Lecture 18: Disjoint Sets

CSE 373: Data Structures and Algorithms

Administrivia

Project 4 Due Wednesday May 20th

Exercise 4 due Friday May 15th

Grades Posted

- Exercise 1
- Project 0
- Project 1
- Project 2
- Project 3
- Midterm 1

Zach & Kasey Grade Office hours

- Today, Monday May 11 9:30-11:30am PDT
- Tuesday May 12 5:45-7:15pm PDT

Roadmap

- Disjoint Sets ADT
- Context, examples
- Different implementations (most of them are just optimizations of the previous)!

1. QuickFind implementation (HashMap based)
2. QuickUnionTrees
3. QuickUnionBySizeTrees
4. QuickUnionBySizeCompressingTrees
5. ArrayQuickUnionBySizeCompressing

DisjointSets

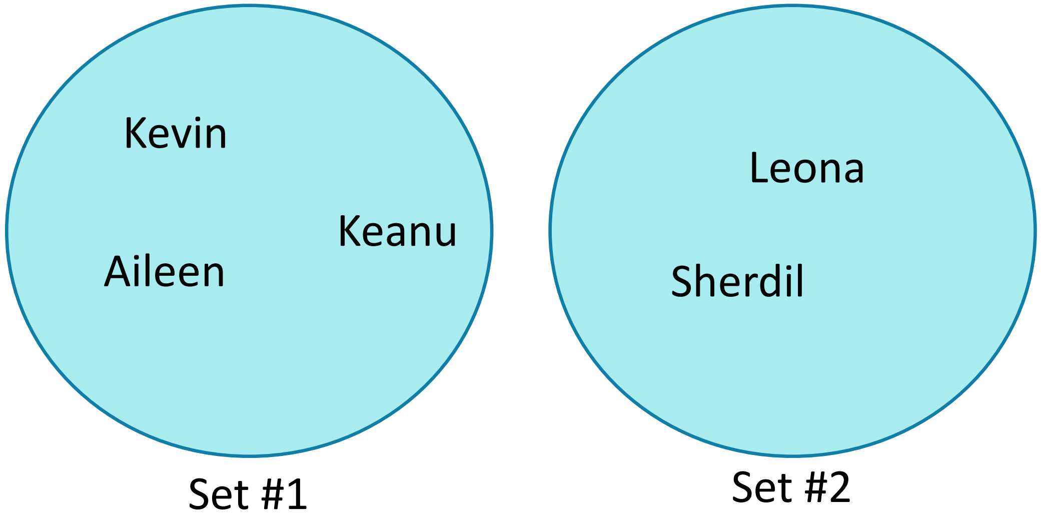
- are a cool recap of topics / touches on a bunch of different things we've seen in this course (trees, arrays, graphs, optimizing runtime, etc.)

- have 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)

DisjointSets 😍 😍 is Zach's favorite ADT/data structure that we talk about in this class (if you ever need to bribe Zach just start a conversation w zach about DisjointSets)

Disjoint Sets in mathematics

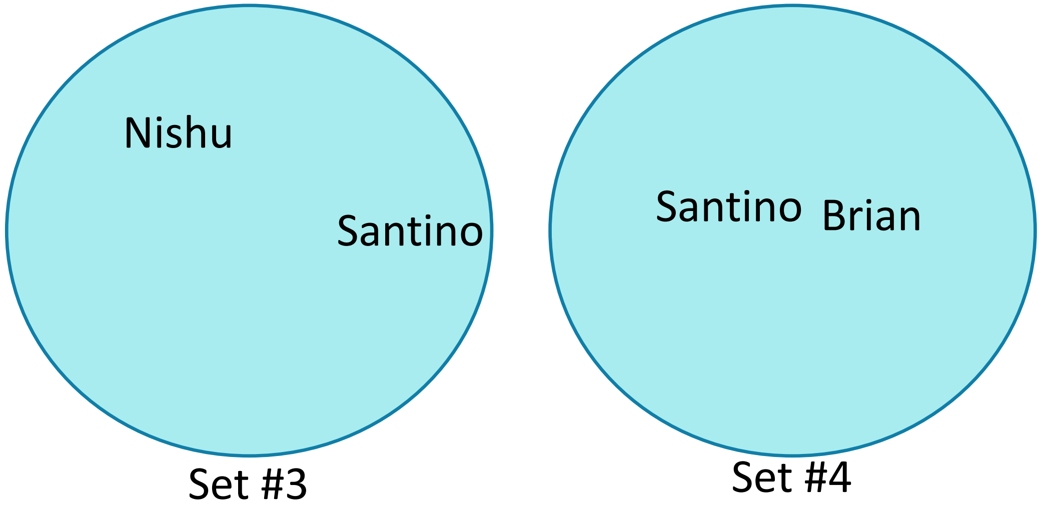
- “In mathematics, two **sets** are said to be **disjoint sets** if they have no element in common.” - Wikipedia
- disjoint = not overlapping



Kevin
Keanu
Aileen
Set #1

Leona
Sherdil
Set #2

These two sets are disjoint sets



Nishu
Santino
Set #3

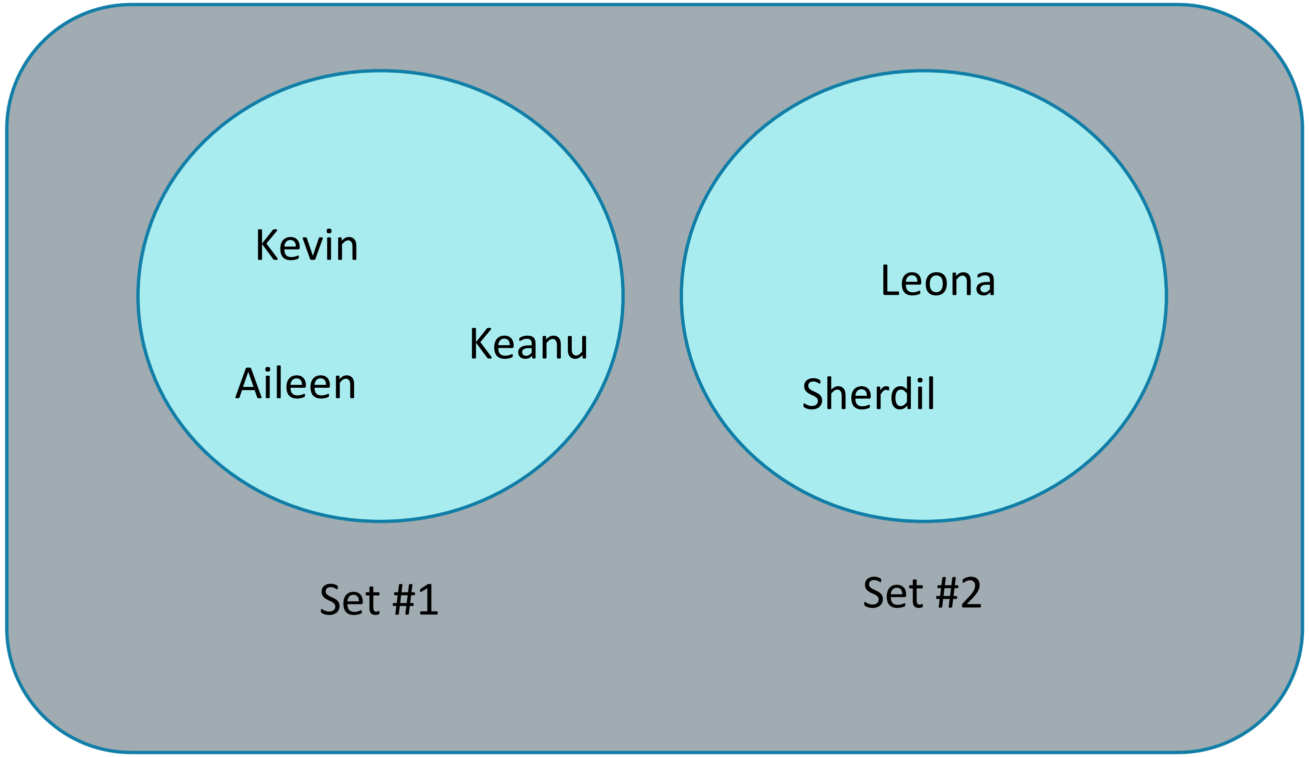
Santino Brian
Set #4

These two sets are not disjoint sets

Disjoint Sets in computer science

new ADT!

In computer science, disjointsets can refer to this ADT/data structure that keeps track of the multiple "mini" sets that are disjoint (confusing naming, I know)



This overall grey blob thing is the actual disjoint sets, 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.

DisjointSets ADT methods

Just 3 methods (and makeSet is pretty simple!)

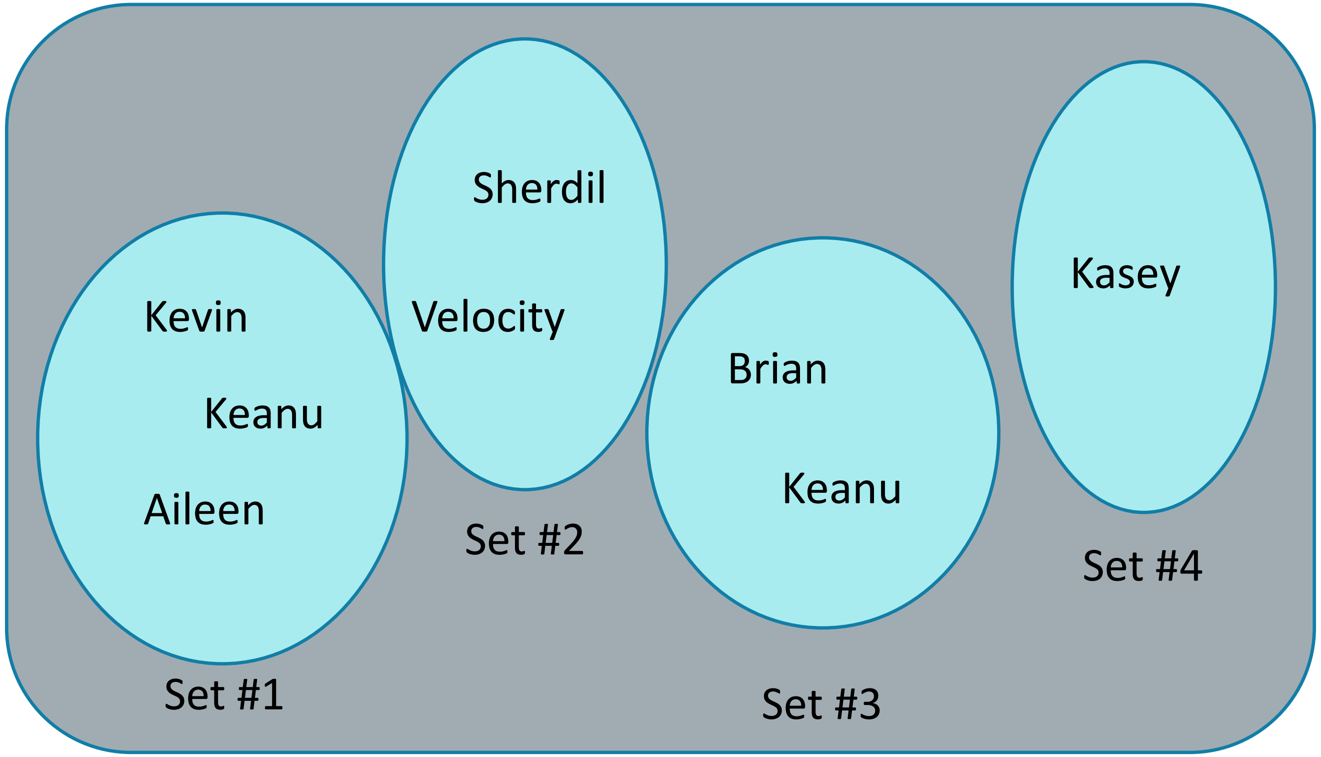
- findSet(value)
- union(valueA, valueB)
- makeSet(value)

findSet(value)

findSet(value) returns some ID for which particular set the value is in. For Disjoint Sets, we often call this the **representative** (as it’s a value that represents the whole set).

Examples:

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

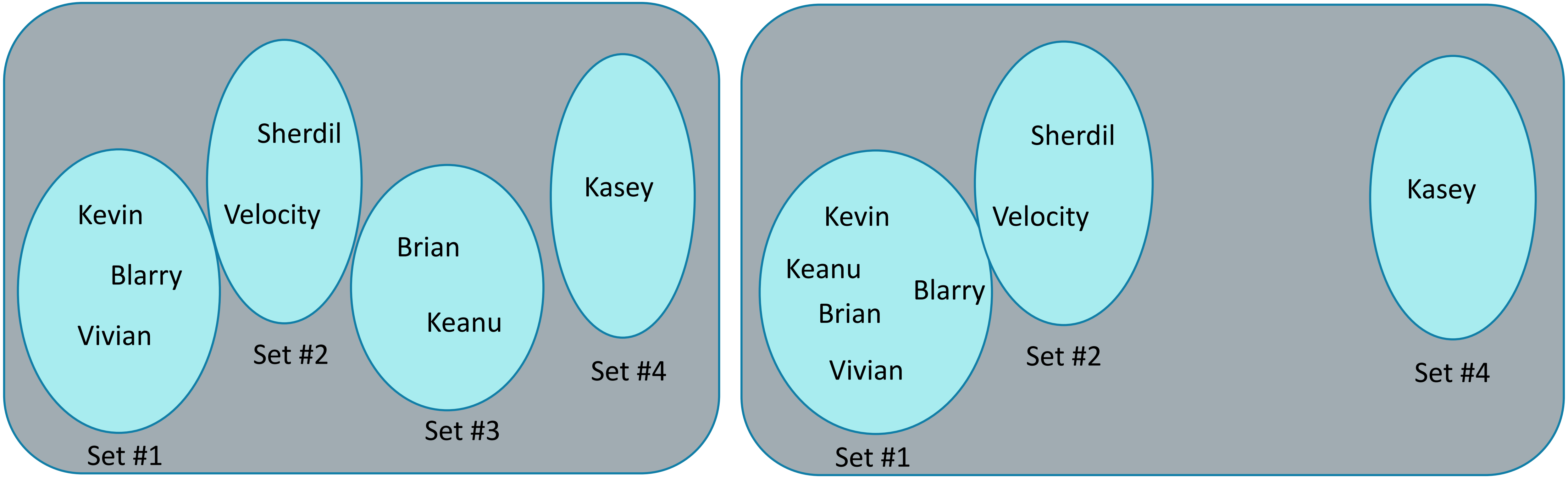


3

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)



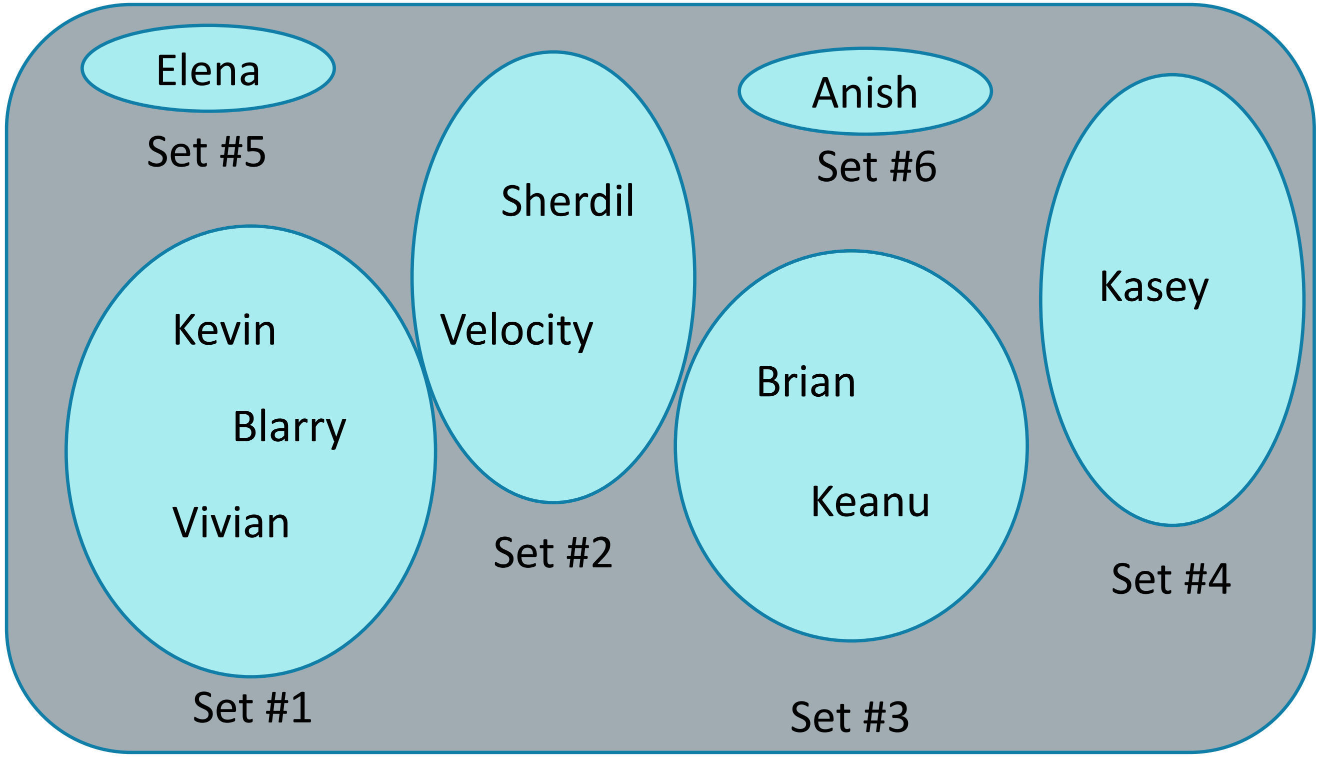
makeSet(value)

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

Examples:

makeSet(Elena)

makeSet(Anish)



Disjoint Sets ADT Summary

Disjoint-Sets 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.

Roadmap

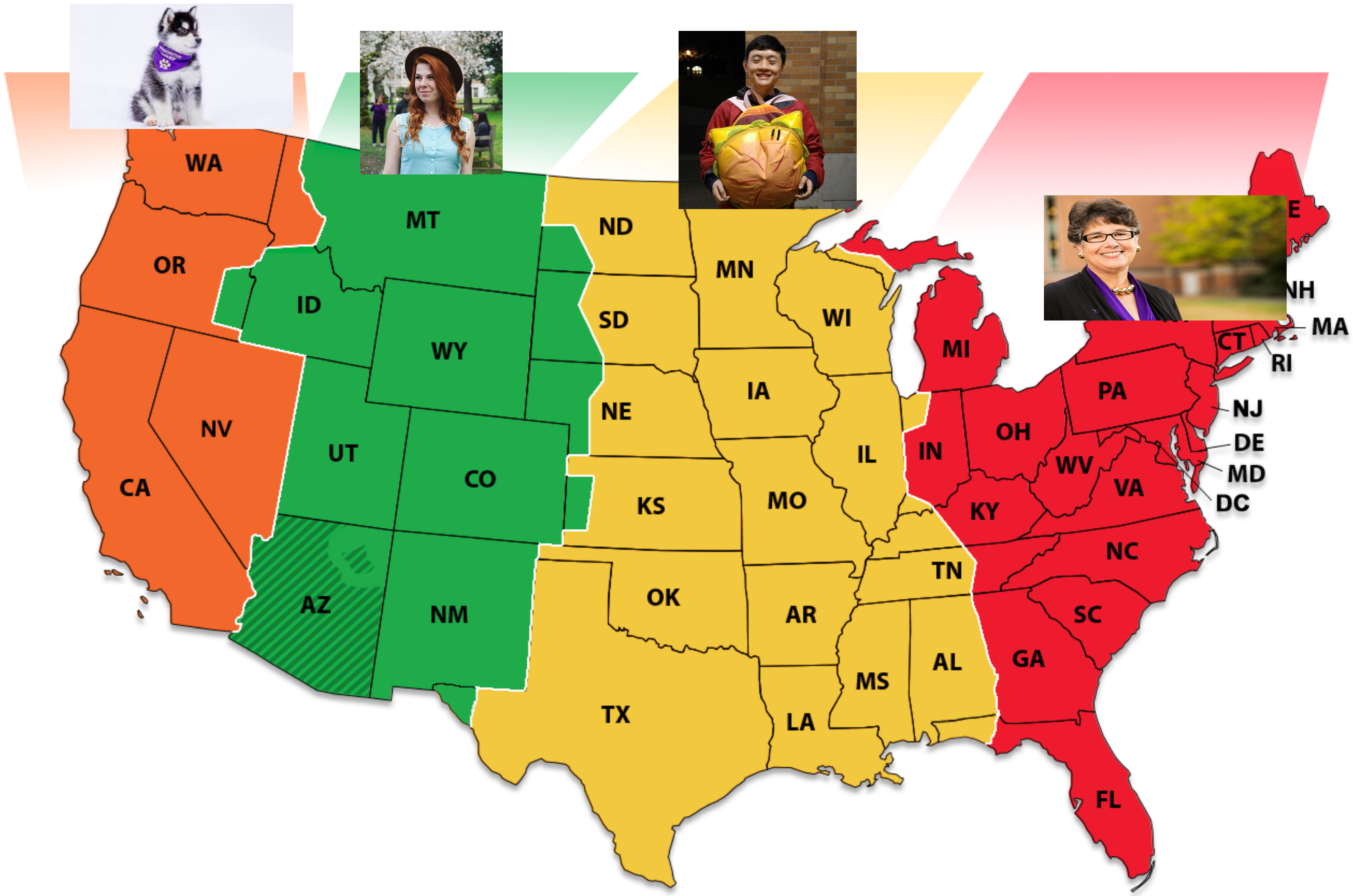
- Disjoint Sets ADT

- Context, examples

- Different implementations (most of them are just optimizations of the previous)! The data structures we talk about are going to be highly optimized for these interesting ADT methods we just talked about!

1. QuickFind implementation (HashMap based)
2. QuickUnionTrees
3. QuickUnionBySizeTrees
4. QuickUnionBySizeCompressingTrees
5. ArrayQuickUnionBySizeCompressing

DisjointSets example: running for President and tracking voters



Imagine that Dubs, Kasey, Zach, and Ana Mari are all running for president, and are currently dominating one region of the country in the polls and have secured that these states will vote for these candidates.

You could imagine that some common questions about this data is:

- what happens when Zach has to resign and wants his supporters to now join Kasey's campaign, and Ana Mari does the same for dubs? How do we keep track of the followers all moving over to a different group?
- You can imagine we track the voters ←→ nominee data with disjoint sets

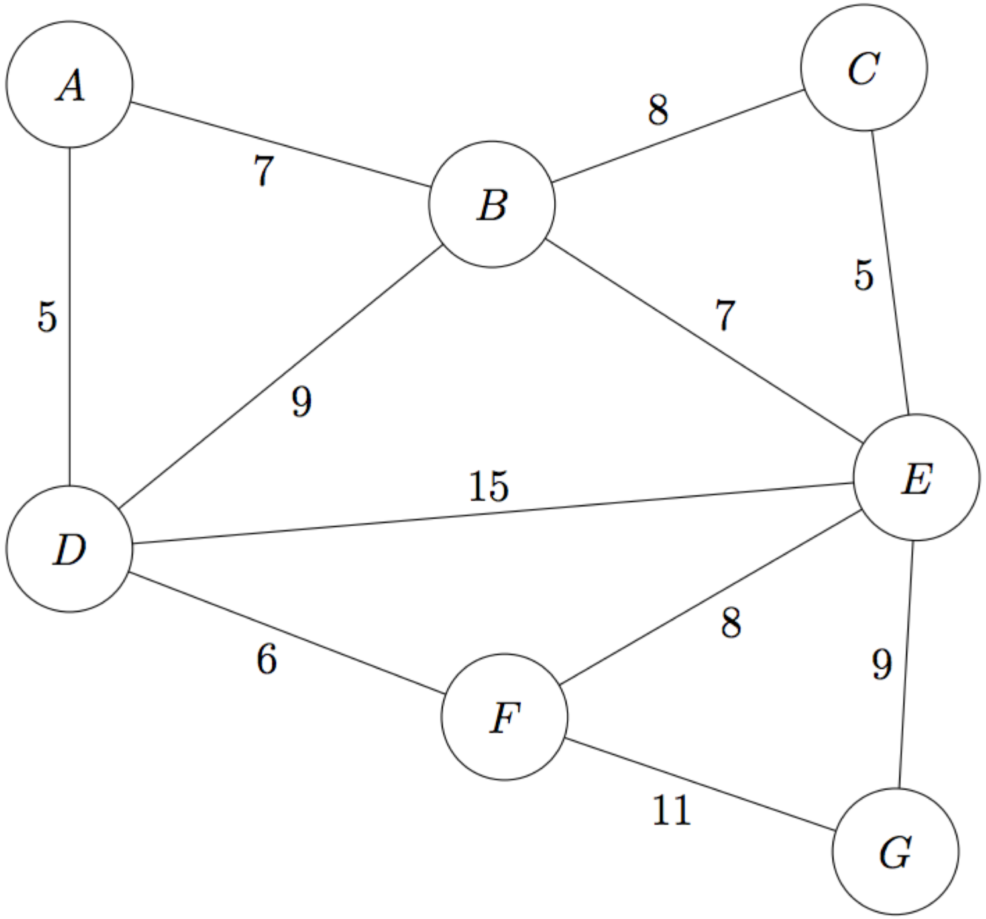
Kruskal’s Algorithm Implementation

```
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 for questions / review your notes

Roadmap

- Disjoint Sets ADT

- Context, examples

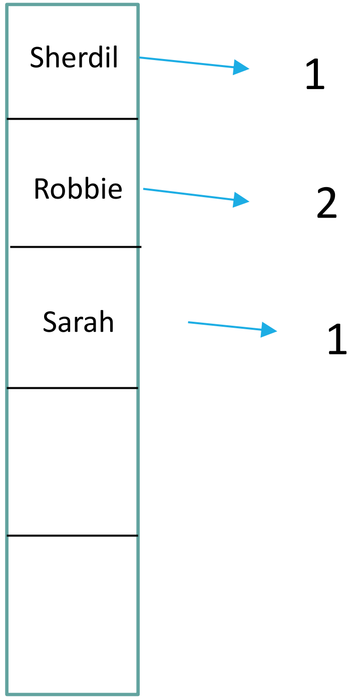
- Different implementations (most of them are just optimizations of the previous)!

1. QuickFind implementation (HashMap based)
2. QuickUnionTrees
3. QuickUnionBySizeTrees
4. QuickUnionBySizeCompressingTrees
5. ArrayQuickUnionBySizeCompressing

1. QuickFind implementation

Calculate the worst case Big-Theta runtimes for each of the methods (makeSet, findSet, union) for this QuickFind implementation of a DisjointSets.

map of value -> set ID/representative



Sherdil -> 1

Robbie -> 2

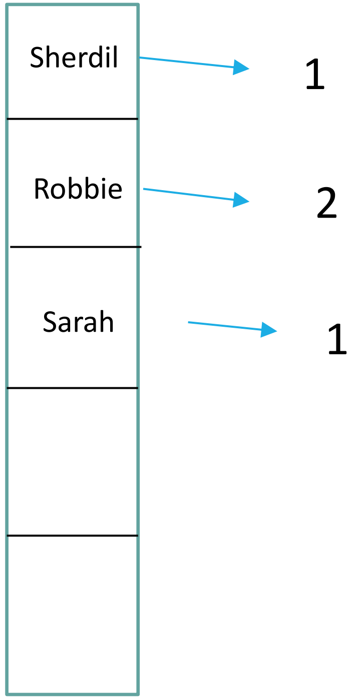
Sarah -> 1

method	runtime (assume using HashMap)
makeSet(value)	
findSet(value)	
union(valueA, valueB)	

1. QuickFind implementation

Calculate the worst case Big-Theta runtimes for each of the methods (makeSet, findSet, union) for this QuickFind implementation of a DisjointSets.

map of value -> set ID/representative



Sherdil -> 1

Robbie -> 2

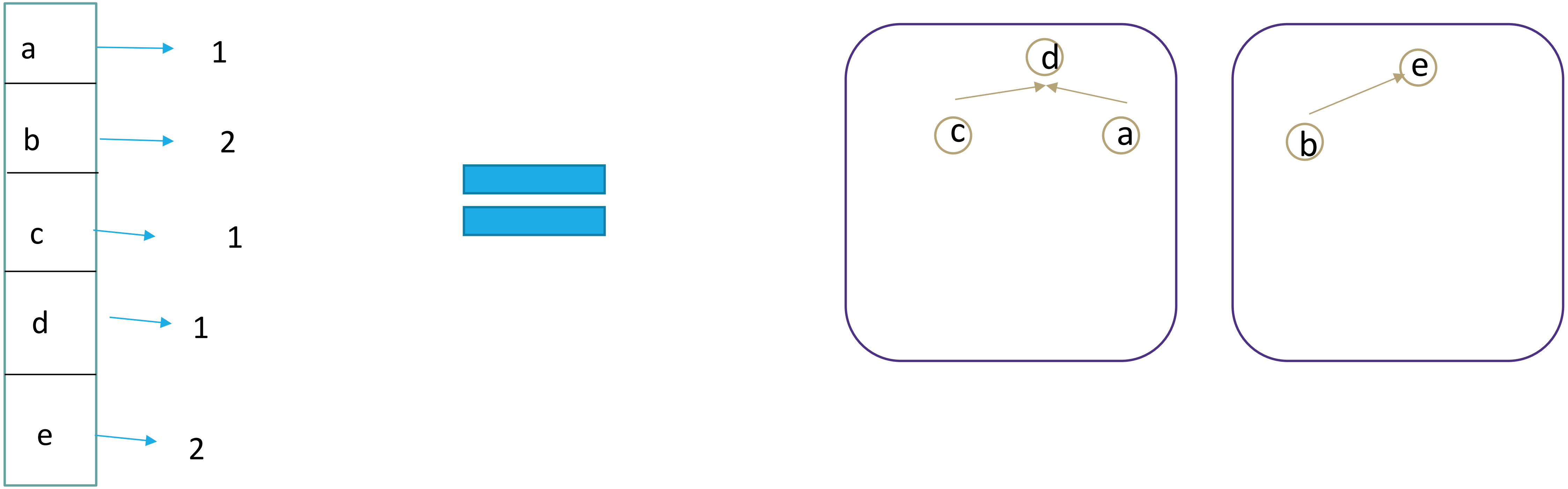
Sarah -> 1

method	runtime (with HashMaps)
makeSet(value)	O(1)
findSet(value)	O(1)
union(valueA, valueB)	O(n)

2. QuickUnionTrees implementation

Each mini-set is now represented as a separate tree. If values are somehow connected / in the same tree, they're in the same mini-set!

(Note: unlike other trees we've seen before, the arrows go upwards! We'll see this is useful so all nodes can access the root)



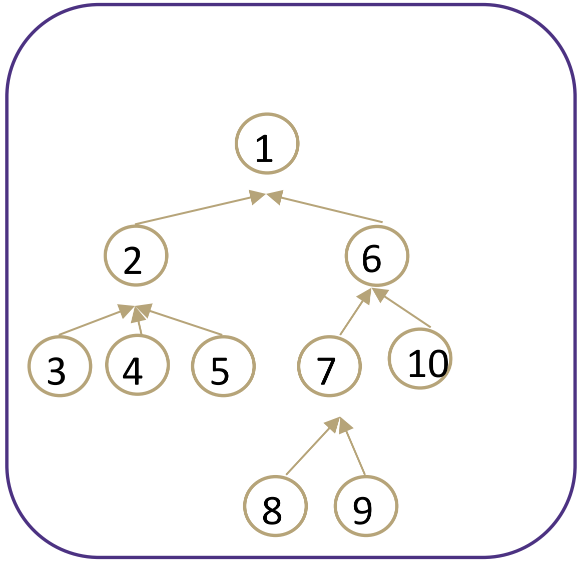
a	1
b	2
c	1
d	1
e	2

2. QuickUnionTrees implementation: `findSet(valueA)`

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.

```
findSet(valueA) {
    jump to valueA node
    travel upwards till root
    return ID for set (in this case the node itself)
}
```

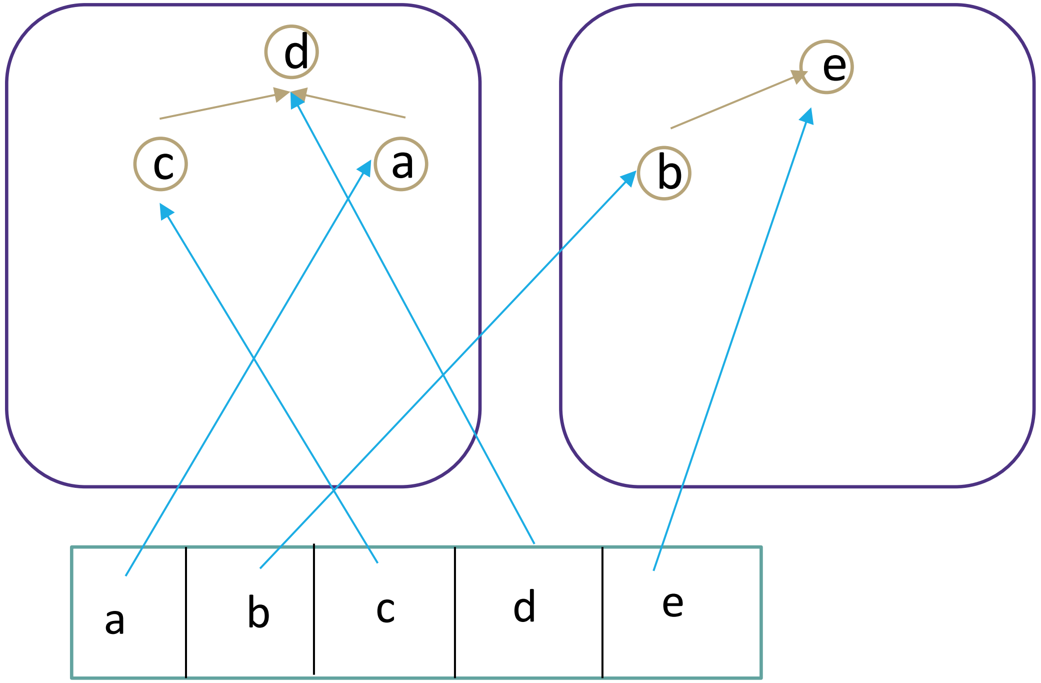


findSet(5) == 1 node

findSet(9) == 9 node

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

jumping to nodes:



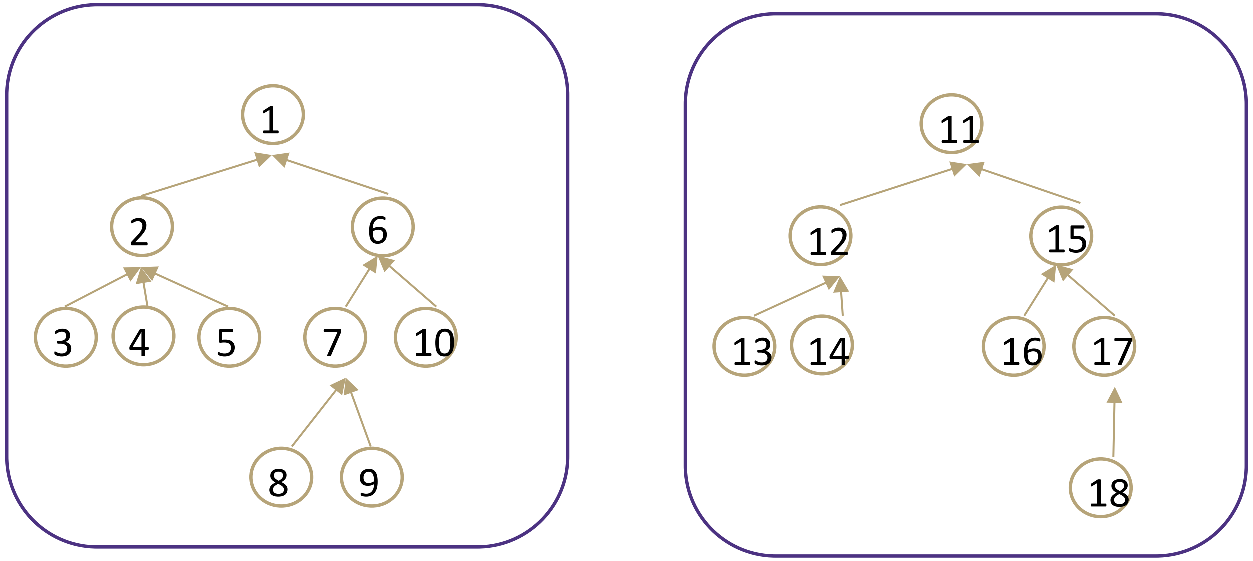
You can use a Map<T, Node> to jump to each node easily (so even though it's not drawn on the future slides, assume we can just jump to any node)

2. QuickUnionTrees implementation: `union(valueA, valueB)`

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 pointer 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 nodes in the tree will be updated to be in the same set. It turns out it will be most efficient if we have the root point to the other tree’s root so we can connect all of the values at once and keep a low height (for findSet)

```
union(valueA, valueB) {
   rootA = findSet(valueA)
   rootB = findSet(valueB)
   set rootA to point to rootB
}
```

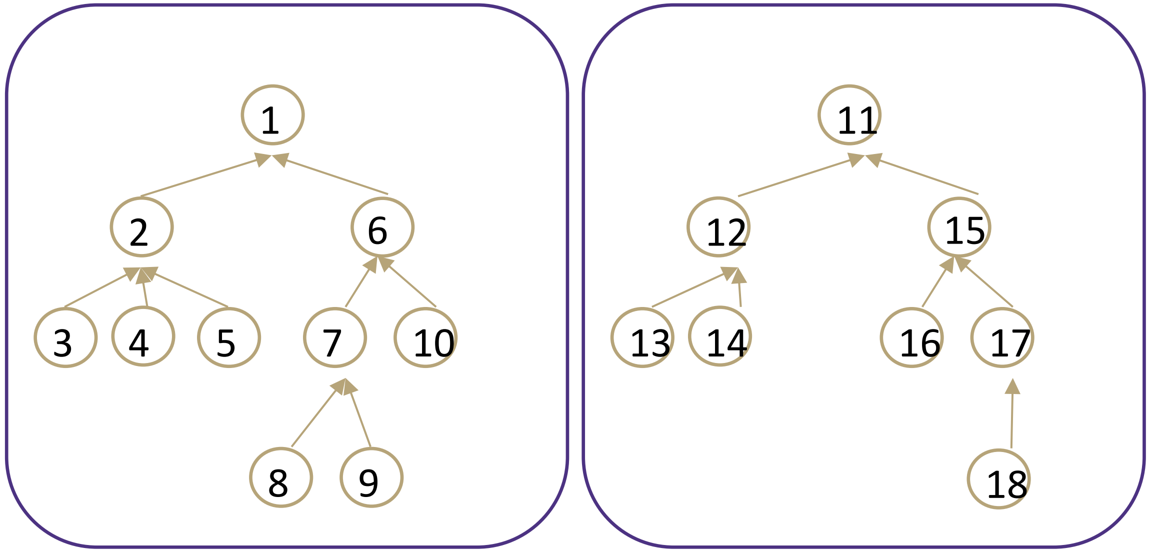


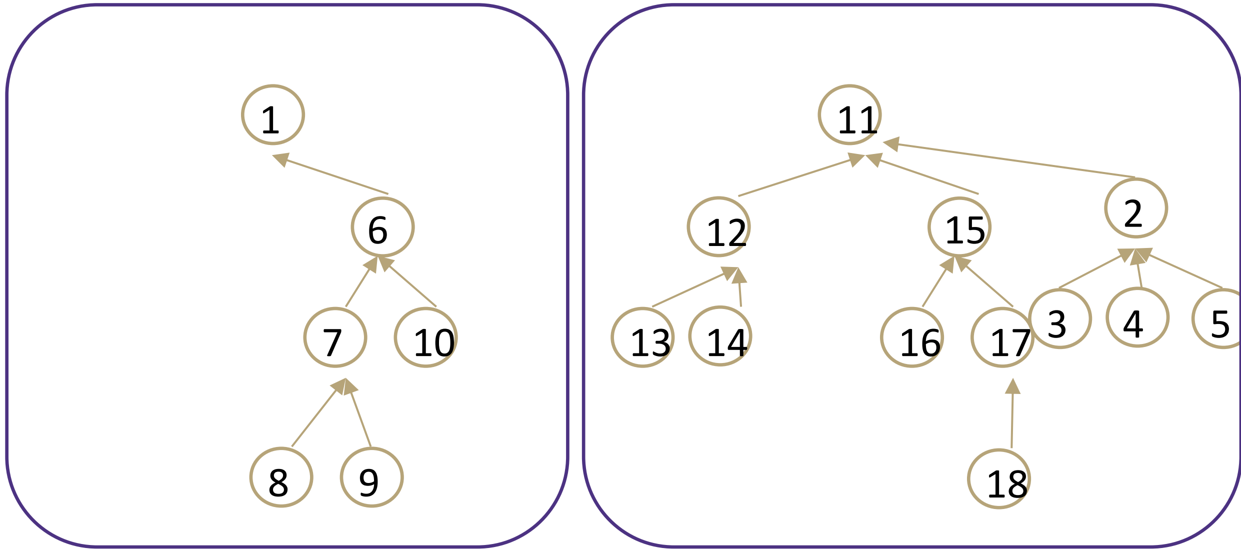
2. QuickUnionTrees implementation: `union(valueA, valueB)`

Note: we need to change one of the roots to point to the tree for correctness. If we did union(2, 11) for example and just set the 2 node to point directly to 11… what's wrong about this picture?

```
original union(valueA, valueB) {
    rootA = findSet(valueA)
    rootB = findSet(valueB)
    set rootA to point to rootB
}
```

```
badUnion1(valueA, valueB) {
    ~~rootA = findSet(valueA)~~
    rootB = findSet(valueB)
    set valueA to point to rootB
}
```



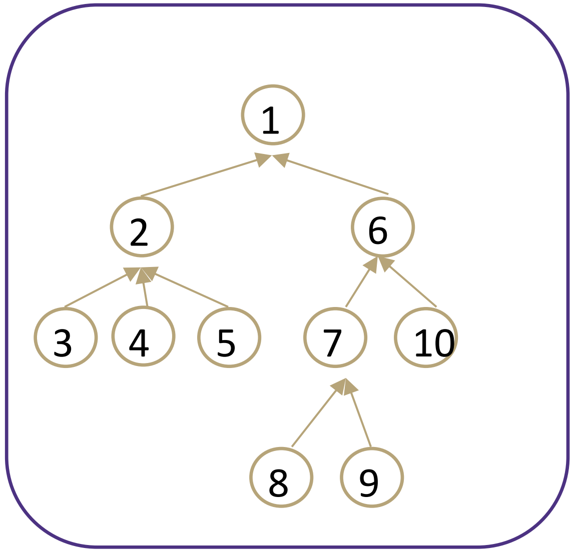


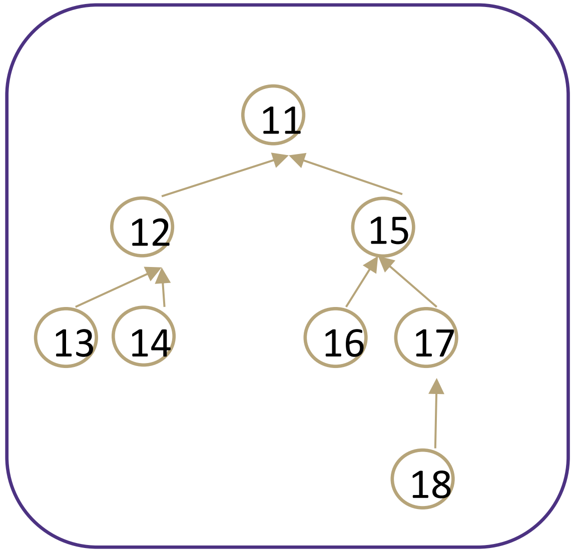
2. QuickUnionTrees implementation: `union(valueA, valueB)`

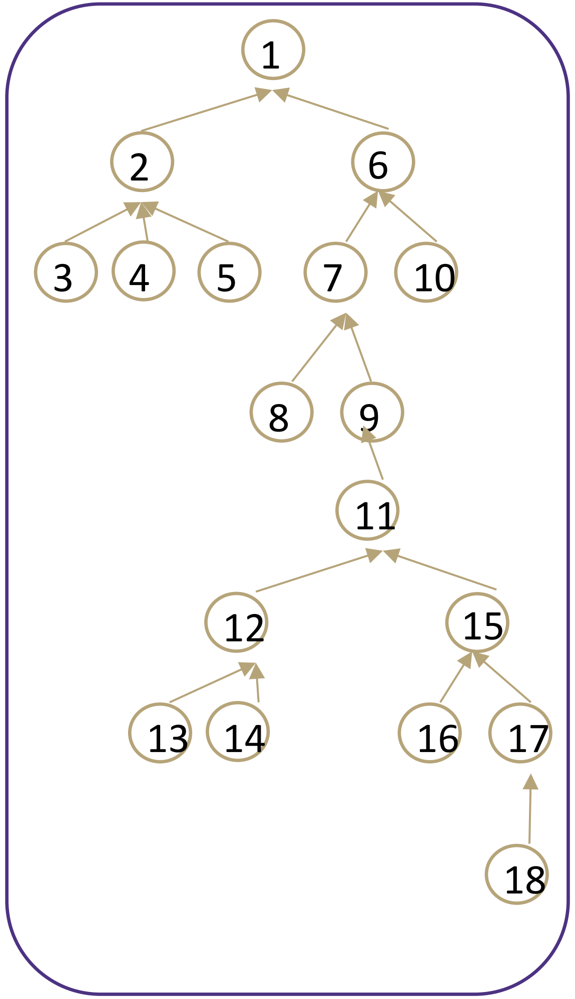
What about if we did the other way around, what happens? It's a little bit inefficient for future calls! Try badUnion2(11, 9) – what does this do to future findSets?

```
original union(valueA, valueB) {
   rootA = findSet(valueA)
   rootB = findSet(valueB)
   set rootA to point to rootB
}
```

```
badUnion2(valueA, valueB) {
    rootA = findSet(valueA)
    ~~rootB = findSet(valueB)~~
    set rootA to point to valueB
}
```







union(valueA, valueB) why modify the roots summary

Recap:

What should we change? If we change the root of one to point to the other tree, then all the lower nodes in the tree will be updated to be in the same set. It turns out it will be most efficient if we have the root point to the other tree's root so we can connect all of the values at once and keep a low height (for findSet).

See the previous slides for the visual examples!

Questions break

QuickFind implementation

QuickUnionTrees

- union
- find

Roadmap

- Disjoint Sets ADT
- Context, examples
- Different implementations (most of them are just optimizations of the previous)!

1. QuickFind implementation (HashMap based)
2. QuickUnionTrees
3. QuickUnionBySizeTrees
4. QuickUnionBySizeCompressingTrees
5. ArrayQuickUnionBySizeCompressing

Let’s try to construct a worst-case scenario ☺

makeSet(a)

makeSet(b)

makeSet(c)

makeSet(d)

makeSet(e)

```
union(valueA, valueB) {
    rootA = findSet(valueA)
    rootB = findSet(valueB)
    set rootA to point to rootB
}
```

Take 1 min to figure out how: what is a worst case scenario for QuickUnionTrees’s findSet/union runtime?   What type of union()s do we need to call to produce this?

Let’s try to construct a worst-case scenario ☺

makeSet(a)

makeSet(b)

makeSet(c)

makeSet(d)

makeSet(e)

```
union(valueA, valueB) {
    rootA = findSet(valueA)
    rootB = findSet(valueB)
    set rootA to point to rootB
}
```

Take 1 min to figure out how: what is a worst case scenario for QuickUnionTrees’s findSet/union runtime? What type of union()s do we need to call to produce this?

union(e, d)
union(d, c)
union(c, b)
union(b, a)
findSet(e)

3. QuickUnionBySizeTrees

Problem: Trees can be unbalanced (and look linked-list-like) so our findSet runtime can be linear runtime in the worst case (if it's linked-list like and we findSet a node towards the bottom of the linked list)

Solution: When union'ing, choose the parent to be the bigger tree

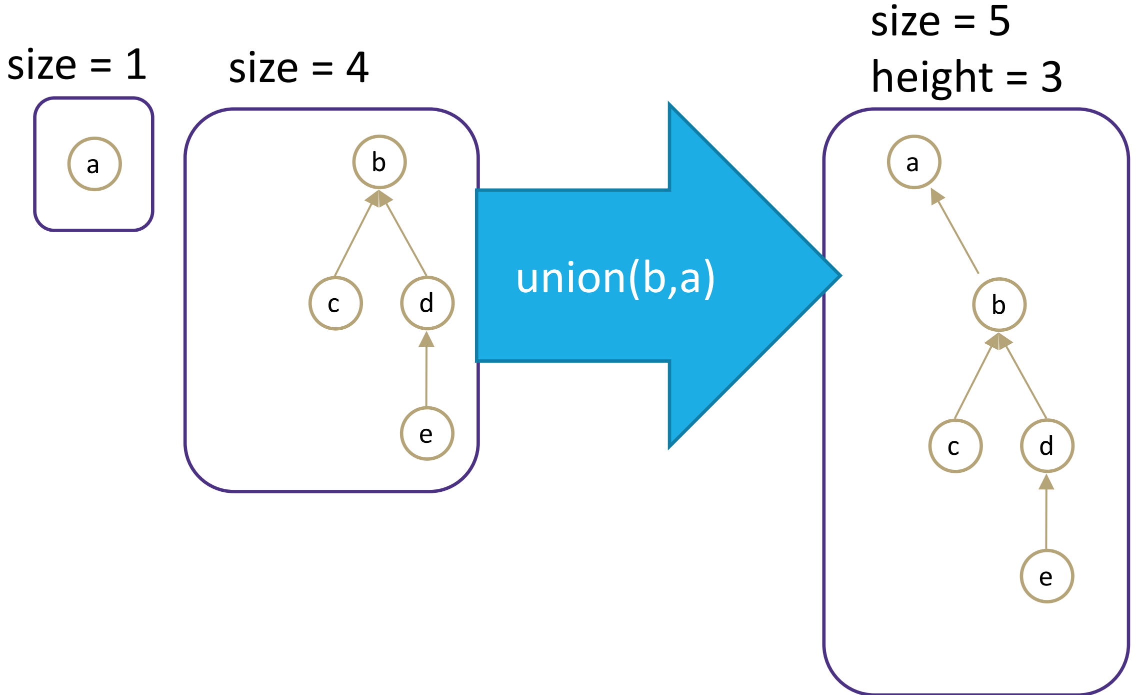
- have the root of each mini-set tree store that tree's size
- **When union'ing make the tree with larger size the root (If it's a tie, pick one arbitrarily)**
- increase the size of the new mini-set as appropriate

3. QuickUnionBySizeTrees

Solution: When union'ing, choose the parent to be the bigger tree

- have the root of each mini-set tree store that tree's size
- **When union'ing make the tree with larger size the root (If it's a tie, pick one arbitrarily)**
- increase the size of the new mini-set as appropriate

<u>possible without union by size</u>



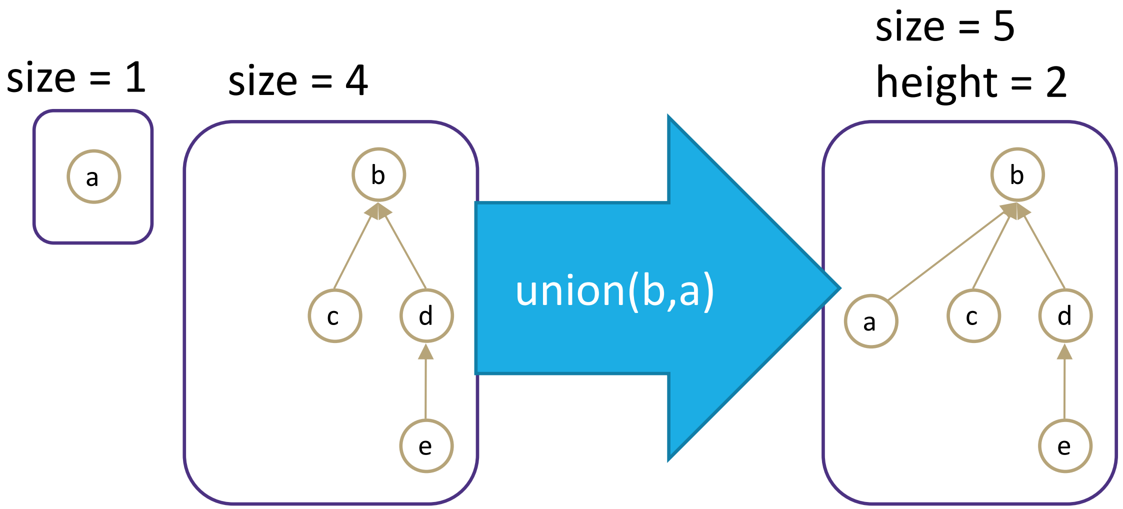
size = 1

size = 4

union(b,a)

size = 5
height = 3

<u>with union by size</u>



size = 1

size = 4

union(b,a)

size = 5
height = 2

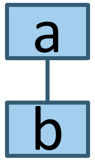
3. QuickUnionBySizeTrees worst case heights

Consider the worst case where the tree height grows as fast as possible for the number of nodes it has

# nodes	height
1	0

3. QuickUnionBySizeTrees worst case heights

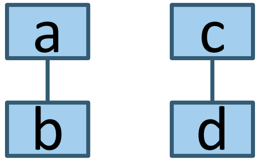
Consider the worst case where the tree height grows as fast as possible for the number of nodes it has



# nodes	height
1	0
2	1

3. QuickUnionBySizeTrees worst case heights

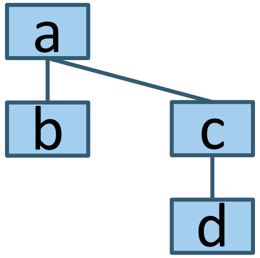
Consider the worst case where the tree height grows as fast as possible for the number of nodes it has



# nodes	height
1	0
2	1

3. QuickUnionBySizeTrees worst case heights

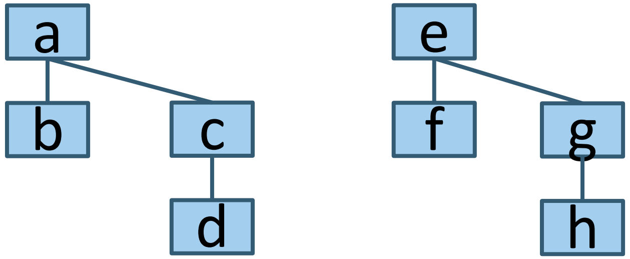
Consider the worst case where the tree height grows as fast as possible for the number of nodes it has



# nodes	height
1	0
2	1
4	2

3. QuickUnionBySizeTrees worst case heights

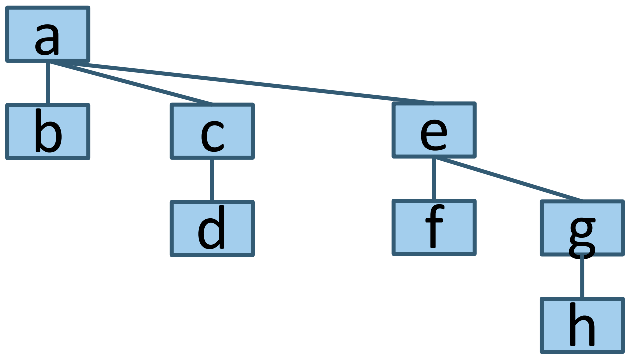
Consider the worst case where the tree height grows as fast as possible for the number of nodes it has



# nodes	height
1	0
2	1
4	2

3. QuickUnionBySizeTrees worst case heights

Consider the worst case where the tree height grows as fast as possible for the number of nodes it has

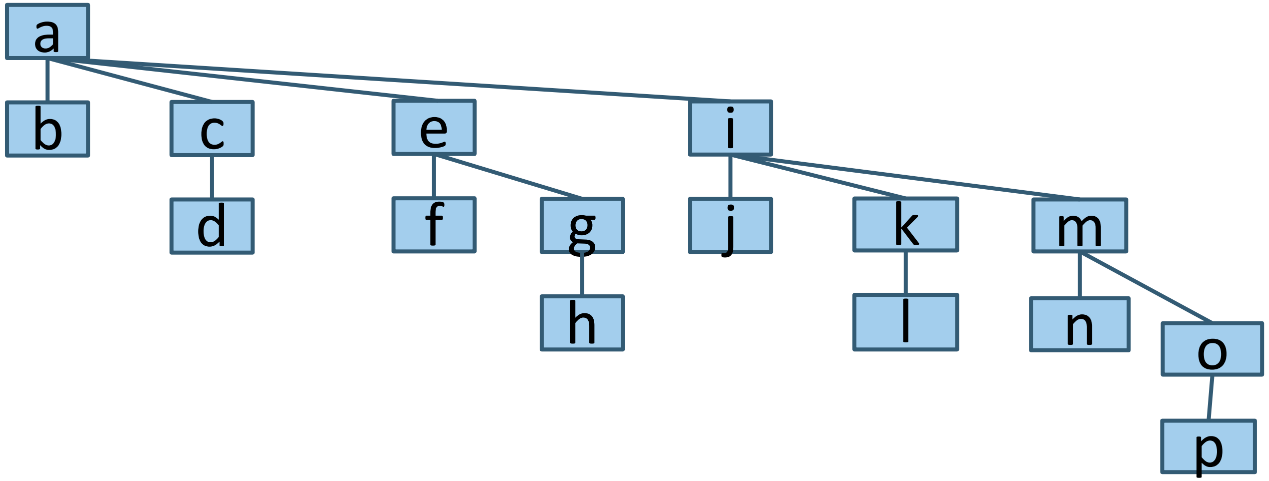


# nodes	height
1	0
2	1
4	2
8	3

3. QuickUnionBySizeTrees worst case heights

Consider the worst case where the tree height grows as fast as possible for the number of nodes it has

Worst case tree height is Θ(log N)

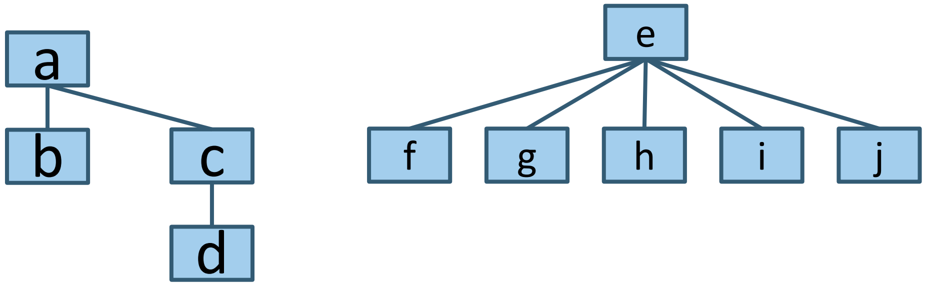


# nodes	height
1	0
2	1
4	2
8	3
16	4

3. QuickUnionBySizeTrees bad situations are still bounded by the worst case heights

union(a, e) – which one becomes the parent when doing union-by-size?

a will point to e because a’s tree size is 4, but e’s tree size is 6. The height increases by one even though it didn’t need to! If we had e point to a the height (the max distance) would have stayed the same.



Why not use the height of the tree?

- QuickUnionByHeightTrees runtime is asymptotically the same: Θ(log(N))
- It’s easier to track weights than heights

main point of this slide: QuickUnionBySizeTrees produces a suboptimal structures, such as this one, in specific cases. But for the most part it works out as you increase the number of nodes towards infinity.

Roadmap

- Disjoint Sets ADT
- Context, examples
- Different implementations (most of them are just optimizations of the previous)!

1. QuickFind implementation (HashMap based)
2. QuickUnionTrees
3. QuickUnionBySizeTrees
4. QuickUnionBySizeCompressingTrees
5. ArrayQuickUnionBySizeCompressing

Modifying Data Structures To Preserve Invariants

Thus far, the modifications we've studied are designed to *preserve invariants* (aka "repair the data structure")

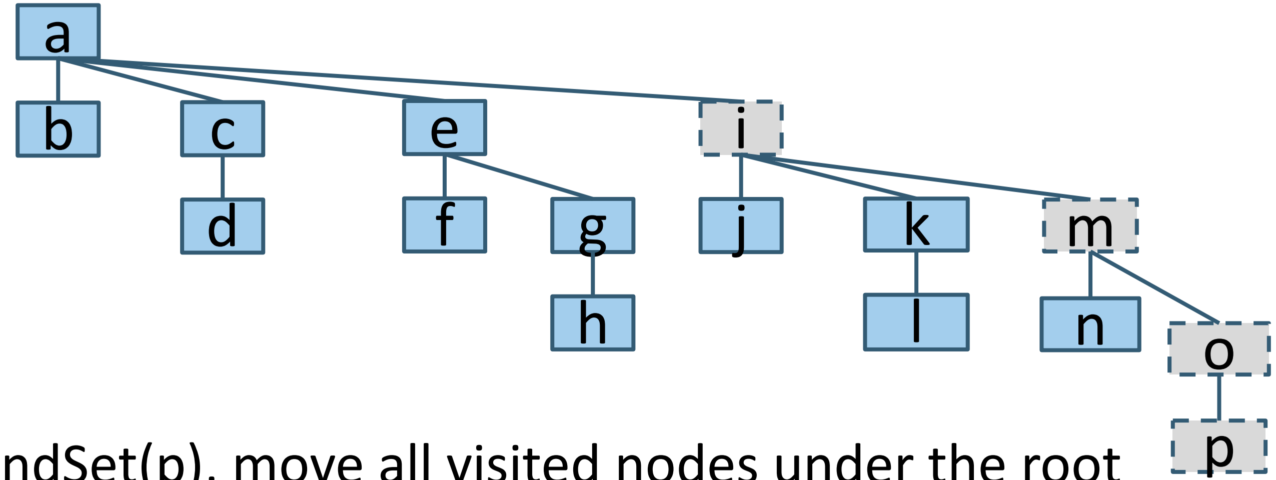
- **Tree rotations**: preserve AVL height invariant so we guarantee log(n) height and log(n) runtime for worst case if we need to traverse to the bottom of the tree
- **heap percolations:** preserve heap sorted invariants so we can find Min/Max still in constant time

Notably, the modifications don't improve runtime between identical method calls

Path compression is entirely different: we are modifying the tree structure to *improve future performance*

4. QuickUnionBySizeCompressingTrees
Path Compression: Idea

This is the worst-case topology if we use WeightedQuickUnion

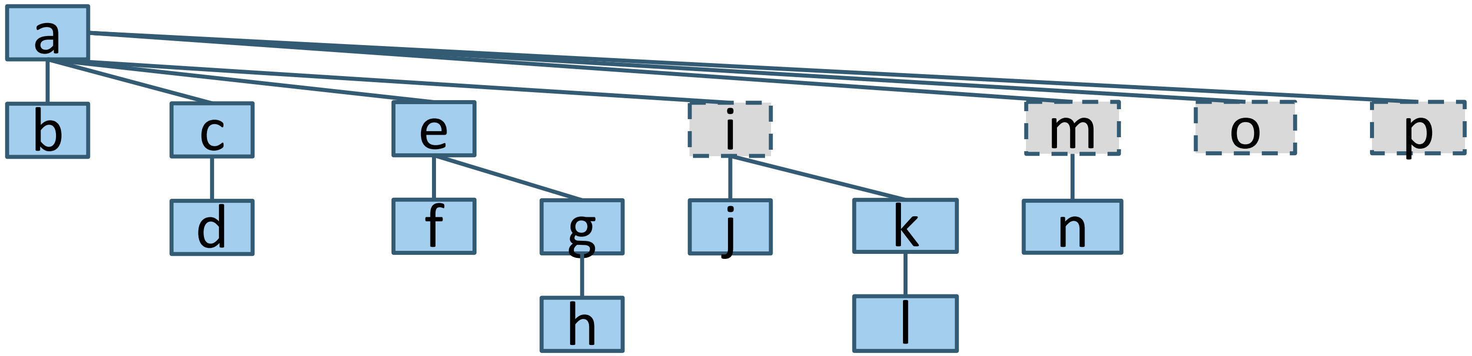


Idea: When we do findSet(p), move all visited nodes under the root

- Additional cost is insignificant (same order of growth)

4. QuickUnionBySizeCompressingTrees Path Compression: Example

This is the worst-case topology if we use WeightedQuickUnion



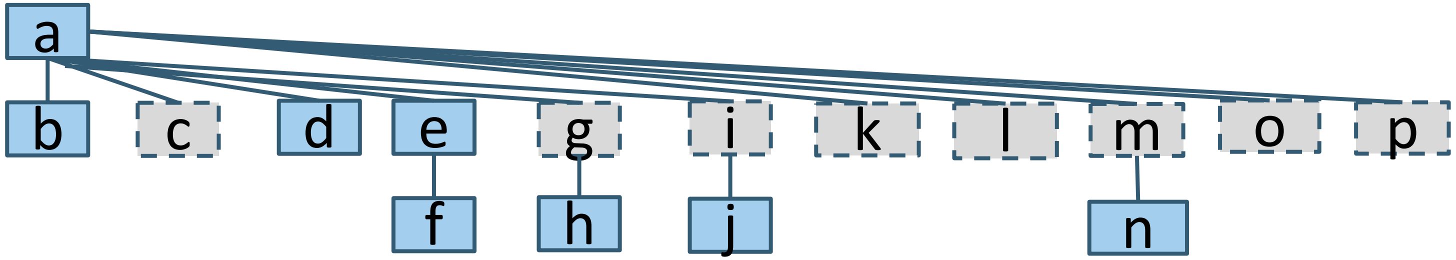
Idea: When we do findSet(p), move all visited nodes under the root

- Doesn’t meaningfully change runtime for *this* invocation of findSet(p), but subsequent findSet(p)s (and subsequent findSet(o)s and findSet(m)s and …) will be faster

4. QuickUnionBySizeCompressingTrees Path Compression: Details and Runtime

Run path compression on every findSet()!

- Including the findSet()s that are invoked as part of a union()



Understanding the performance of M>1 operations requires *amortized analysis*

We won’t go into it here, but we’ve sort of seen this before

- It’s how we can actually say that appending to an array is “O(1) on average” if we double whenever we resize. You can google it more if you’re curious!

4. QuickUnionBySizeCompressingTrees
Subtleties of Path Compression

Path compression is an optimization written into the `findSet` code.

It does not appear directly in the `union` code.

- It's not worth it; you'd have to rewrite the entire `findSet` code inside `union` to make it happen.

But `union` does make two `findSet` calls,

- So path compression will happen when you do a `union` call, just indirectly.

Questions break

QuickunionBySizeTrees

QuickUnionBySizeCompressingTrees

4. QuickUnionBySizeCompressingTrees 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 $O(1)$.

$\log^* n$ 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“

4. QuickUnionBySizeCompressingTrees methods recap

findSet(value):

1. jump to the node of value and traverse up to get to the root (representative)
2. after finding the representative do *path compression* (point every node from the path you visited to the root directly)
3. return the root (representative) of the set value is in

union(valueA, valueB):

1. call findSet(valueA) and findSet(valueB) to get access to the root (representative) of both
2. merge by setting one root to point to the other root (one root becomes the parent of the other root). Have the smaller sized tree's root point to the bigger tree's root
   - if treeA's rank == treeB's size, It doesn't matter which is the parent so choose arbitrarily

Roadmap

- Disjoint Sets ADT
- Context, examples
- Different implementations (most of them are just optimizations of the previous)!

1. QuickFind implementation (HashMap based)
2. QuickUnionTrees
3. QuickUnionBySizeTrees
4. QuickUnionBySizeCompressingTrees
5. ArrayQuickUnionBySizeCompressing

Array implementation motivation

Instead of nodes, let’s use an array implementation!

Just like heaps, the trees and node objects will exist in our mind, but not in our programs.

It won’t be asymptotically faster, but check out all these benefits:

- this will be more memory compact
- get better caching benefits because we’ll be using arrays
- simplify the implementation

What are we going to put in the array and what is it going to mean?

One of the most common things we do with Disjoint Sets is: go to a node and traverse upwards to the root (go to your parent, then go to your parent's parent, then go to your parent's parent's parent, etc.).

A couple of ideas:

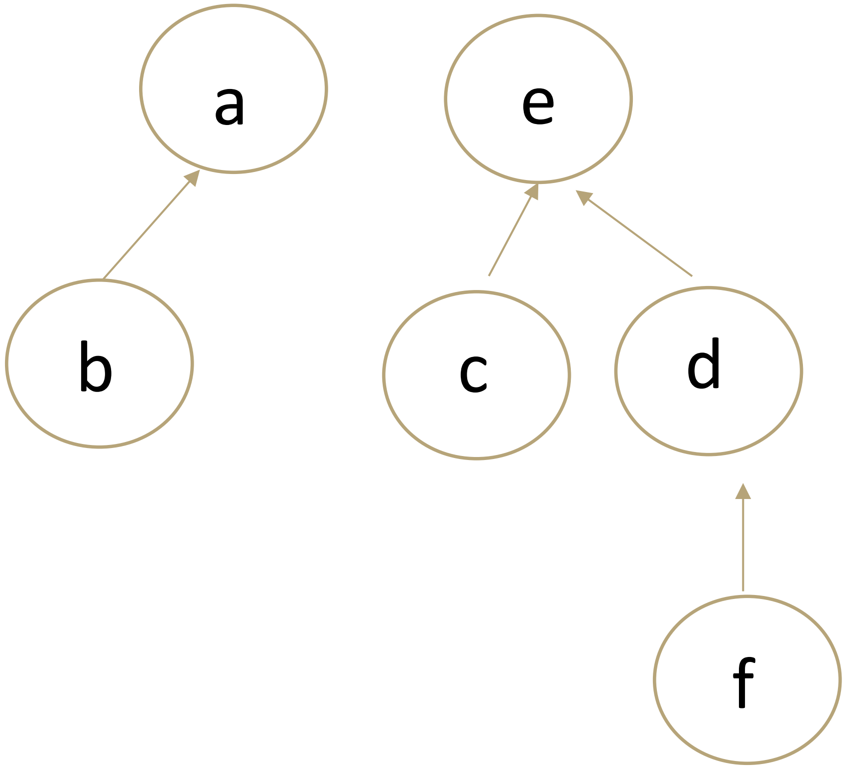
- represent each node as a position in our array

- at each node's position, store the index of the parent node. This will let us jump to the parent node position in the array, and then we can look up our parent's parent node position, etc.
    - if we're storing indices, this mean this is an array of ints

This is a big idea!

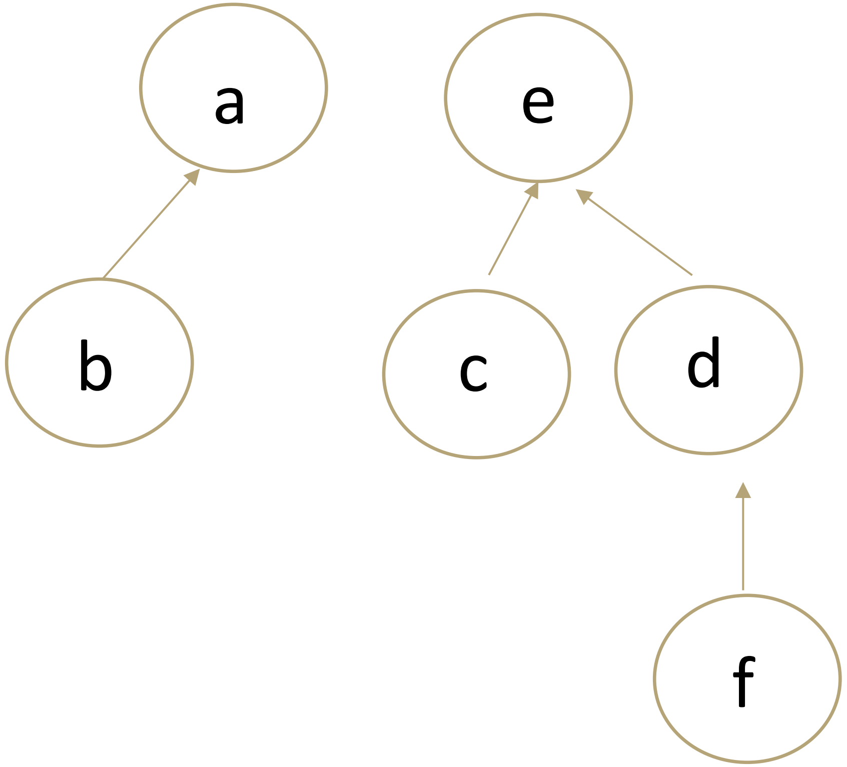
at each node's position, store the index of the parent node

	a	e	d	c	b	f
index	0	1	2	3	4	5
value	-	-	1	1	0	?



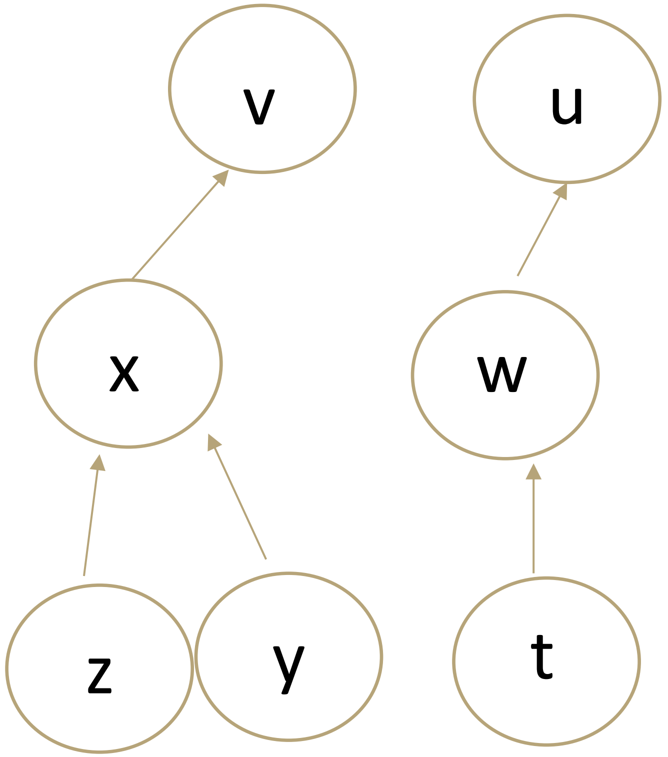
at each node’s position, store the index of the parent node

	a	e	d	c	b	f
index	0	1	2	3	4	5
value	-	-	1	1	0	2



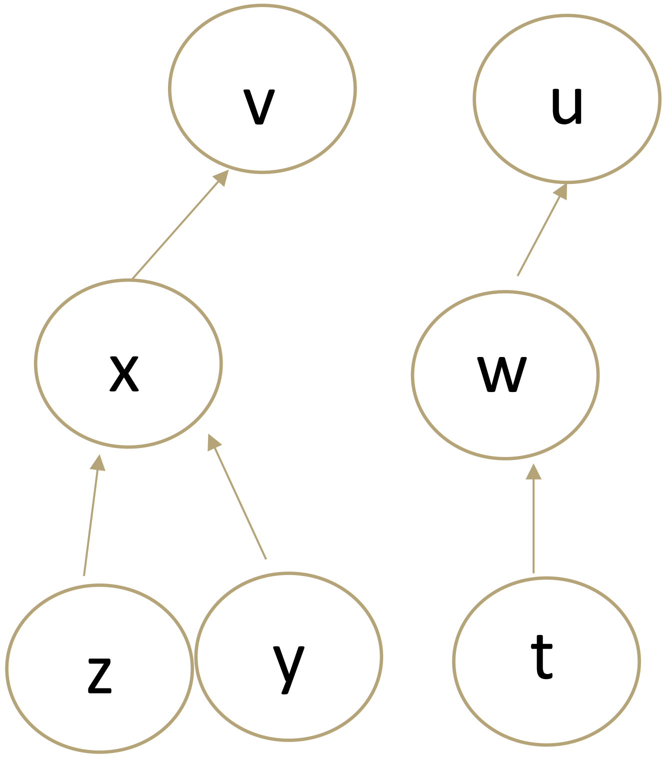
Exercise (1 min)
at each node’s position, store the index of the parent node

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	?	?	?	?	?	-	-



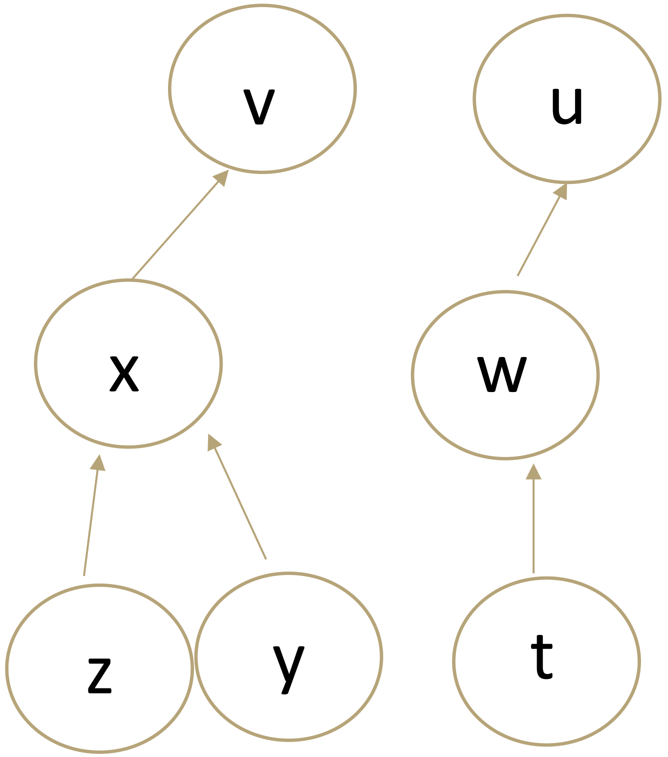
Exercise (1 min)
at each node’s position, store the index of the parent node

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	?	?	?	?	-	-



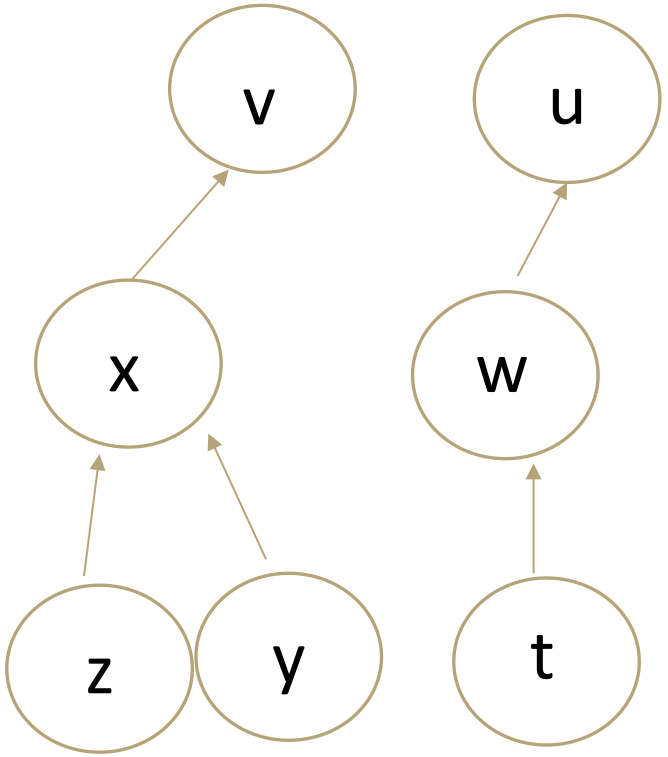
Exercise (1 min)
at each node's position, store the index of the parent node

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	3	?	?	?	-	-



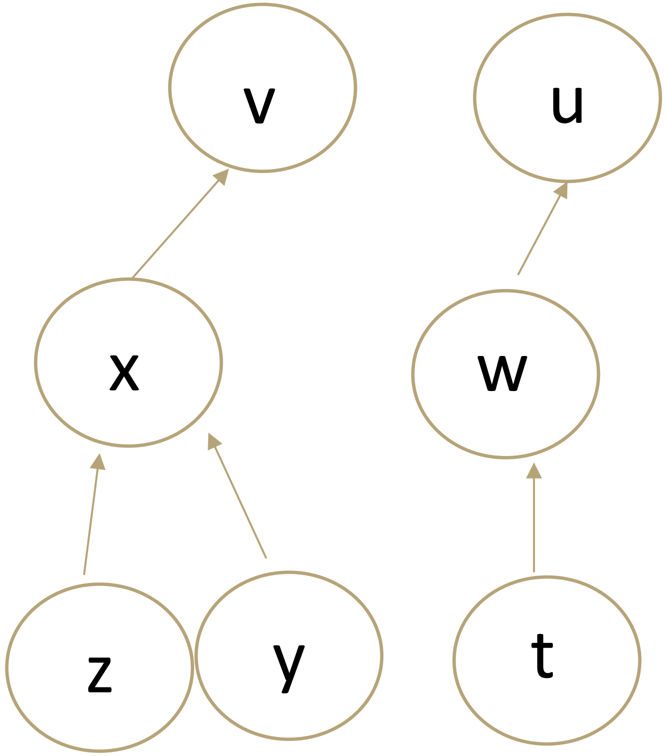
Exercise (1 min)
at each node's position, store the index of the parent node

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	3	4	?	?	-	-



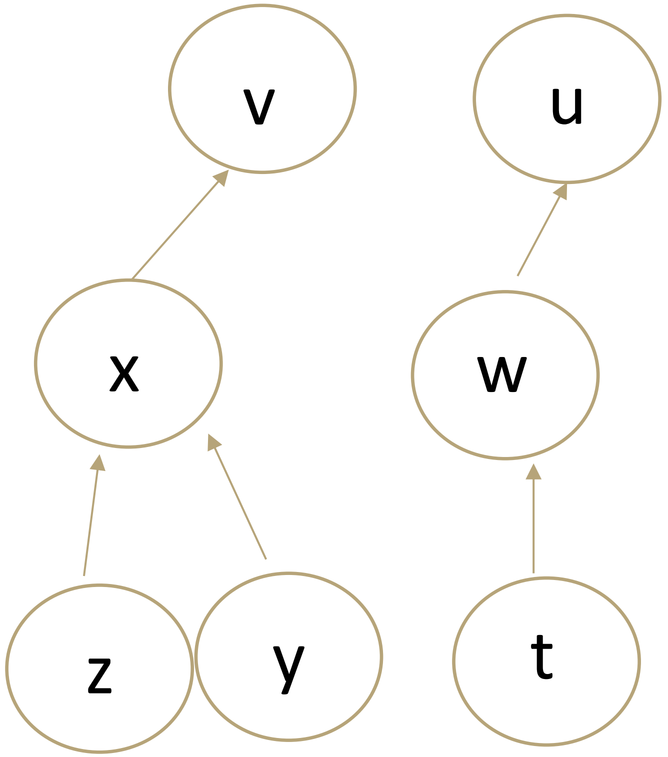
Exercise (1 min)
at each node's position, store the index of the parent node

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	3	4	5	?	-	-



Exercise (1 min)
at each node's position, store the index of the parent node

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	3	4	5	6	-	-

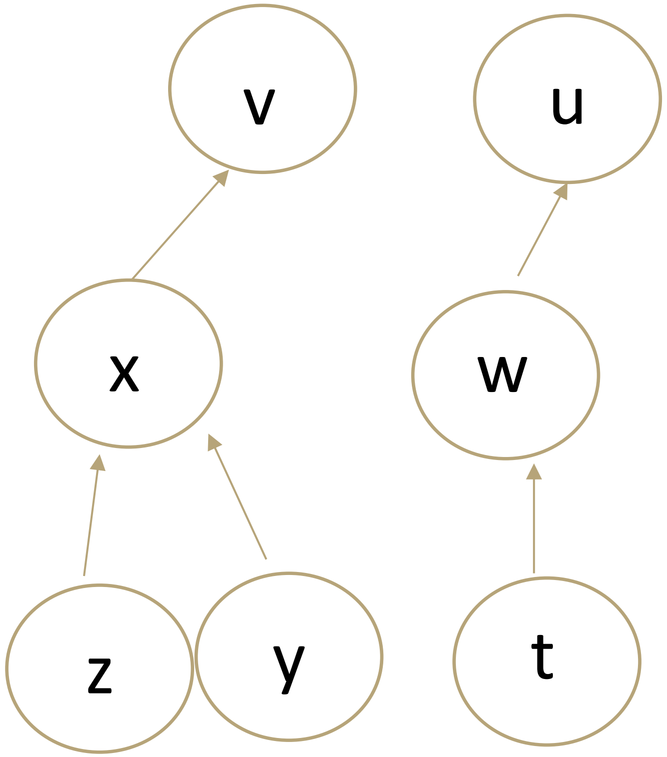


How would findSet work for array implementation?

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	3	4	5	6	-	-

example : findSet(y)

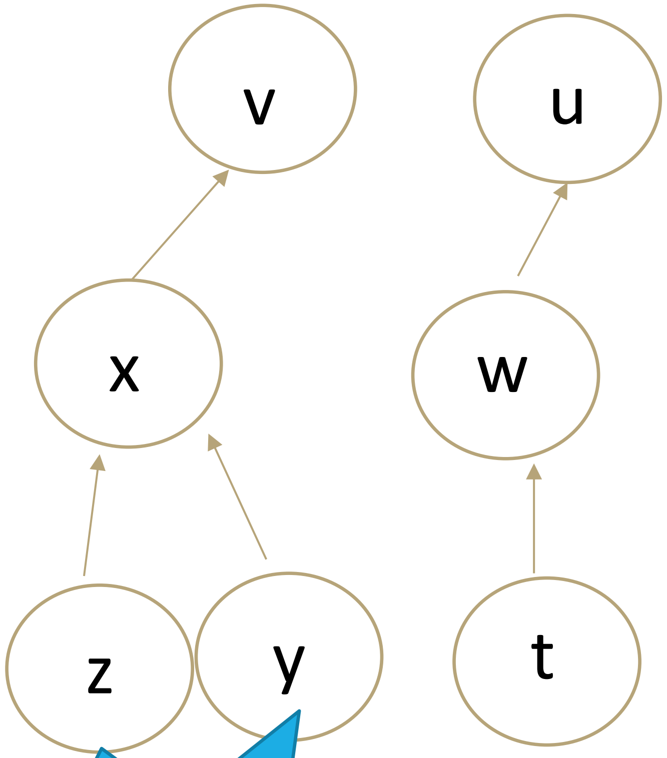
- look up the index of y in our array (index 1)
- keep traversing till we get to the root / no more parent indices available
- path compression (set everything to point to the index of the root - in this case set everything on the path to 5)
- return the **index** of the root (in this case return 5). Instead of the actual node itself, we now have access to an index which is a simpler, but still unique ID



How would findSet work for array implementation?

(Looking up the index for a given value)

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	3	4	5	6	-	-



In findSet we have to figure out where to start traversing upwards from …
so what index do we use and how do we keep track of the values indices?
(In the above example) basically, how would we map each letter to a position?

Whenever you add new values into your disjoint set,
keep track of what index you stored it at with a **dictionary of value to index**!
This is similar to the thing as what we did in our ArrayHeap.

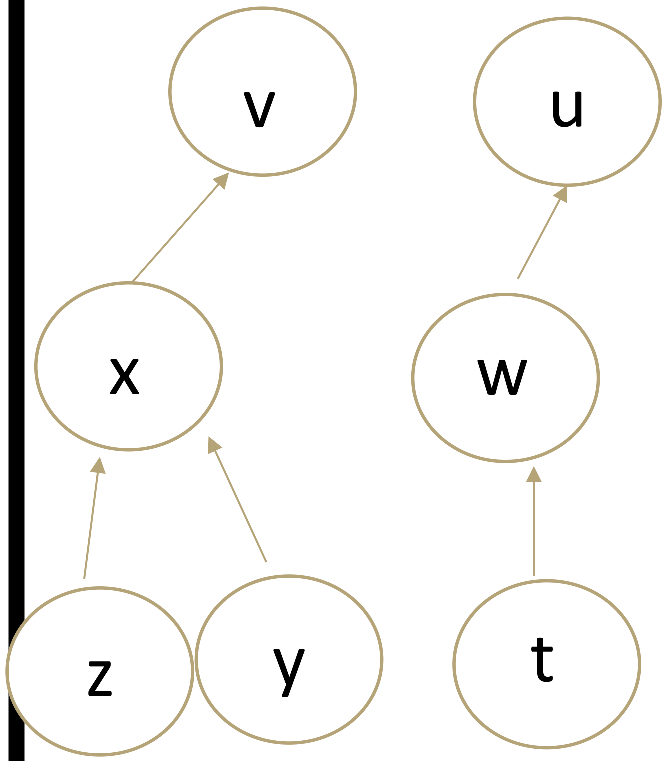
This is a big idea!

How would findSet work for array implementation?

(What do we store at the root position so we know when to stop?)

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	3	4	5	6	-	-

We just mentioned for findSet that we need to traverse starting from a node (like y) to its parent and then its parent's parent until we get to a root. What type of int could we put there as a sign that we've reached the root?



How would findSet work for array implementation?

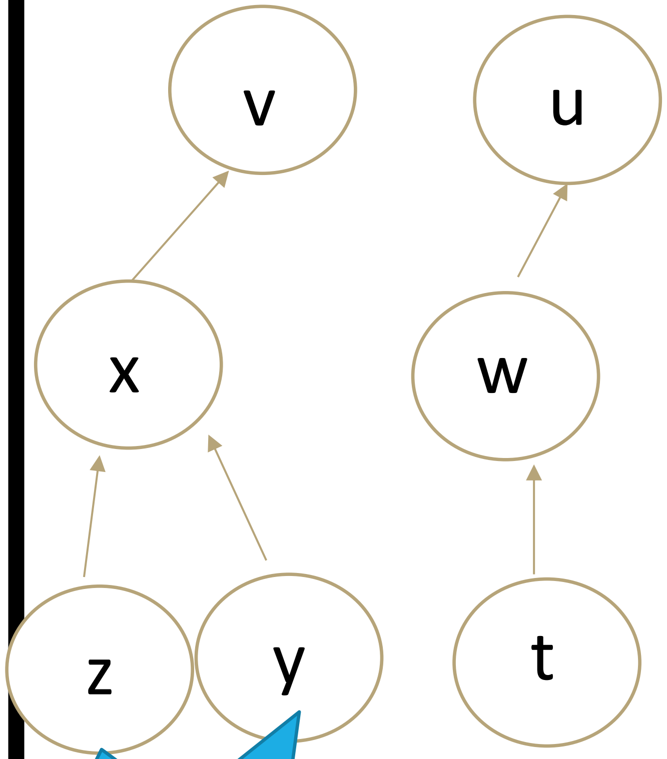
(What do we store at the root position so we know when to stop?)

	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	3	4	5	6	-4	-3

We just mentioned for findSet that we need to traverse starting from a node (like y) to its parent and then its parent’s parent until we get to a root. What type of int could we put there as a sign that we’ve reached the root?

A negative number! (since valid array indices are only 0 and positive numbers)

We’re going to actually be extra clever and store a strictly negative version of th e: for our root nodes.



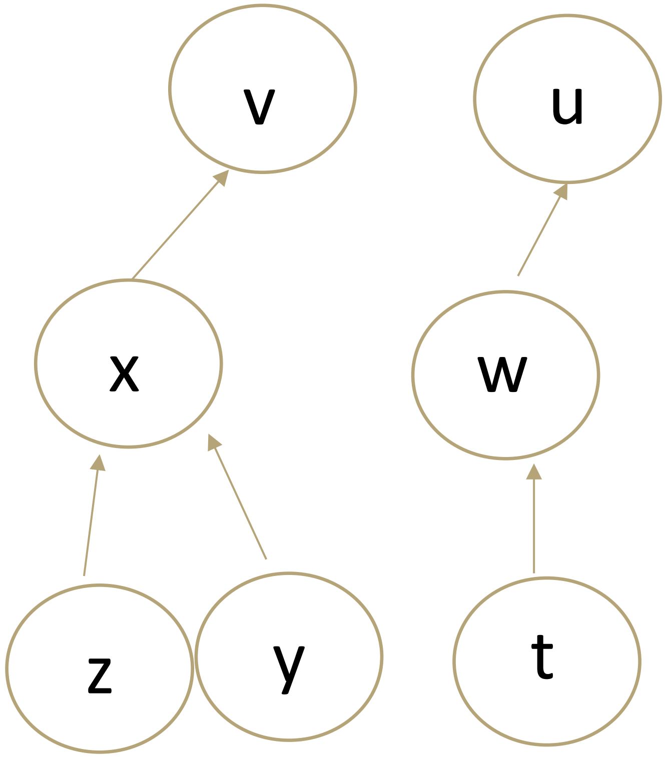
This is a big idea!

How would findSet work for array implementation? (after ironing out details)

	z	y	t	x	w	v	u
index	**0**	**1**	**2**	**3**	**4**	**5**	**6**
value	3	3	4	5	6	-4	-3

example : findSet(y)

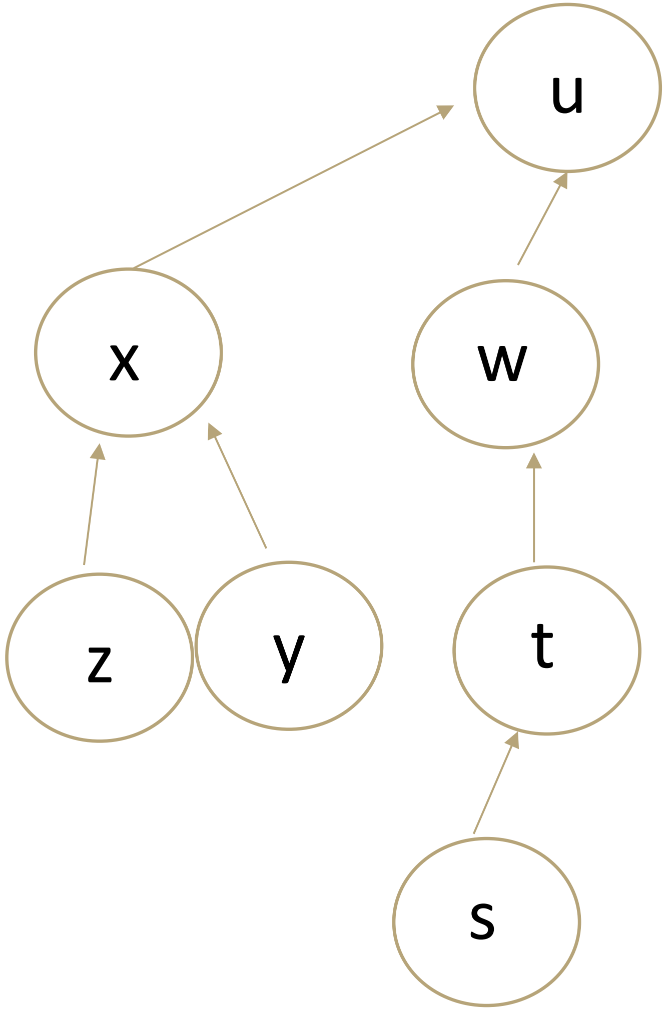
- look up the index of y in our array with index dictionary (index 1)
- keep traversing till we get to the root, signified by negative numbers
- path compression (set everything to point to the index of the root - in this case set everything on the path to 5)
- return the **index** of the root (in this case return 5). Instead of the actual node itself, we now have access to an index which is a simpler, but still unique ID



Exercise (1.5 min) – what happens for findSet(s)

	z	y	t	x	w	u	s
index	0	1	2	3	4	5	6
value	3	3	4	5	5	-7	2

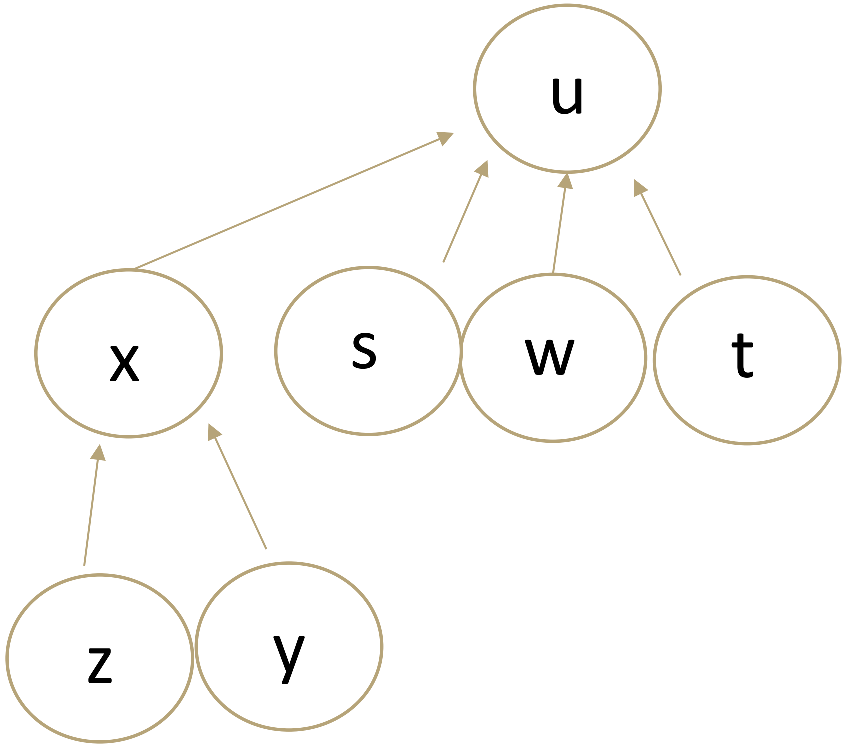
- look up the index of value in our array with index dictionary keep traversing till we get to the root, signified by negative numbers
- path compression (set everything to point to the index of the root)
- return the **index** of the root (in this case return 5). Instead of the actual node itself, we now have access to an index which is a simpler, but still unique ID



Exercise (1.5 min) – what happens for findSet(s)

	z	y	t	x	w	u	s
index	0	1	2	3	4	5	6
value	3	3	5	5	5	-7	5

- look up the index of value in our array with index dictionary keep traversing till we get to the root, signified by negative numbers
- path compression (set everything to point to the index of the root)
- return the **index** of the root (in this case return 5). Instead of the actual node itself, we now have access to an index which is a simpler, but still unique ID



returns 5

How would union work for array implementation?

note: formula to store in root nodes is negative size

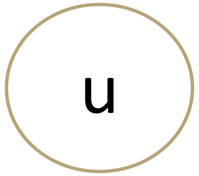
index	0	1	2	3
value	/	/	/	/

makeSet(u)
makeSet(v)
union(u, v)

How would union work for array implementation?

note: formula to store in root nodes is negative size

	u			
index	0	1	2	3
value	-1	/	/	/



~~makeSet(u)~~

makeSet(v)

union(u, v)

How would union work for array implementation?

note: formula to store in root nodes is negative size

	u	v		
index	0	1	2	3
value	-1	-1	/	/

~~makeSet(u)~~

~~makeSet(v)~~

union(u, v)

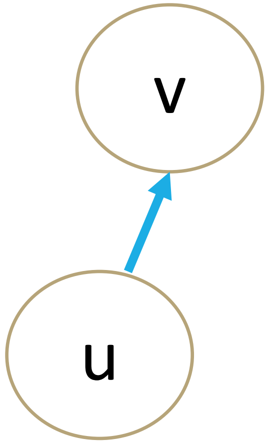
How would union work for array implementation?

note: formula to store in root nodes is negative size

	u	v		
index	0	1	2	3
value	1	-1	/	/

union – almost the same as before

- update one of the roots to point to the other root (in this case we had node u's position in the array store index 1, as v is now its parent)



~~makeSet(u)~~

~~makeSet(v)~~

~~union(u, v)~~

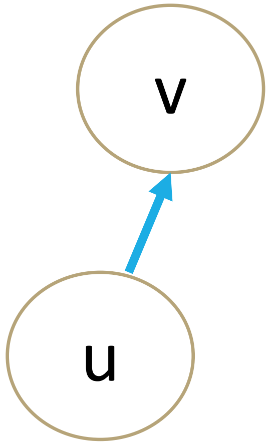
How would union work for array implementation?

note: formula to store in root nodes is negative size

	u	v		
index	0	1	2	3
value	1	-2	/	/

union – almost the same as before

- update one of the roots to point to the other root (in this case we had node u's position in the array store index 1, as v is now its parent)
- Note: calculate the new size and then multiply it by -1 to turn it into the negative version.



~~makeSet(u)~~

~~makeSet(v)~~

~~union(u, v)~~

Exercise maybe

a	b	c	d	e	f	g
0	**1**	**2**	**3**	**4**	**5**	**6**
-1	-1	-1	-1	-1	-1	-1

already set up all the makeSet calls in the area

- union(a, b)
- union(c, d)
- union(e, f)
- union(a, g)
- union(c, e)
- union(a, c)

Summary of the big ideas

- each node is represented by a position in the int array
- each position stores either:
  - the index of its parent, if not the root node
  - -1 * size if the root node
- keep track of a dictionary of value to index to be able to jump to a node's position in the array
- apply all the same high level ideas of how the Disjoint Set methods work (findSet and union) for trees, but to the array representation
  - makeSet – store -1 (size of 1) in a new slot in the array
  - findSet(value) – jump to the value's position in your array, and traverse till you reach a negative number (signifies the root). Do path compression and return the index of the root (the representative of this set).
  - union(valueA, valueB) – call findSet(valueA) and findSet(valueB) to access the sizes and indices of valueA and valueB's sets. Compare the sizes like in the tree representation. Make sure to update the size when you union the two of them together.