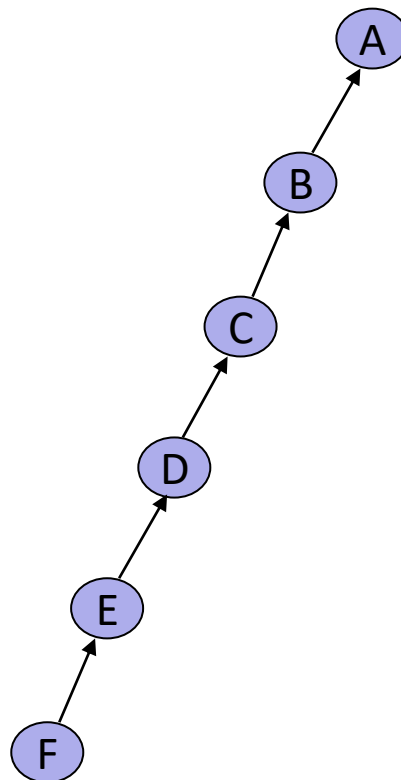
```
void union(int x, int y) {  
    up[y] = x;  
}
```

```
int find(int x) {  
    while (up[x] != 0) {  
        x = up[x];  
    }  
    return x;  
}
```

- ❖ Worst-case runtime for union():
- ❖ Worst-case runtime for find():
- ❖ Total runtime for  $n-1$  union()s and  $m$  find()s:

# Worst-case Union

```
union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)
```



 *If only I could keep these trees (semi-?)balanced*

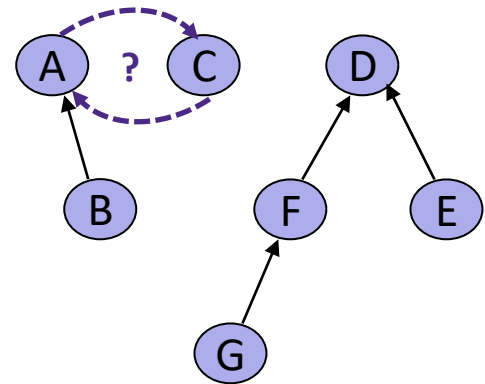
# Lecture Outline

- ❖ Minimum Spanning Tree
  - Prim's Algorithm
  - Kruskal's Algorithm
- ❖ Disjoint Sets ADT
- ❖ Up-Tree Data Structure
  - Representation
  - **Optimization: Weighted Union**
  - Optimization: Path Compression

*[pollev.com/332summer](http://pollev.com/332summer) :: [tinyurl.com/332-08-17A](http://tinyurl.com/332-08-17A)*

# Weighted Union (1 of 2)

- ❖ Our naïve union() always picked the same argument (the second argument) to become the child in the unioned structure

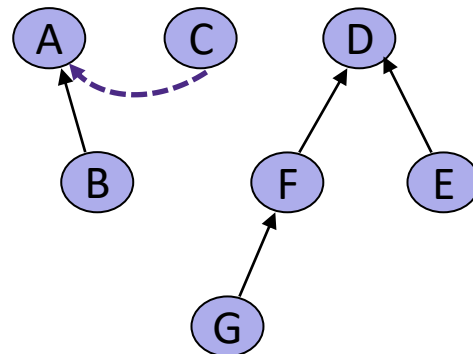


→

```
union(A, B)
union(A, C)
union(A, D)
...
```

## Weighted Union (2 of 2)

- ❖ Our naïve union() always picked the same argument (the second argument) to become the child in the unioned structure



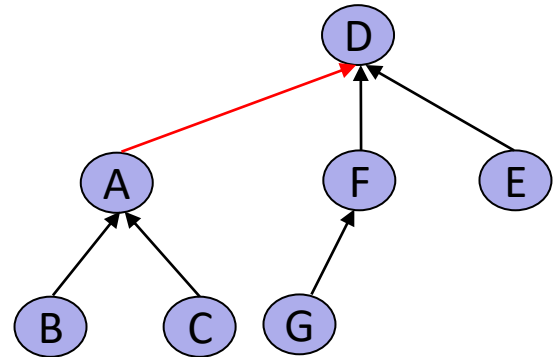
- ❖ Instead:
  - Pick the smaller tree (ie, tree with fewer nodes) to be the new child
    - Let “weight” = “num nodes”
  - Add the new child to the root




```
union(A, B)
union(A, C)
union(A, D)
...
```

## Weighted Union (2 of 2)

- ❖ Our naïve union() always picked the same argument (the second argument) to become the child in the unioned structure



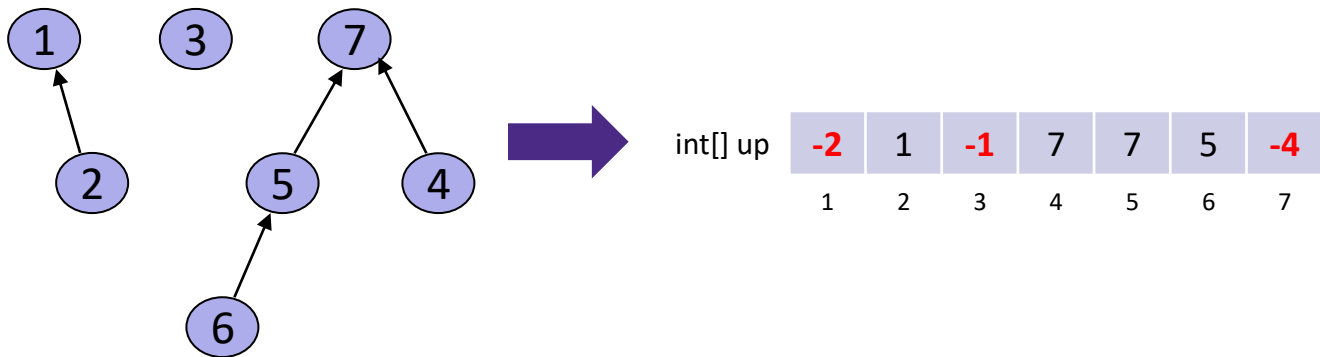
- ❖ Instead:
  - Pick the smaller tree (ie, tree with fewer nodes) to be the new child
    - Let “weight” = “num nodes”
  - Add the new child to the root



```
union(A, B)
union(B, C)
union(A, D)
...
```

# Weighted Union: Representation

- ❖ Need to store *number of nodes* (or “weight”) of each tree
- ❖ Instead of ‘0’, we can store the root’s weight instead!
  - Use negative values to indicate they’re not indices
  - See Weiss, 8.4



# Weighted Union: Implementation

```
void union(int x, int y) {  
    up[y] = x;  
}
```

```
weightedUnion(int x, int y) {  
    wx = weight[x];  
    wy = weight[y];  
    if (wx < wy) {  
        up[x] = y;  
        weight[y] = wx + wy;  
    } else {  
        up[y] = x;  
        weight[x] = wx + wy;  
    }  
}
```

*union()'s runtime is still  $O(1)$ !*

*Does this (slightly) added complexity help us  
balance the up-trees and improve find()?*

# Weighted Union: Performance

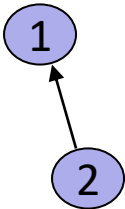
- ❖ Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are “spindly”

N	H
1	0

1

# Weighted Union: Performance

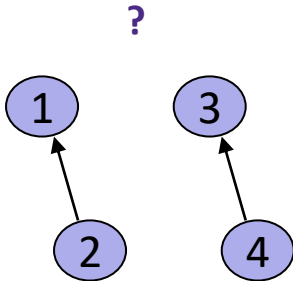
- ❖ Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are “spindly”



N	H
1	0
2	1

# Weighted Union: Performance

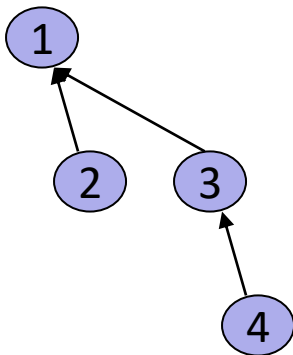
- ❖ Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are “spindly”



N	H
1	0
2	1
4	?

# Weighted Union: Performance

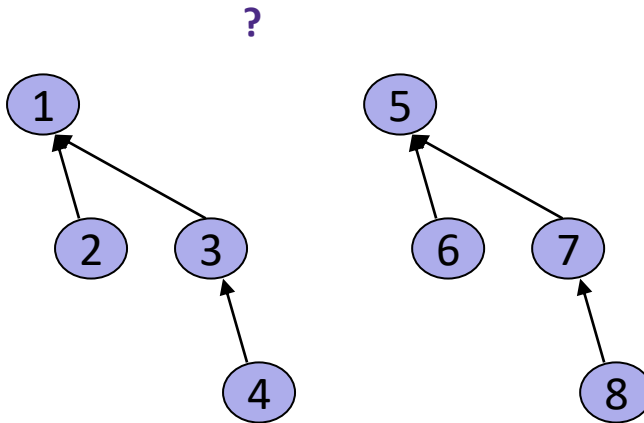
- ❖ Consider the worst case: tree height grows as fast as possible



N	H
1	0
2	1
4	2

# Weighted Union: Performance

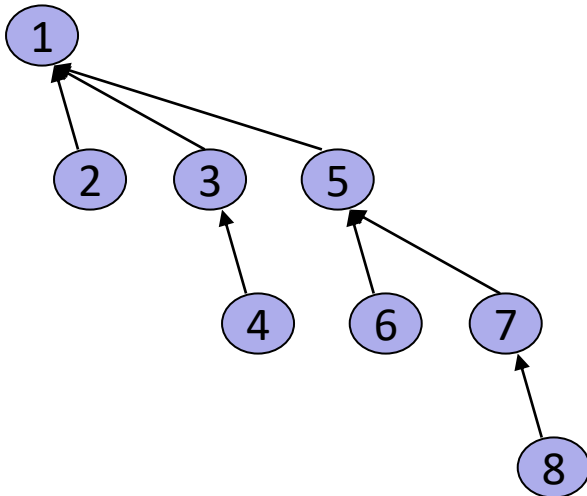
- ❖ Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are “spindly”



N	H
1	0
2	1
4	2
8	?

# Weighted Union: Performance

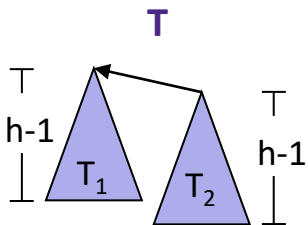
- ❖ Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are “spindly”
- ❖ Worst-case height and worst-case find() is  $\Theta(\log N)$



N	H
1	0
2	1
4	2
8	3
$2^n$	$n$

# Weighted Union Performance: Proof

- ❖ An up-tree with height  $h$  using weighted union has weight at least  $2^h$
- ❖ Proof by induction
  - *Base-case*:  $h = 0$ . The up-tree has one node and  $2^0 = 1$
  - *Inductive step*: Assume true for all  $h' < h$



Minimum weight up-tree of height  $h$  formed by weighted unions

We know:

$$W(T_1) \geq 2^{h-1}$$

$$W(T_2) \geq 2^{h-1}$$

} *Induction hypothesis*

$$W(T_1) \geq W(T_2)$$

} *Definition of weighted union*

Since  $W(\mathbf{T}) = W(T_1) + W(T_2)$ ,

we know that

$$W(\mathbf{T}) \geq W(T_1) + W(T_2)$$

$$= 2^{h-1} + 2^{h-1}$$

$$= 2^h$$

Therefore  $W(\mathbf{T}) \geq 2^h$



# Poll Everywhere

[pollev.com/332summer](https://pollev.com/332summer)

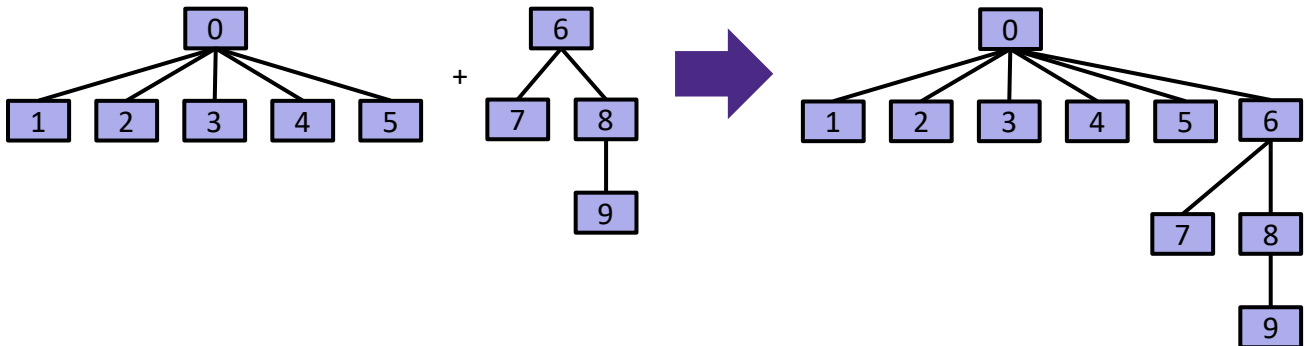
- ❖ What is the runtime for ...
  - weighted union(), worst-case
  - find(), worst-case
  - n-1 union()s + m find()s
  
- A.  $\Theta(1) / \Theta(1) / O(n + m)$
- B.  $\Theta(1) / \Theta(n) / O(n + m^2)$
- C.  $\Theta(1) / \Theta(\log n) / O(n + m^2)$
- D.  $\Theta(1) / \Theta(\log n) / O(n + m \log n)$
- E. I'm not sure ...

```
weightedUnion(int x, int y) {
    wx = weight[x];
    wy = weight[y];
    if (wx < wy) {
        up[x] = y;
        weight[y] = wx + wy;
    } else {
        up[y] = x;
        weight[x] = wx + wy;
    }
}
```

```
int find(int x) {
    while (up[x] > 0) {
        x = up[x];
    }
    return x;
}
```

# Why Weights Instead of Heights?

- ❖ We used the *number of items* in a tree to decide upon the root
- ❖ Why not use the *height* of the tree?
  - Heighted Union's runtime is asymptotically the same:  $\Theta(\log(N))$ 
    - Proof is left as an exercise to the reader ;)



- Easier to track weights than heights, and heighted union doesn't combine very well with the next optimization technique for find()

# Lecture Outline

- ❖ Minimum Spanning Tree
  - Prim's Algorithm
  - Kruskal's Algorithm
  
- ❖ Disjoint Sets ADT
  
  
- ❖ Up-Tree Data Structure
  - Representation
  - Optimization: Weighted Union
  - **Optimization: Path Compression**

*[pollev.com/332summer](https://pollev.com/332summer) :: [tinyurl.com/332-08-17A](https://tinyurl.com/332-08-17A)*

## Modifying Data Structures To Preserve Invariants

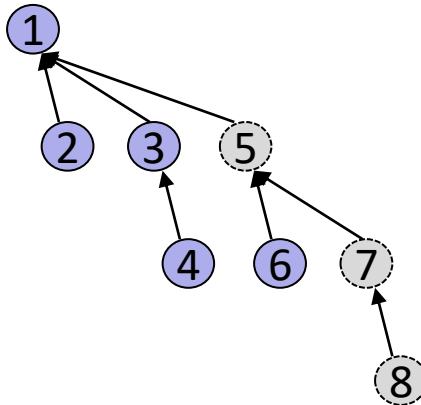
- ❖ Thus far, the modifications we've studied are designed to *preserve invariants* (aka “repair the data structure”)
  - **Tree rotations:** preserve AVL tree invariants
  - **Promoting keys / splitting leaves:** preserve B-tree invariants (eg, L+1 keys stored in a leaf node)
- ❖ Notably, the modifications don't improve runtime *between identical method calls*
  - If `avl.find(x)` takes  $2 \mu\text{s}$ , we expect future calls to take  $\sim 2 \mu\text{s}$
  - If we call `avl.find(x)`  $m$  times, the total runtime should be  $\sim 2m \mu\text{s}$

# Modifying Data Structures for Future Gains

- ❖ Path compression is entirely different: we are modifying the up-tree to *improve future performance*
  - If `uptree.find(x)` takes  $2 \mu\text{s}$ , we expect future calls to take  $<2 \mu\text{s}$
  - If we call `uptree.find(x)`  $m$  times, the total runtime should be  $<2m \mu\text{s}$ 
    - ... and possibly even  $\ll 2m \mu\text{s}$

# Path Compression: Idea

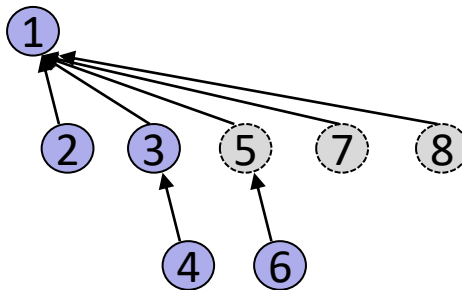
- ❖ Recall the worst-case structure if we use weighted union:



- ❖ *Idea*: When we find(8), move all visited nodes under the root
  - Additional cost is insignificant (same order of growth) , so run path compression on every find()

# Path Compression: Example

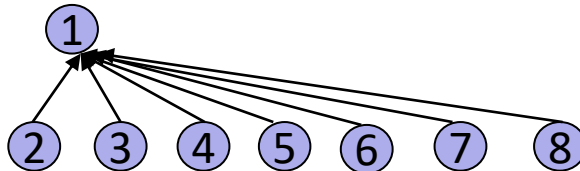
- ❖ Recall the worst-case structure if we use weighted union



- ❖ *Idea*: When we find(8), move all visited nodes under the root
  - Additional cost is insignificant (same order of growth), so run path compression on every find()
  - Doesn't meaningfully change runtime for *this* invocation of find(8), but *subsequent* find(8)s (and subsequent find(7)s and find(5)s and ...) will be faster!

# Path Compression: Details and Runtime

- ❖ With “enough” path compression aka “enough” find()s ...



- How much is “enough”? Probably  $m > n$

# Path Compression: Implementation

```
int find(int x) {
    while (up[x] != 0) {
        x = up[x];
    }
    return x;
}
```

```
int pathCompressionFind(int x) {
    while (up[x] > 0) {
        x = up[x];
    }
    int root = x;

    // Change the parent for all
    // nodes along this path
    while (up[x] > 0) {
        x = up[x];
        up[x] = root;
    }
    return root;
}
```

*find()'s worst-case runtime is still  $O(\log n)$ !*

*Does this (slightly) added complexity help us make the up-trees shallower and improve sequences of find()?*

# Path Compression: Runtime

- ❖ In total,  $m > 1$  find()s will take  $O(m \log^* n)$  time
  - $\log^* n$  is the “iterated log”: the number of times you need to apply log to  $n$  before the result is  $\leq 1$
  - See Weiss for proof

- ❖ find()s and union()s are now amortized constant time
  - $\log^* n < 5$  for any realistic  $n$
  - If  $\log^* n \approx 5$ , then  $m$  find()/union()s amortizes to  $O(1)$ ! 🤖

n	$\log^* n$
1	0
2	1
4	2
16	3
65536	4
$2^{65536}$	5

$2^{16}$

Number of atoms in the known universe is  $2^{256}$ ish

# Interlude: A Really Slow Function

- ❖ Ackermann's function is a really big function  $A(x, y)$  with inverse  $\alpha(x, y)$  which is really small
- ❖  $\alpha$  shows up in:
  - Computation Geometry (surface complexity)
  - Combinatorics of sequences
- ❖ How fast does  $\alpha(x, y)$  grow?
  - Even slower than iterated log!
  - For all practical purposes,  $\alpha(x, y) < 4$

# Path Compression: Tighter Runtime

- ❖ A sequence of  $p$  union()s + find()s on a set of  $n$  elements has worst-case time of  $O(p \cdot \alpha(p, n))$ 
  - Assumes weighted union and path compression
  - Proved by Robert Tarjan in 1984 (see also Fibonacci heaps and splay trees)
- ❖ So find()'s amortized runtime is  $O(1)$ 
  - Since  $O(p \cdot 4)$  for  $p$  operations!
- ❖ Complex analysis, but inverse-Ackermann's is a tighter bound than iterated-log

# Summary

- ❖ Worst-case runtime for up-trees with weighted union and path compression:
  - union is  $O(1)$
  - find is  $O(\log n)$
- ❖ Total time for  $m \geq n$  operations on  $n$  elements is  $O(m \cdot \alpha(m, n))$ 
  - An individual operation can be costly, but over time the “average” cost per operation is not
  - Logic is similar to array-resizing: an individual `insert()` can be costly (*due to resizing*), but over time the “average” `insert()` is not
- ❖ Using “ranked union” gives an even better bound theoretically