Lecture 23: Minimum Spanning Trees

CSE 373: Data Structures and Algorithms

Administriva

Midterm solutions are on the exams section of the webpage.

Project 3 is due today

Proejct 4 (the last project) out soon (probably sometime tomorrow)
- The last project!
- Due Monday the 19th

Exercise 4 due Friday.

Dijkstra’s Runtime

Just like when we analyzed BFS, don’t just work inside out; try to figure out how many times each line will be executed.

```
Dijkstra(Graph G, Vertex source)
    for (Vertex v : G.getVertices()) { v.dist = INFINITY; }
    G.getVertex(source).dist = 0;
    initialize MPQ as a Min Priority Queue, add source
    while(MPQ is not empty){
        u = MPQ.removeMin(); +logV
        for (Edge e : u.getEdges(u)){
            oldDist = v.dist; newDist = u.dist+weight(u,v)
            if(newDist < oldDist){
                v.dist = newDist
                v.predecessor = u
                if(oldDist == INFINITY) { MPQ.insert(v) } +logV
                else { MPQ.updatePriority(v, newDist) }
            }
        }
    }
```

This actually doesn’t run m times for every iteration of the outer loop. It actually will run m times in total; if every vertex is only removed from the priority queue (processed) once, then we examine each edge once. Each line inside this foreach gets multiplied by a single E instead of E * V.

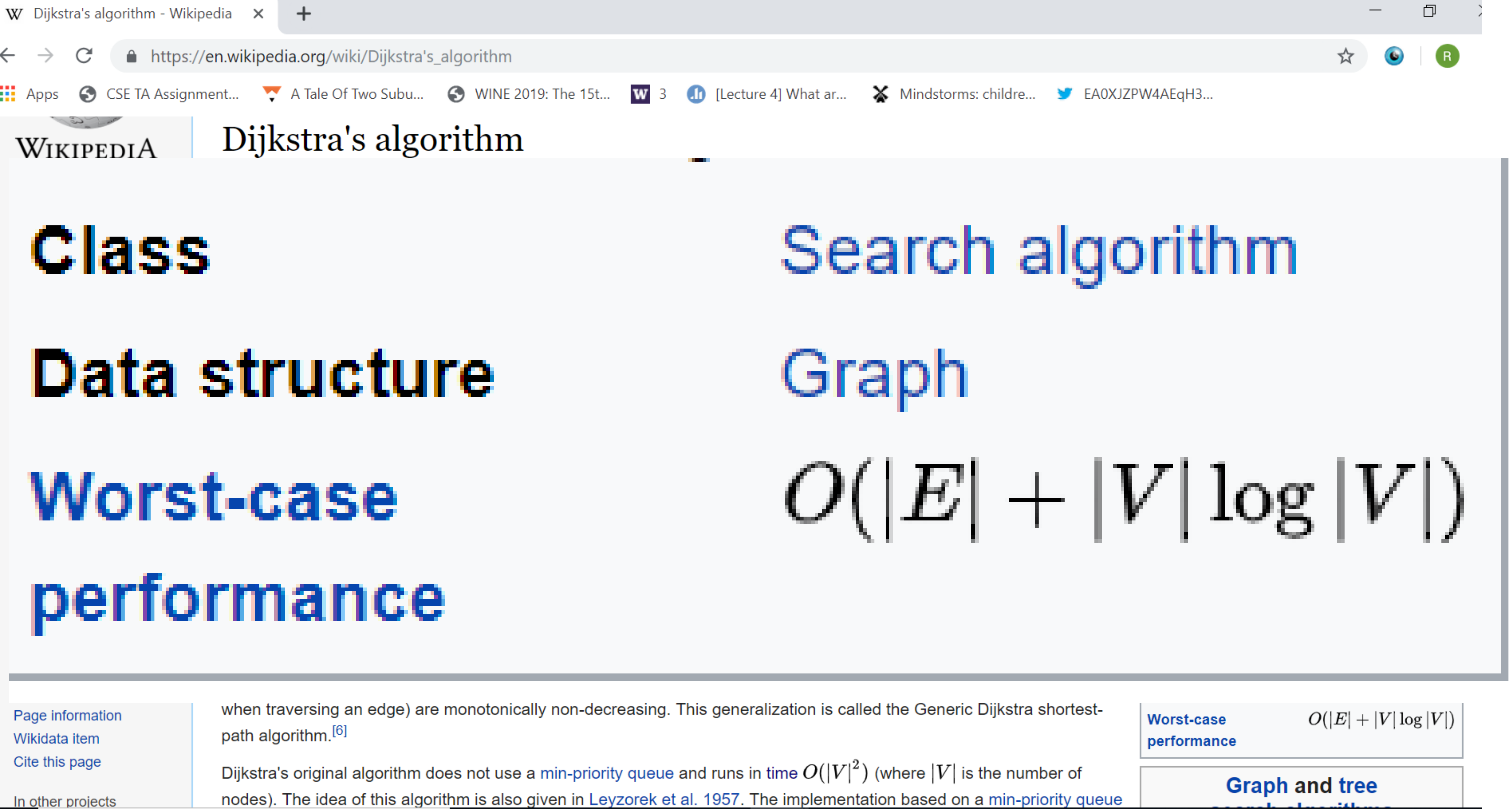
Θ-Bound = **Θ**(n log n + m log n)

Dijkstra's Wrap-up

The details of the implementation depend on what data structures you have available.

Your implementation in the programming project will be different in a few spots.

Our running time is $\Theta(E \log V + V \log V)$ i.e. $\Theta(m \log n + n \log n)$.



Dijkstra's algorithm

Class	Search algorithm
Data structure	Graph
Worst-case performance	$O(\|E\|+\|V\|\log\|V\|)$

when traversing an edge) are monotonically non-decreasing. This generalization is called the Generic Dijkstra shortest-path algorithm.[6]

Dijkstra's original algorithm does not use a min-priority queue and runs in time $O(|V|^2)$ (where $|V|$ is the number of nodes). The idea of this algorithm is also given in Leyzorek et al. 1957. The implementation based on a min-priority queue

Worst-case performance	$O(\|E\|+\|V\|\log\|V\|)$

Graph and tree

Dijkstra’s Wrap-up

The details of the implementation depend on what data structures you have available.

Your implementation in the programming project will be different in a few spots.

Our running time is $\Theta(E \log V + V \log V)$ i.e. $\Theta(m \log n + n \log n)$.

If you go to Wikipedia right now, they say it’s $O(E + V \log V)$

They’re using a Fibonacci heap instead of a binary heap.

$\Theta(E \log V + V \log V)$ is the right running time for this class.

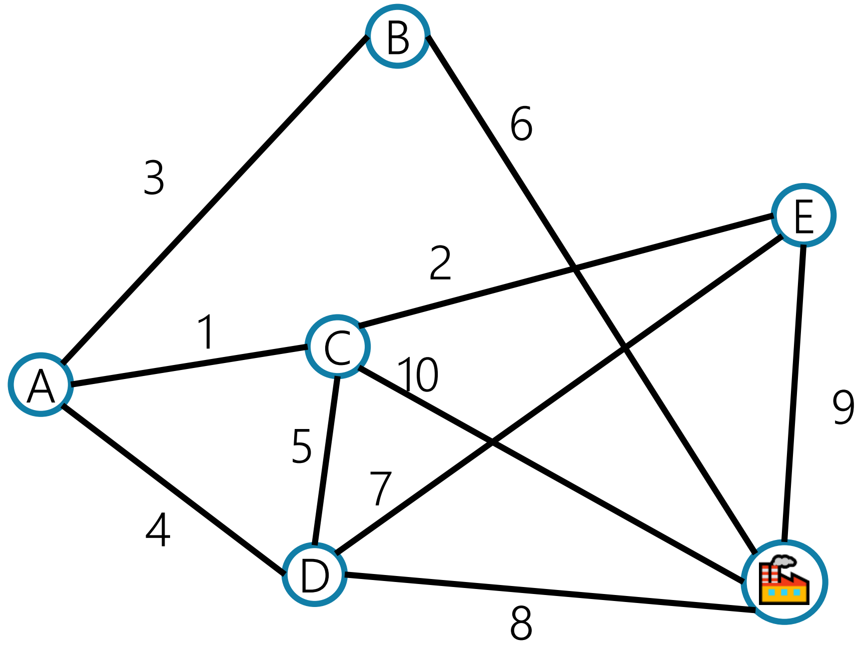
Shortest path summary:

- BFS works great (and fast -- $\Theta(m + n)$ time) if graph is unweighted.
- Dijkstra’s works for weighted graphs with no negative edges, but a bit slower $\Theta(m \log n + n \log n)$
- Reductions!

Minimum Spanning Trees

Minimum Spanning Trees

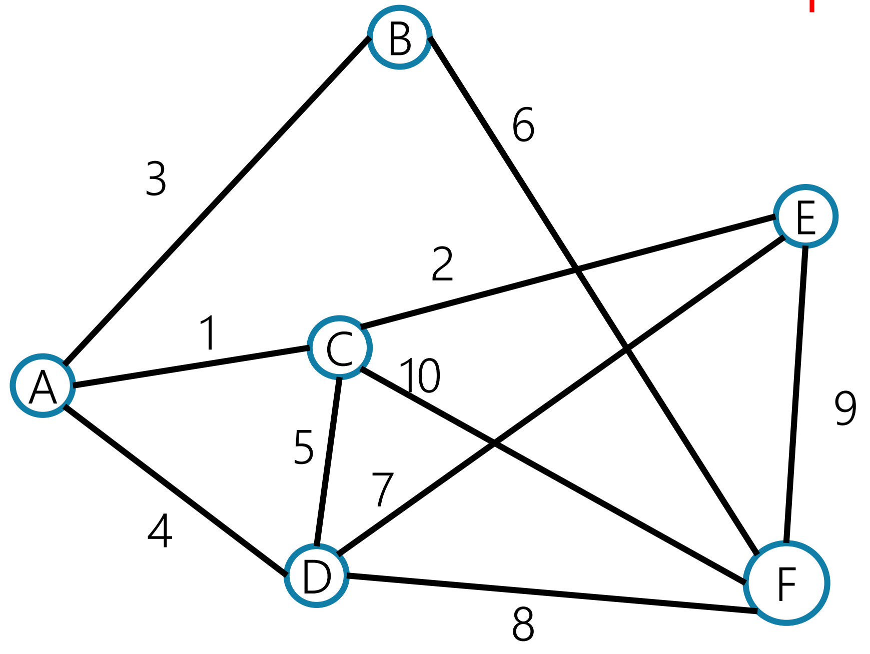
It’s the 1920’s. Your friend at the electric company needs to choose where to build wires to connect all these cities to the plant.



She knows how much it would cost to lay electric wires between any pair of cities, and wants the cheapest way to make sure electricity from the plant to every city.

Minimum Spanning Trees

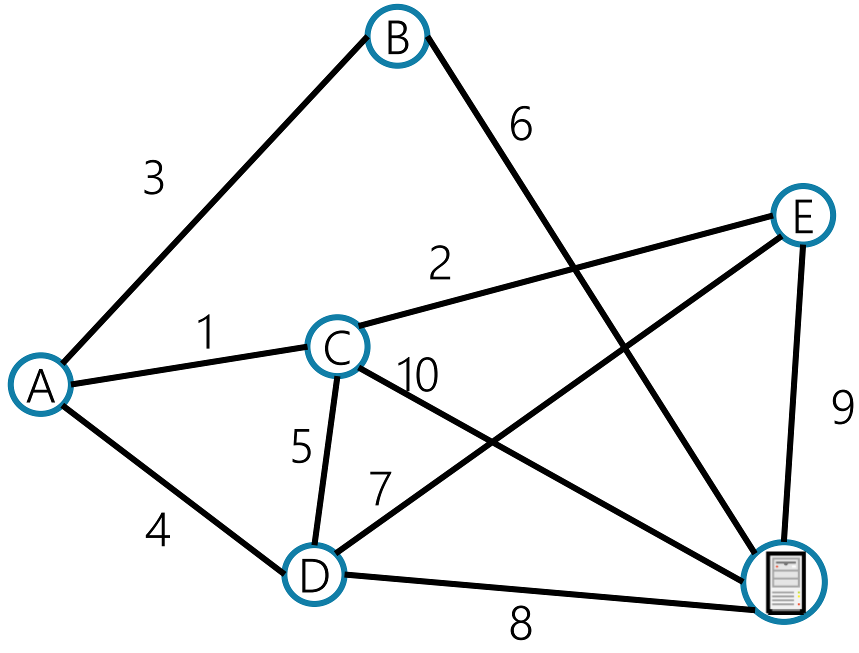
It's the 1950's Your boss at the phone company needs to choose where to build wires to connect all these phones to each other.



She knows how much it would cost to lay phone wires between any pair of locations, and wants the cheapest way to make sure Everyone can call everyone else.

Minimum Spanning Trees

It's today Your friend at the ISP needs to choose where to build wires to connect all these cities to the Internet.



She knows how much it would cost to lay cable between any pair of locations, and wants the cheapest way to make sure Everyone can reach the server

Minimum Spanning Trees

What do we need? A set of edges such that:

- Every vertex touches at least one of the edges. (the edges **span** the graph)
- The graph on just those edges is **connected**.
- The minimum weight set of edges that meet those conditions.

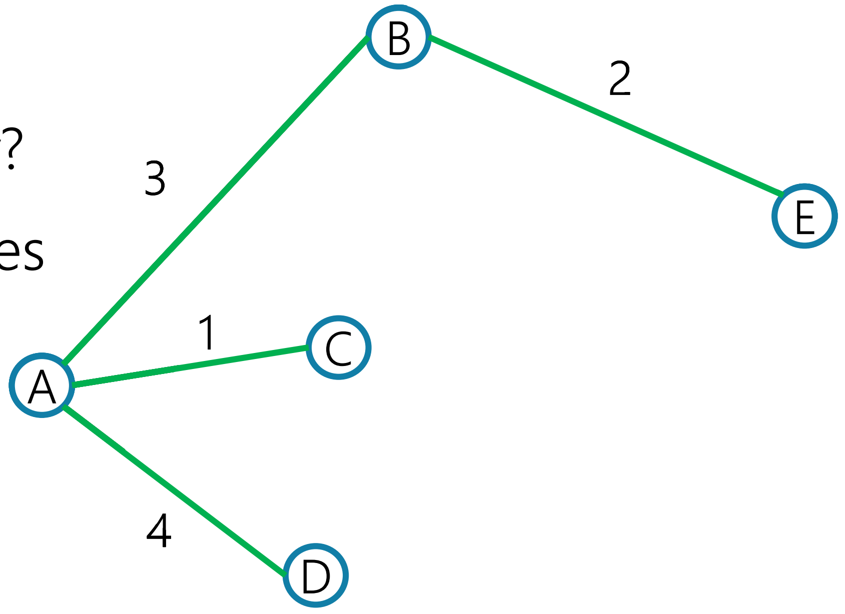
Notice we do not need a directed graph!

Assume all edge weights are positive.

Claim: The set of edges we pick never has a cycle. Why?

MST is the exact number of edges to connect all vertices

- taking away 1 edge breaks connectiveness
- adding 1 edge makes a cycle
- contains exactly V – 1 edges



Aside: Trees

Our BSTs had:

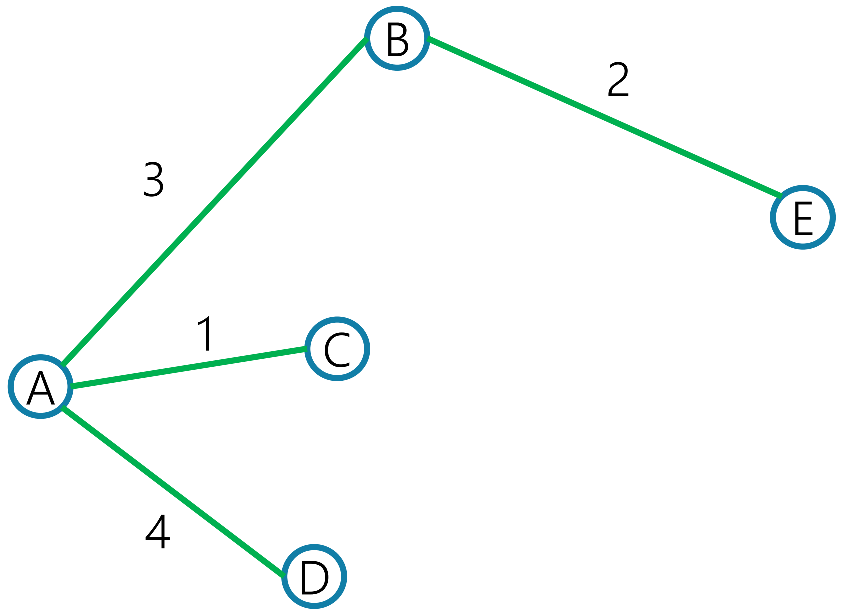
- A root
- Left and/or right children
- Connected and no cycles

Our heaps had:

- A root
- Varying numbers of children
- Connected and no cycles

On graphs our tees:

- Don't need a root (the vertices aren't ordered, and we can start BFS from anywhere)
- Varying numbers of children
- Connected and no cycles



Tree (when talking about graphs)

An undirected, connected acyclic graph.

MST Problem

What do we need? A set of edges such that:

- Every vertex touches at least one of the edges. (the edges **span** the graph)
- The graph on just those edges is **connected**.
- The minimum weight set of edges that meet those conditions.

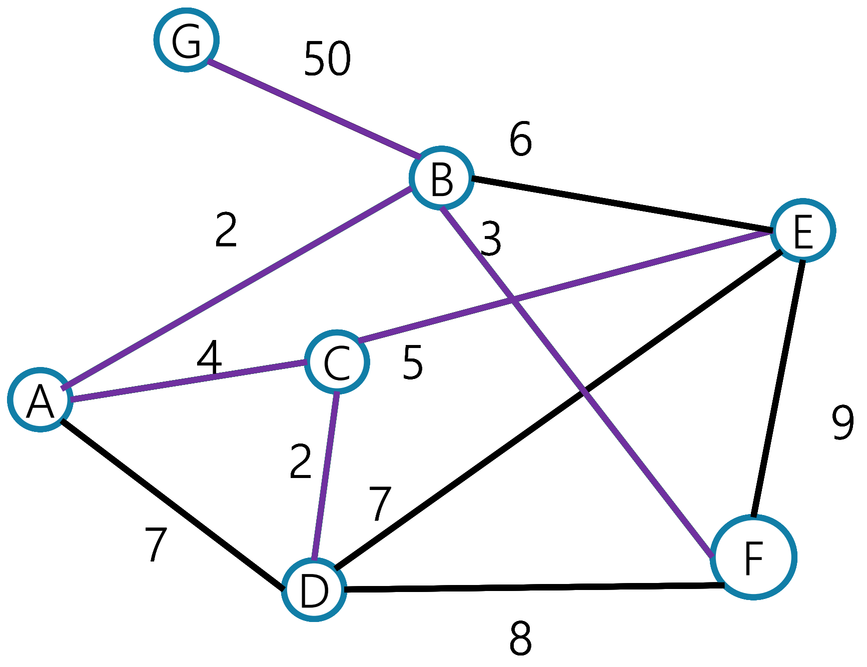
Our goal is a tree!

Minimum Spanning Tree Problem

Given: an undirected, weighted graph G
Find: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges.

Example

Try to find an MST of this graph



Finding an MST

pollEV.com/cse373su19

Which of these sounds like more likely to work?

Here are two ideas for finding an MST:

Think vertex-by-vertex

- Maintain a tree over a set of vertices
- Have each vertex remember the cheapest edge that could connect it to that set.
- At every step, connect the vertex that can be connected the cheapest.

Think edge-by-edge

- Sort edges by weight. In increasing order:
- add it if it connects new things to each other (don't add it if it would create a cycle)

Both ideas work!!

Kruskal's Algorithm

Let's start with the edge-by-edge version.

We'll need one more vocab word:

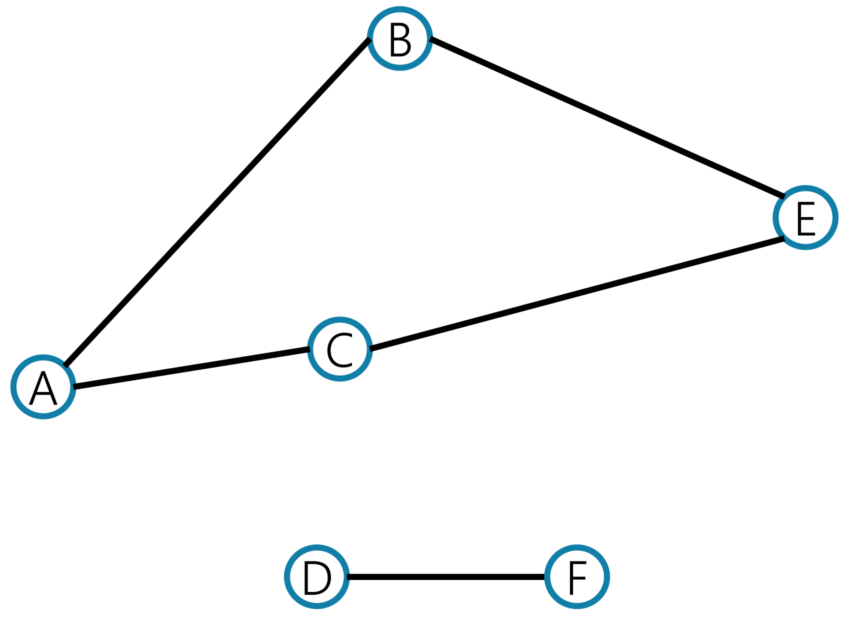
A **connected component** (or just **"component"**) is a "piece" of an undirected graph.

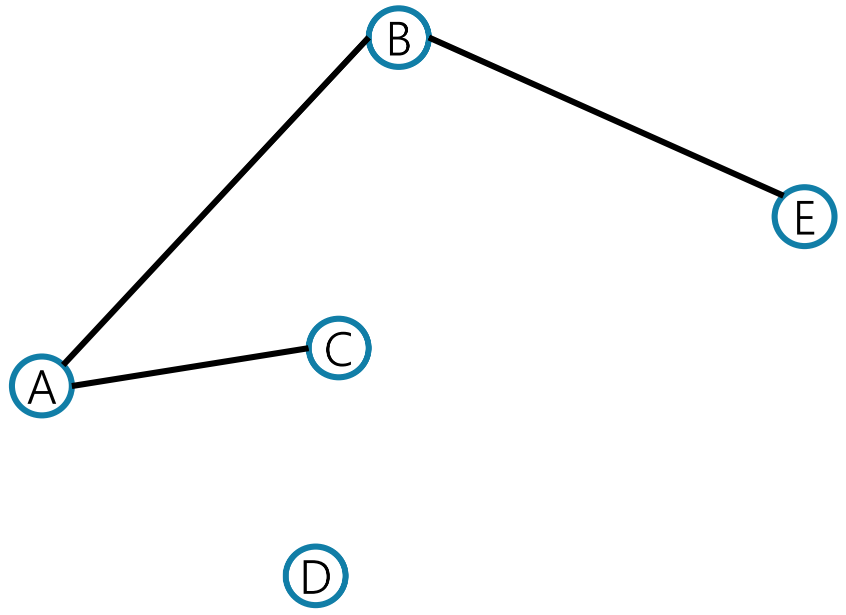
Connected component

A set S of vertices is a connected component (of an undirected graph) if:

1. It is connected, i.e. for all vertices u, v in S: there is a walk from u to v
2. It is maximal:
   - Either it's the entire set of vertices, or
   - For **every** vertex u that's not in S, $S \cup \{u\}$ is not connected.

Find the connected components



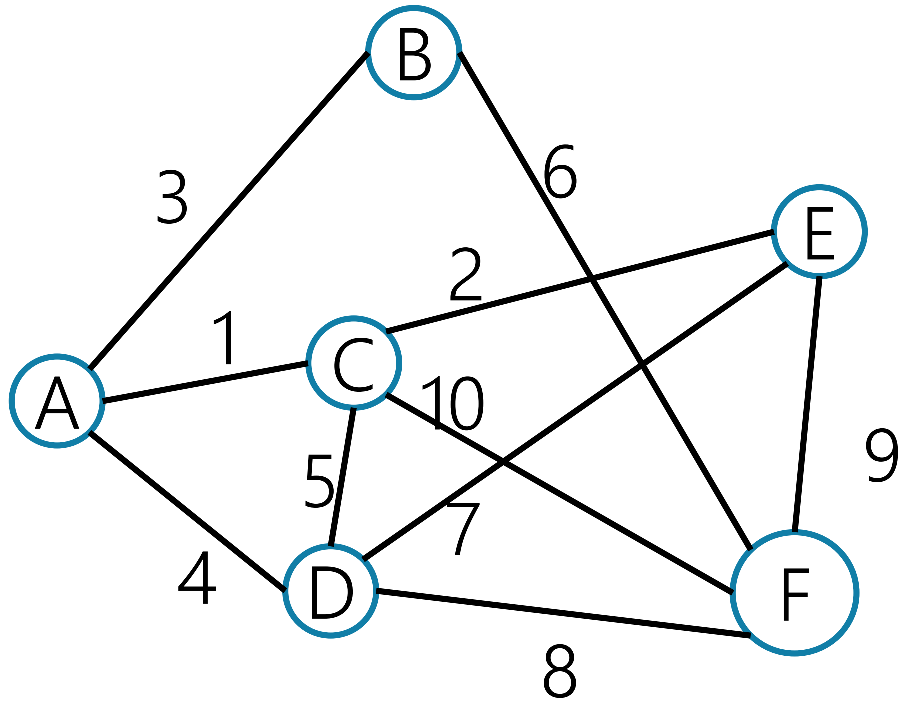


Kruskal’s Algorithm

```
KruskalMST(Graph G)
   initialize each vertex to be its own component
   sort the edges by weight
   foreach(edge (u, v) in sorted order){
      if(u and v are in different components){
         add (u,v) to the MST
         Update u and v to be in the same component
      }
   }
```

Try It Out

```
KruskalMST(Graph G)
    initialize each vertex to be its own component
     sort the edges by weight
     foreach(edge (u, v) in sorted order){
          if(u and v are in different components){
               add (u,v) to the MST
               Update u and v to be in the same component
          }
     }
```

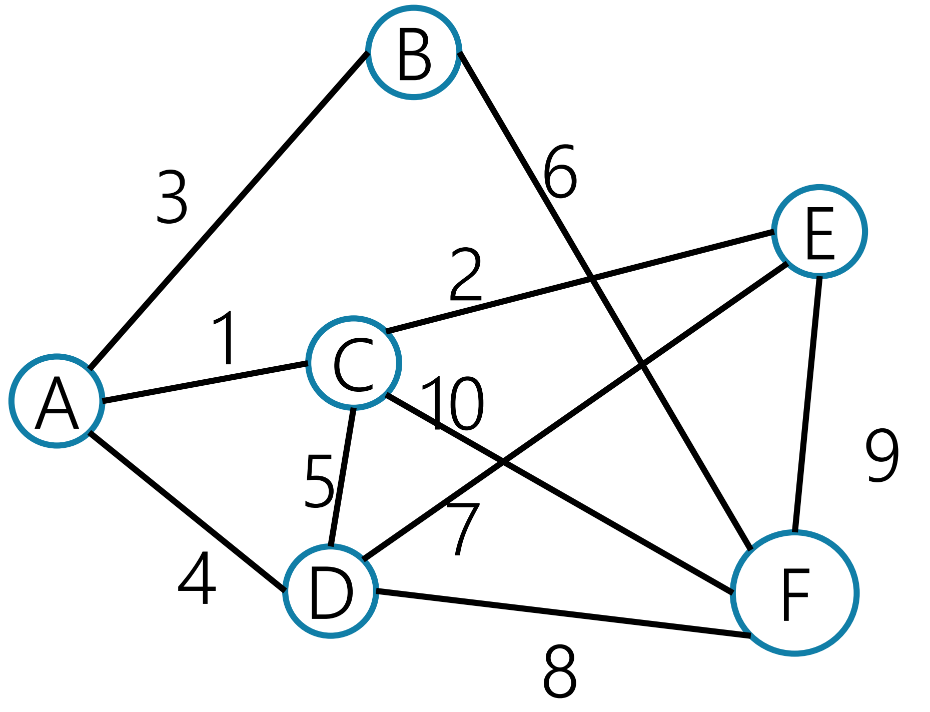


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

Edge (cont.)	Inc?	Reason
(B,F)		
(D,E)		
(D,F)		
(E,F)		
(C,F)		

Try It Out

```
KruskalMST(Graph G)
    initialize each vertex to be its own component
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(u and v are in different components){
            add (u,v) to the MST
            Update u and v to be in the same component
        }
    }
```



Edge	Include?	Reason
(A,C)	Yes	
(C,E)	Yes	
(A,B)	Yes	
(A,D)	Yes	
(C,D)	No	Cycle A,C,D,A

Edge (cont.)	Inc?	Reason
(B,F)	Yes	
(D,E)	No	Cycle A,C,E,D,A
(D,F)	No	Cycle A,D,F,B,A
(E,F)	No	Cycle A,C,E,F,D,A
(C,F)	No	Cycle C,A,B,F,C

Kruskal’s Implementation

Some lines of code there were a little sketchy.

```
> initialize each vertex to be its own component
> Update u and v to be in the same component
```

Last time we solved sketchy lines of code with a data structure.

Can we use one of our data structures?

A new ADT

We need a new ADT!

Disjoint-Sets (aka Union-Find) ADT

state

Family of Sets

- **sets are disjoint:** No element appears in more than one set
- No required order (neither within sets, nor between sets)
- Each set has a representative (use one of its members as a name)

behavior

makeSet(value) – creates a new set where the only member is the value. Picks value as the representative

findSet(value) – looks up the representative of the set containing value, returns the representative of that set

union(x, y) – looks up set containing x and set containing y, combines two sets into one. All of the values of one set are added to the other, and the now empty set goes away. Chooses a representative for combined set.

Disjoint sets implementation

There's only one common implementation of the Disjoint sets/Union-find ADT.

We'll call it "forest of up-trees" or just "up-trees"

It's very common to conflate the ADT with the data structure

- Because the standard implementation is basically the "only one"
- Don't conflate them!

We're going to slowly design/optimize the implementation over the next lecture-plus.

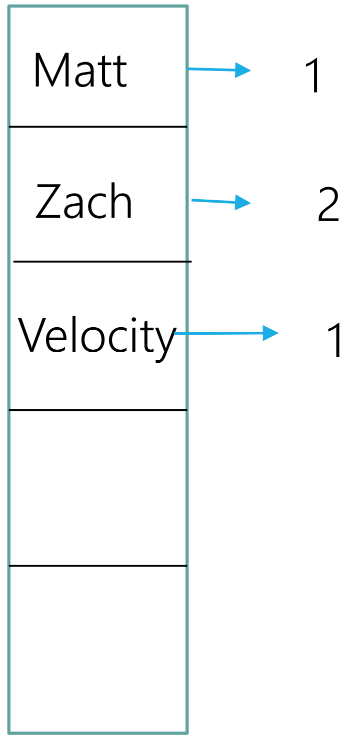
It'll take us a while, but it'll be a great review of some key ideas we've learned this quarter.

Implementing Union-Find

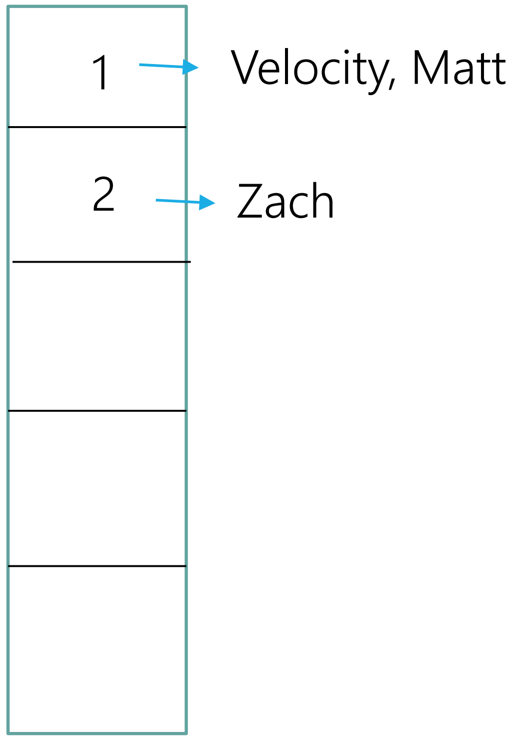
Implementing Disjoint-Sets with Dictionaries

Let's start with a not-great implementation to see why we really need a new data structure.

Approach 1: dictionary of value -> set ID/representative



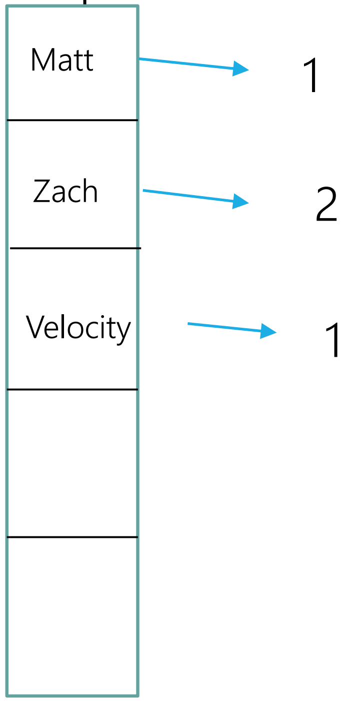
Approach 2: dictionary of ID/representative of set -> all the values in that set



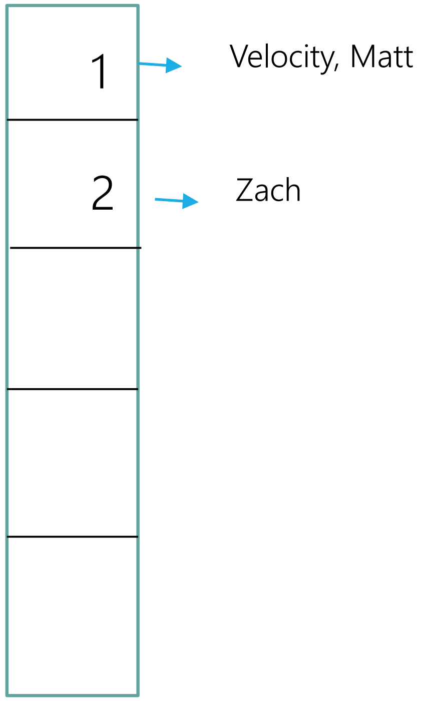
Exercise (2 mins)

Calculate the worst case Big-Θ runtimes for each of the methods (makeSet, findSet, union) for both approaches.

Approach 1: dictionary of value -> set ID/representative



Approach 2: dictionary of ID/representative of set -> all the values in that set



	approach 1	approach 2
makeSet(value)	$\Theta(1)$	$\Theta(1)$
findSet(value)	$\Theta(1)$	$\Theta(n)$
union(valueA, valueB)	$\Theta(n)$	$\Theta(n)$

A better idea

Here’s a better idea:

We need to be able to combine things easily.
- Pointer based data structures are better at that.

But given a value, we need to be able to find the right set.
- Sounds like we need a dictionary somewhere

And we need to be able to find a certain element (“the representative”) within a set quickly.
- Trees are good at that (better than linked lists at least)

The Real Implementation

Disjoint-Set ADT

state

Set of Sets
- Disjoint: Elements must be unique across sets
- No required order
- Each set has representative

Count of Sets

behavior

makeSet(x) – creates a new set within the disjoint set where the only member is x. Picks representative for set

findSet(x) – looks up the set containing element x, returns representative of that set

union(x, y) – looks up set containing x and set containing y, combines two sets into one. Picks new representative for resulting set

UpTreeDisjointSet<E>

state

```
Collection<TreeSet> forest
Dictionary<NodeValues,
NodeLocations> nodeInventory
```

behavior

```
makeSet(x)-create a new
tree of size 1 and add to
our forest

findSet(x)-locates node with
x and moves up tree to find
root

union(x, y)-append tree
with y as a child of tree
with x
```

TreeSet<E>

state

```
SetNode overallRoot
```

behavior

```
TreeSet(x)
add(x)
remove(x, y)
getRep()-returns data of
overallRoot
```

SetNode<E>

state

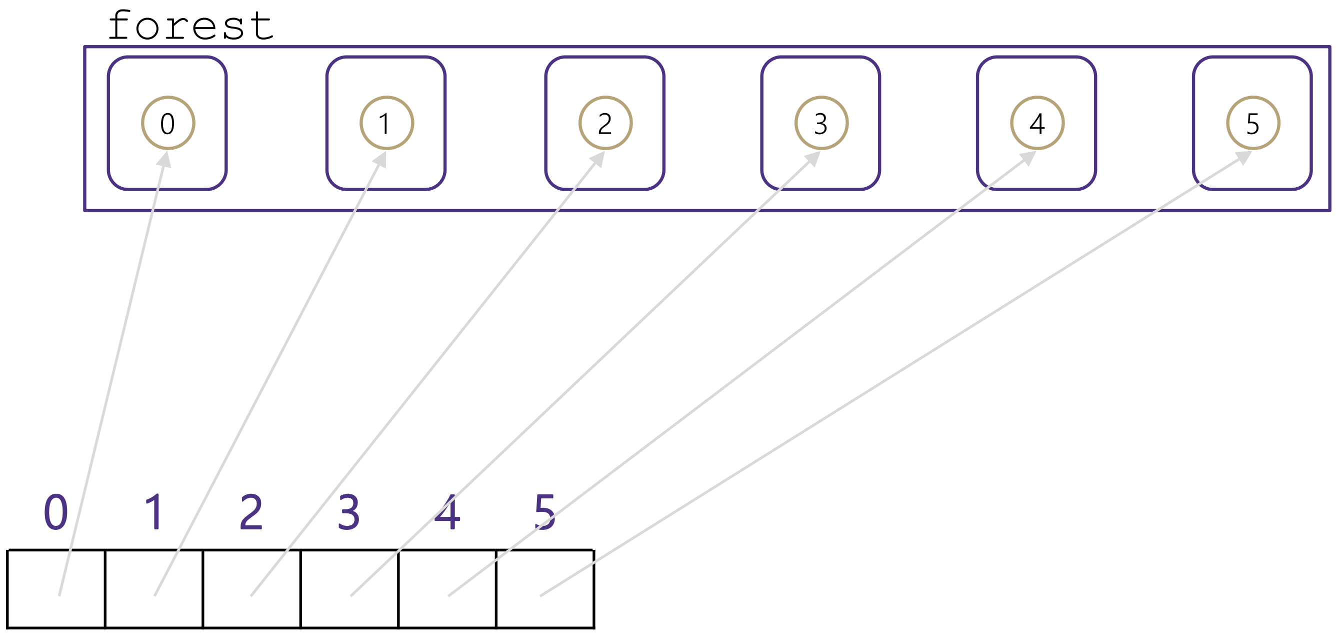
```
E data
Collection<SetNode>
children
```

behavior

```
SetNode(x)
addChild(x)
removeChild(x, y)
```

Implement makeSet(x)

```
makeSet(0)

makeSet(1)

makeSet(2)

makeSet(3)

makeSet(4)

makeSet(5)
```



TreeDisjointSet<E>

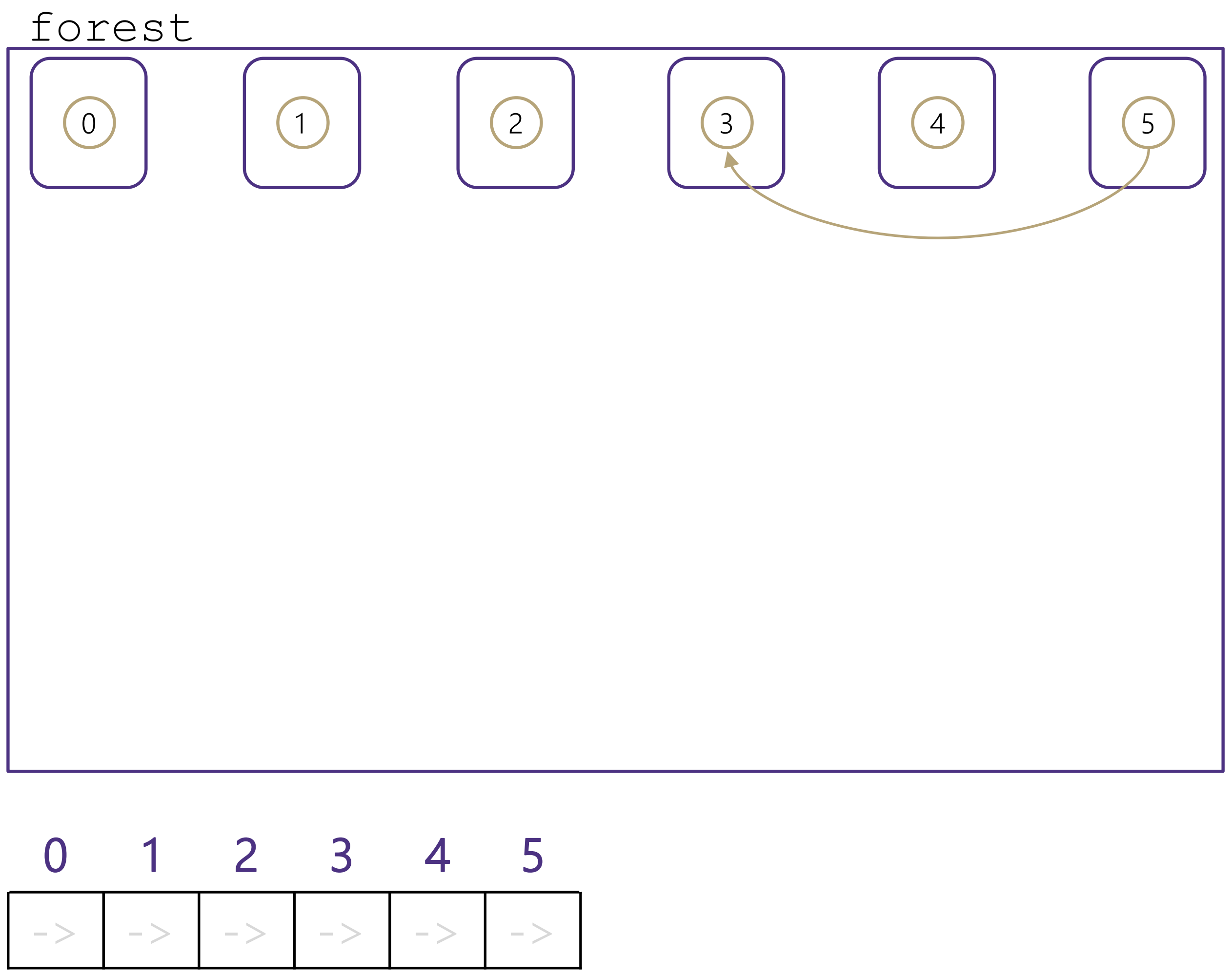
state
```
Collection<TreeSet> forest
Dictionary<NodeValues,
NodeLocations> nodeInventory
```
behavior
```
makeSet(x)-create a new tree
of size 1 and add to our
forest
findSet(x)-locates node with x
and moves up tree to find root
union(x, y)-append tree with y
as a child of tree with x
```

Worst case runtime? Just like with graphs, we're going to assume we have control over the dictionary keys and just say we'll always have $\Theta(1)$ dictionary behavior.

$O(1)$

Implement union(x, y)

```
union(3, 5)
```



TreeDisjointSet<E>

state

```
Collection<TreeSet> forest
Dictionary<NodeValues,
NodeLocations> nodeInventory
```

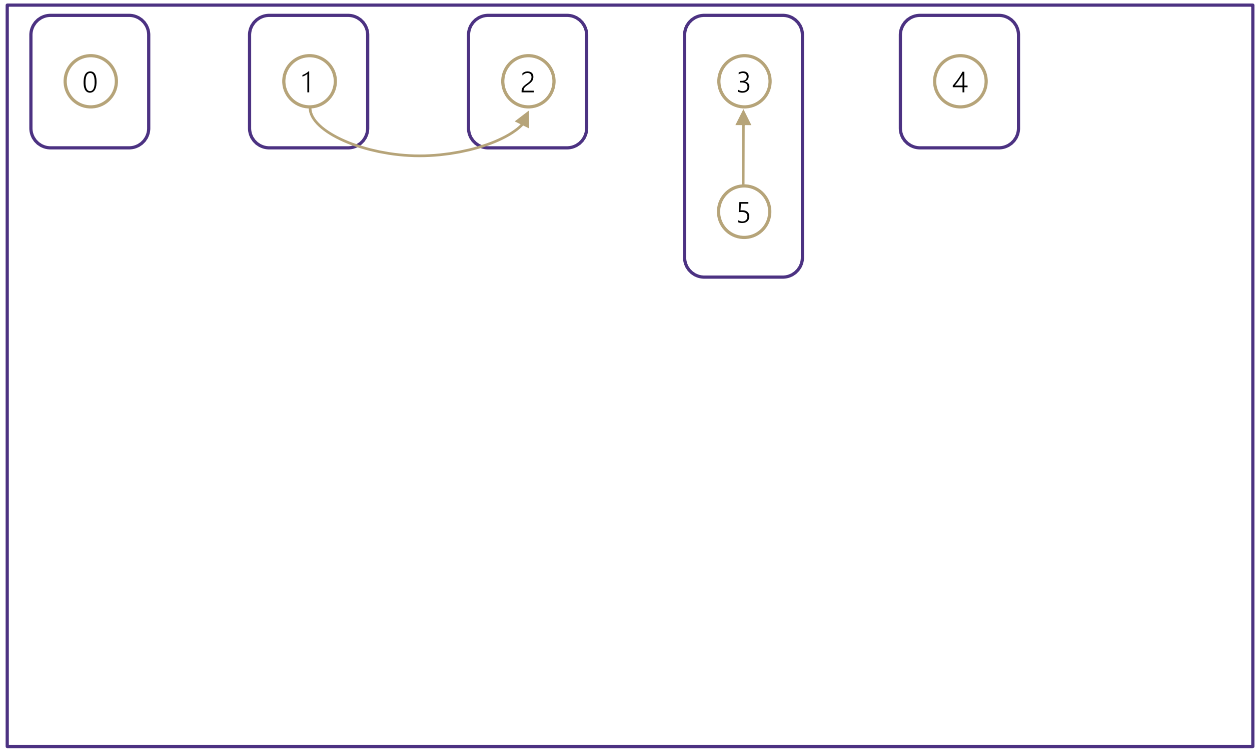
behavior

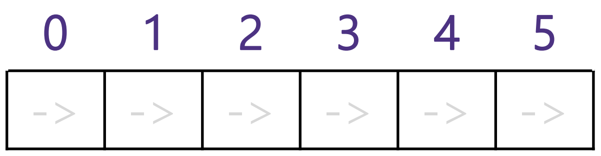
```
makeSet(x)-create a new tree
of size 1 and add to our
forest
findSet(x)-locates node with x
and moves up tree to find root
union(x, y)-append tree with y
as a child of tree with x
```

Implement union(x, y)

```
union(3, 5)

union(2, 1)
```

forest

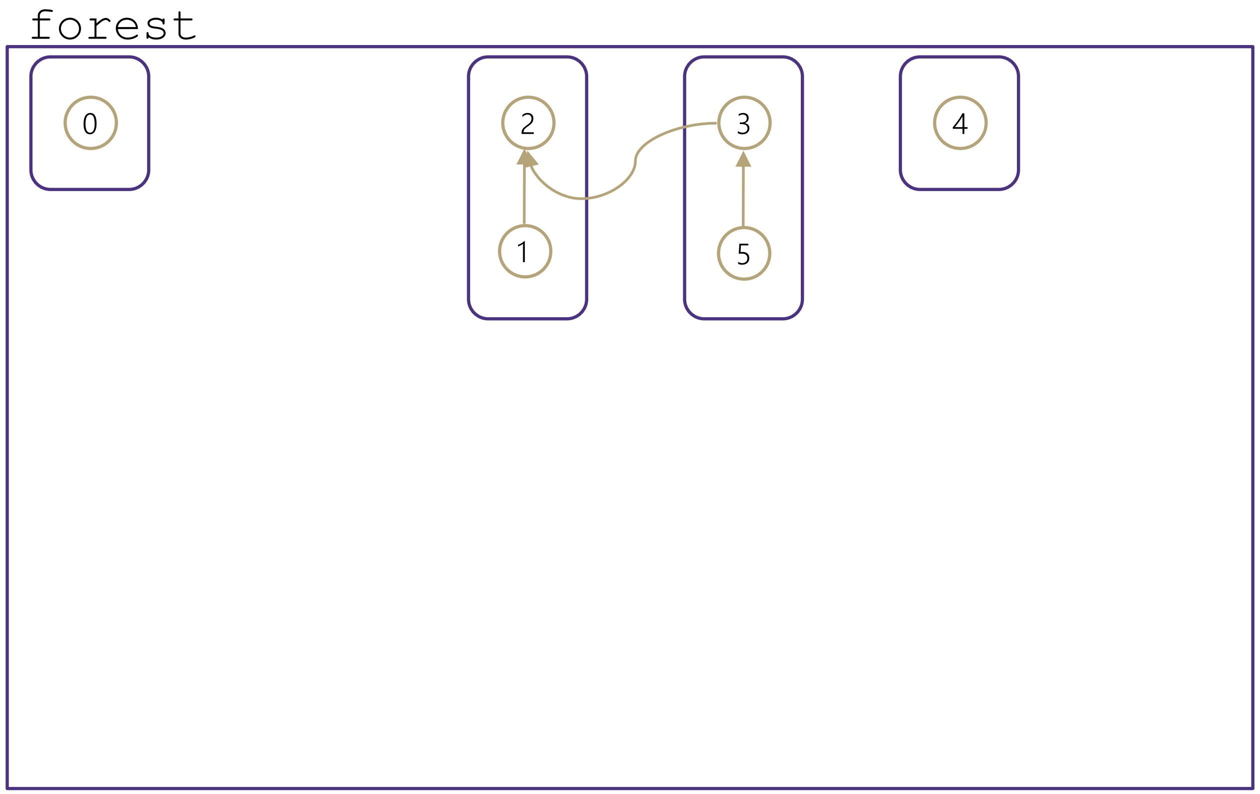


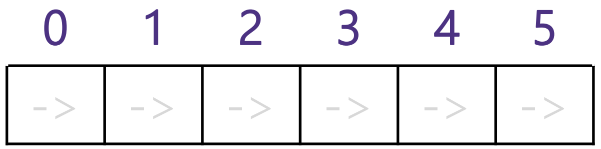


TreeDisjointSet<E>
state
`Collection<TreeSet> forest`
`Dictionary<NodeValues, NodeLocations> nodeInventory`
behavior
`makeSet(x)`-create a new tree of size 1 and add to our forest
`findSet(x)`-locates node with x and moves up tree to find root
`union(x, y)`-append tree with y as a child of tree with x

Implement union(x, y)

```
union(3, 5)

union(2, 1)

union(2, 5)
```





TreeDisjointSet<E>

state

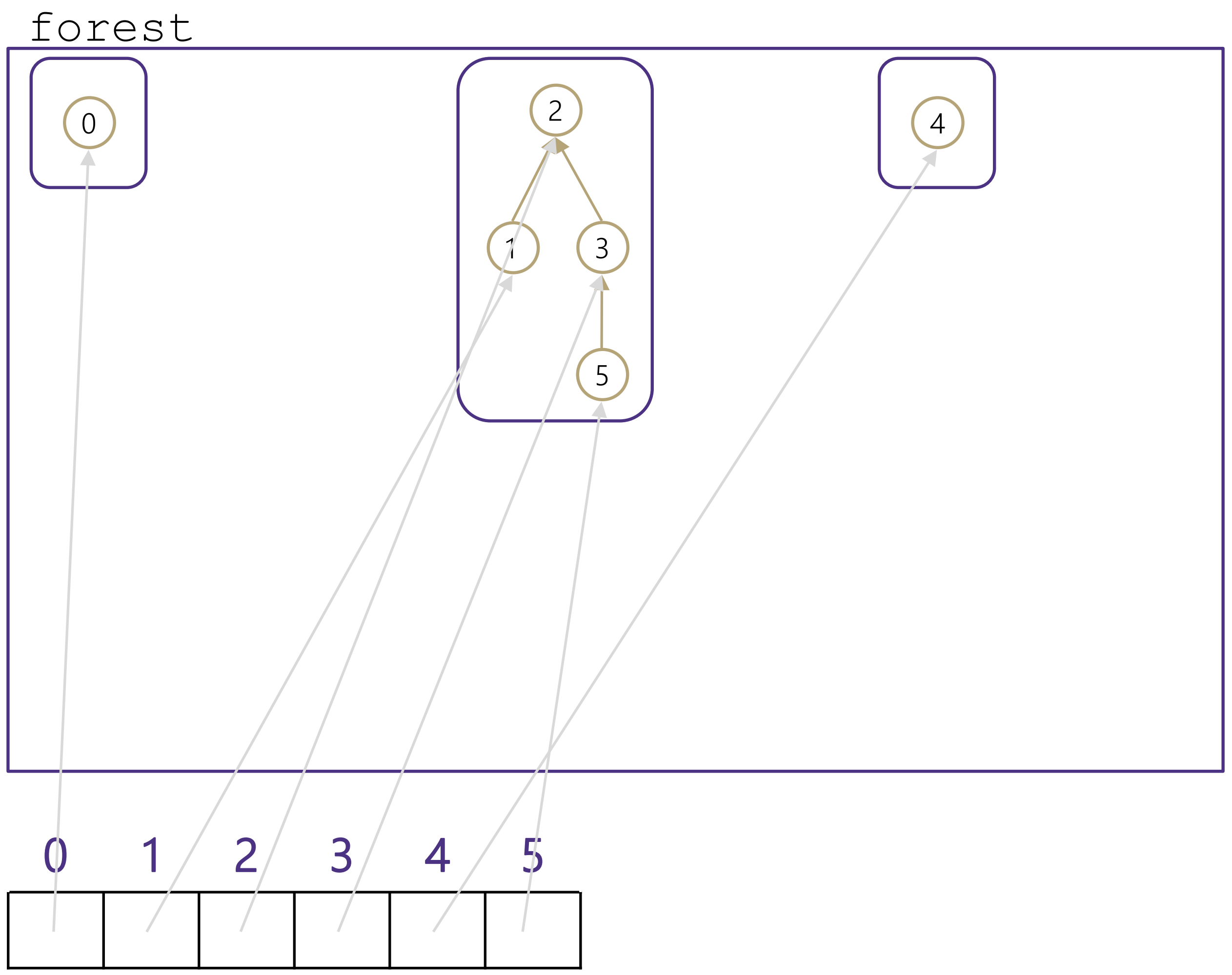
```
Collection<TreeSet> forest
Dictionary<NodeValues,
NodeLocations> nodeInventory
```

behavior

```
makeSet(x)-create a new tree
of size 1 and add to our
forest
findSet(x)-locates node with x
and moves up tree to find root
union(x, y)-append tree with y
as a child of tree with x
```

Implement union(x, y)

```
union(3, 5)

union(2, 1)

union(2, 5)
```



TreeDisjointSet<E>

state

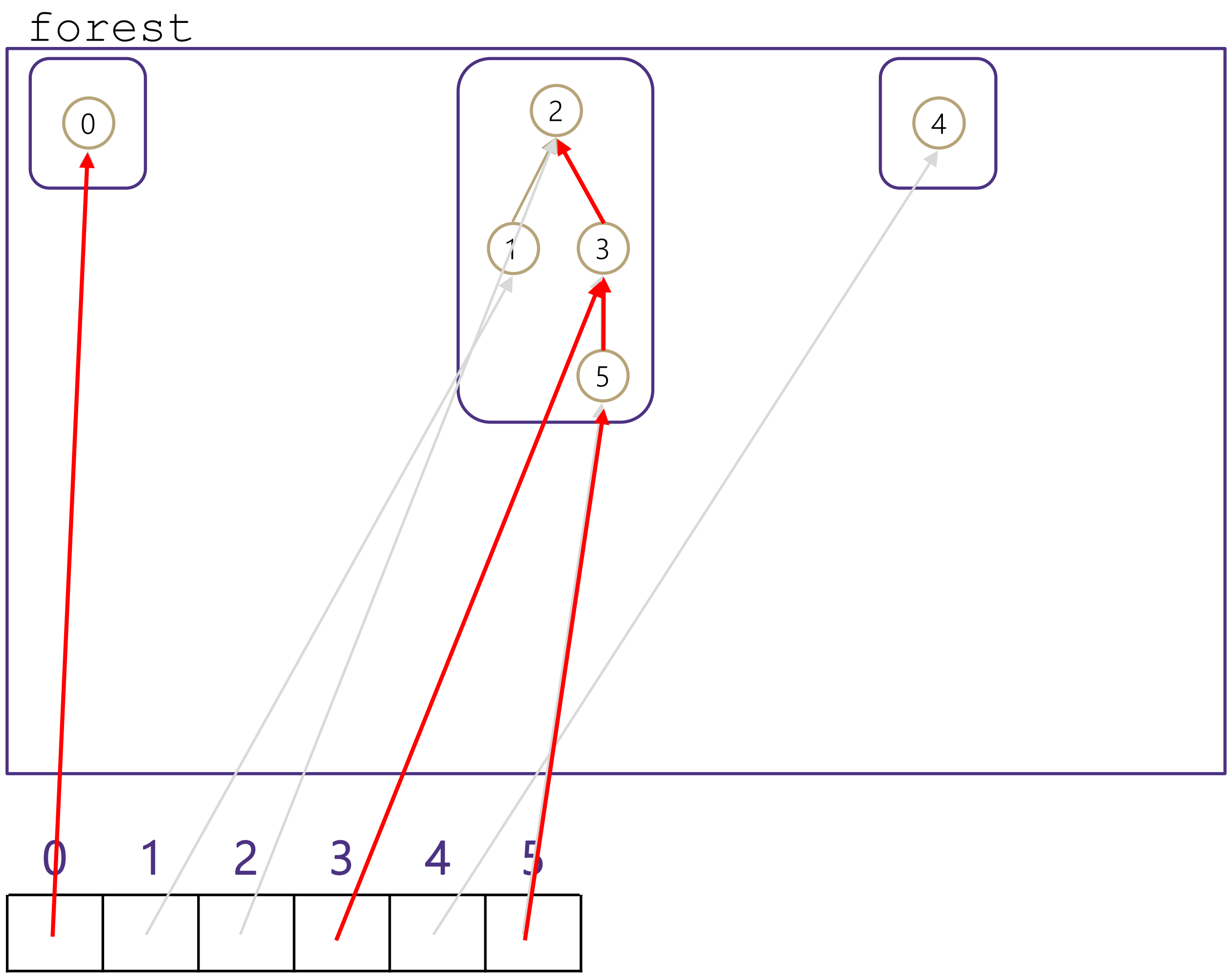
```
Collection<TreeSet> forest
Dictionary<NodeValues,
NodeLocations> nodeInventory
```

behavior

```
makeSet(x)-create a new tree
of size 1 and add to our
forest
findSet(x)-locates node with x
and moves up tree to find root
union(x, y)-append tree with y
as a child of tree with x
```

Implement findSet(x)

```
findSet(0)

findSet(3)

findSet(5)
```



TreeDisjointSet<E>

state

```
Collection<TreeSet> forest
Dictionary<NodeValues,
NodeLocations> nodeInventory
```

behavior

`makeSet(x)`-create a new tree of size 1 and add to our forest

`findSet(x)`-locates node with x and moves up tree to find root

`union(x, y)`-append tree with y as a child of tree with x

Worst case runtime of findSet?

$\Theta(n)$

Worst case runtime of union?

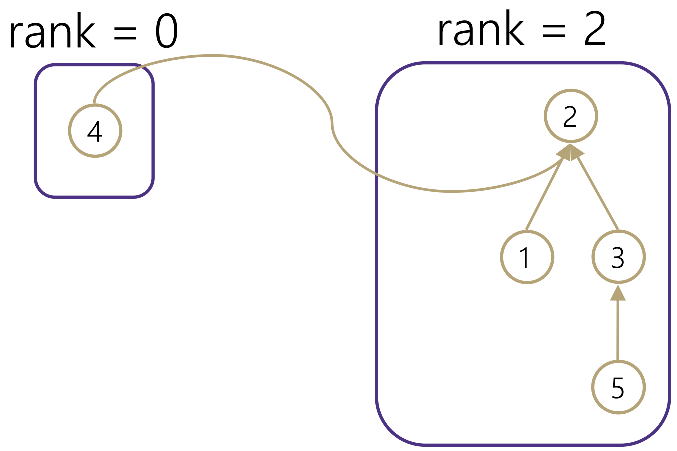
$\Theta(n)$ – union has to call find!

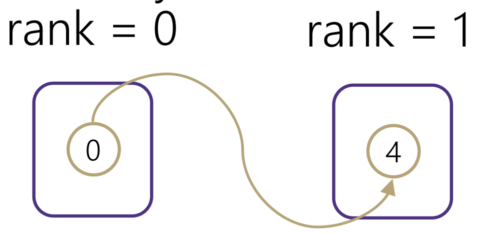
Improving union

Problem: Trees can be unbalanced

Solution: Union-by-rank!

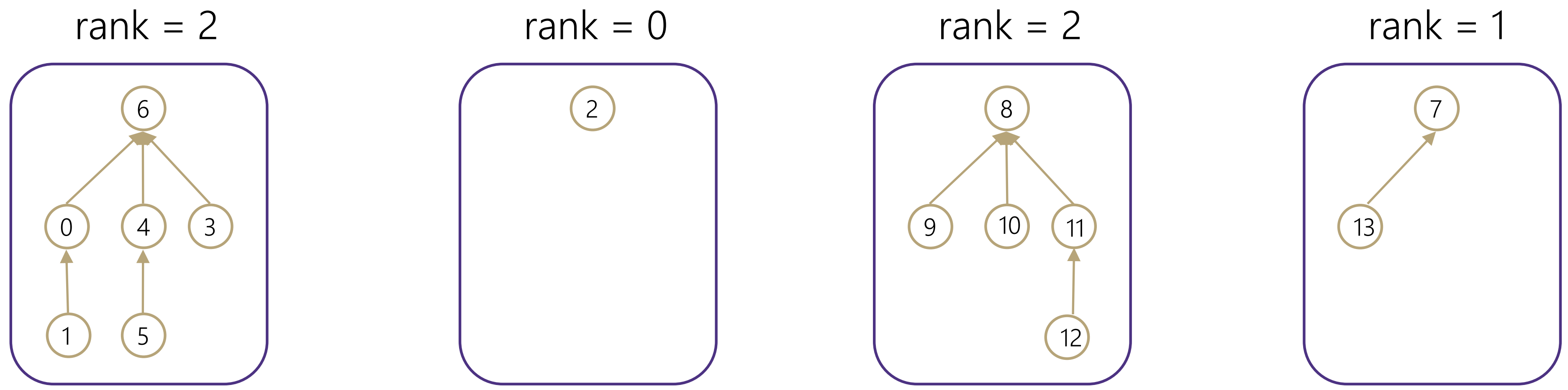
- rank is a lot like height (it's not quite height, for reasons we'll see tomorrow)
- Keep track of rank of all trees
- makeSet creates a tree of rank 0.
- When unioning make the tree with larger rank the root. New rank is larger of two merged ranks.
- If it's a tie, pick one to be root arbitrarily and increase rank by one.





Practice

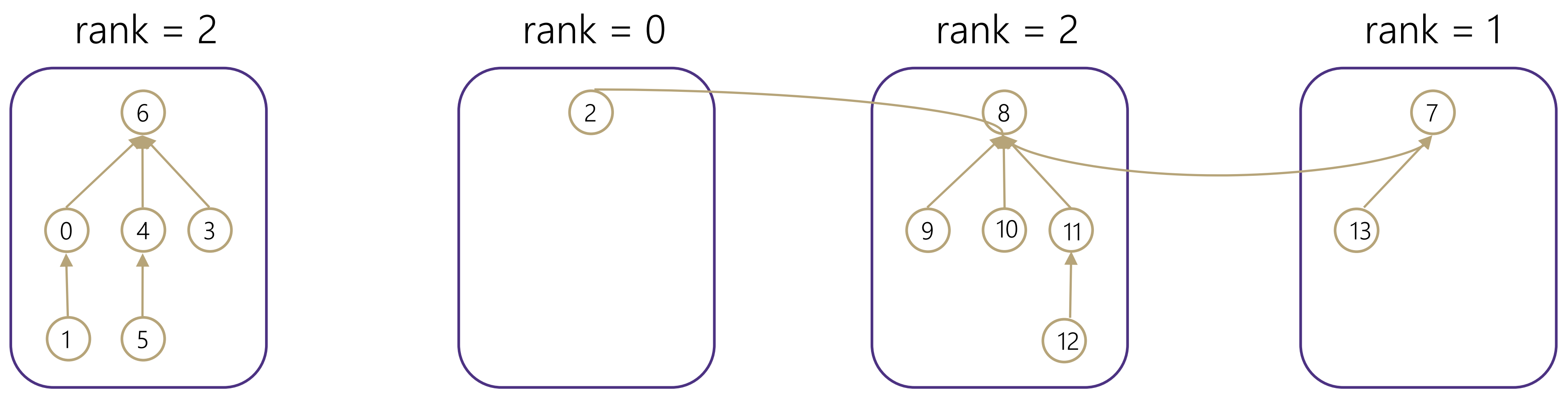
Given the following disjoint-set what would be the result of the following calls on union if we add the "union-by-rank" optimization. Draw the forest at each stage with corresponding ranks for each tree.



```
union(2, 13)
union(4, 12)
union(2, 8)
```

Practice

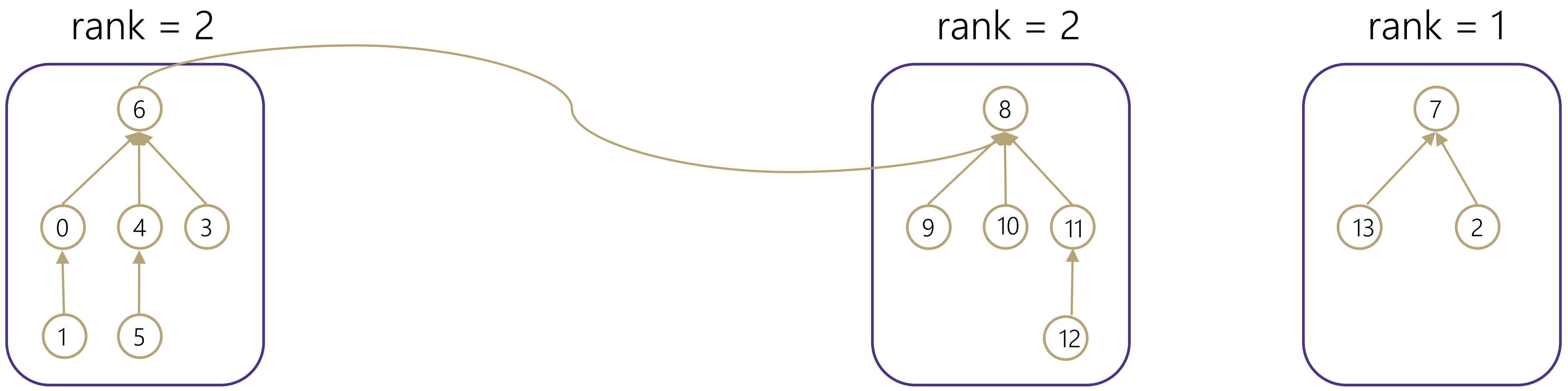
Given the following disjoint-set what would be the result of the following calls on union if we add the "union-by-rank" optimization. Draw the forest at each stage with corresponding ranks for each tree.



```
union(2, 13)
```

Practice

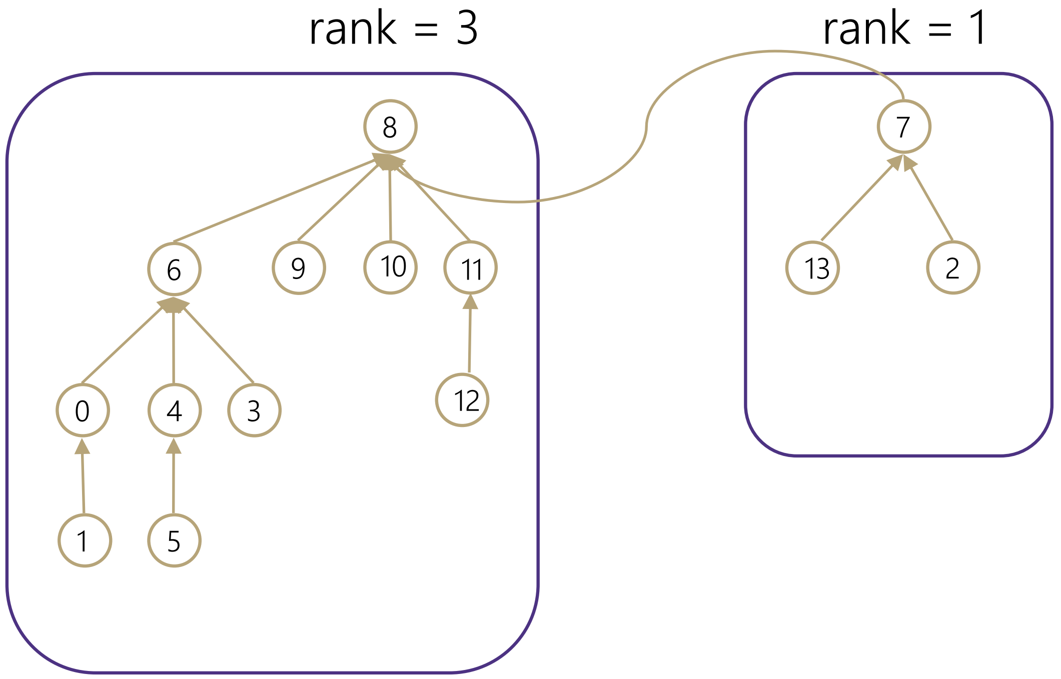
Given the following disjoint-set what would be the result of the following calls on union if we add the "union-by-rank" optimization. Draw the forest at each stage with corresponding ranks for each tree.



```
union(2, 13)

union(4, 12)
```

Practice

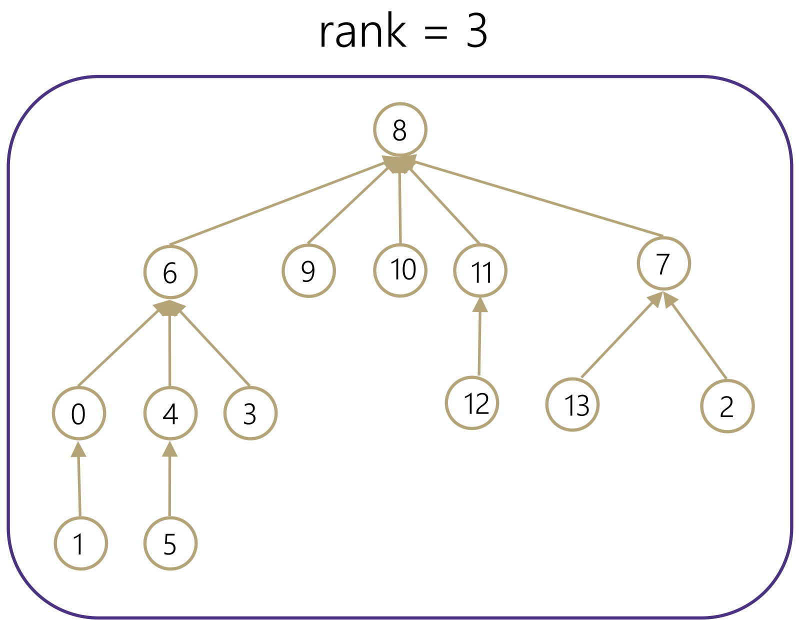
Given the following disjoint-set what would be the result of the following calls on union if we add the "union-by-rank" optimization. Draw the forest at each stage with corresponding ranks for each tree.



```
union(2, 13)

union(4, 12)

union(2, 8)
```

Practice

Given the following disjoint-set what would be the result of the following calls on union if we add the "union-by-rank" optimization. Draw the forest at each stage with corresponding ranks for each tree.



```
union(2, 13)
union(4, 12)
union(2, 8)
```

Does this improve the worst case runtimes?

findSet is $\Theta(\log(n))$ now, not $\Theta(n)$!

Improving findSet()

Problem: Every time we call findSet() you must traverse all the levels of the tree to find representative

Solution: Path Compression

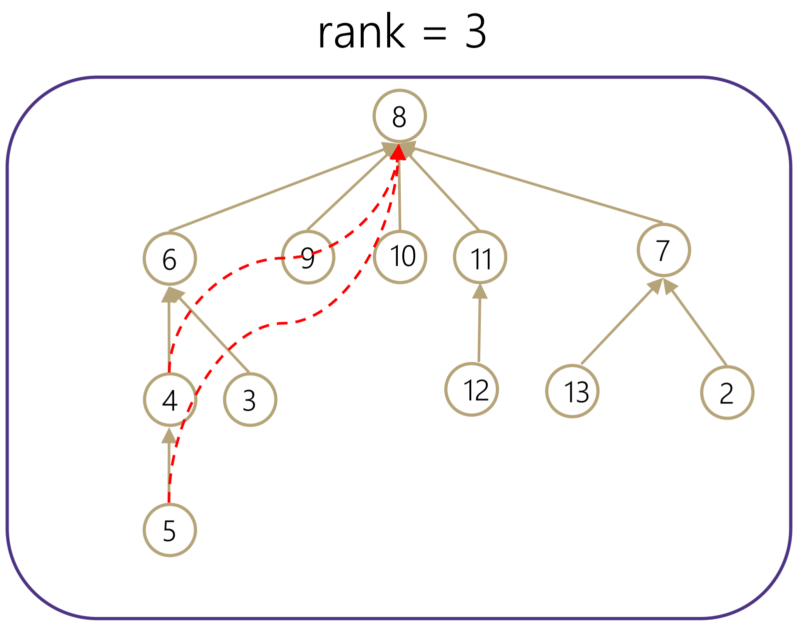
- Collapse tree into fewer levels by updating parent pointer of each node you visit
- Whenever you call findSet() update each node you touch's parent pointer to point directly to overallRoot

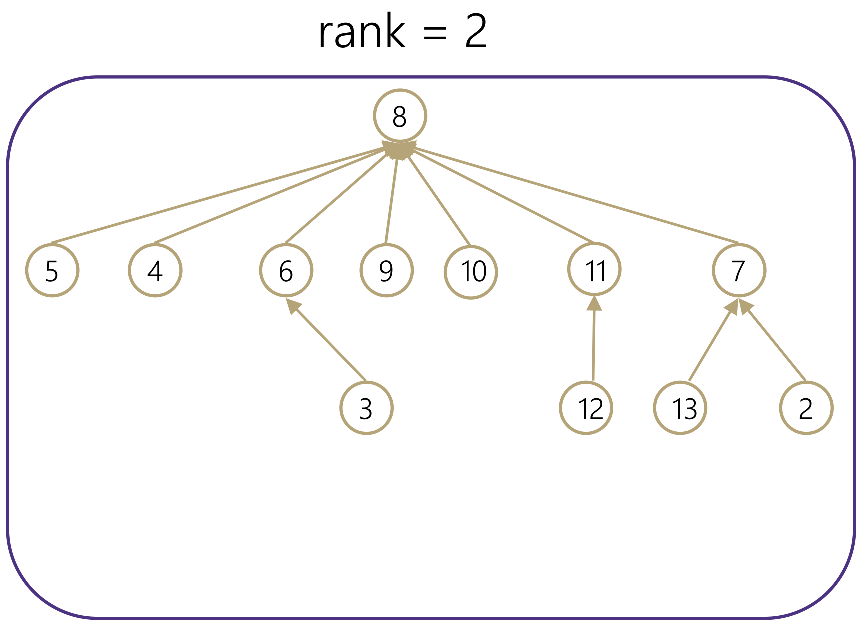
`findSet(5)`

`findSet(4)`

Does this improve the worst case runtimes?

Not the worst-case, but…

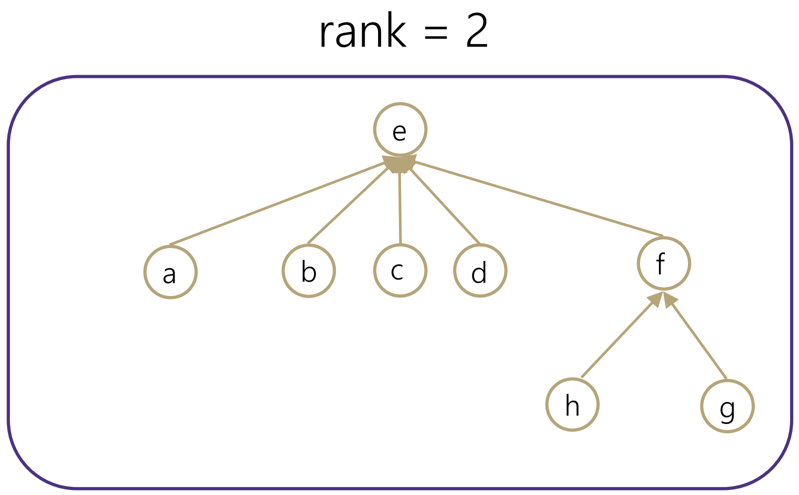




Example

Using the union-by-rank and path-compression optimized implementations of disjoint-sets draw the resulting forest caused by these calls:

```
1.makeSet(a)
2.makeSet(b)
3.makeSet(c)
4.makeSet(d)
5.makeSet(e)
6.makeSet(f)
7.makeSet(h)
8.union(c, e)
9.union(d, e)
10.union(a, c)
11.union(g, h)
12.union(b, f)
13.union(g, f)
14.union(b, c)
```



Optimized Up-trees Runtimes

	makeSet	findSet	Union
Worst-Case	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Best-Case	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
In-Practice	$\Theta(1)$	$O(\log^* n)$	$O(\log^* n)$

Hey why are some of those $O()$ not $\Theta()$?
And…wait what's that * above the log?

$\log^*(n)$

$\log^*(n)$ is the “iterated logarithm”

It answers the question “how many times do I have to take the log of this to get a number at most 1?”

E.g. $\log^*(16) = 3$

$\log(16) = 4 \qquad \log(4) = 2 \qquad \log(2) = 1.$

$\log^* n$ grows ridiculously slowly.

$\log^*(10^{80}) \ = 5.$

10^{80} is the number of atoms in the observable universe. For all practical purposes these operations are constant time.

But they aren’t $O(1)$.

Optimized Up-tree Runtimes

$\log^* n$ isn't tight – that's why those $\Theta()$ bounds became $O()$ bounds.

There is a tight bound. It's a function that grows even slower than $\log^* n$

- Google "inverse Ackerman function"