Lecture 17: Minimum Spanning Trees

CSE 373: Data Structures and Algorithms

Administrivia

Exercise 3 due tonight

- Don't forget you have 3 extra late days ☺
- note on N vs R and P: please actually use the R and P variables if the runtime is actually based on the number of possible reaction types or number of possible persons. You use N (the normal variable) bc there are two factors here – if you write N, we can't tell which one you mean so it's not correct.

Project 4 due Wednesday May 20th

- Two-week assignment, two weeks of work
- project 3 part 1 out, highly recommend you try to finish in 1 week. Part 2 topics covered today / Monday/Wednesday next week

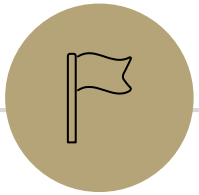
Come hang out with us this Saturday!

- How to ace the technical interview with Zach and Kasey
- TA career panel discussion
- Lots of resources & real talk to share

We're really sorry but we're a bit behind on grading…

- Next week we'll release midterm grades
- Put everything into canvas
- Special Zach & Kasey office hours to talk grades

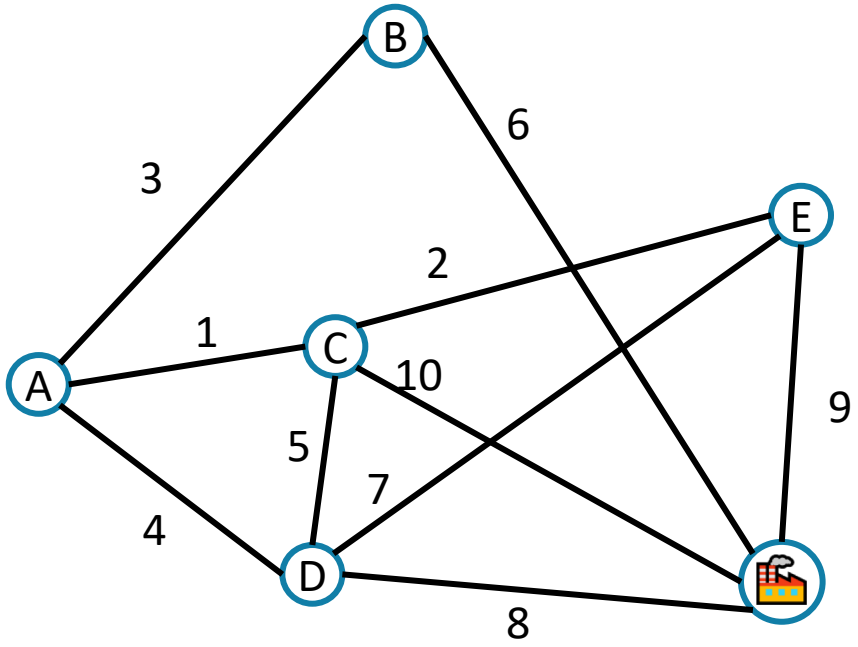
https://www.reddit.com/r/udub/comments/gffvse/ive_crashed_and_my_motivation_is_dead/



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.

MST Problem

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

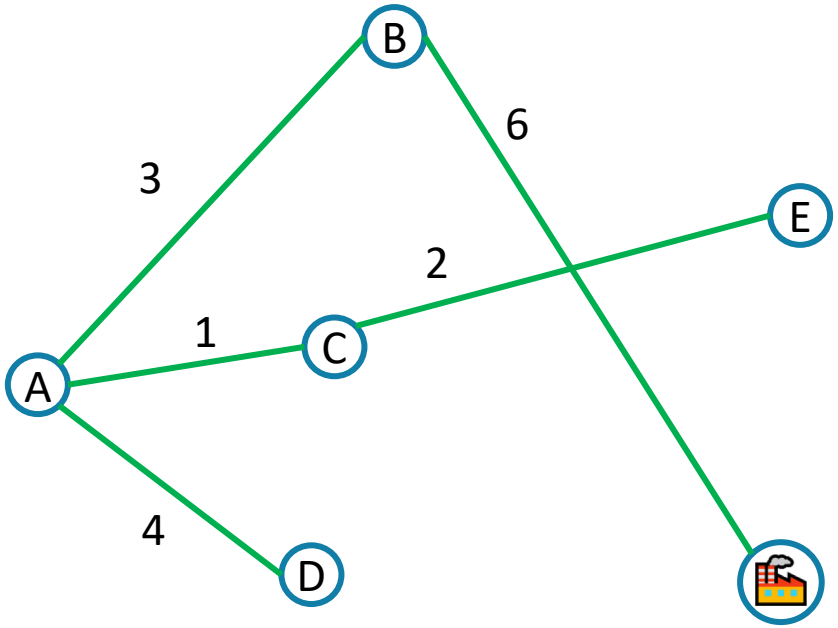
- Every vertex touches at least one of the edges. The edges "**span**" the graph.
- The graph on just those edges is **connected**.
- The minimum weight set of edges that meet those conditions.

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

Our result 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.

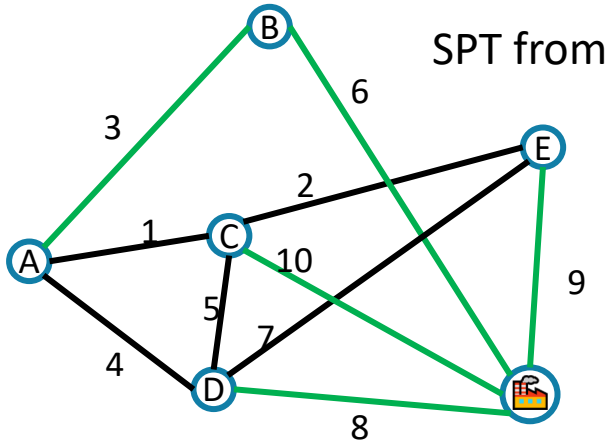
Interaction Pane Question:

Is there always a unique MST for a given graph, yes or no?

Shortest Path vs Minimum Spanning

Shortest Path Problem

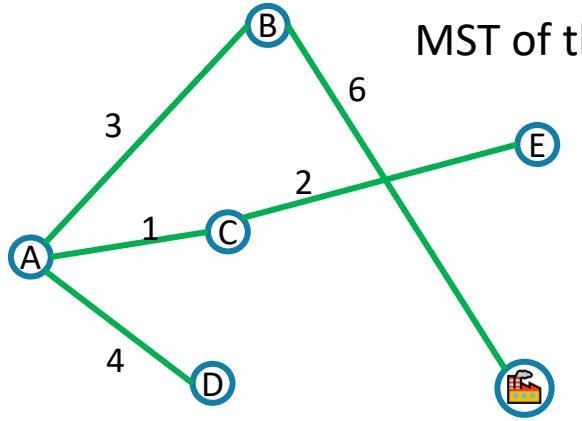
Given: a directed graph G and vertices s,t
Find: the shortest path from s to t.



SPT from Factory

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.



MST of the graph

Shortest Path Tree

Specific start node (if you have a different start node, that changes the whole SPT, so there are multiple SPTs for graphs frequently)
Keeps track of total path length.

Minimum Spanning Tree

No specific start node, since the goal is just to minimize the edge weights sum. Often only one possible MST that has the minimum sum.
All nodes connected
Keeps track of cheapest edges that maintain connectivity

Finding an MST

Here are two ideas for finding an MST:

Prim's

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.

Kruskal's

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!!

Interaction Pane Question:

Which of these do you think are more likely to work?

- Thumbs up for vertex by vertex
- Thumbs down for edge by edge
- Clap for both

Prim's Algorithm

Dijkstra's

1. Start at source
2. Update distance from current to unprocessed neighbors
3. Add closest unprocessed neighbor to solution
4. Repeat until all vertices have been marked processed

```
1Dijkstra(Graph G, Vertex source)
2    initialize distances to ∞
3    mark source as distance 0
4
5    mark all vertices unprocessed
5    while(there are unprocessed vertices){
6        let u be the closest unprocessed vertex
7        foreach(edge (u,v) leaving u){
8            if(u.dist+weight(u,v) < v.dist){
9                v.dist = u.dist+weight(u,v)
10               v.predecessor = u
11           }
12       }
13       mark u as processed
14   }
15
```

Algorithm idea:

1. Start at any node
2. Investigate edges that connect unprocessed vertices
3. Add the lightest edge that grows connectivity to solution
4. Repeat until all vertices have been marked processed

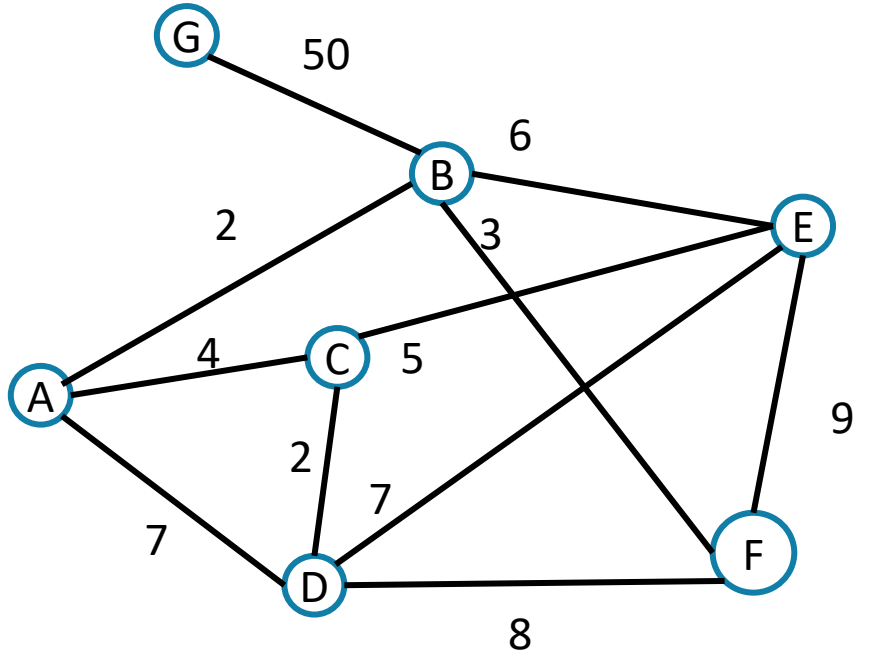
```
1Prims(Graph G, Vertex source)
2    initialize distances to ∞
3    mark source as distance 0
4
5    mark all vertices unprocessed
5    while(there are unprocessed vertices){
6        let u be the closest unprocessed vertex
7        foreach(edge (u,v) leaving u){
8            if(weight(u,v) < v.dist){
9                v.dist = u.dist+weight(u,v)
10               v.predecessor = u
11           }
12       }
13       mark u as processed
14   }
15
```

In the Chat

Which lines of Dijkstra can we change to create our new algorithm?

Try it Out

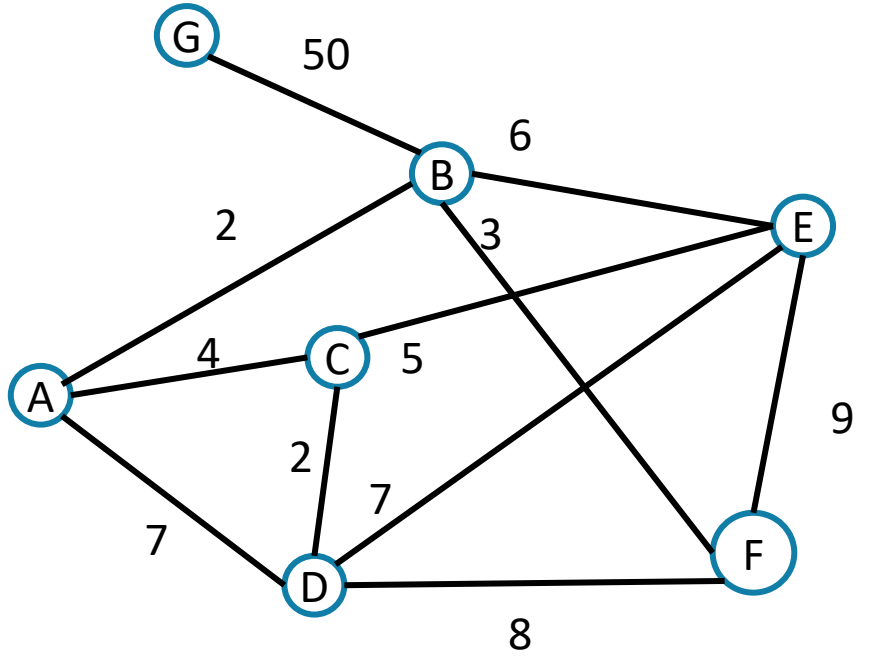
```
PrimMST(Graph G)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  foreach(edge (source, v) ) {
    v.dist = weight(source,v)
    v.bestEdge = (source,v)
  }
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    add u.bestEdge to spanning tree
    foreach(edge (u,v) leaving u){
      if(weight(u,v) < v.dist && v unprocessed ){
        v.dist = weight(u,v)
        v.bestEdge = (u,v)
      }
    }
    mark u as processed
  }
```



Vertex	Distance	Best Edge	Processed
A			
B			
C			
D			
E			
F			
G			

Try it Out

```
PrimMST(Graph G)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  foreach(edge (source, v) ) {
    v.dist = weight(source,v)
    v.bestEdge = (source,v)
  }
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    add u.bestEdge to spanning tree
    foreach(edge (u,v) leaving u){
      if(weight(u,v) < v.dist && v unprocessed ){
        v.dist = weight(u,v)
        v.bestEdge = (u,v)
      }
    }
    mark u as processed
  }
```



Vertex	Distance	Best Edge	Processed
A	-	X	✓
B	2	(A, B)	✓
C	4	(A, C)	✓
D	~~7~~-2	~~(A,-D)~~(C, D)	✓
E	~~6~~-5	~~(B,-E)~~ (C, E)	✓
F	3	(B, F)	✓
G	50	(B, G)	✓

A different Approach

Prim’s Algorithm started from a single vertex and reached more and more other vertices.

Prim’s thinks vertex by vertex (add the closest vertex to the currently reachable set).

Prim's Algorithm Visualization

What if you think edge by edge instead?

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)

This is Kruskal’s Algorithm.

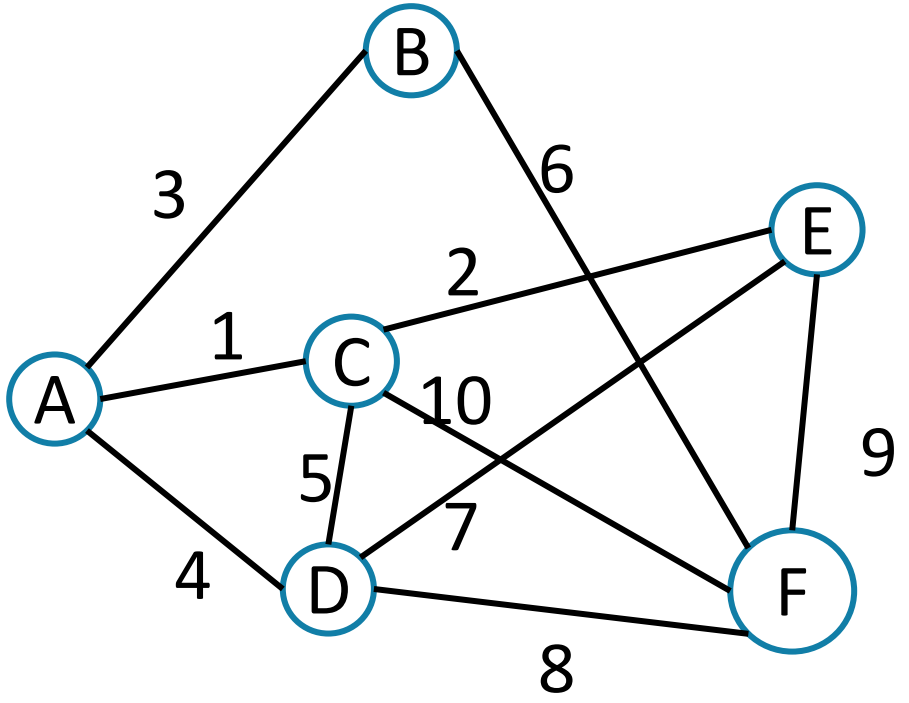
Kruskal's Algorithm Visualization

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
        }
    }
```

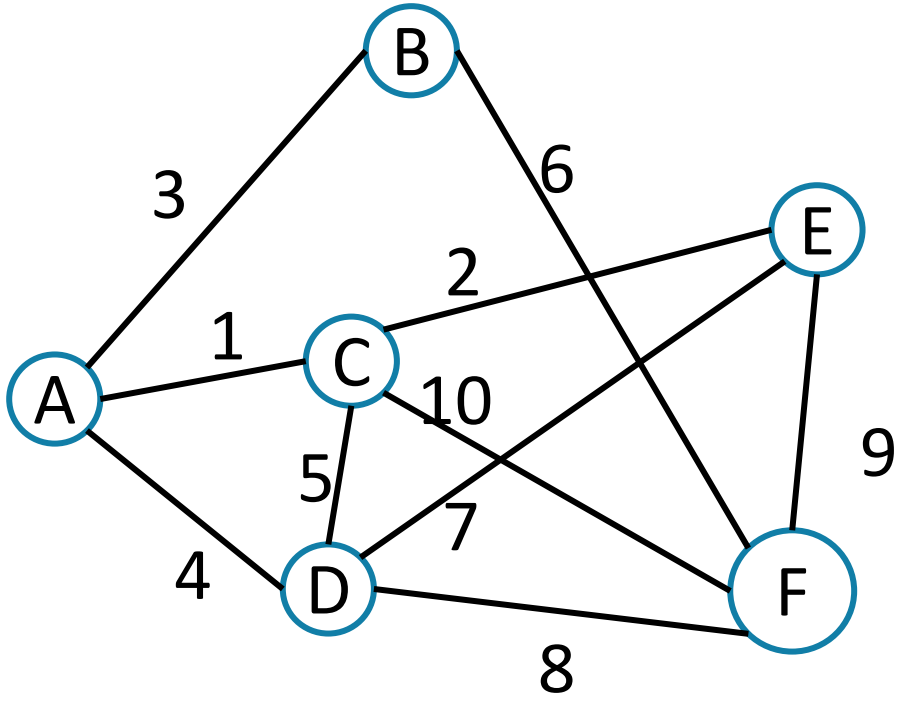


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

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

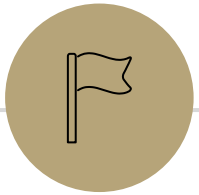
Kruskal's Implementation

```
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
        }
    }
```

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
```

Can we use one of our data structures?



Disjoint Sets

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!

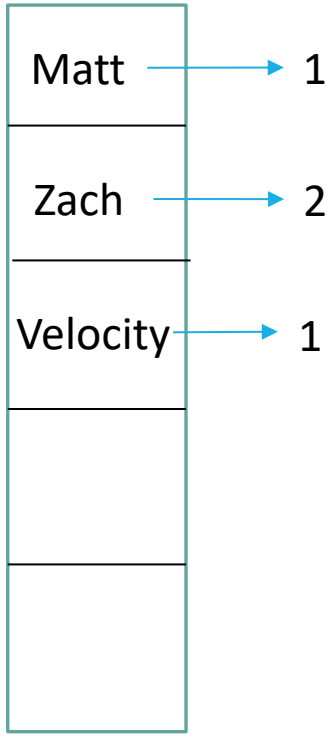
We're going to slowly design/optimize the implementation over the next lecture-plus.

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

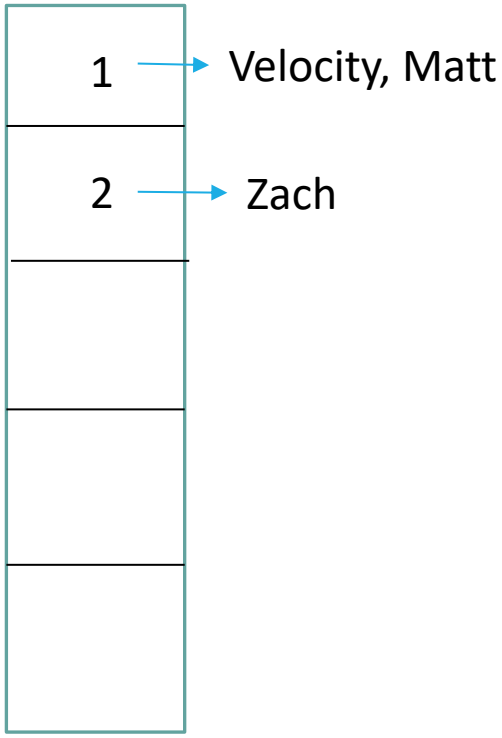
Implementing Disjoint-Sets with Dictionaries

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



	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

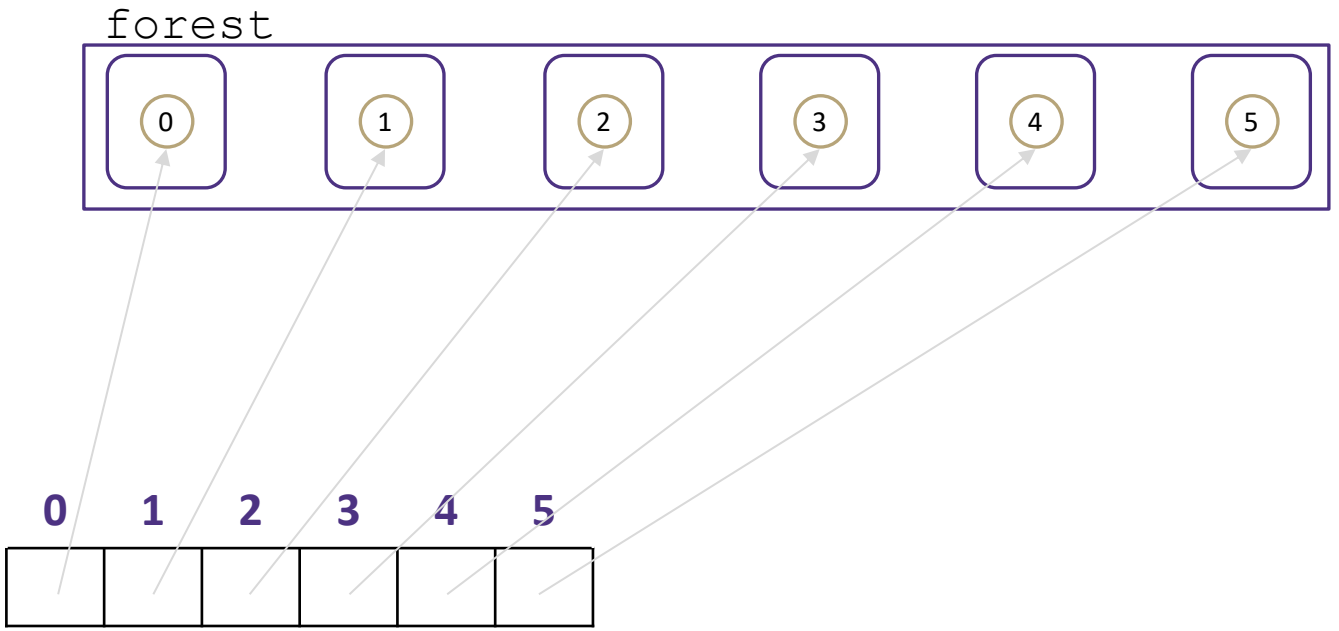
`SetNode(x)`

`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

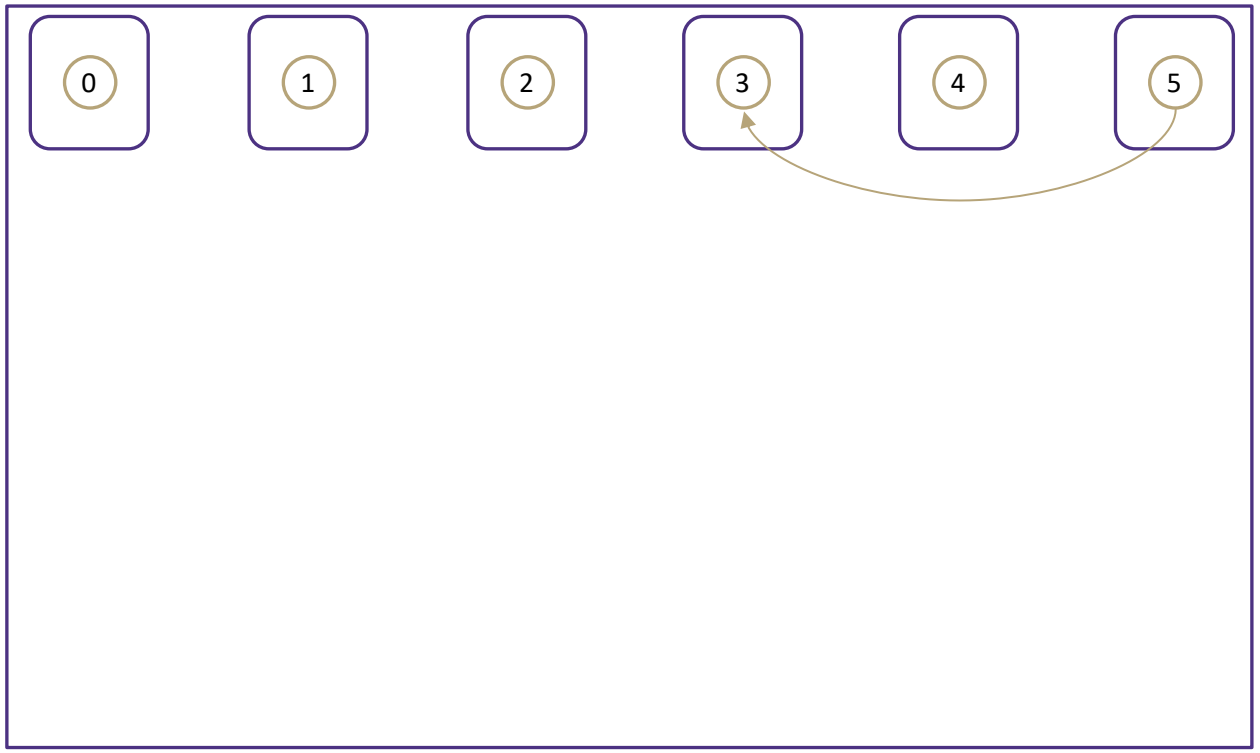
Worst case runtime? Just like with graphs, we're going to assume we have control over the dictionary keys and just say we'll always have $\Theta(1)$ dictionary behavior.

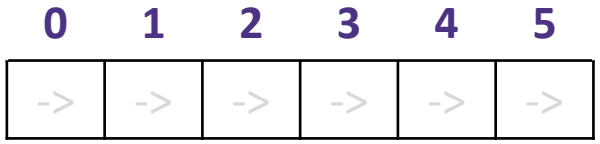
$O(1)$

Implement union(x, y)

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

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

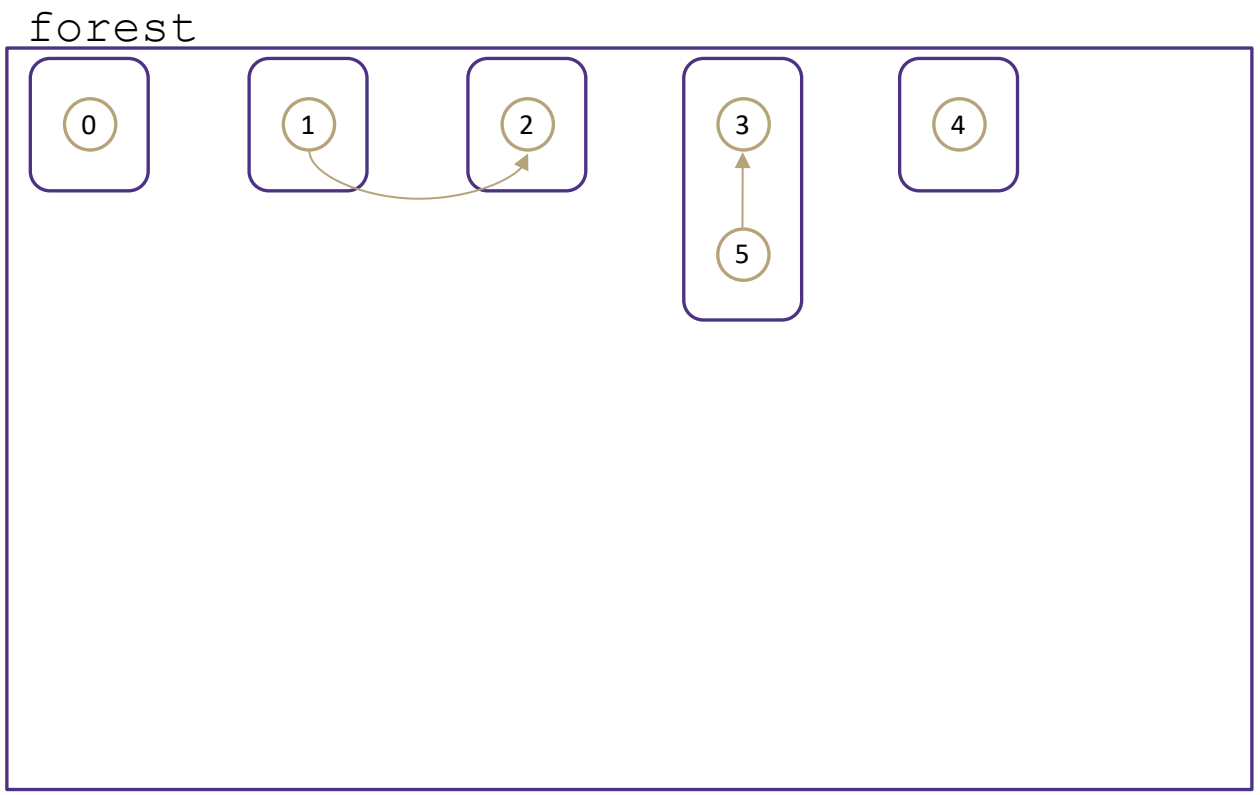
`findSet(x)`-locates node with x and moves up tree to find root

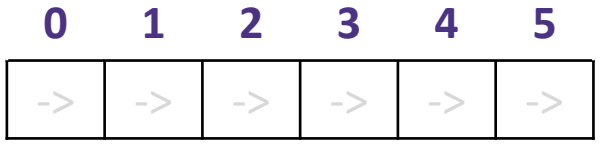
`union(x, y)`-append tree with y as a child of tree with x

Implement union(x, y)

```
union(3, 5)

union(2, 1)
```





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

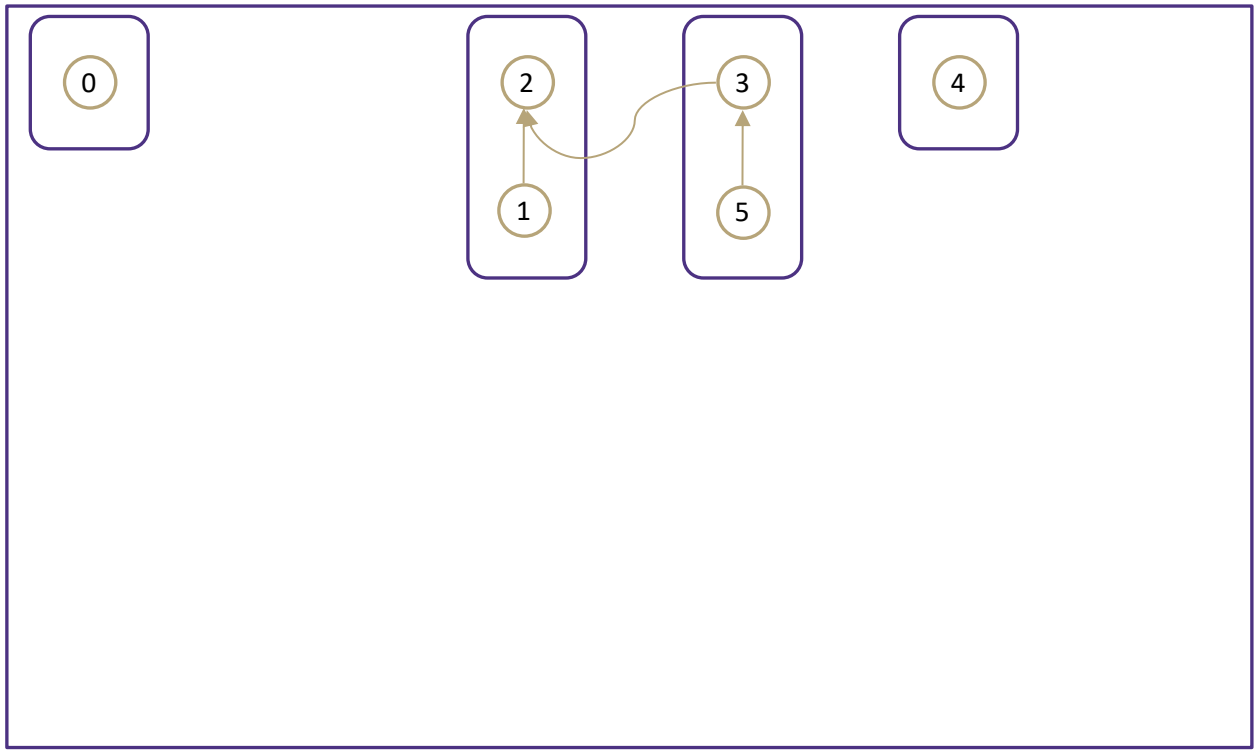
Implement union(x, y)

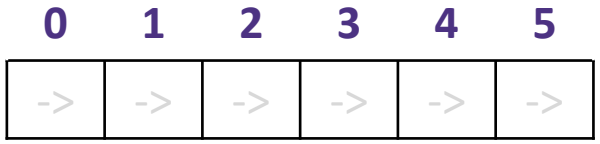
```
union(3, 5)

union(2, 1)

union(2, 5)
```

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

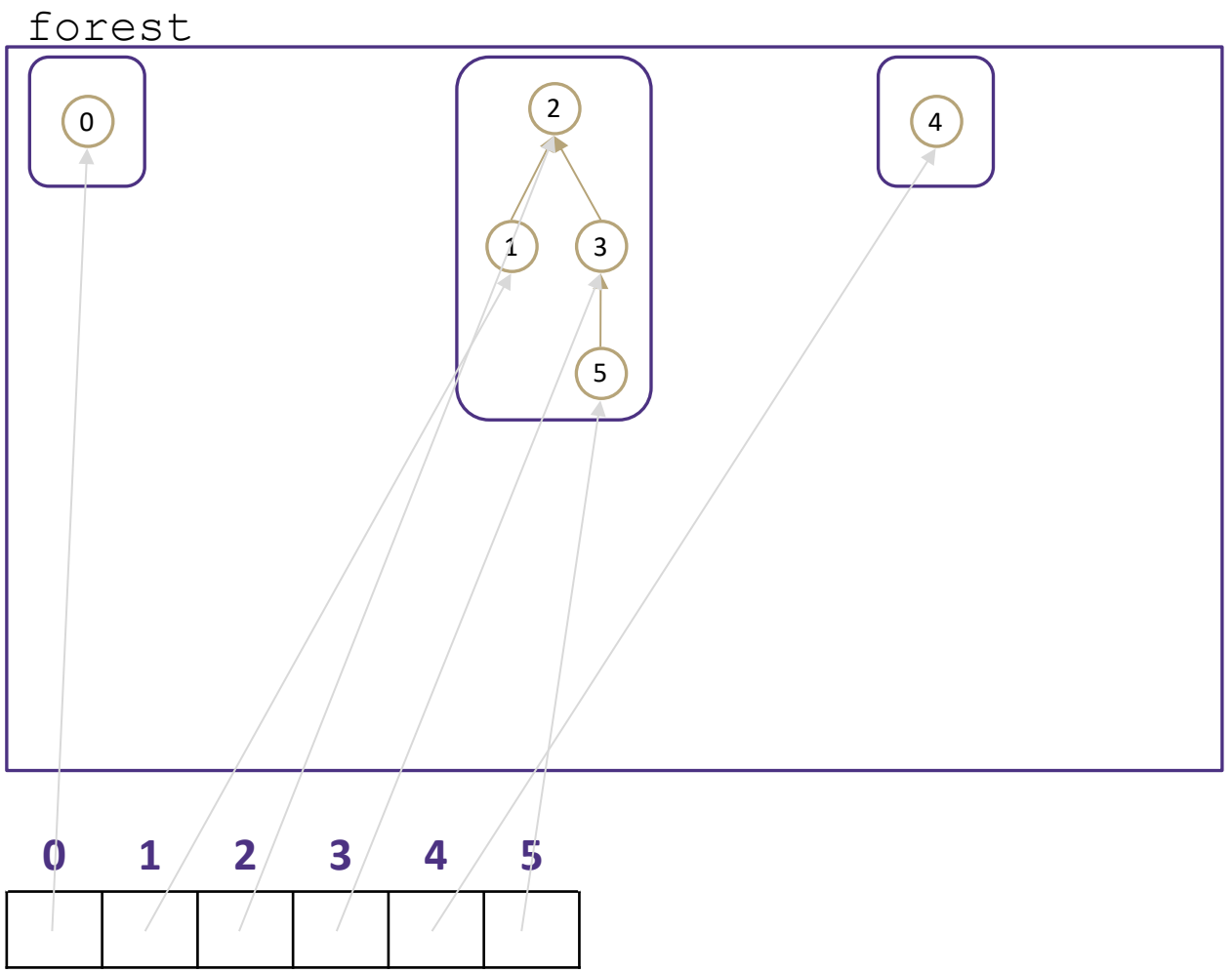
`union(x, y)`-append tree with y as a child of tree with x

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

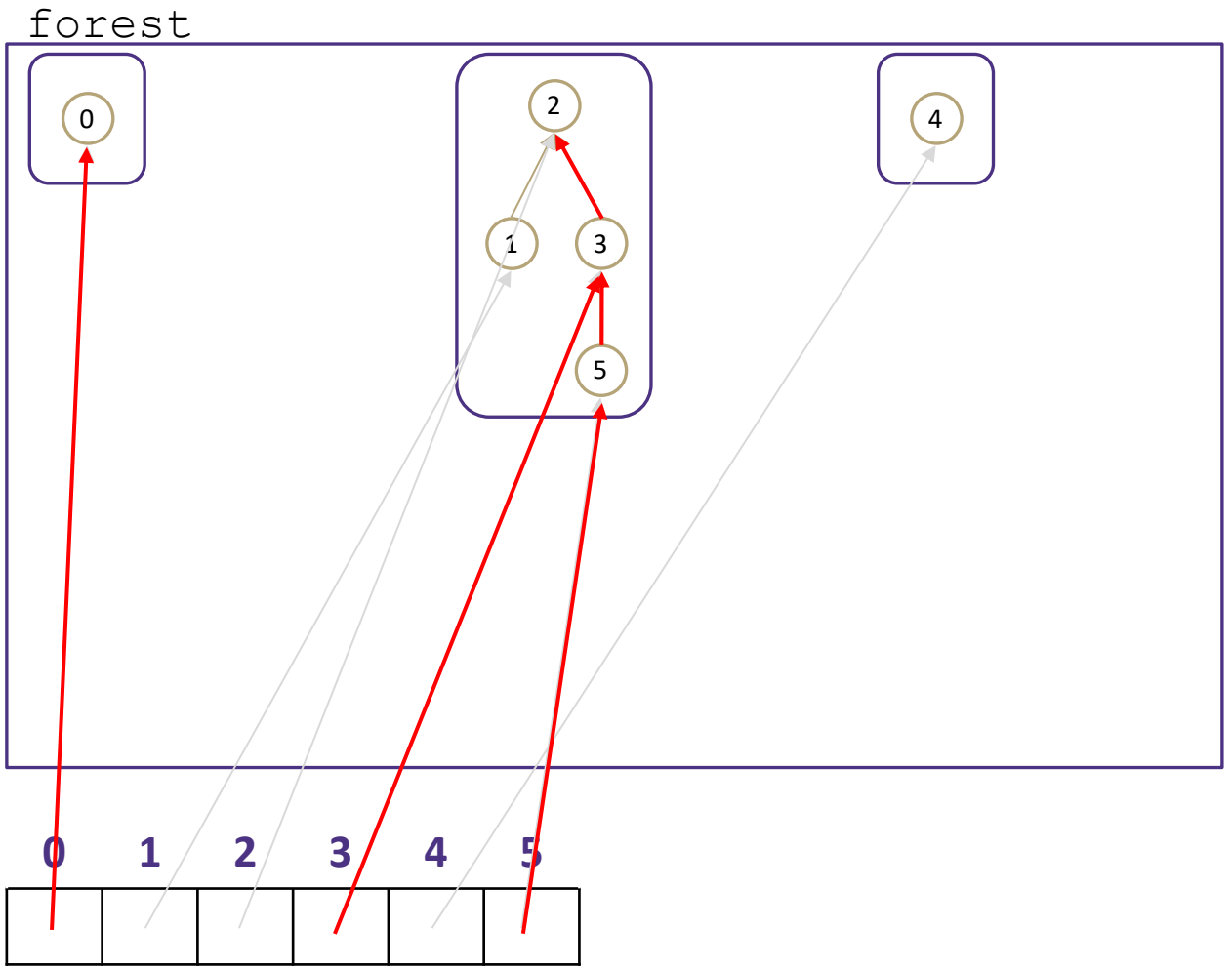
`union(x, y)`-append tree with y as a child of tree with x

Implement 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

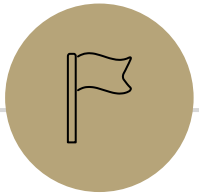
`union(x, y)`-append tree with y as a child of tree with x

Worst case runtime of findSet?

$\Theta(n)$

Worst case runtime of union?

$\Theta(n)$ – union has to call find!



Appendix: MST Properties, Another MST Application

Why do all of these MST Algorithms Work?

MSTs satisfy two very useful properties:

Cycle Property: The heaviest edge along a cycle is NEVER part of an MST.

Cut Property: Split the vertices of the graph any way you want into two sets A and B. The lightest edge with one endpoint in A and the other in B is ALWAYS part of an MST.

Whenever you add an edge to a tree you create exactly one cycle, you can then remove any edge from that cycle and get another tree out.

This observation, combined with the cycle and cut properties form the basis of all of the greedy algorithms for MSTs.

One More MST application

Let's say you're building a new building.

There are very important building donors coming to visit TOMORROW,

- and the hallways are not finished.

You have n rooms you need to show them, connected by the unfinished hallways.

Thanks to your generous donors you have n-1 construction crews, so you can assign one to each of that many hallways.

- Sadly the hallways are narrow and you can't have multiple crews working on the same hallway.

Can you finish enough hallways in time to give them a tour?

Minimum Bottleneck Spanning Tree Problem

Given: an undirected, weighted graph G
Find: A spanning tree such that the weight of the maximum edge is minimized.

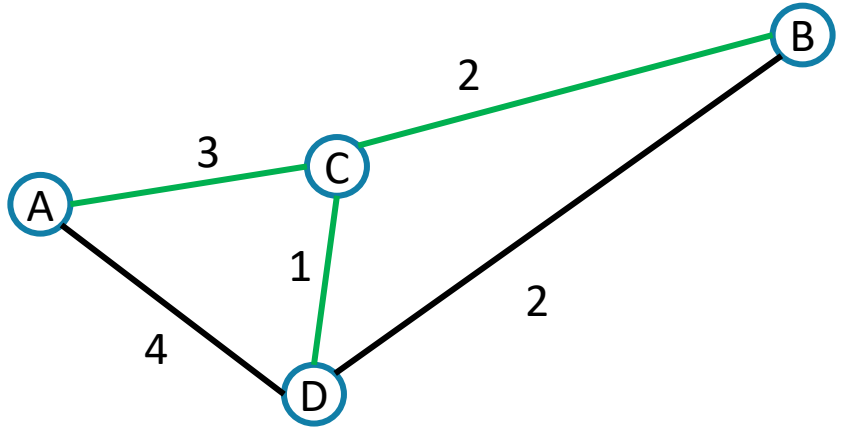
MSTs and MBSTs

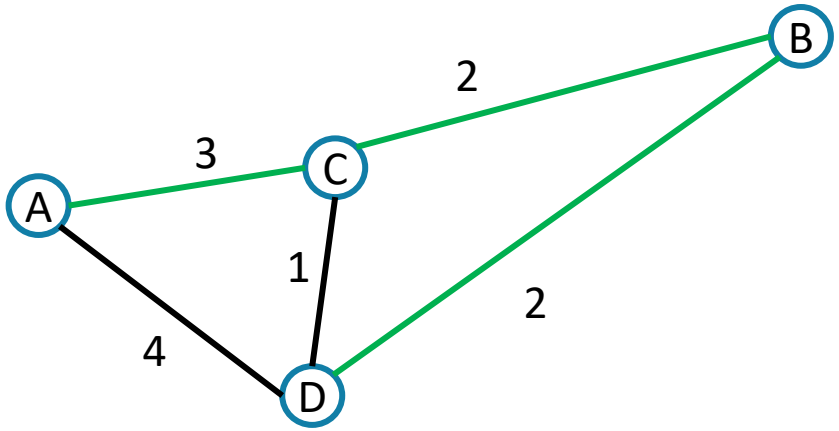
Minimum Spanning Tree Problem

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.

Minimum Bottleneck Spanning Tree Problem

Given: an undirected, weighted graph G
Find: A spanning tree such that the weight of the maximum edge is minimized.





Graph on the right is a minimum bottleneck spanning tree, but not a minimum spanning tree.

Finding MBSTs

Algorithm Idea: want to use smallest edges. Just start with the smallest edge and add it if it connects previously unrelated things (and don't if it makes a cycle).

Hey wait…that's Kruskal's Algorithm!

Every MST is an MBST (because Kruskal's can find any MST when looking for MBSTs)

but not vice versa (see the example on the last slide).

If you need an MBST, any MST algorithm will work.

There are also some specially designed MBST algorithms that are *faster* (see Wikipedia)

Takeaway: When you're modeling a problem, be careful to really understand what you're looking for. There may be a better algorithm out there.