Lecture 24: Disjoint Sets

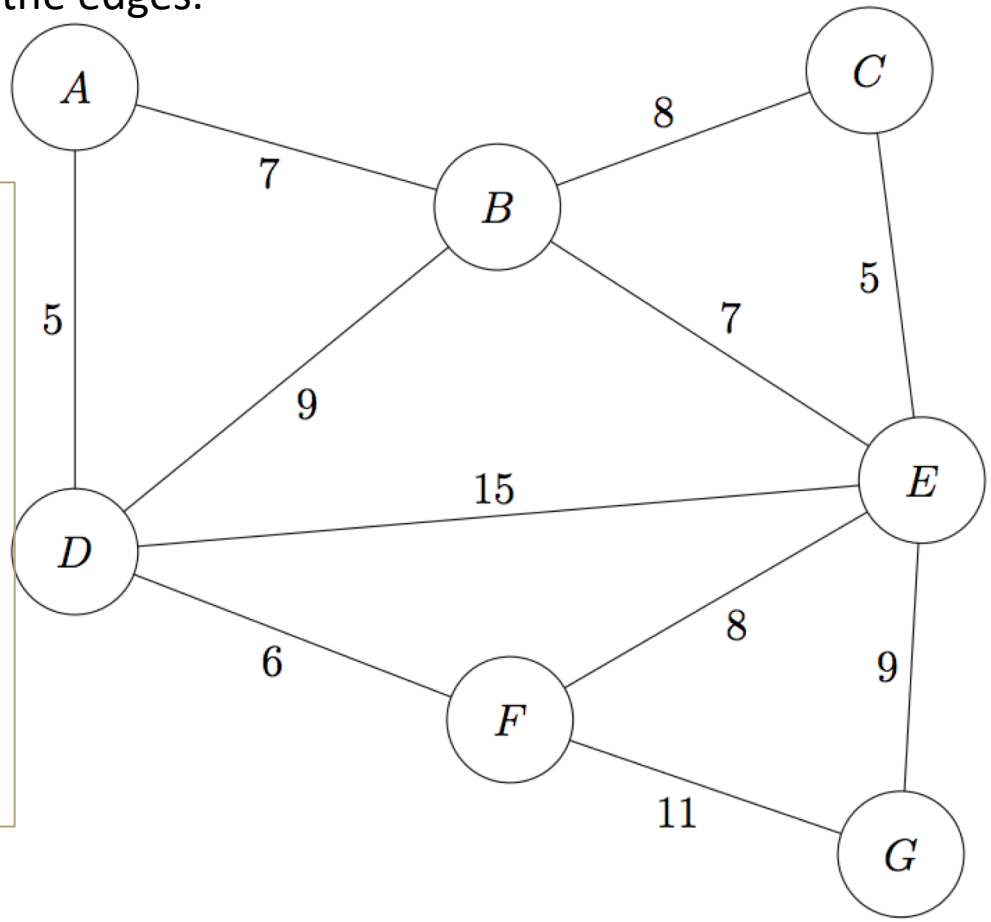
CSE 373: Data Structures and Algorithms

Warmup

Run Kruskal's algorithm on the following graph to find the MST (minimum spanning tree) of the graph below. Recall the definition of a minimum spanning tree: a minimum-weight set of edges such that you can get from any vertex of the graph to any other on only those edges. The set of these edges form a valid tree in the graph. Below is the provided pseudocode for Kruksal's algorithm to choose all the edges.

PollEv.com/373lecture

```
KruskalMST(Graph G)
   initialize each vertex to be an independent
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
      }
   }
```



Announcements

- Kasey out today (no Kasey 2:30 office hours)
- Hw6 released, due next Wednesday
- Hw7 partner form out now, due Monday 11:59pm.
  - If you do not fill out the partner form out on time, Brian will be sad because he has to do more work unnecessarily to fix it
- No office hours Monday (Memorial Day)

What are we doing today?

- Disjoint Set ADT

- Implementing Disjoint Set

Disjoint Set is honestly a very specific ADT/Data structure that has pretty limited realistic uses … but it's exciting because:

- is a cool recap of topics / touches on a bunch of different things we've seen in this course (trees, arrays, graphs, optimizing runtime, etc.)

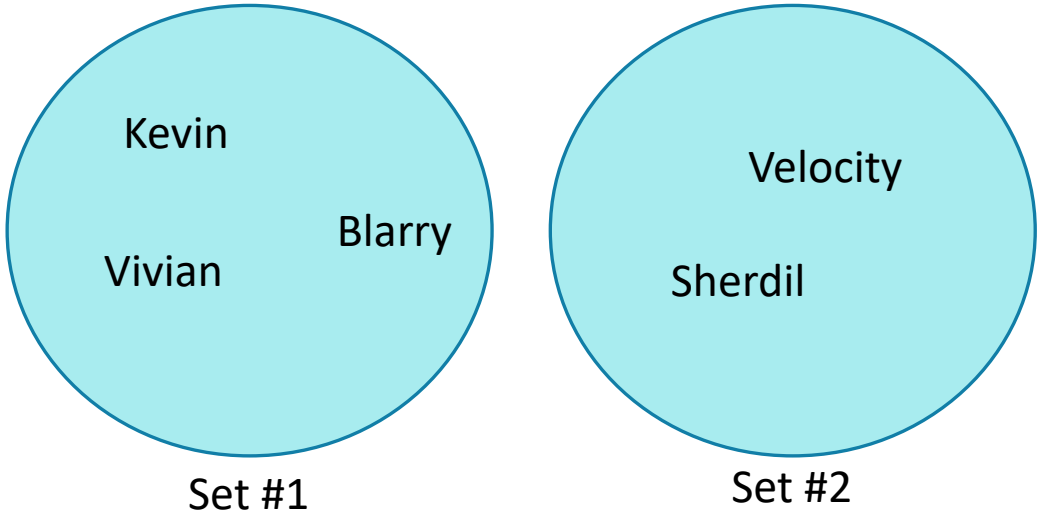
- it has a lot of details and is fairly complex – it doesn't seem like a plus at first, but after you learn this / while you're learning this…you've come along way since lists and being able to learn new complex data structures is a great skill to have built)

The Disjoint Set ADT

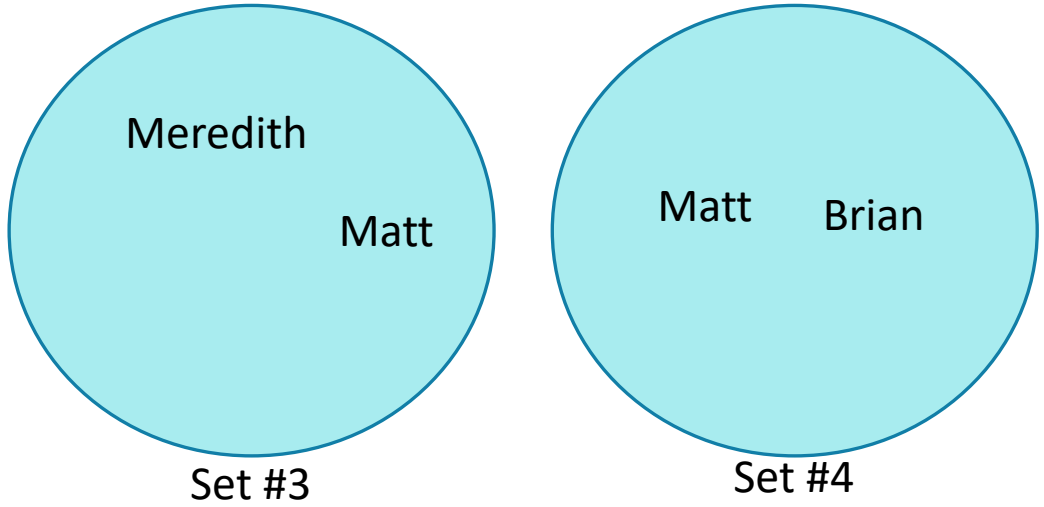
Disjoint Sets in mathematics

- “In mathematics, two **sets** are said to be **disjoint sets** if they have no element in common.” - Wikipedia

- disjoint = not overlapping



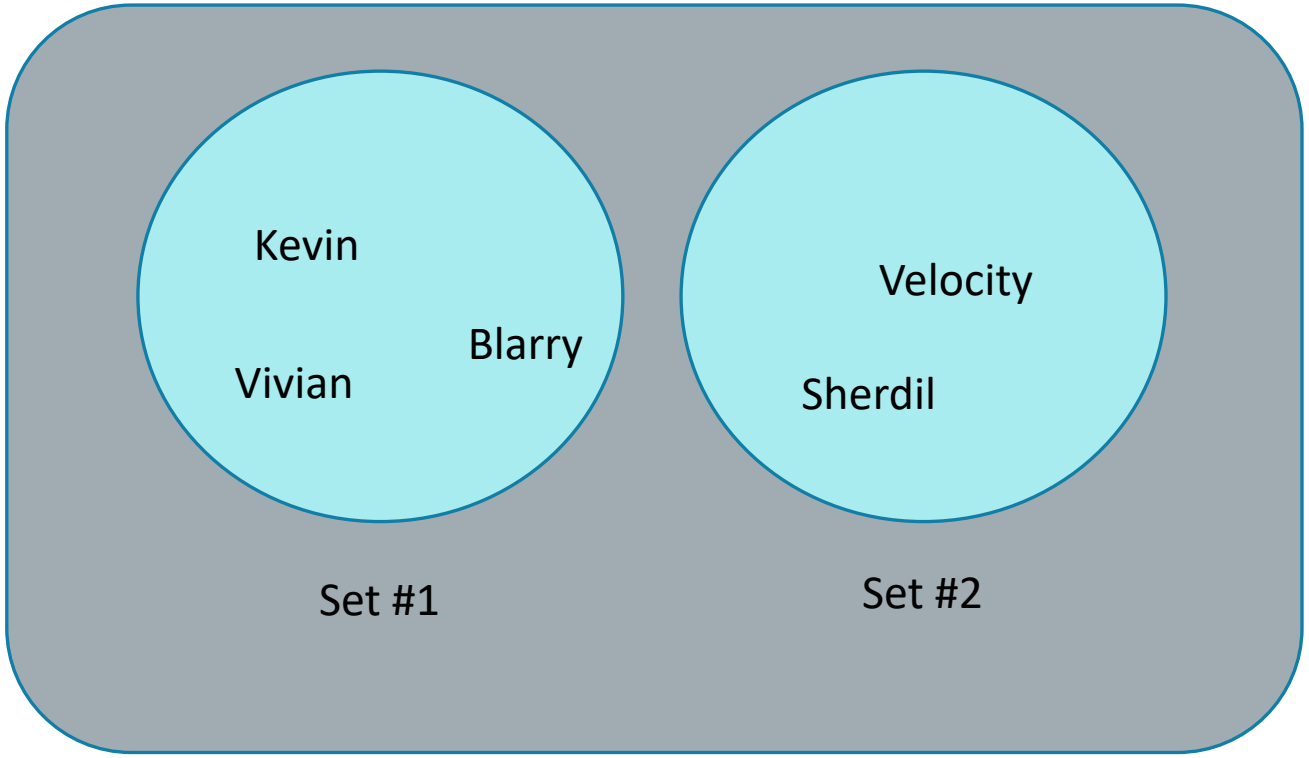
These two sets are disjoint sets



These two sets are not disjoint sets

Disjoint Sets in computer science

In computer science, a disjoint set keeps track of multiple "mini" disjoint sets (confusing naming, I know)



This overall grey blob thing is the actual disjoint set, and it's keeping track of any number of mini-sets, which are all disjoint (the mini sets have no overlapping values).

Note: this might feel really different than ADTs we've run into before. The ADTs we've seen before (dictionaries, lists, sets, etc.) just store values directly. But the Disjoint Set ADT is particularly interested in letting you group your values into sets and keep track of which particular set your values are in.

What methods does the DisjointSet ADT have?

Just 3 methods (and one of them is pretty simple!)

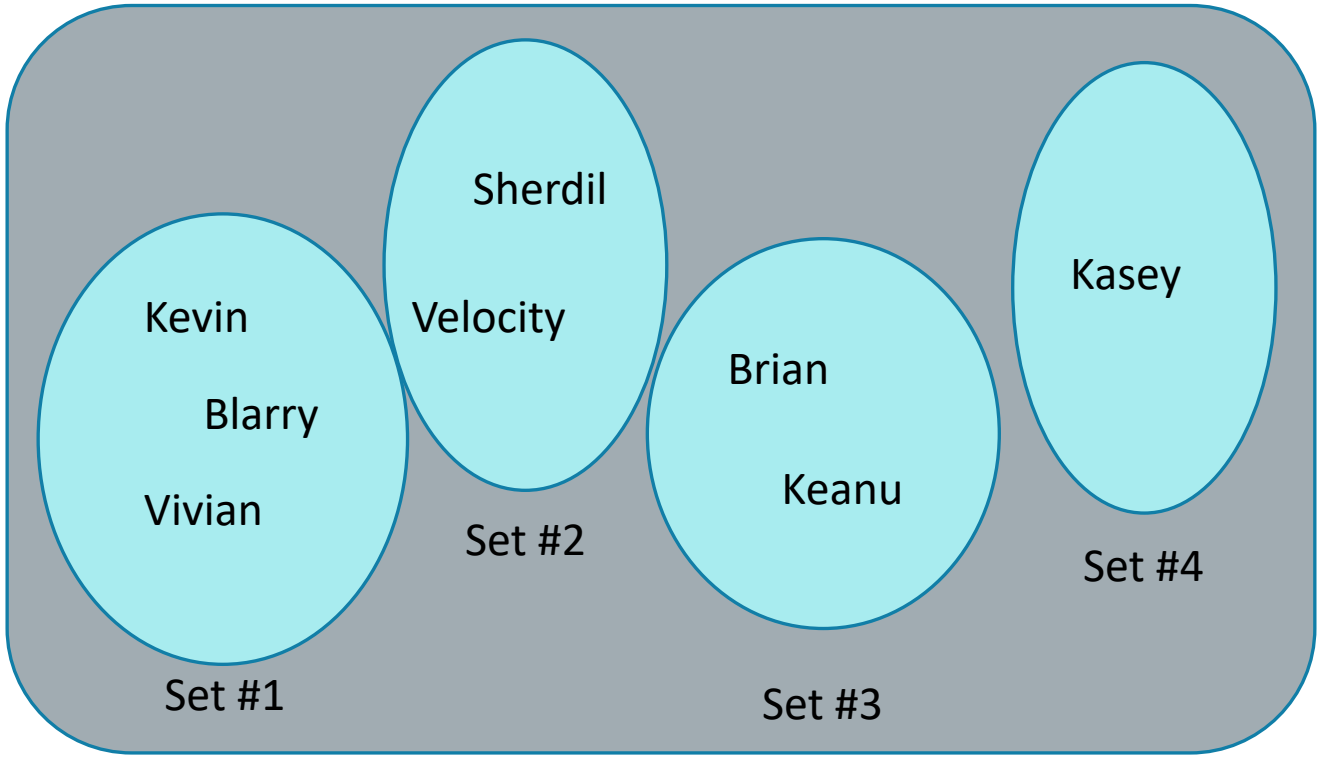
- findSet(value)
- union(valueA, valueB)
- makeSet(value)

findSet(value)

findSet(value) returns some indicator for which particular set the value is in. You can think of this as an ID. For Disjoint Sets, we often call this the **representative**.

Examples:

findSet(Brian)	3
findSet(Sherdil)	2
findSet(Velocity)	2
findSet(Kevin) == findSet(Blarry)	true



What methods does the Disjoint Set ADT have?

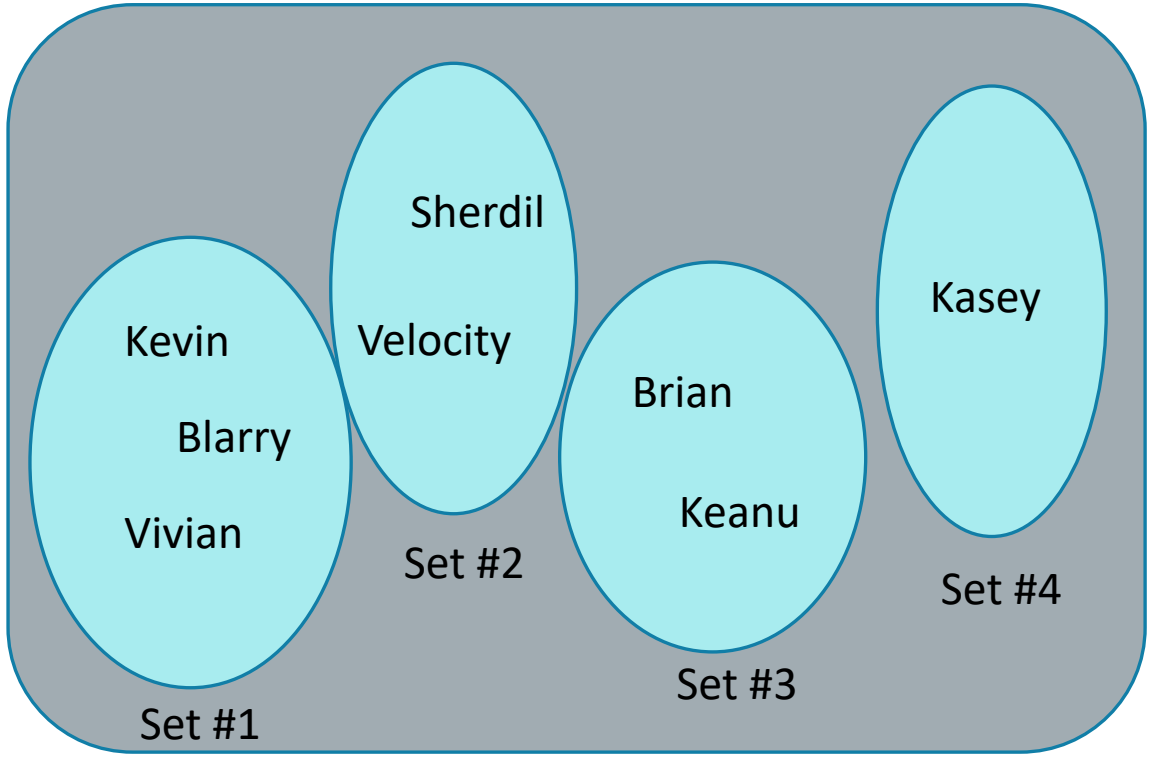
Just 3 methods (and one of them is pretty simple!)

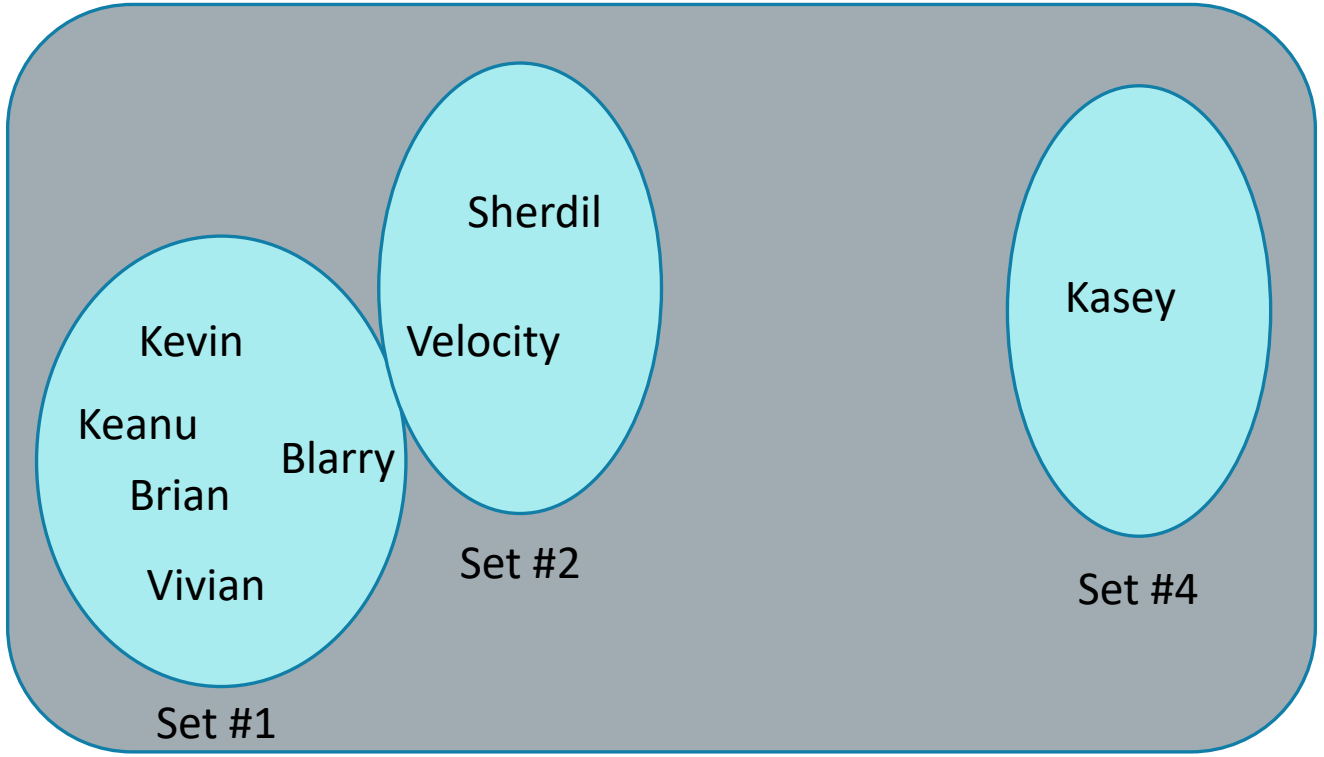
- findSet(value)
- union(valueA, valueB)
- makeSet(value)

union(valueA, valueB)

union(valueA, valueB) merges the set that A is in with the set that B is in. (basically add the two sets together into one)

Example: union(Blarry,Brian)





What methods does the DisjointSet ADT have?

Just 3 methods (and one of them is pretty simple!)

- findSet(value)
- union(valueA, valueB)
- makeSet(value)

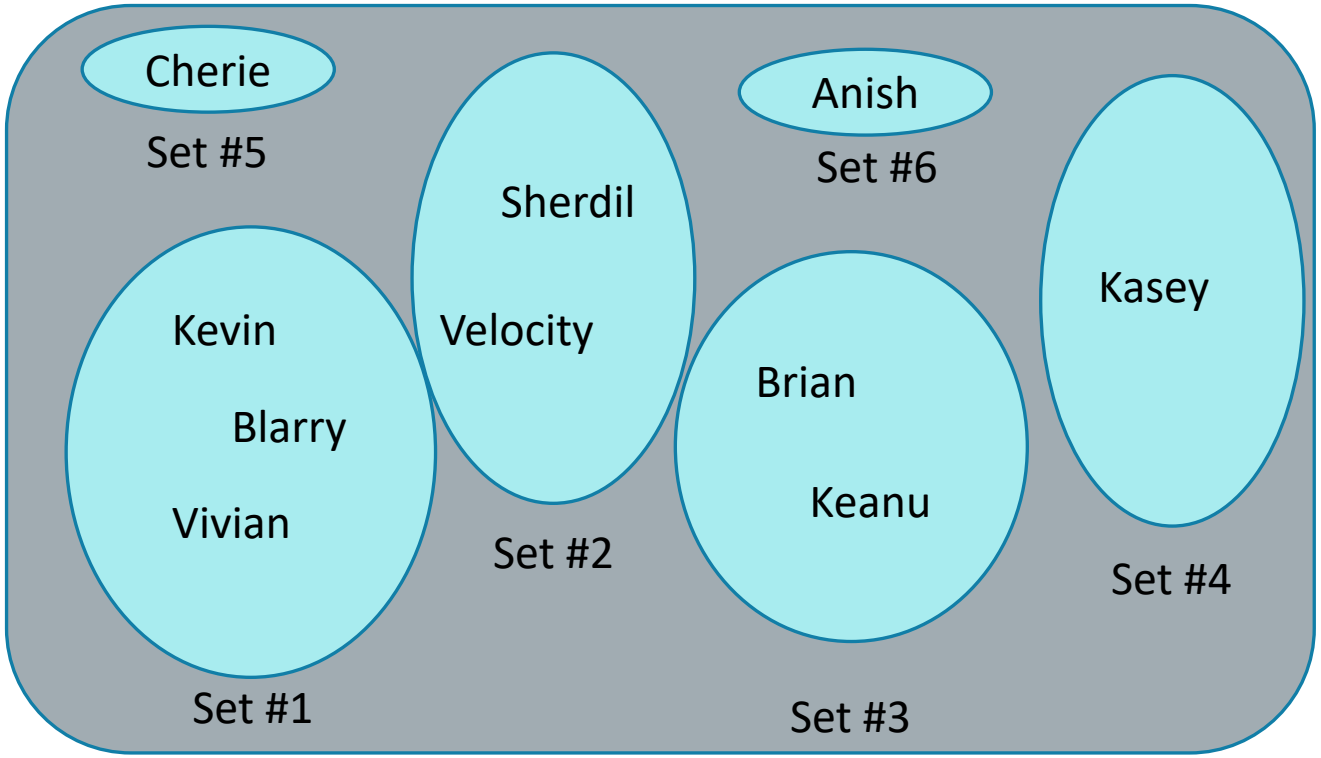
makeSet(value)

makeSet(value) makes a new mini set that just has the value parameter in it.

Examples:

makeSet(Cherie)

makeSet(Anish)



Disjoint Set ADT Summary

Disjoint-Set ADT

state

Set of Sets

- **Mini sets are disjoint:** Elements must be unique across mini sets
- No required order
- Each set has id/representative

behavior

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

findSet(value) – looks up the set containing the value, returns id/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.

Why are we doing this again?

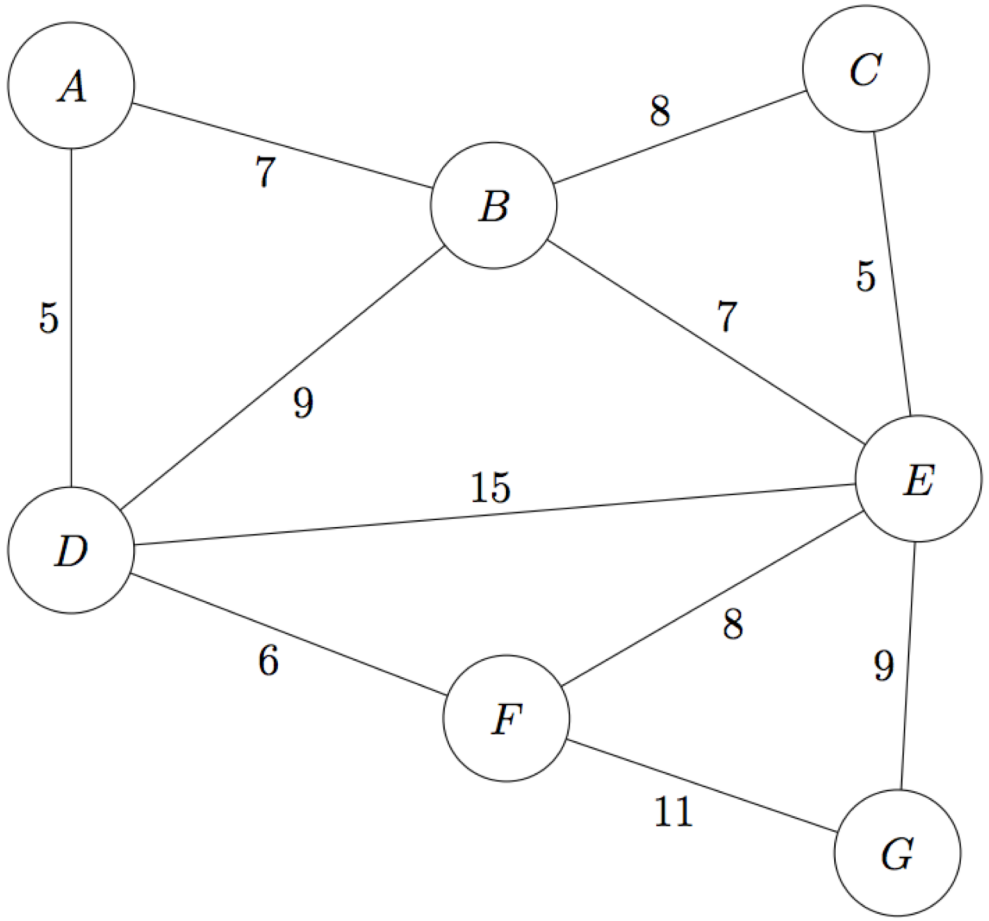
Kruskal's Algorithm Implementation

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

```
KruskalMST(Graph G)
   foreach (V : G.vertices) {
      makeSet(v);
   }
   sort the edges by weight
   foreach(edge (u, v) in sorted order){
      if(findSet(v) is not the same as findSet(u)
         union(u, v)
         add (u, v) to the MST
      }
   }
```

Kruskal's with disjoint sets on the side example

```
KruskalMST(Graph G)
   foreach (V : G.vertices) {
      makeSet(v);
   }
   sort the edges by weight
   foreach(edge (u, v) in sorted order){
      if(findSet(v) is not the same as
         findSet(u)){
         union(u, v)
      }
   }
```



Why are we doing this again? (continued)

Disjoint Sets help us **manage groups of distinct values**.

This is a common idea in graphs, where we want to keep track of different connected components of a graph.

In Kruskal's, if each connected-so-far-island of the graph is its own mini set in our disjoint set, we can easily check that we don't introduce cycles. If we're considering a new edge, we just check that the two vertices of that edge are in different mini sets by calling findSet.

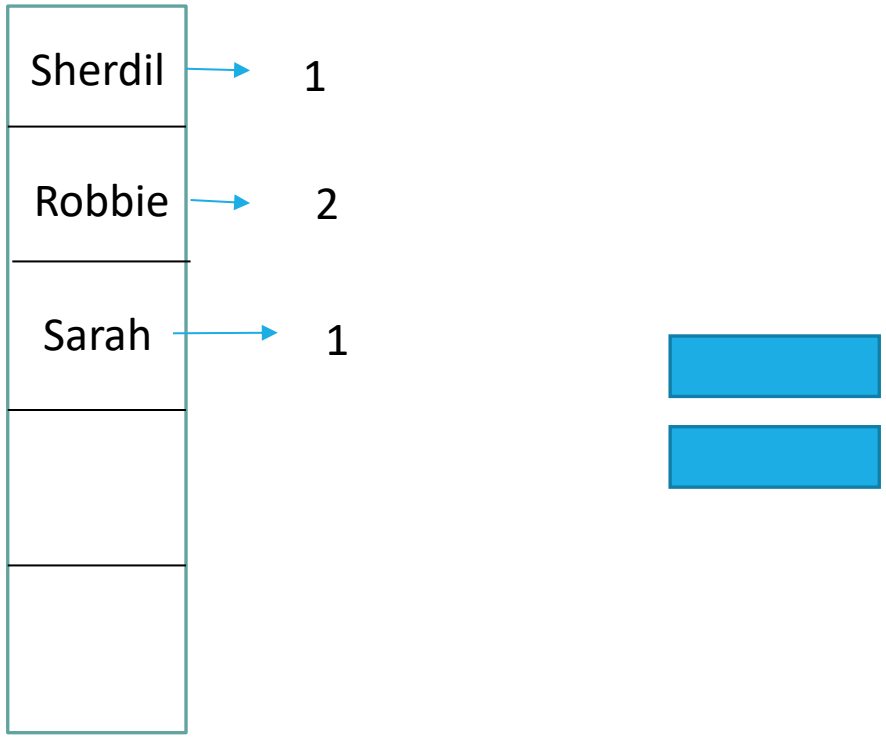
1 min break

Take a second to review notes with your neighbors, ask questions, try to clear up any confusions you have… we'll group back up and see if there are still any unanswered questions then!

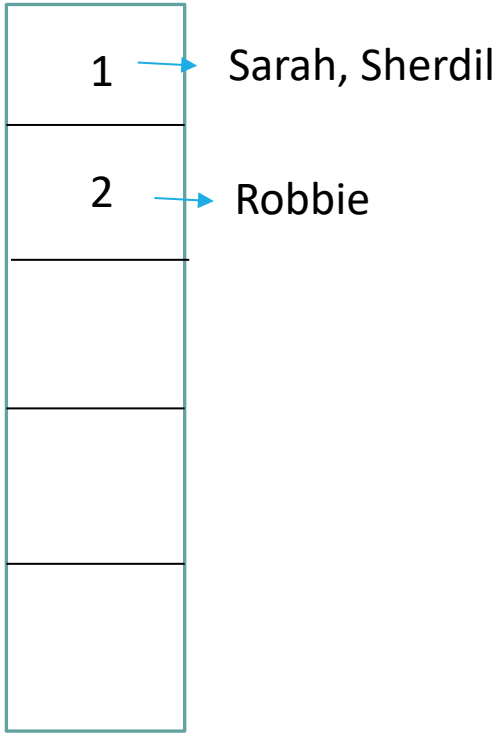
Implementing Disjoint Set

Implementing Disjoint Set with Dictionaries

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



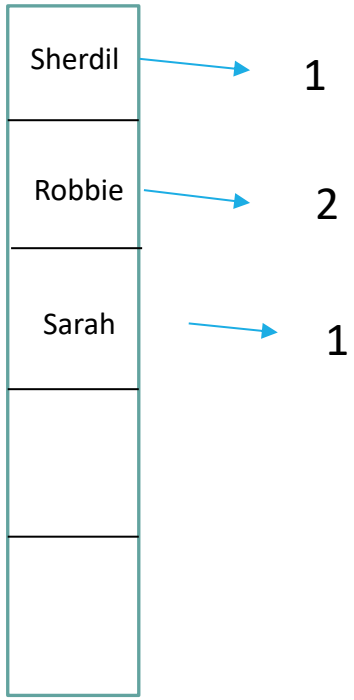
Approach 2: dictionary of ID/representative of set -> all the values in that set



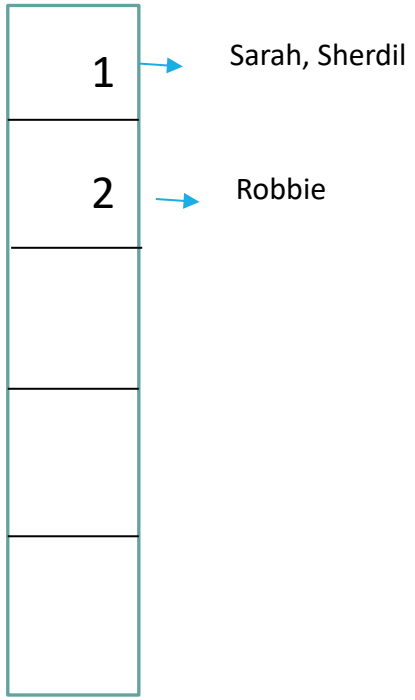
Exercise (1.5 min)

Calculate the worst case Big O 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

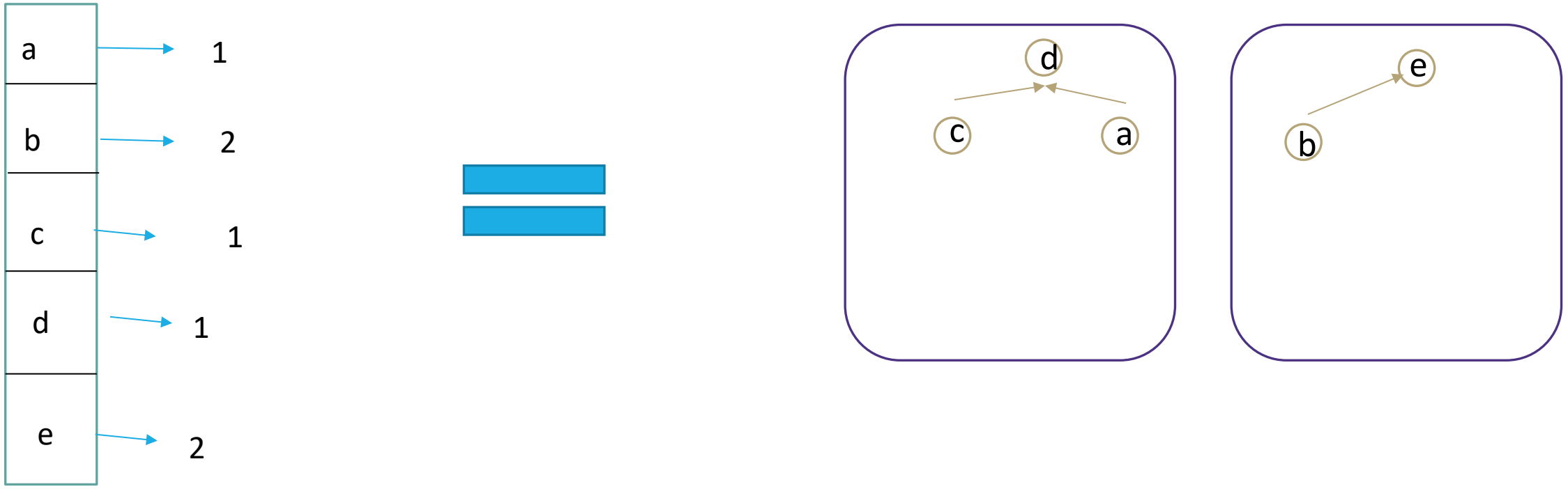


	approach 1	approach 2
makeSet(value)	O(1)	O(1)
findSet(value)	O(1)	O(n)
union(valueA, valueB)	O(n)	O(n)

Implementing Disjoint Set with Trees (and dictionaries) (1)

Each mini-set is now represented as a separate tree.

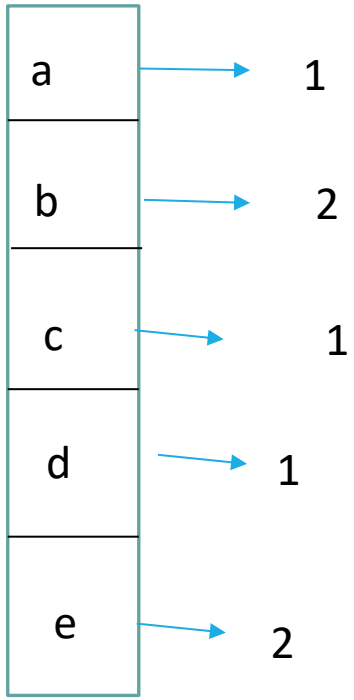
(Note: using letters/numbers from now on as the values because they're easier to fit inside the nodes)

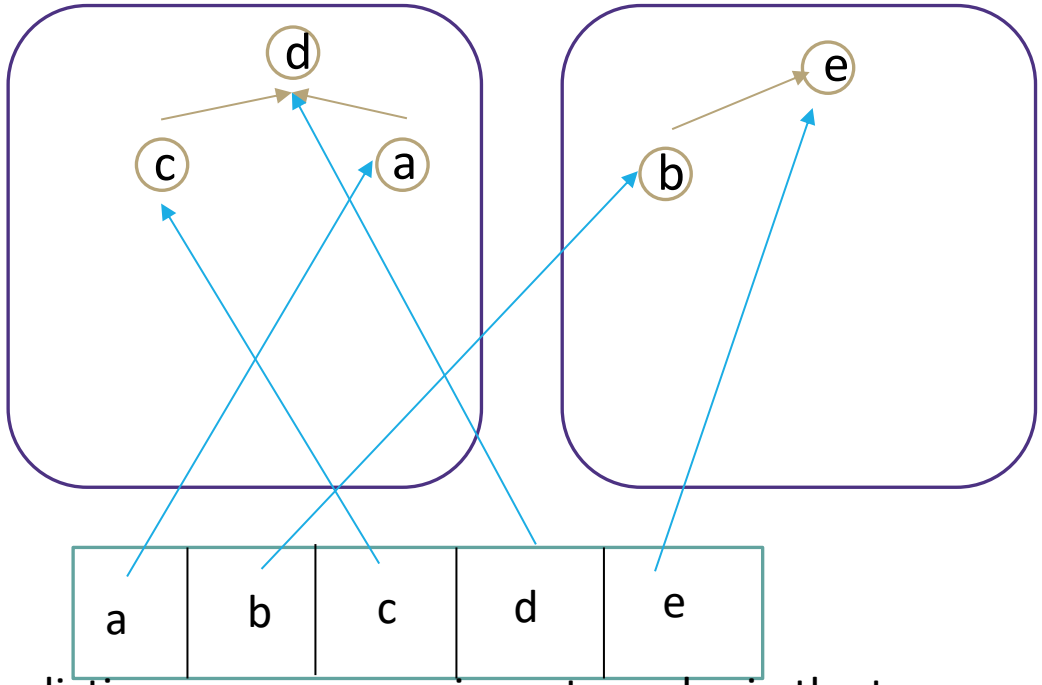


Implementing Disjoint Set with Trees (and dictionaries) (1)

Each mini-set is now represented as a different tree.

(Note: using letters/numbers from now on as the values because they're easier to fit inside the nodes)



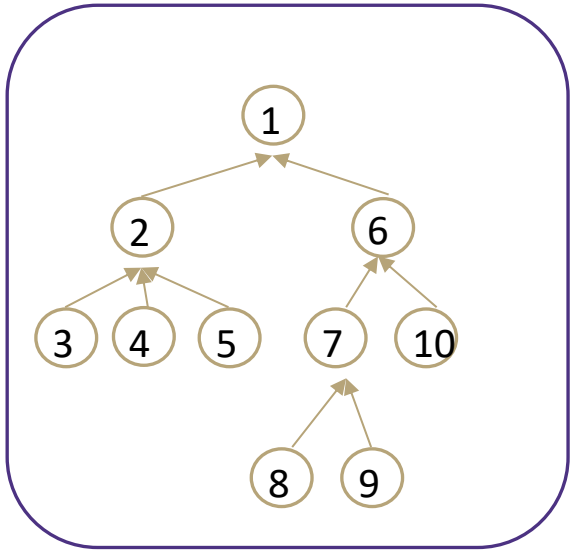


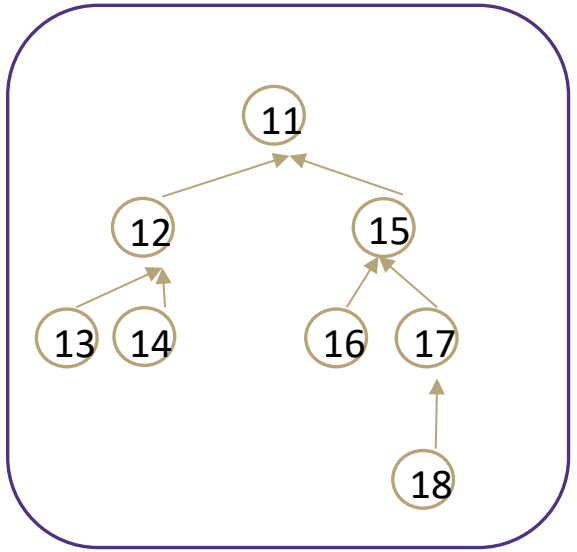
dictionary so you can jump to nodes in the tree

Implementing Disjoint Set with Trees (and dictionaries) (2)

union(valueA, valueB) -- the method with the problem runtime from before -- should look a lot easier in terms of updating the data structure – all we have to do is change one link so they're connected.

What should we change? If we change the root of one to point to the other tree, then all the lower things will be updated. It turns out it will be most efficient if we have the root point to the other tree's root.

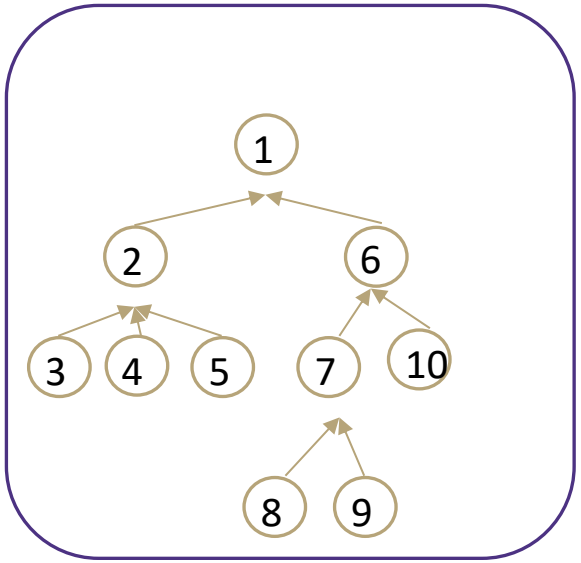




Implementing Disjoint Set with Trees (and dictionaries) (3)

findSet has to be different though …

They all have access to the root node because all the links point up – we can use the root node as our id / representative. For example:



findSet(5)

findSet(9)

they're in the same set because they have the same representative!

Seems great so far but let's abuse some stuff

makeSet(a)

makeSet(b)

makeSet(c)

makeSet(d)

makeSet(e)

union(a, b)

union(a, c)

union(a, d)

union(a, e)

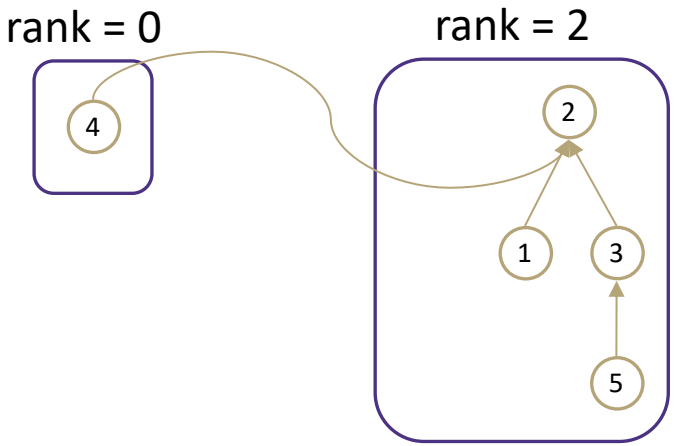
findSet (a) – how long will this take? Could turn into a linked list where you might have to start at the bottom and loop all the way to the top.

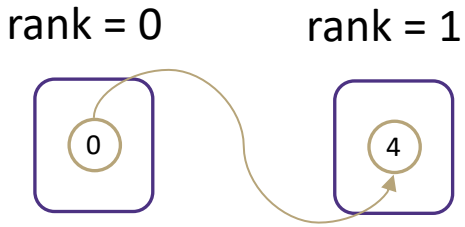
Improving union

Problem: Trees can be unbalanced (and look linked-list-like) so our findSet runtime becomes closer to N

Solution: Union-by-rank!

- let rank(x) be a number representing the upper bound of the height of x so rank(x) >= height(x)
- Keep track of rank of all trees
- When unioning make the tree with larger rank the root
- If it's a tie, pick one randomly and increase rank by one

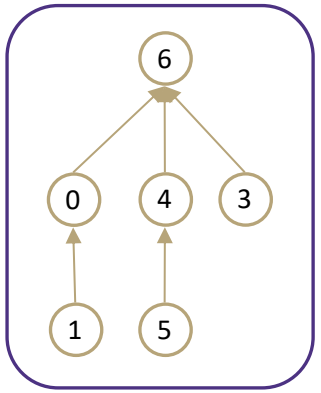




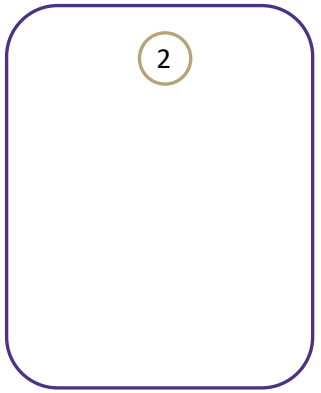
Practice

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.

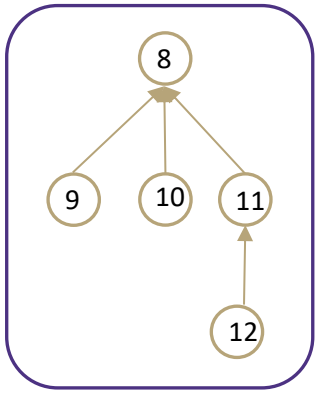
rank = 2



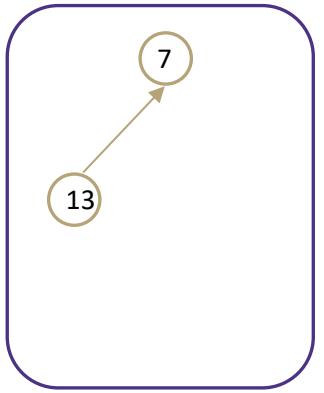
rank = 0



rank = 2



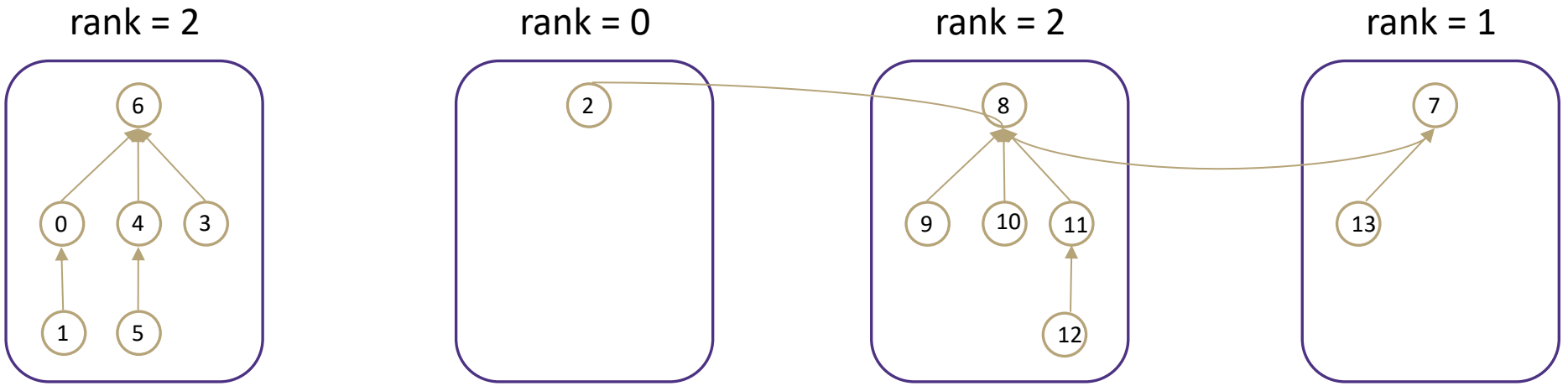
rank = 1




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

Practice

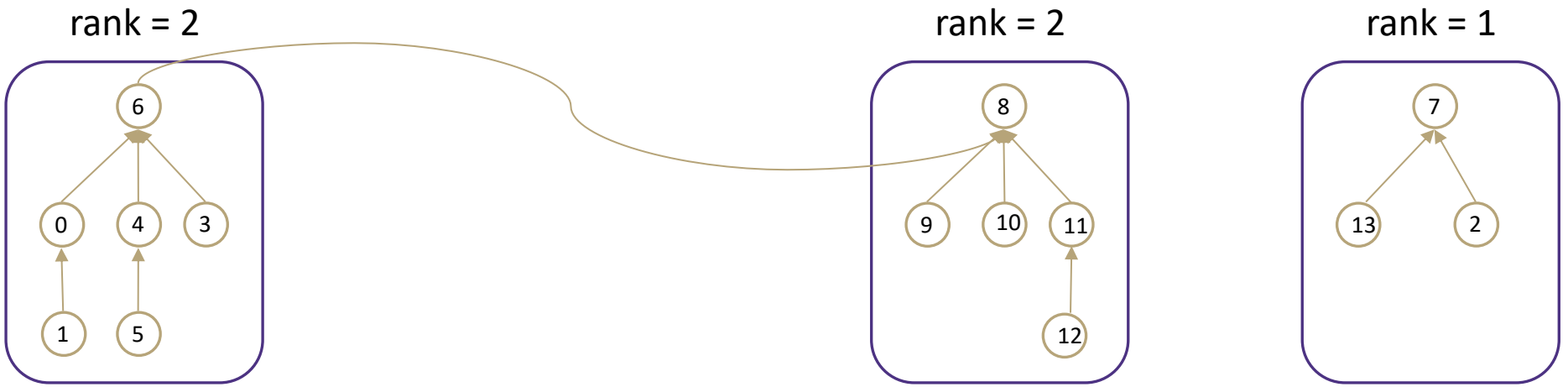
Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-rank” optimization. Draw the forest at each stage with corresponding ranks for each tree.




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

Practice

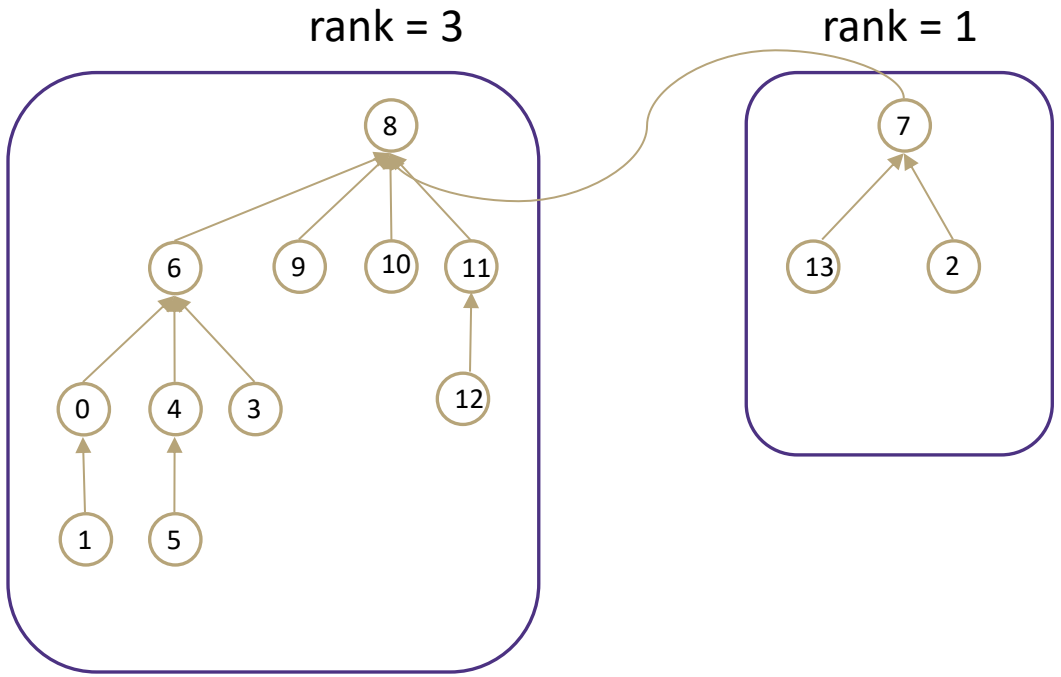
Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-rank” optimization. Draw the forest at each stage with corresponding ranks for each tree.




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

Practice

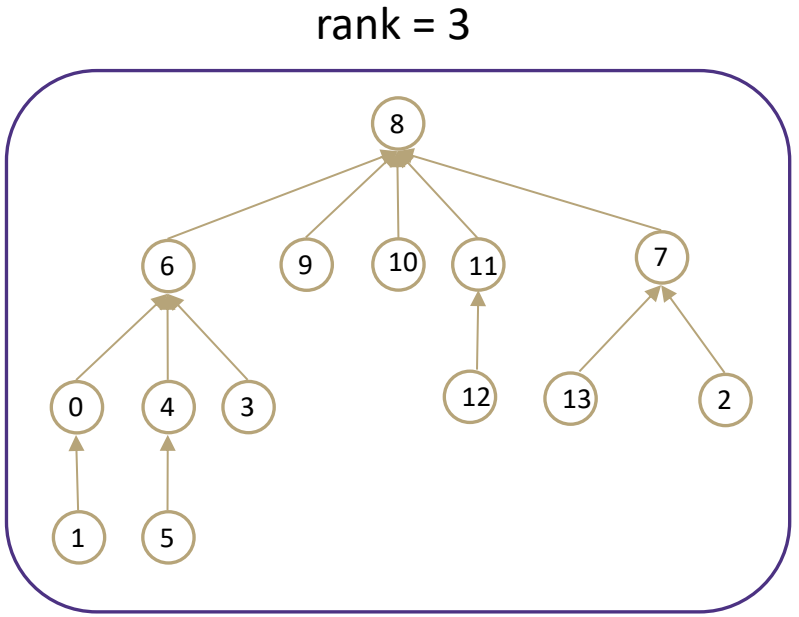
Given the following disjoint-set what would be the result of the following calls on union if we add the “union-by-rank” optimization. Draw the forest at each stage with corresponding ranks for each tree.




```
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)
```

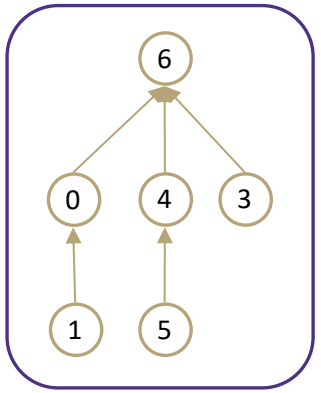
Does this improve the worst case runtimes?

findSet is more likely to be O(log(n)) than O(n)

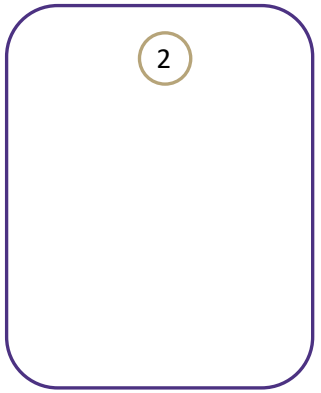
Exercise (2 min)

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.

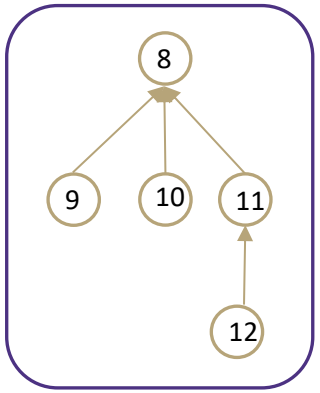
rank = 2



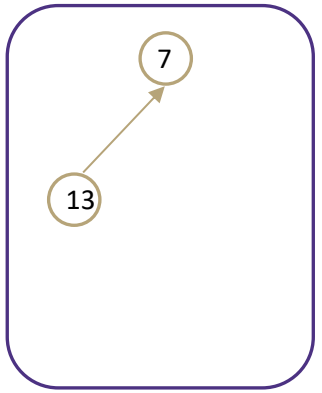
rank = 0



rank = 2



rank = 1




```
union(5, 8)
union(1, 2)
union(7, 3)
```

Improving findSet()

Problem: Every time we call findSet() you must traverse all the levels of the tree to find representative. If there are a lot of levels (big height), this is more inefficient than need be.

Solution: Path Compression

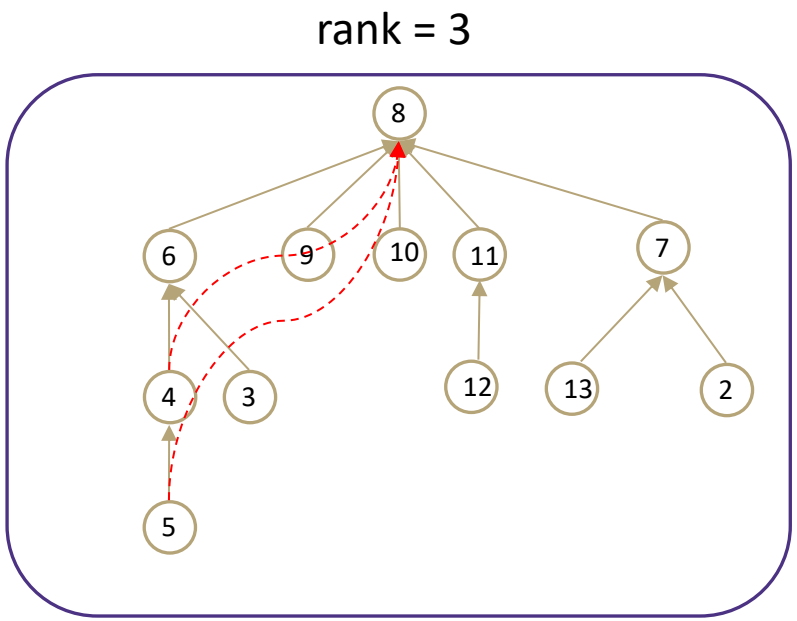
- Collapse tree into fewer levels by updating parent pointer of each node you visit
- Whenever you call findSet() update each node you touch's parent pointer to point directly to overallRoot

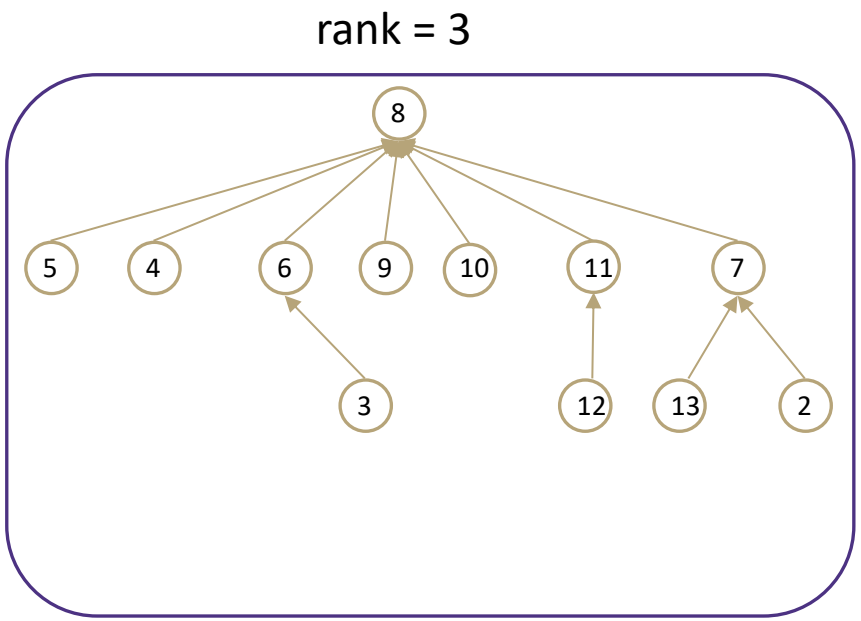
```
findSet(5)

findSet(4)
```

Does this improve the worst case runtimes?

findSet is more likely to be O(1) than O(log(n))





Exercise if time?

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 Disjoint Set Runtime

makeSet(x)

Without Optimizations	**O(1)**
With Optimizations	**O(1)**

findSet(x)

Without Optimizations	**O(n)**
With Optimizations	**Best case: O(1) Worst case: O(logn)**

union(x, y)

Without Optimizations	**O(n)**
With Optimizations	**Best case: O(1) Worst case: O(logn)**

Next time

- union should call findMin to get access to the root of the trees
- why rank is an approximation of height
- array representation instead of tree
- more practice!