CSE 332: Data Structures and Parallelism

Fall 2022

Richard Anderson

Lecture 27: Minimum Spanning Trees

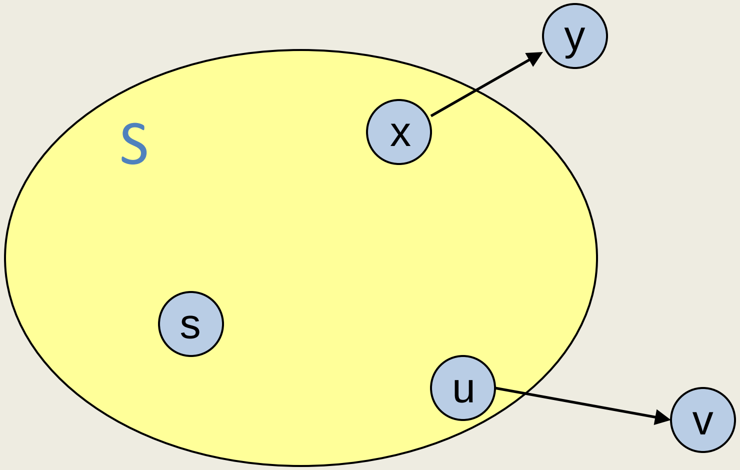
Announcements

- Upcoming lectures
  - Graph Algorithms
    - ~~Intro to graphs~~
    - ~~Topological Sort~~
    - ~~Graph Traversal~~
    - ~~Shortest Paths~~
    - Minimum Spanning Tree
  - Theory of NP-Completeness (2 lectures)
  - Review session (Tuesday, Dec 13 (?))
  - Final Exam, Thursday, Dec 15, 8:30-10:20 AM

Assume all edges have non-negative cost

Dijkstra's Algorithm
What about negative cost edges?

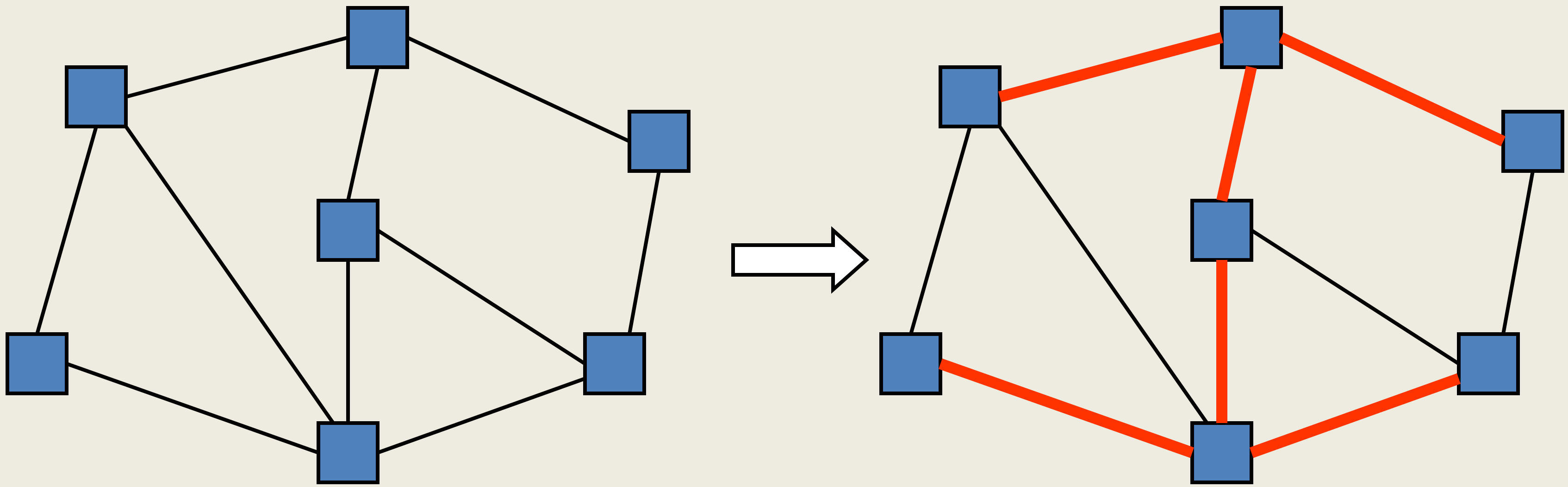
```
S = { };    d[s] = 0;     d[v] = infinity for v != s
while S != V
     Choose v in V-S with minimum d[v]
     Add v to S
     for each  w in the neighborhood of v
          newCost = d[v] + c(v, w)
          if (newCost < d[w])
               d[w] = newCost
               prev[w] = v
```



Graph Theory

- $G = (V, E)$
  - V: vertices, $|V| = n$
  - E: edges, $|E| = m$
- Undirected graphs
  - Edges sets of two vertices {u, v}
- Directed graphs
  - Edges ordered pairs (u, v)
- Many other flavors
  - Edge / vertices weights
  - Parallel edges
  - Self loops
- Path: $v_1, v_2, \ldots, v_k$, with (v_i, v_{i+1}) in E
  - Simple Path
  - Cycle
  - Simple Cycle
- Neighborhood
  - N(v)
- Distance
- Connectivity
  - Undirected
  - Directed (strong connectivity)
- Trees
  - Rooted
  - Unrooted

Spanning Tree in an Undirected Graph

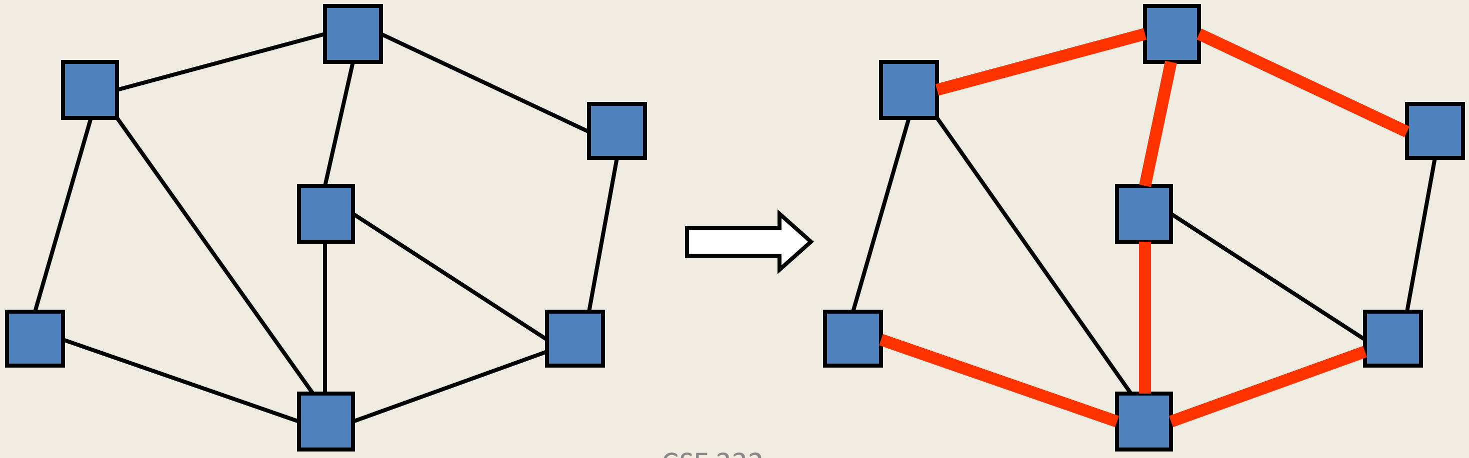


Spanning tree
- Connects all the vertices
- No cycles

Note: this is a problem where there is a difference between undirected graphs and directed graphs

Spanning Tree Problem

- Input: An undirected graph G = (V,E). G is connected.
- Output: **T** ⊂ E such that
  – (V,**T**) is a connected graph
  – (V,**T**) has no cycles



Spanning Tree Algorithm

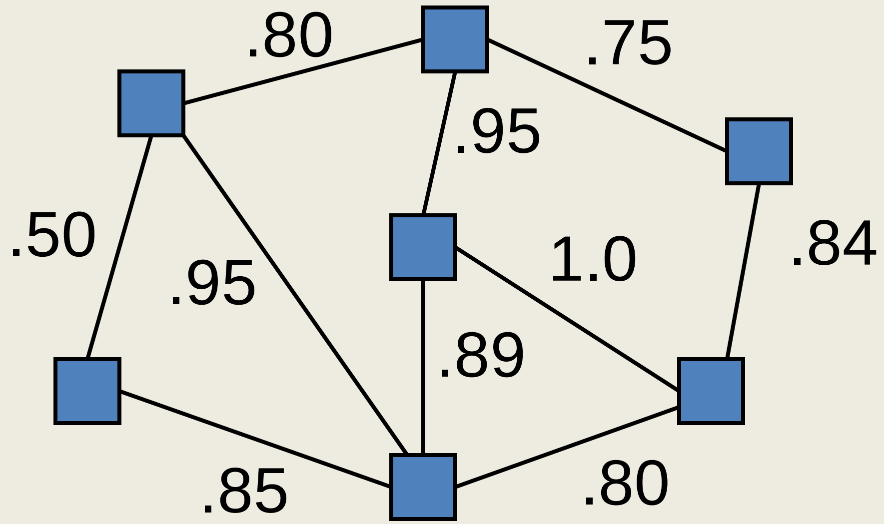
```
ST(Vertex i) {
  mark i;
  for each j adjacent to i {
    if (j is unmarked) {
      Add (i,j) to T;
      ST(j);
    }
  }
}
```

```
Main( ) {
  T = empty set;
  ST(1);
}
```

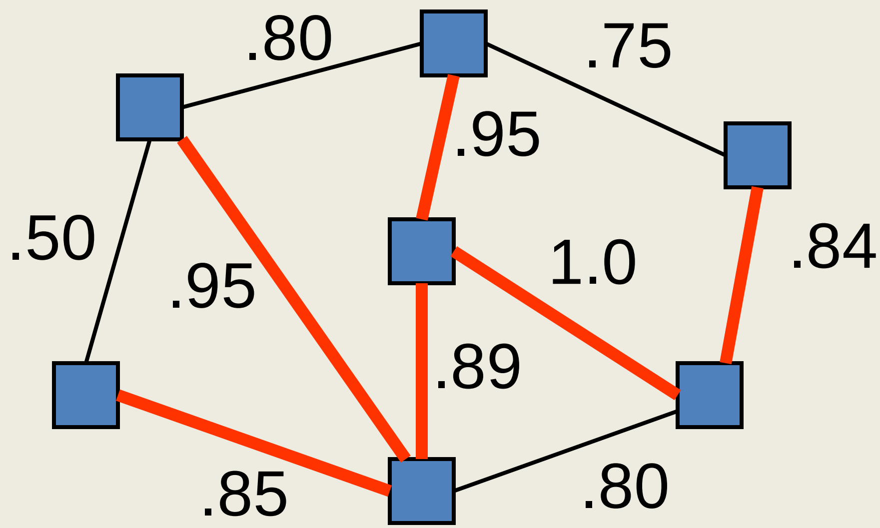
Best Spanning Tree

Finding a reliable routing subnetwork:

- edge cost = probability that it won't fail
- Find the spanning tree that is least likely to fail



Example of a Spanning Tree



Probability of success = .85 x .95 x .89 x .95 x 1.0 x .84
= .5735

Minimum Spanning Trees

Given an undirected graph **G=(V,E)**, find 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

G**'** is a **minimum spanning tree**.

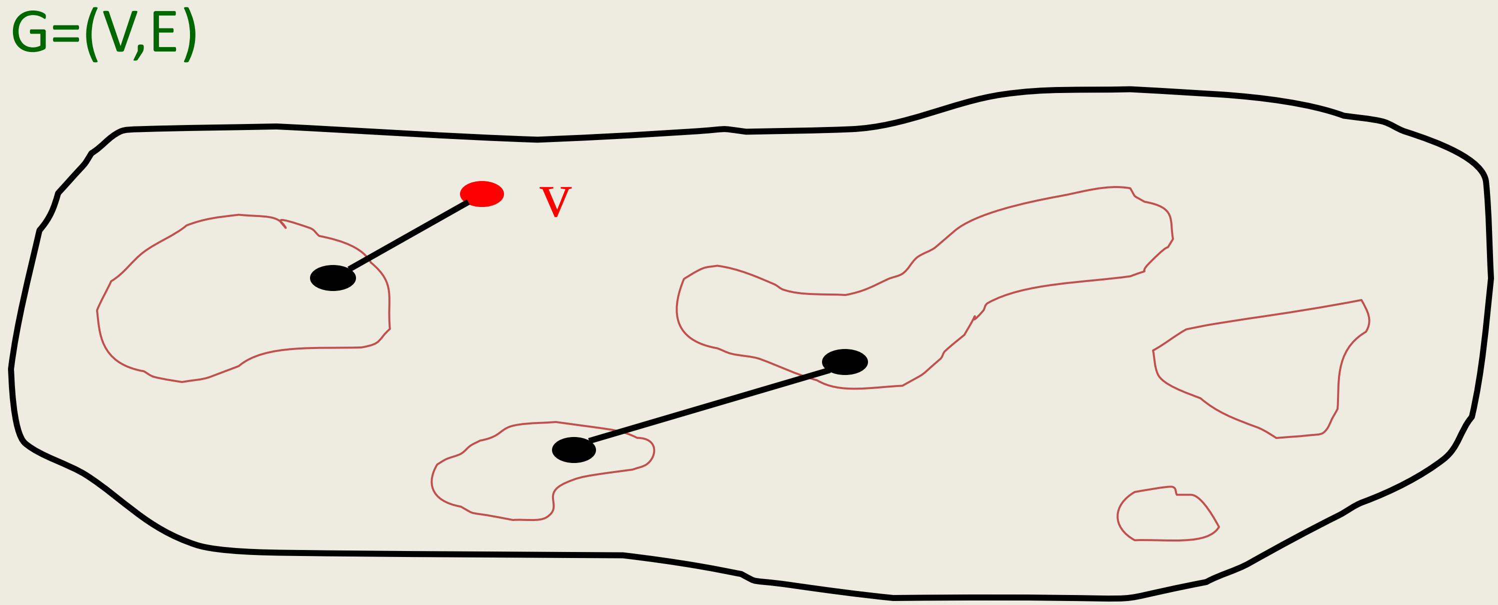
Minimum Spanning Tree Problem

- Input: Undirected Graph G = (V,E) and C(e) is the cost of edge e.
- Output: A spanning tree T with minimum total cost. Find a tree T that minimizes

$$C(T) = \sum_{e \in T} C(e)$$

Kruskal's MST Algorithm

Idea: Grow a forest out of edges that do not create a cycle.
Pick an edge with the smallest weight.



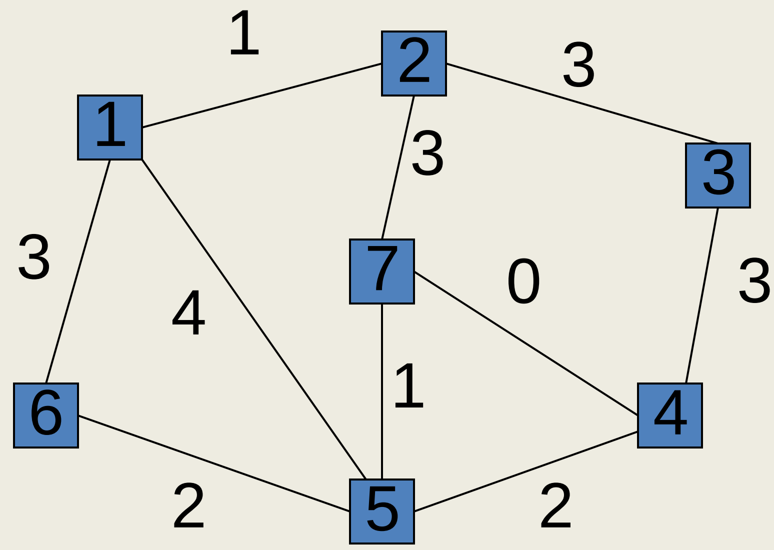
Kruskal's Algorithm for MST

An *edge-based* greedy algorithm

Builds MST by greedily adding edges

1. Initialize with
   - empty MST
   - all vertices marked unconnected
   - all edges unmarked
2. While there are still unmarked edges
   a. Pick the <u>lowest cost edge</u> **`(u,v)`** and mark it
   b. If **`u`** and **`v`** are not already connected, add **`(u,v)`** to the MST and mark **`u`** and **`v`** as connected to each other

Example of for Kruskal



(7,4) (2,1) (7,5) (5,6) (5,4) (1,6) (2,7) (2,3) (3,4) (1,5)
0 1 1 2 2 3 3 3 3 4

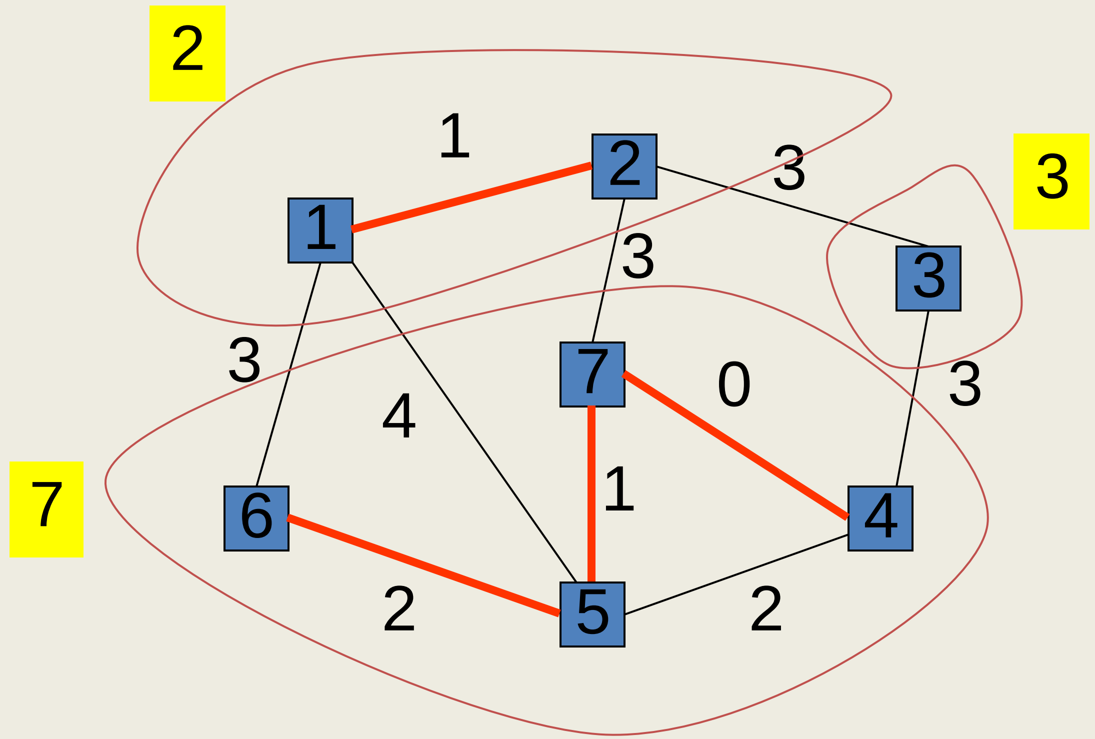
Data Structures for Kruskal

- Sorted edge list

(7,4)	(2,1)	(7,5)	(5,6)	(5,4)	(1,6)	(2,7)	(2,3)	(3,4)	(1,5)
0	1	1	2	2	3	3	3	3	4

- Disjoint Union / Find
  - Union(*a,b*) - merge the disjoint sets named by *a* and *b*
  - Find(a) returns the name of the set containing *a*
- Union / Find data structure will be presented at end of lecture

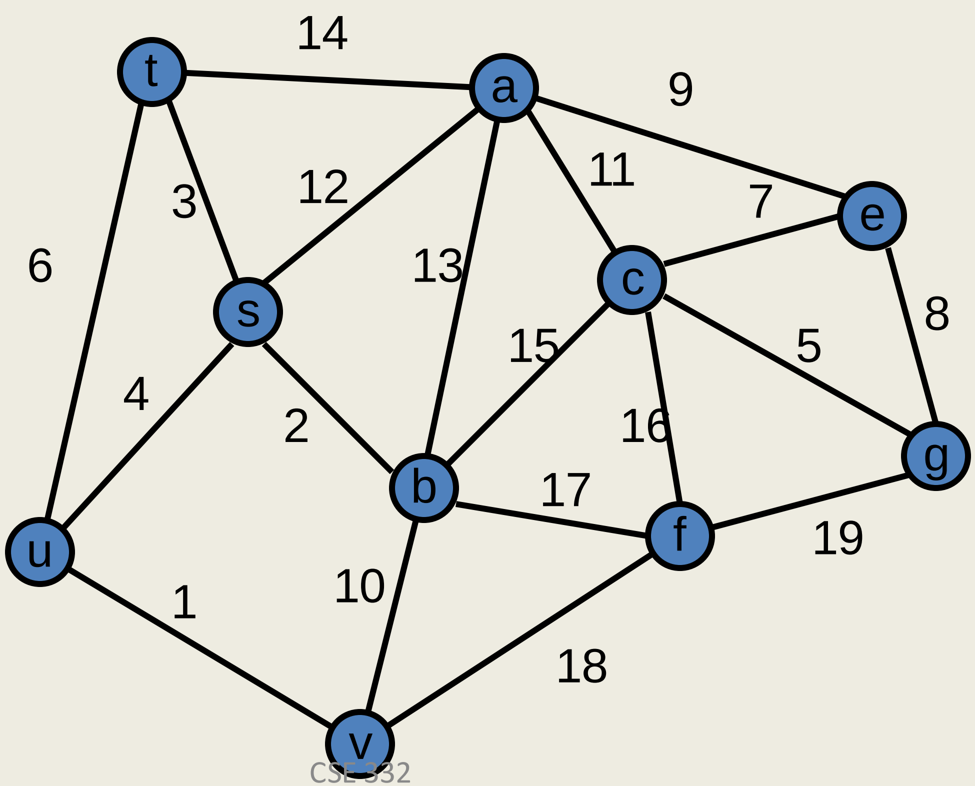
Example of DU/F



~~(7,4)~~	~~(2,1)~~	~~(7,5)~~	~~(5,6)~~	(5,4)	(1,6)	(2,7)	(2,3)	(3,4)	(1,5)
~~0~~	~~1~~	~~1~~	~~2~~	2	3	3	3	3	4

Kruskal's Algorithm

- Add the cheapest edge that joins disjoint components



Kruskal's Algorithm with DU / F

```
Sort the edges by increasing cost;
Initialize A to be empty;
for each edge (i,j) chosen in increasing order do
    u := Find(i);
    v := Find(j);
    if (u != v) then
        add (i,j) to A;
        Union(u,v);
```

This algorithm will work, but it goes through all the edges.

Is this always necessary?

Kruskal code

```
void Graph::kruskal(){
  int edgesAccepted = 0;
  DisjSet s(NUM_VERTICES);

  while (edgesAccepted < NUM_VERTICES – 1){
    e = smallest weight edge not deleted yet;
    // edge e = (u, v)
    uset = s.find(u);
    vset = s.find(v);
    if (uset != vset){
      edgesAccepted++;
      s.unionSets(uset, vset);
    }
  }
}
```

|V| ops to init. sets

|E| heap ops

2|E| finds

|V| unions

Total Cost:

Kruskal's Algorithm: Correctness

It clearly generates a spanning tree. Call it T_K.

Suppose T_K is *not* minimum:

- Pick another spanning tree T_{min} with *lower cost* than T_K
- Pick the smallest edge e_1=(u,v) in T_K that is not in T_{min}
- T_{min} already has a path p in T_{min} from u to v
  - $\Rightarrow$ Adding e_1 to T_{min} will create a cycle in T_{min}
- Pick an edge e_2 in p that Kruskal's algorithm considered *after* adding e_1 (must exist: u and v unconnected when e_1 considered)
  - $\Rightarrow$ cost(e_2) $\geq$ cost(e_1)
  - $\Rightarrow$ can replace e_2 with e_1 in T_{min} without increasing cost!
- Keep doing this until T_{min} is identical to T_K
  - $\Rightarrow$ T_K must also be minimal – contradiction!

Assume the edge costs are distinct

Correctness

Let T_k be the tree found by Kruskal, and let T be a different spanning tree, then T is not a MST

Let e_1 be the minimum cost edge of T_k not in T

If we add e_1 to T, we create a unique cycle A

Let e_2 be the maximum cost edge on A

$c(e_2) > c(e_1)$

$T' = T + \{e_1\} – \{e_2\}$ is a spanning tree

$C(T') < c(T)$

Therefor, T is not a MST

Disjoint Set ADT

- Data: set of pairwise **disjoint sets**.
- Required operations
  - **Union** – merge two sets to create their union
  - **Find** – determine which set an item appears in

Disjoint Sets and Naming

- Maintain a set of pairwise disjoint sets.
  - {3,5,7} , {4,2,8}, {9}, {1,6}
- Each set has a unique name: one of its members (for convenience)
  - {3,5,7} , {4,2,8}, {9}, {1,6}

Union / Find

- Union(x,y) – take the union of two sets named x and y
  - {3,5,7} , {4,2,8}, {9}, {1,6}
  - Union(5,1)

    {3,5,7,1,6}, {4,2,8}, {9},
- Find(x) – return the name of the set containing x.
  - {3,5,7,1,6}, {4,2,8}, {9},
  - Find(1) = 5
  - Find(4) = 8

Union/Find Trade-off

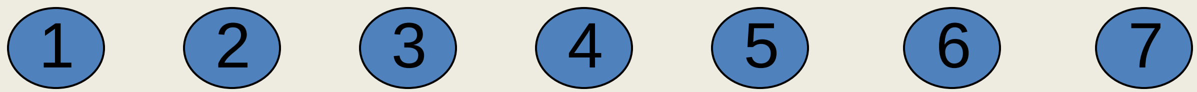
- Known result:
  - Find and Union cannot *both* be done in worst-case $O(1)$ time with any data structure.
- We will instead aim for good *amortized* complexity.
- For m operations on n elements:
  - Target complexity: $O(m)$ *i.e.* $O(1)$ amortized

Up-Tree for DS Union/Find

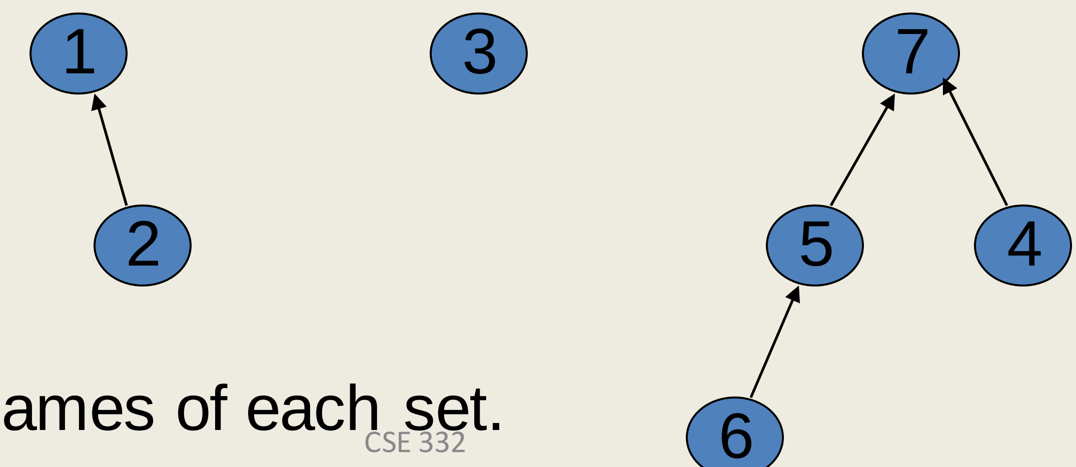
Observation: we will only traverse these trees upward from any given node to find the root.

Idea: *reverse* the pointers (make them point up from child to parent). The result is an **up-tree**.

Initial state



Intermediate state

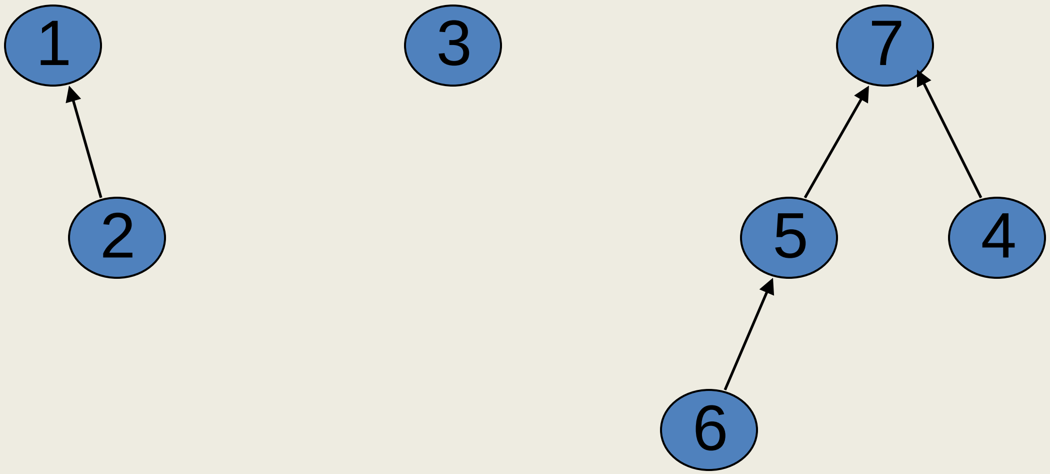


Roots are the names of each set.

Operations

Find(x) follow x to the root and return the root.

Union(i, j) - assuming i and j roots, point j to i.

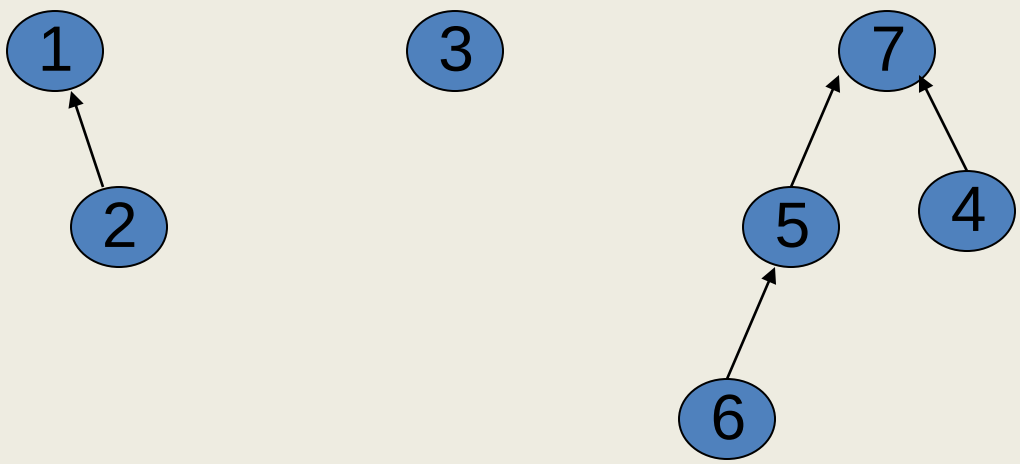


Simple Implementation

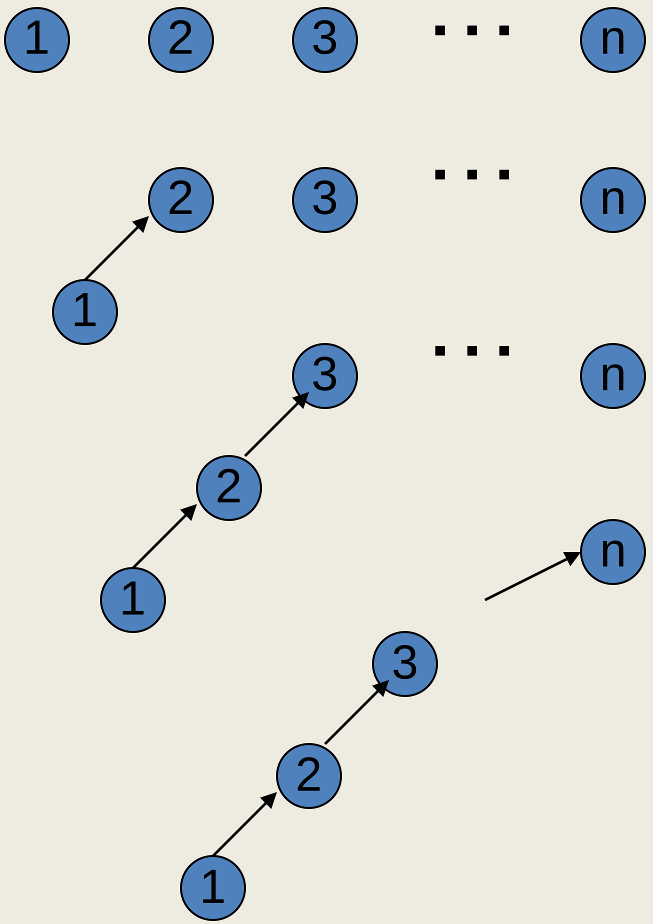
- Array of indices

	1	2	3	4	5	6	7
up	-1	1	-1	7	7	5	-1

up[x] = -1 means
x is a root.



A Bad Case



Union(1,2)

Union(2,3)

⋮

Union(n-1,n)

Find(1) n steps!!

Amortized Cost

- Cost of n Union operations followed by n Find operations is n^2
- $\Theta(n)$ per operation

Two Big Improvements

Can we do better? *Yes!*

1. Union-by-size
   - Improve Union so that *Find* only takes worst case time of $\Theta(\log n)$.

2. Path compression
   - Improve Find so that, with Union-by-size,

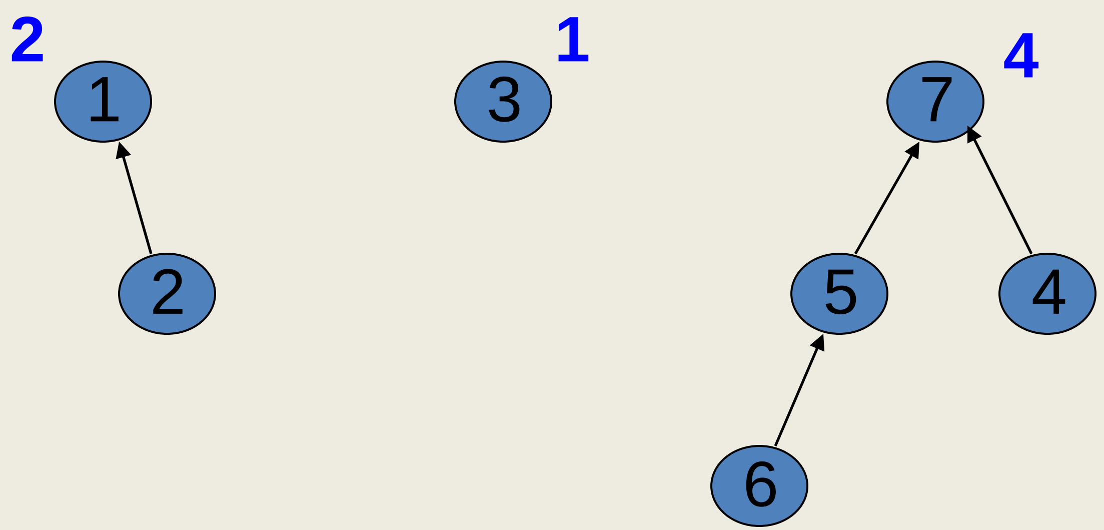
   Find takes amortized time of <u>almost</u> $\Theta(1)$.

Union-by-Size

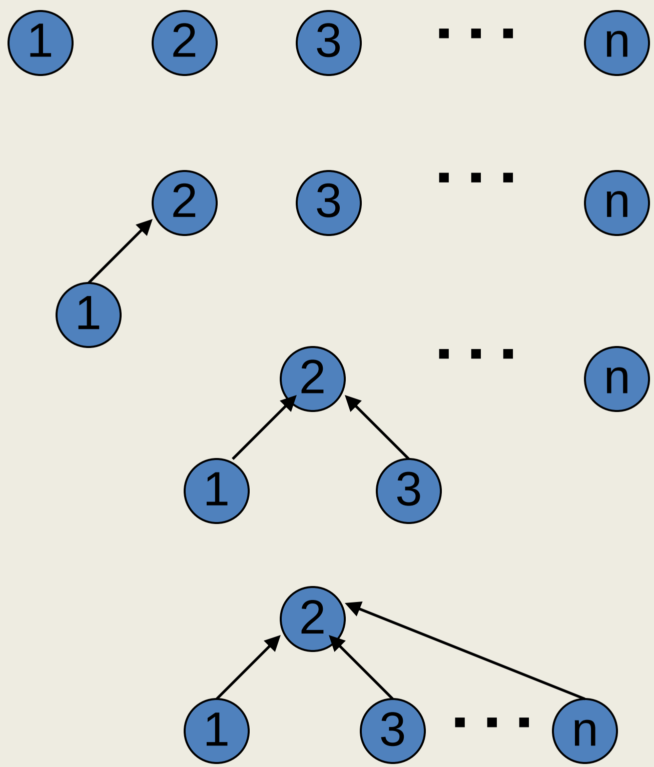
Union-by-size

- Always point the smaller tree to the root of the larger tree

S-Union(7,1)



Example Again



S-Union(1,2)

S-Union(2,3)

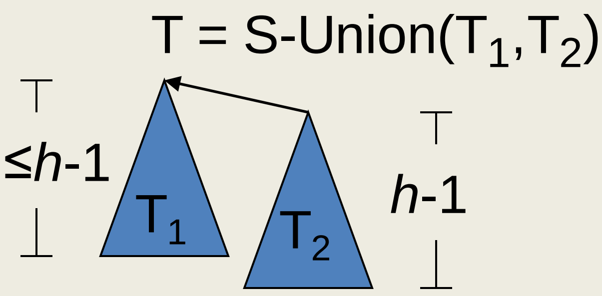
⋮

S-Union(n-1,n)

Find(1) constant time

Analysis of Union-by-Size

- Theorem: With union-by-size an up-tree of height h has size at least 2^h.
- Proof by induction
  - Base case: $h = 0$. The up-tree has one node, $2^0 = 1$
  - Inductive hypothesis: Assume true for h-1
  - Observation: tree gets taller only as a result of a union.



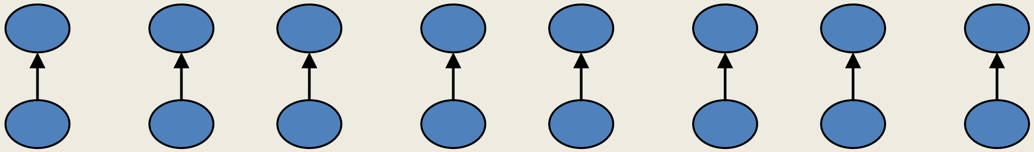
Analysis of Union-by-Size

- What is worst case complexity of Find(x) in an up-tree forest of *n* nodes?

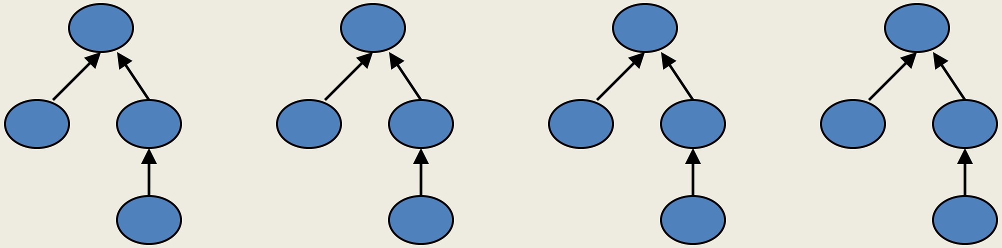
- (Amortized complexity is no better.)

Worst Case for Union-by-Size

n/2 Unions-by-size

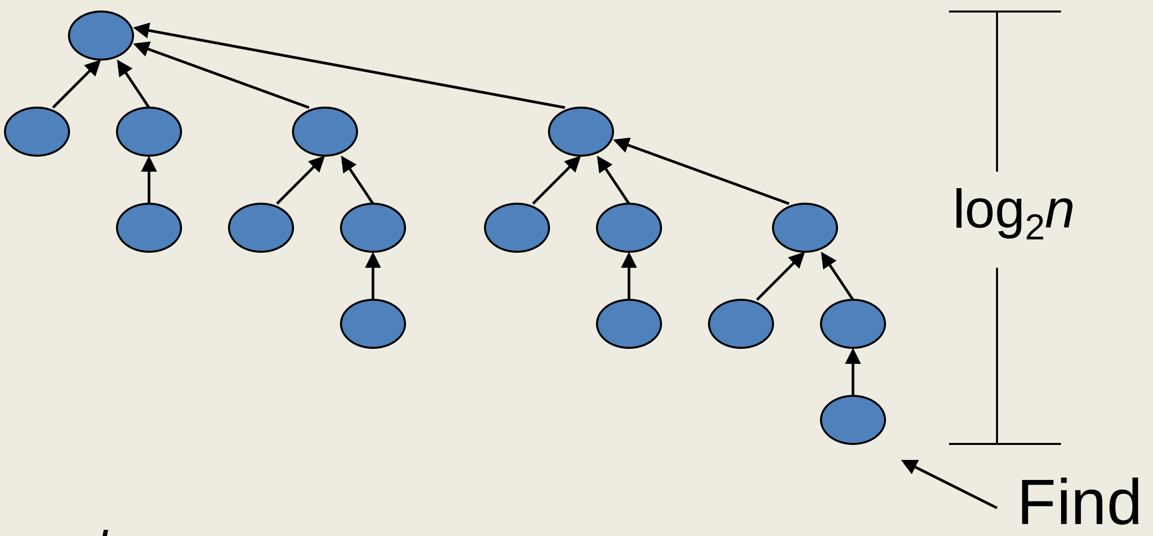


n/4 Unions-by-size



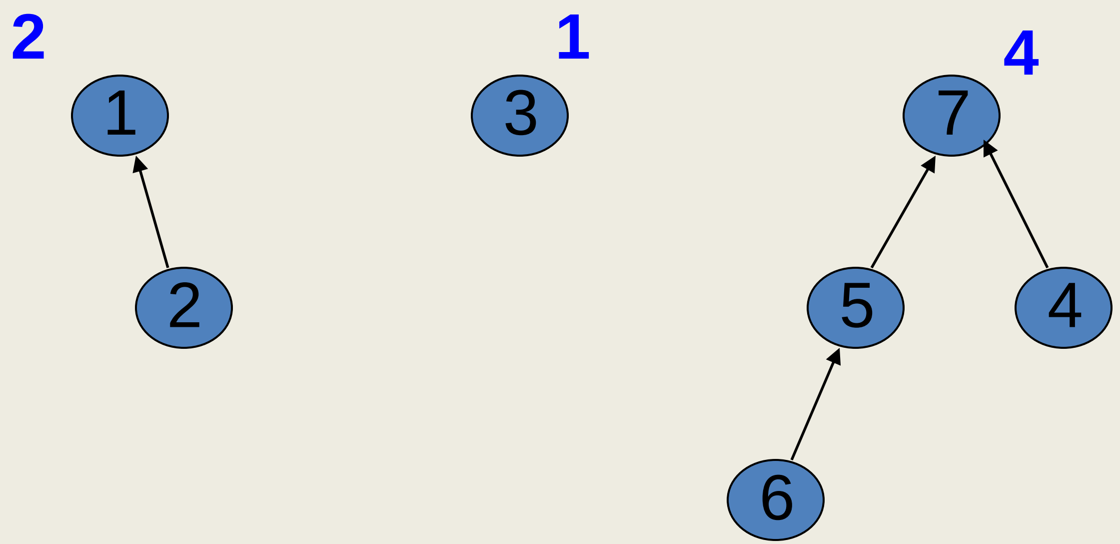
Example of Worst Cast (cont')

After n -1 = n/2 + n/4 + …+ 1 Unions-by-size



If there are $n = 2^k$ nodes then the longest path from leaf to root has length k.

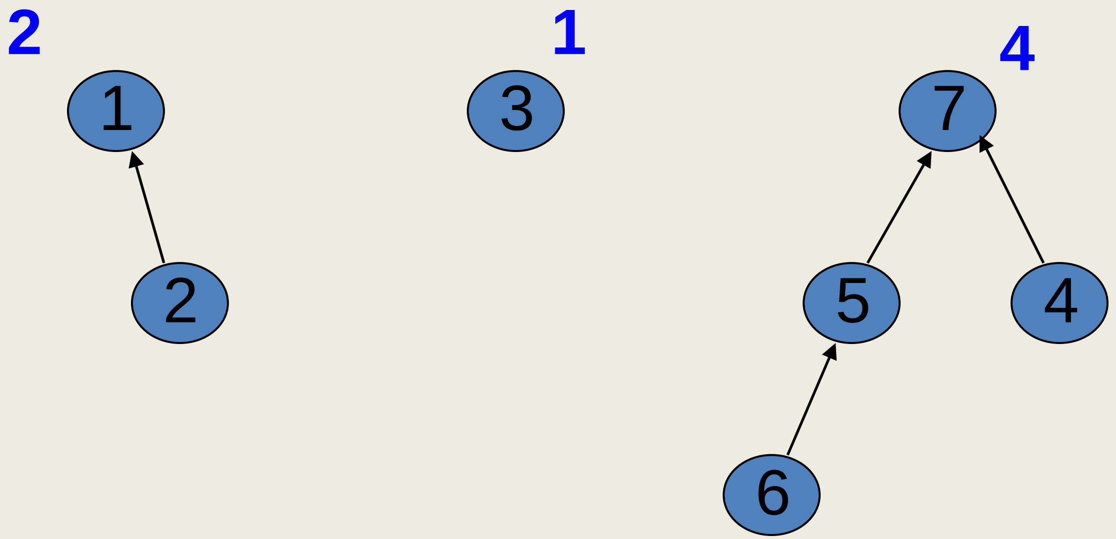
Array Implementation



Can store separate size array:

	1	2	3	4	5	6	7
up	-1	1	-1	7	7	5	-1
size	2		1				4

Elegant Array Implementation



Better, store sizes in the up array:

	1	2	3	4	5	6	7
up	-2	1	-1	7	7	5	-4

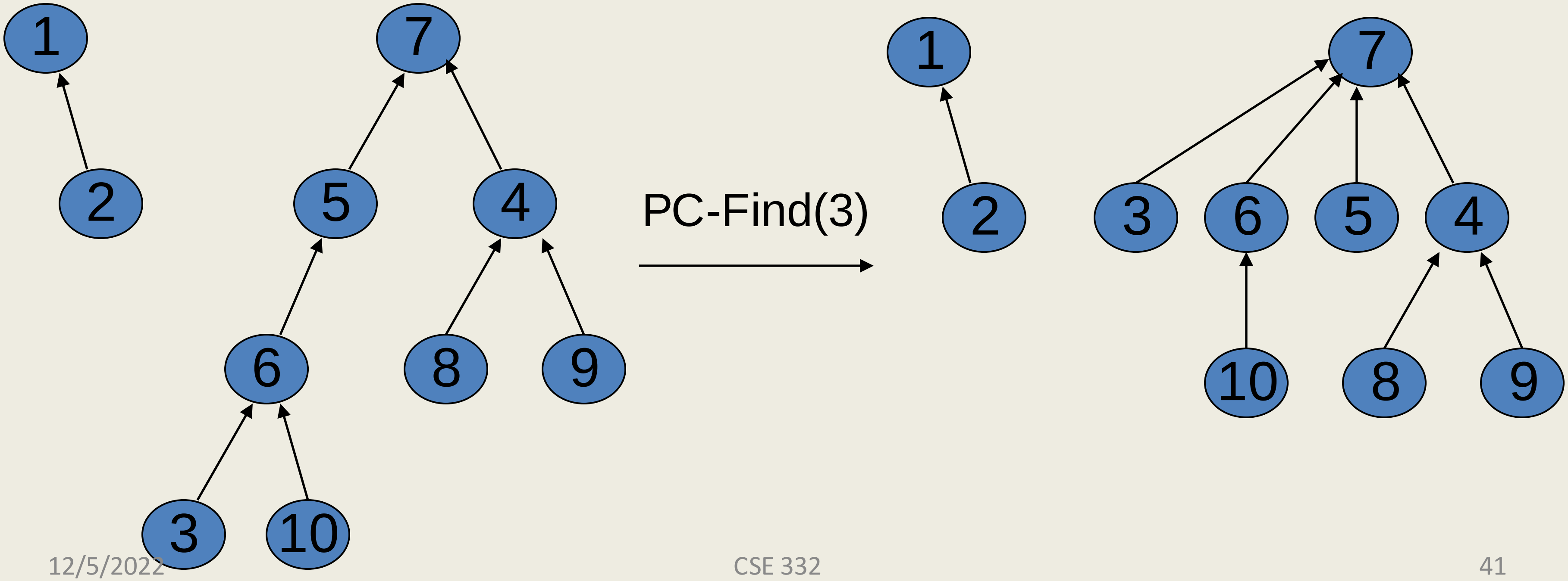
Negative up-values correspond to sizes of roots.

Code for Union-by-Size

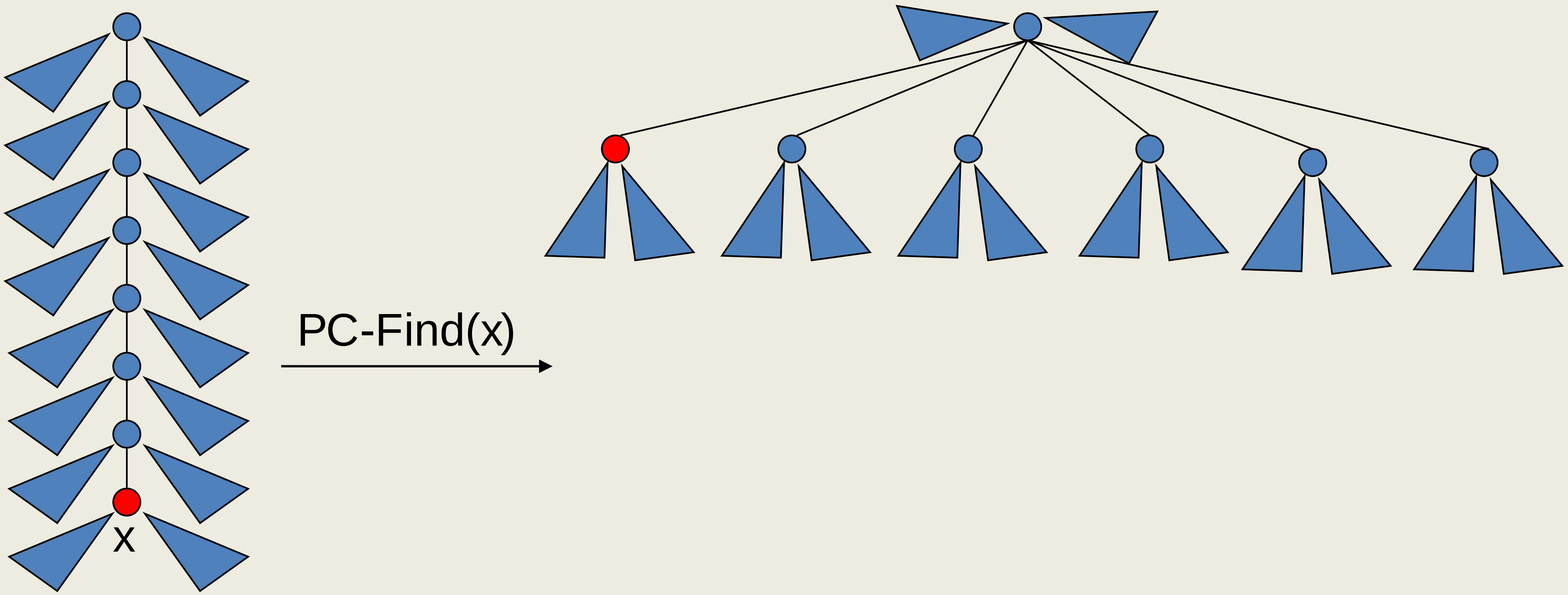
```
S-Union(i,j){
  // Collect sizes
  si = -up[i];
  sj = -up[j];

  // verify i and j are roots
  assert(si >=0 && sj >=0)
  // point smaller sized tree to
  // root of larger, update size
  if (si < sj) {
    up[i] = j;
    up[j] = -(si + sj);
  else {
    up[j] = i;
    up[i] = -(si + sj);
  }
}
```

Path Compression

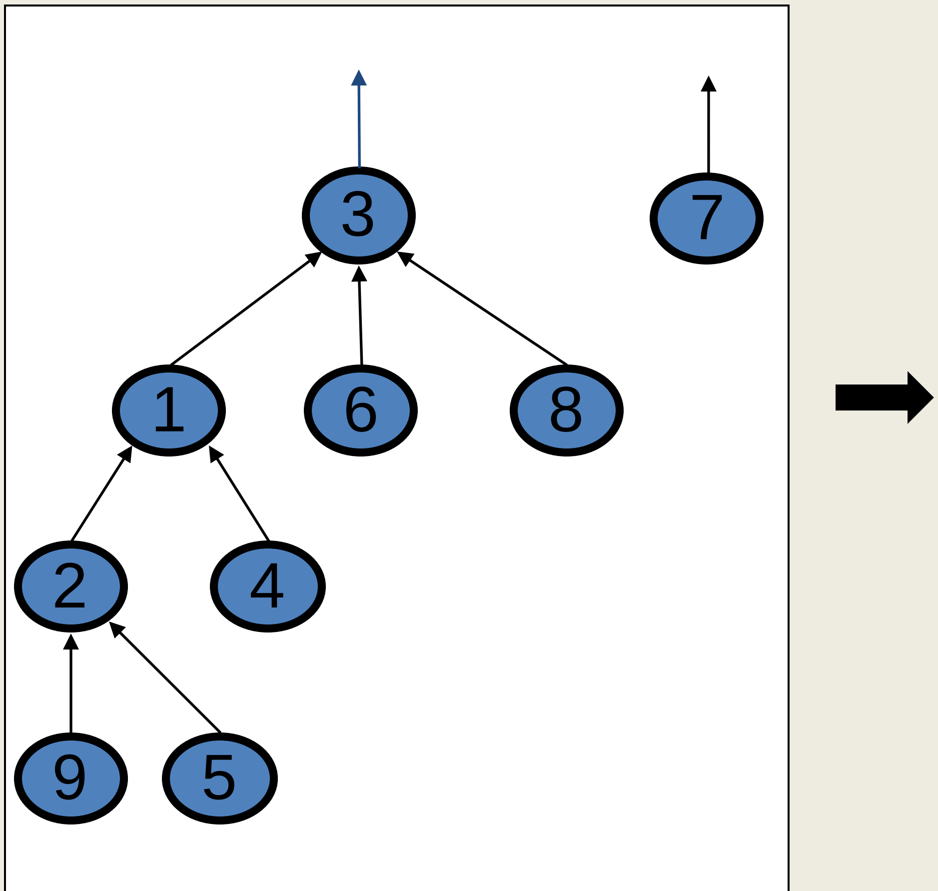
- To improve the amortized complexity, we'll borrow an idea from splay trees:
  - When going up the tree, *improve nodes on the path*!
- On a Find operation point all the nodes on the search path directly to the root. This is called "path compression."



Self-Adjustment Works



Draw the result of Find(5):



Code for Path Compression Find

```
PC-Find(i) {
  //find root
  j = i;
  while (up[j] >= 0) {
    j = up[j];
  root = j;

  //compress path
  if (i != root) {
    parent = up[i];
    while (parent != root) {
      up[i] = root;
      i = parent;
      parent = up[parent];
    }
  }
  return(root)
}
```

Complexity of Union-by-Size + Path Compression

- Worst case time complexity for…
  - …a single Union-by-size is:
  - …a single PC-Find is:

- Time complexity for $m \geq n$ operations on n elements has been shown to be O(m log* n).

  [See Weiss for proof.]
  - Amortized complexity is then O(log* n)
  - What is log* ?

log* n

log* n = number of times you need to apply log to bring value down to at most 1

log* 2 = 1

log* 4 = log* 2^2 = 2

log* 16 = log* 2^{2^2} = 3 (log log log 16 = 1)

log* 65536 = log* $2^{2^{2^2}}$ = 4 (log log log log 65536 = 1)

log* 2^{65536} = …………… ≈ log* ($2 \times 10^{19,728}$) = 5

log * $n \leq 5$ for all reasonable n.

The Tight Bound

In fact, Tarjan showed the time complexity for $m \geq n$ operations on n elements is:

$$\Theta(m\ \alpha(m, n))$$

Amortized complexity is then $\Theta(\alpha(m, n))$.

What is $\alpha(m, n)$?

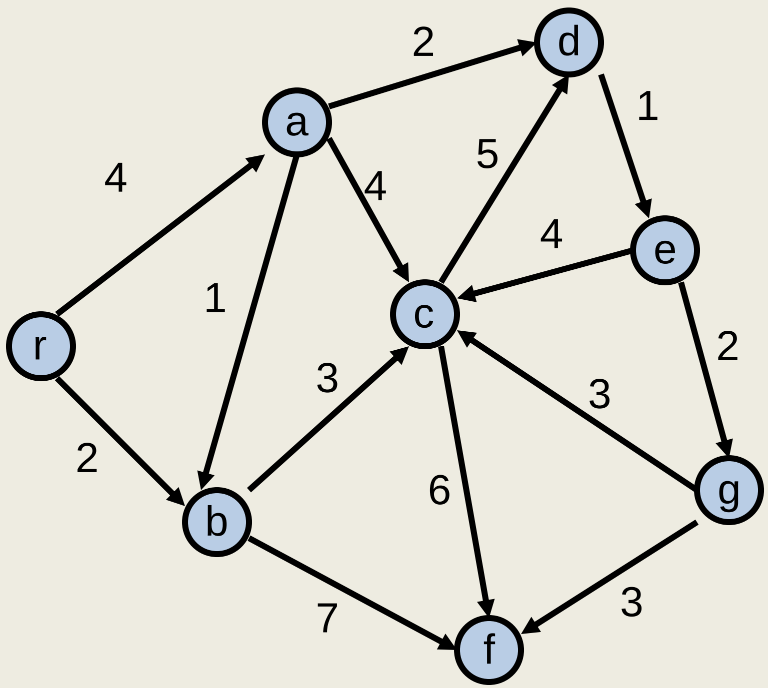
– Inverse of Ackermann's function.

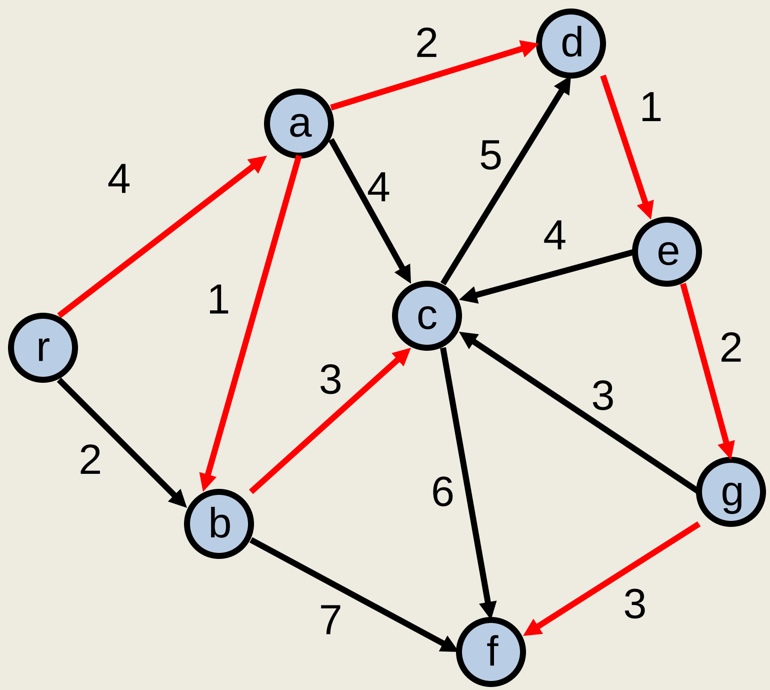
– For reasonable values of *m, n*, grows even slower than log * *n.* So, it's even "more constant."

Proof is beyond scope of this class. A simple algorithm can lead to incredibly hardcore analysis!

What about the minimum spanning tree of a directed graph?

- Must specify the root r
- Branching: Out tree with root r





Assume all vertices reachable from r

Also called an arborescence