Lecture 19: Array Disjoint Sets and more Graphs

CSE 373: Data Structures and Algorithms

Administrivia

Project 4 due Wednesday May 20th

- Please start now
- Last assignment to use late days on
- Will use Dijkstra's code for last programming project

Exercise 4 due Friday May 15th

Roadmap

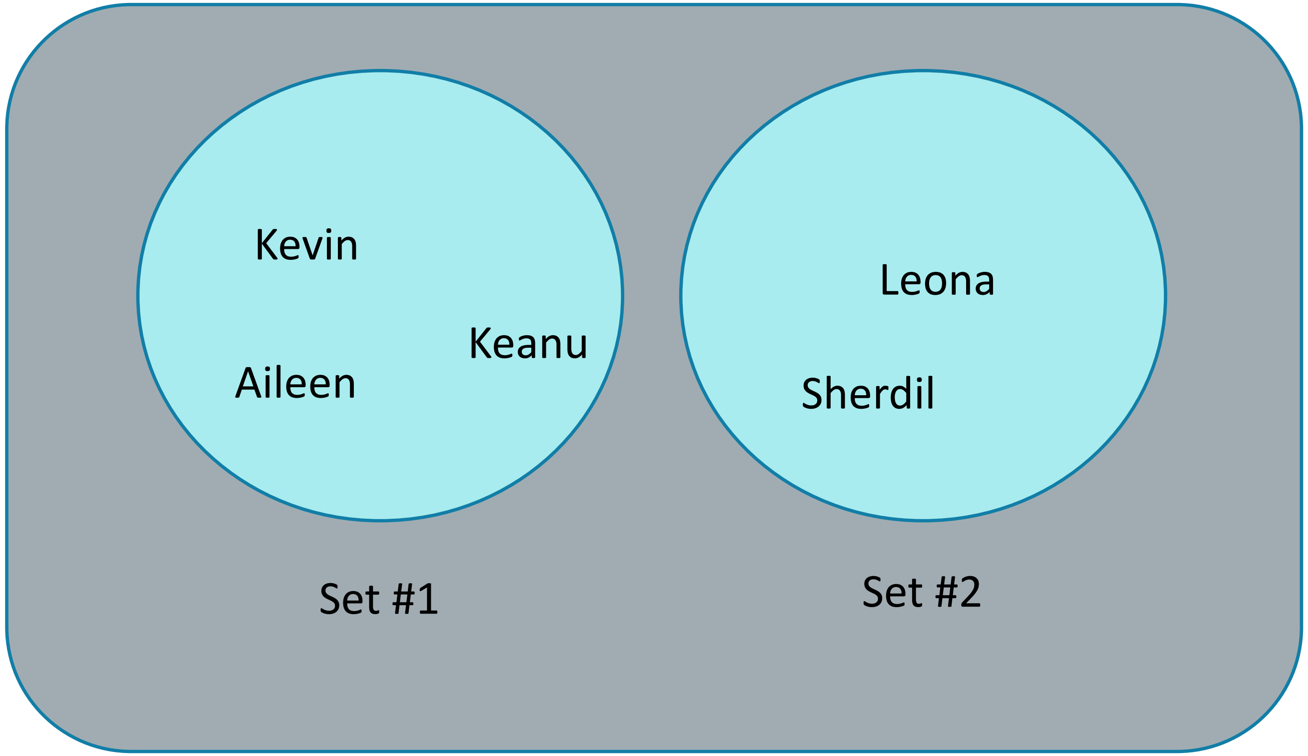
- Disjoint Sets ADT
- Context, examples
- Different implementations (most of them are just optimizations of the previous)!

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

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).

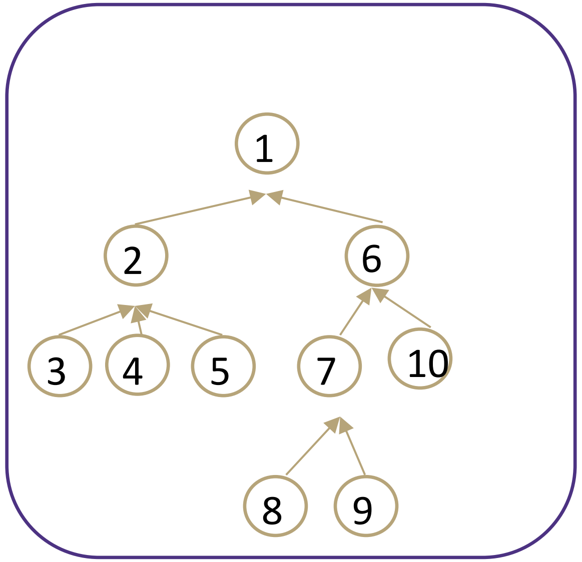
Note: this might feel really different than ADTs we’ve run into before. The ADTs we’ve seen before (dictionaries, lists, sets, etc.) just store values directly. But the Disjoint Set ADT is particularly interested in letting you group your values into sets and keep track of which particular set your values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

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)
}
```

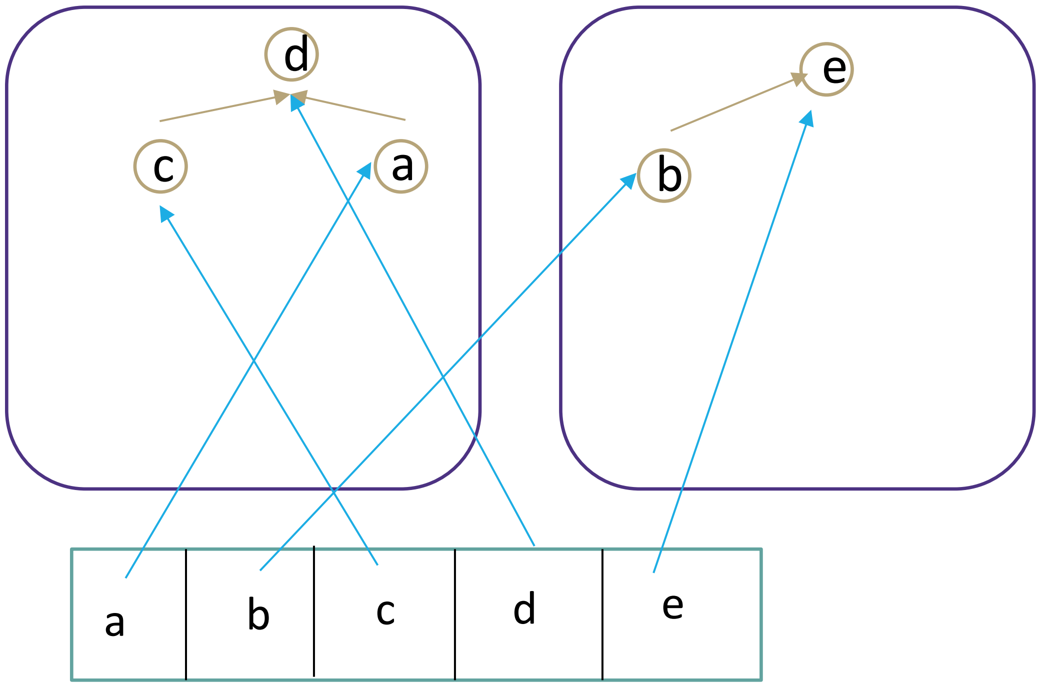


findSet(5) == 1 node

findSet(9) == 9 node

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

jumping to nodes:



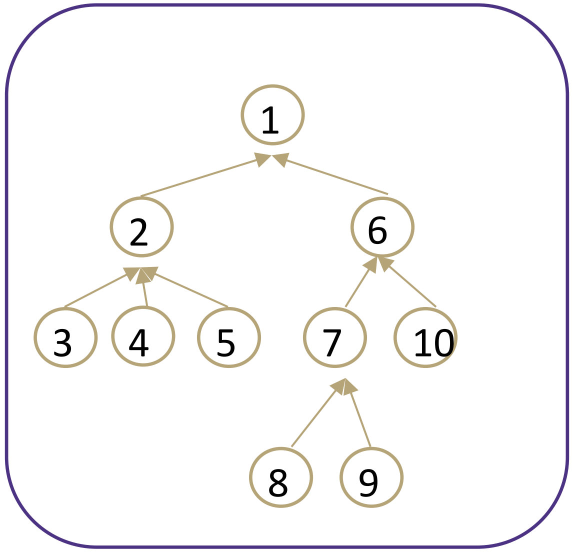
You can use a Map<T, Node> to jump to each node easily (so even though it's not drawn on the future slides, assume we can just jump to any node)

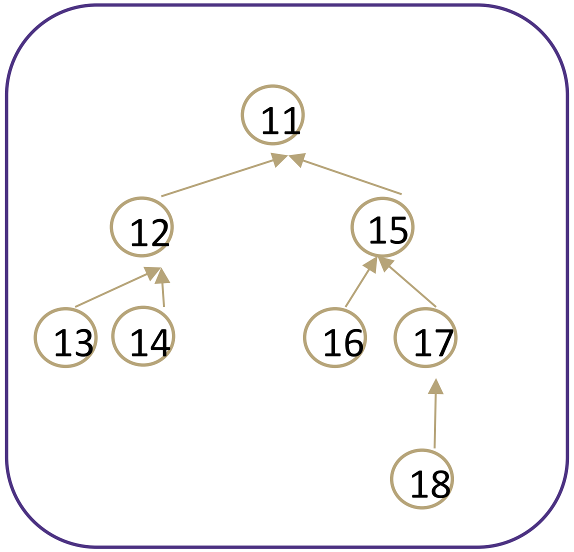
2. QuickUnionTrees implementation: `union(valueA, valueB)`

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
}
```





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)

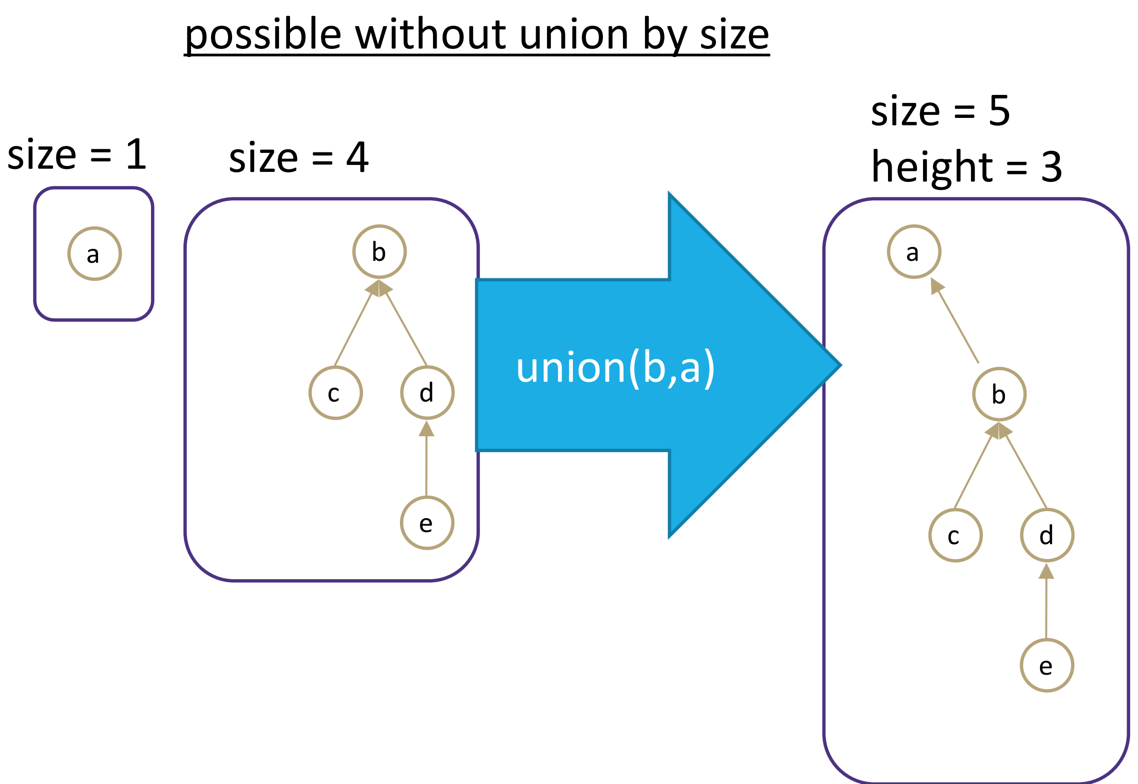
Solution: When union'ing, choose the parent to be the bigger tree

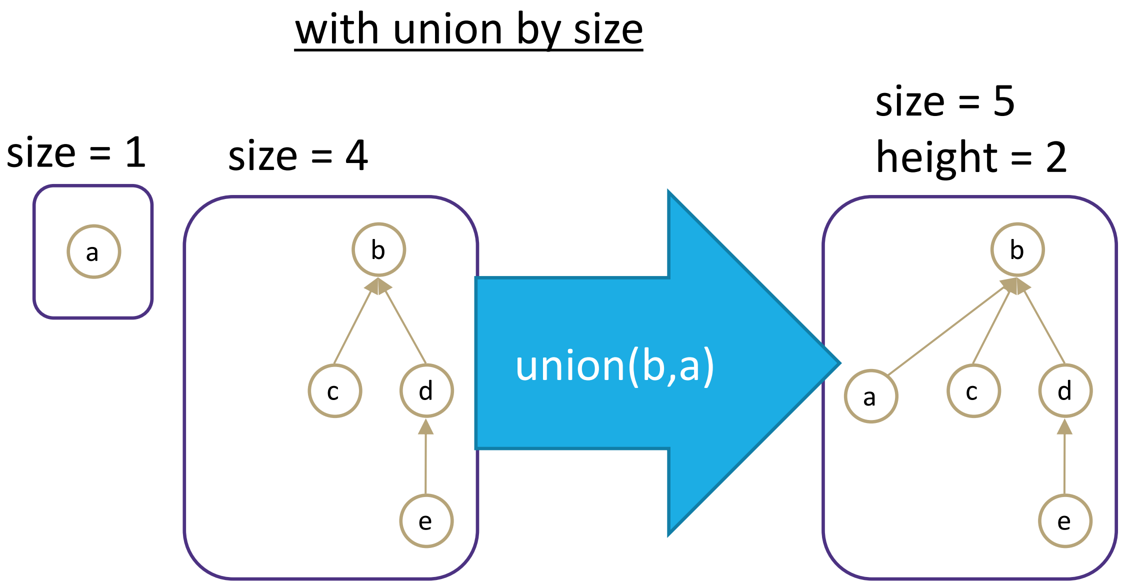
- have the root of each mini-set tree store that tree's size
- **When union'ing make the tree with larger size the root (If it's a tie, pick one arbitrarily)**
- increase the size of the new mini-set as appropriate

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





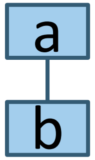
3. QuickUnionBySizeTrees worst case heights

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

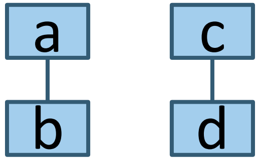
Consider the worst case where the tree height grows as fast as possible for the number of nodes it has



# nodes	height
1	0
2	1

3. QuickUnionBySizeTrees worst case heights

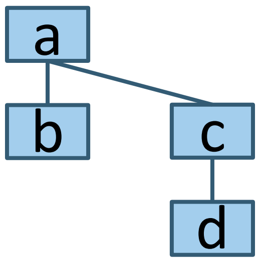
Consider the worst case where the tree height grows as fast as possible for the number of nodes it has



# nodes	height
1	0
2	1

3. QuickUnionBySizeTrees worst case heights

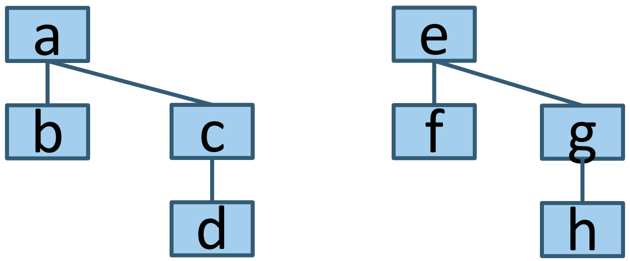
Consider the worst case where the tree height grows as fast as possible for the number of nodes it has



# nodes	height
1	0
2	1
4	2

3. QuickUnionBySizeTrees worst case heights

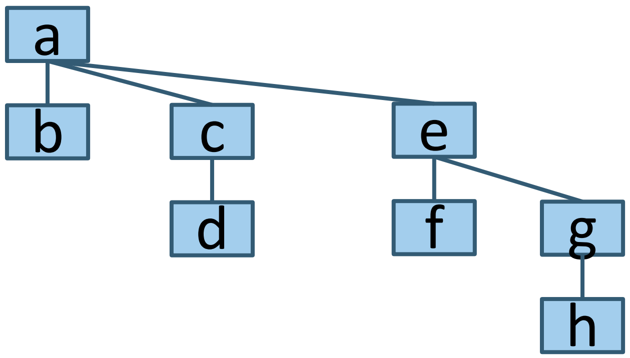
Consider the worst case where the tree height grows as fast as possible for the number of nodes it has



# 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



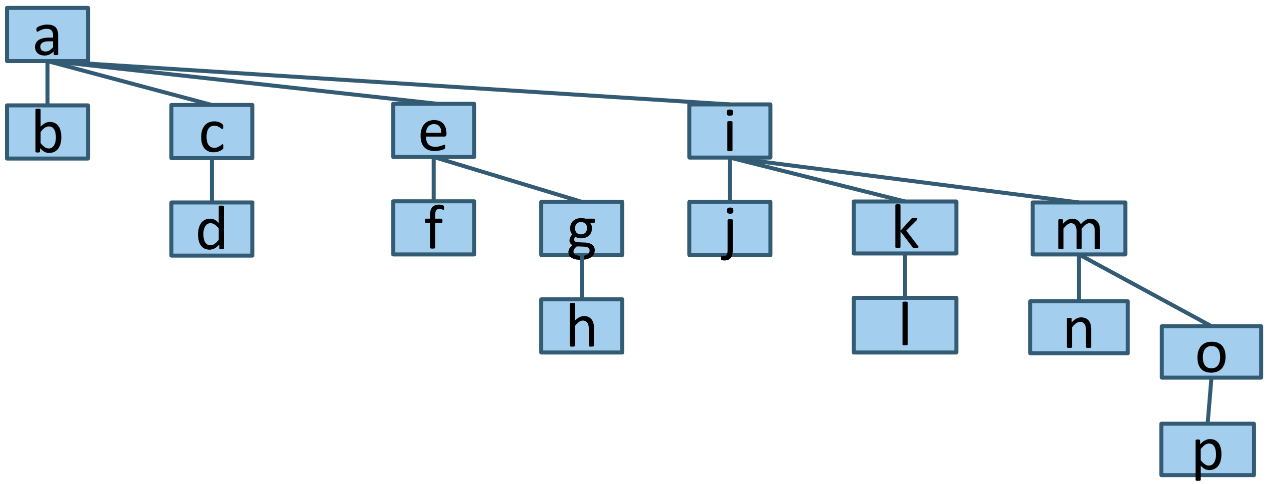
# nodes	height
1	0
2	1
4	2
8	3

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 $\Theta(\log N)$

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



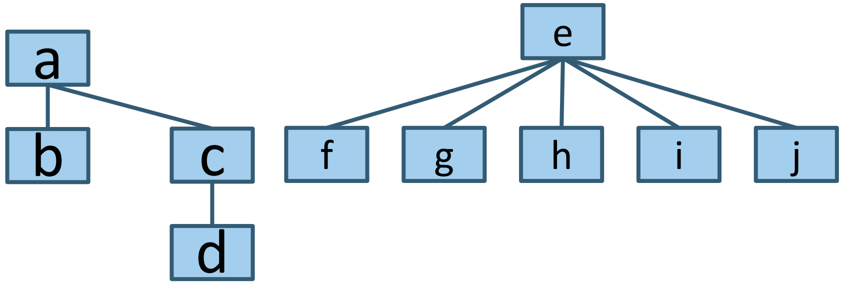
reminder: this is the worst case height since we're trying to increase the height of the tree as much as possible. The best case height can just be at constant height with n nodes if all of them are at level 2 except for the root.

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

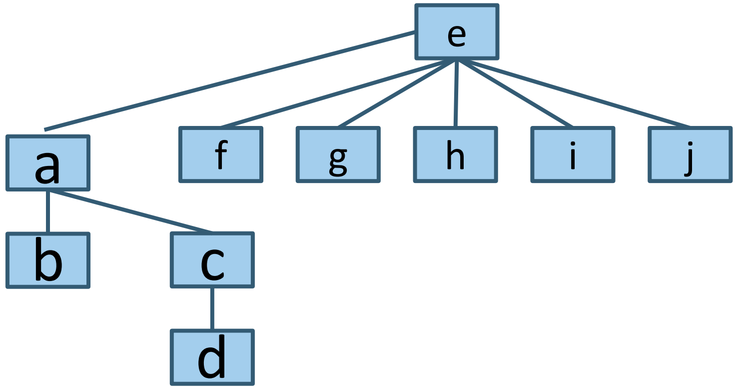
union(a, e) – which one becomes the parent when doing union-by-size?

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.

original disjoint sets

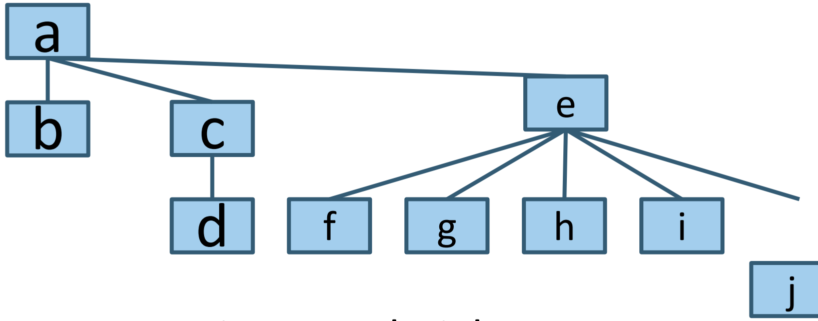


after union by size



size = 10, height = 3

what should happen instead/optimally



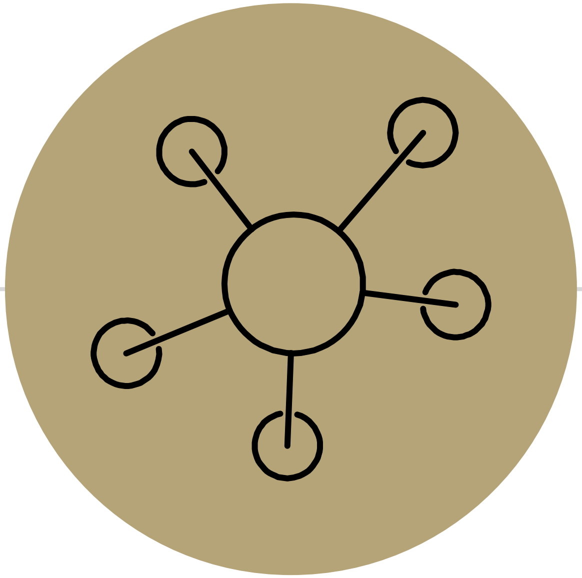
size = 10, height = 2

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. It’s still bounded by the example we did before to show that the height of the tree grows logarithmically in the worst case. If you try to come up with example union calls to create situations like above where union-by-size does the suboptimal thing, you’ll see that the height is still bounded by log(n) – the proof / practice is left as an exercise for the reader ☺.

small aside (just to satisfy curiosity):

why not use the height of the tree?

- QuickUnionByHeightTrees runtime is asymptotically the same: $\Theta(\log(N))$
- It's easier to track weights than heights



Questions break

QuickUnionBySizeCompressingTrees

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")

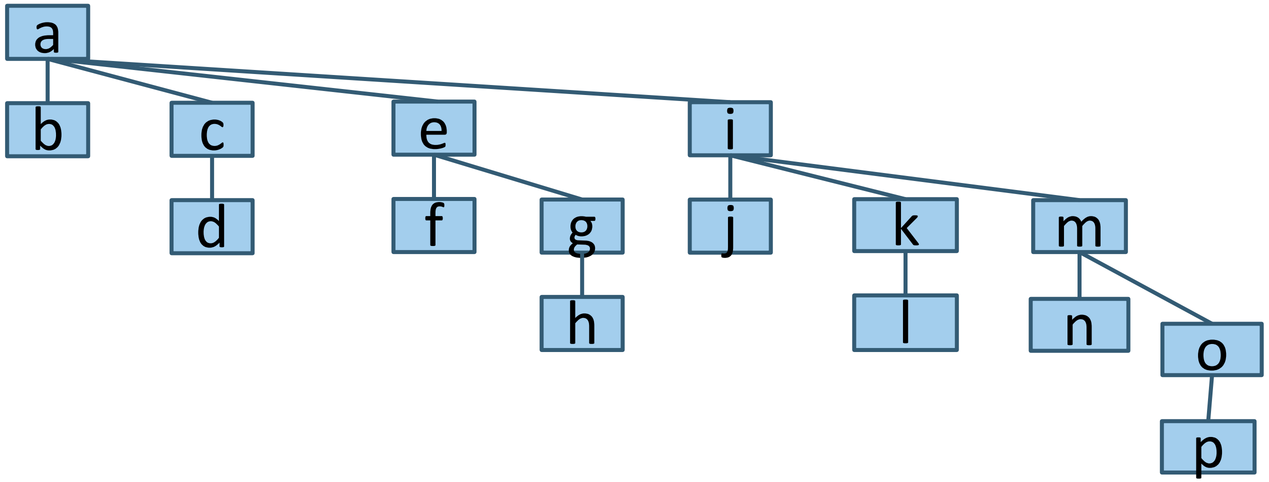
- **Tree rotations**: preserve AVL height invariant so we guarantee log(n) height and log(n) runtime for worst case if we need to traverse to the bottom of the tree
- **heap percolations:** preserve heap sorted invariants so we can find Min/Max still in constant time

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 of related method calls.*

4. QuickUnionBySizeCompressingTrees Path Compression: Idea

This is the worst-case structure / height if we use QuickUnionBySize

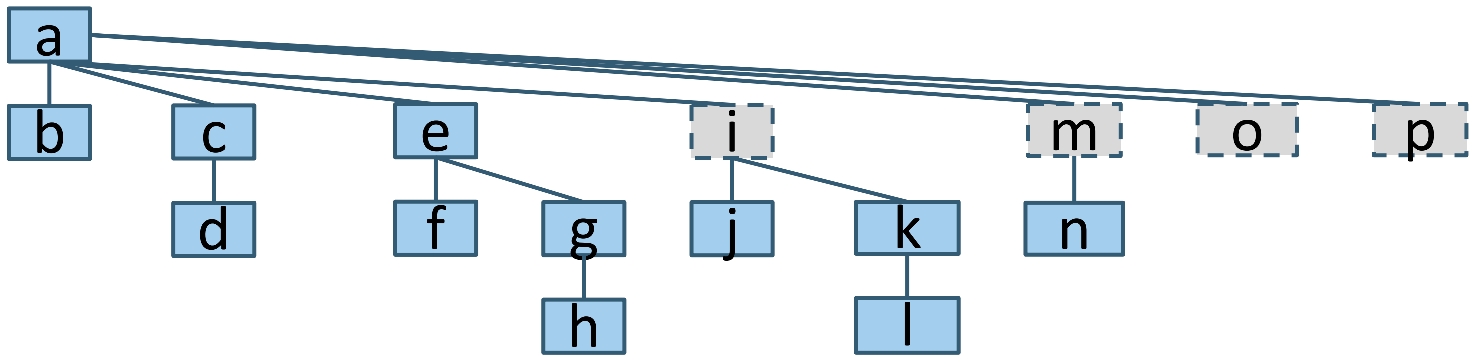


Idea: When we do findSet(p), move all visited nodes to directly point to the root

Additional cost is insignificant (same order of growth to visit all of these nodes one more time)

4. QuickUnionBySizeCompressingTrees Path Compression: Example

This is the worst-case structure / height if we use WeightedQuickUnion

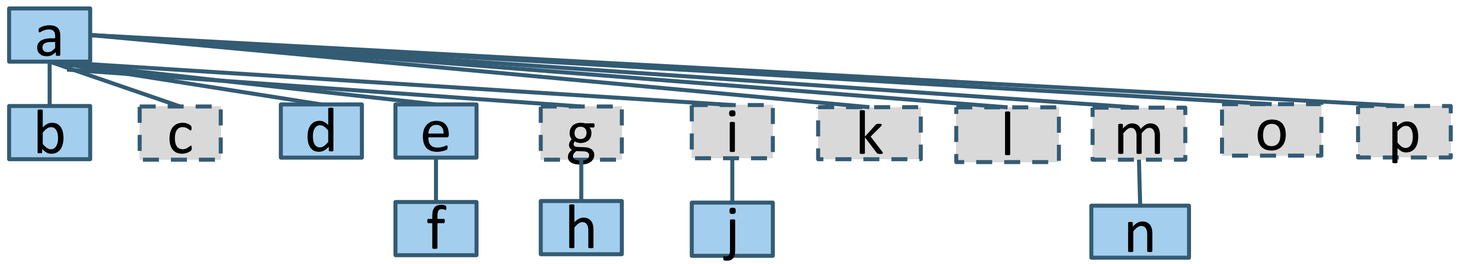


Idea: When we do findSet(p), move all visited nodes under the root

- 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()!



Understanding the performance of more than 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.

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	$\Theta(1)$	$\Theta(1)$*	$\Theta(1)*$

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

* can be thought of as $\Theta(1)$ but technically incorrect notation … it's bounded by a function called the inverse Ackermann function, $\alpha(n)$, that outputs < 5 for any value of *n* that can be written in this physical universe, so the disjoint-sets operations take place in essentially constant time.

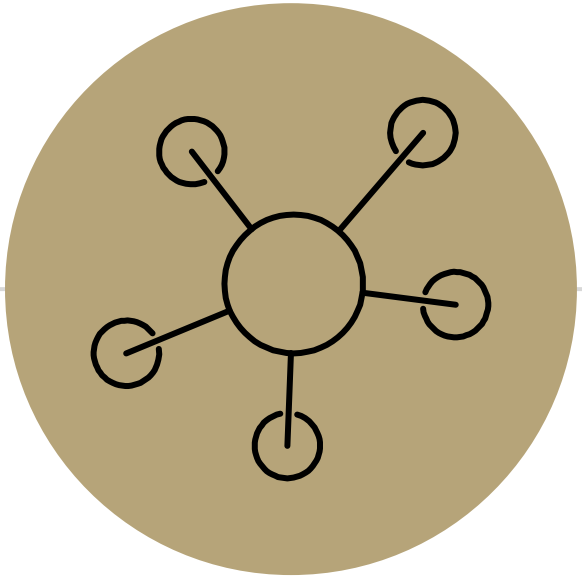
4. QuickUnionBySizeCompressingTrees methods recap

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



Questions break

QuickUnionBySizeCompressingTrees

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

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. So everything we learned about the tree versions conceptually will still exist, we'll just store the data a little differently.

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

5. ArrayQuickUnionBySizeCompressing 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.).

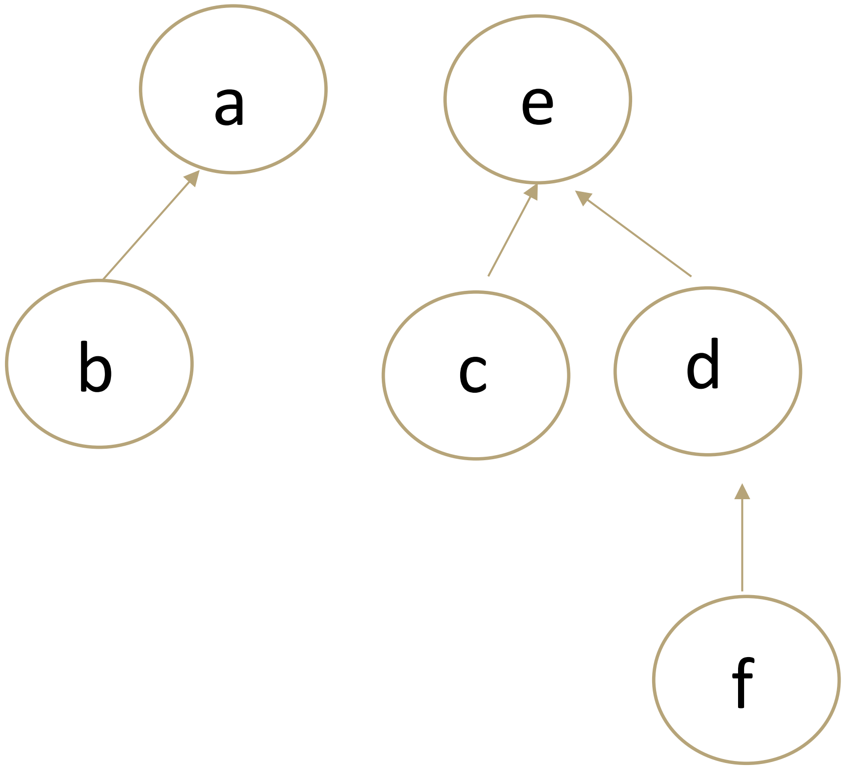
A couple of ideas:

- 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!

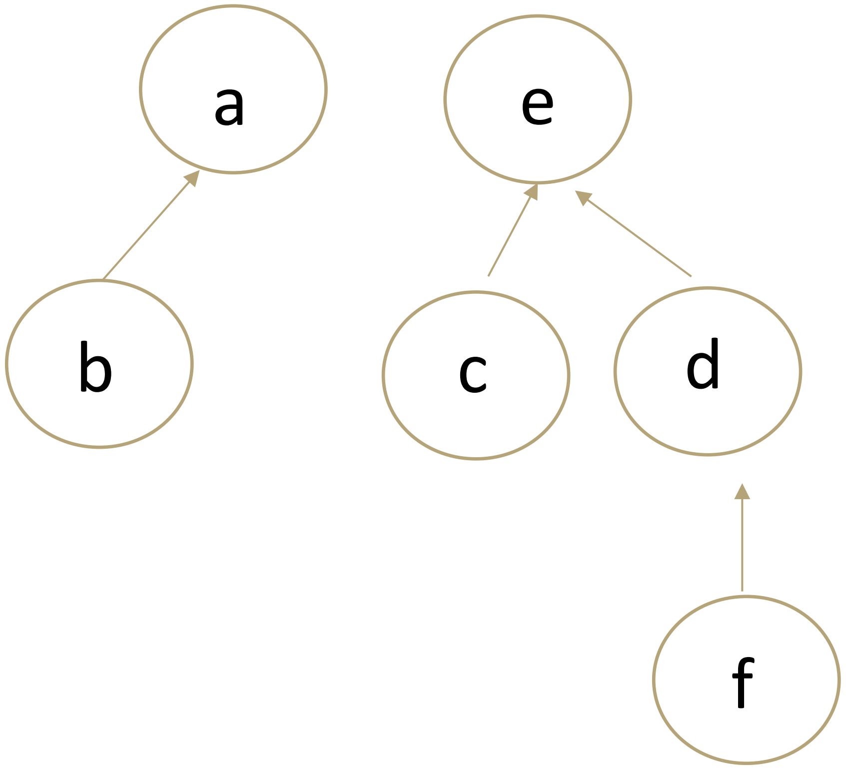
5. ArrayQuickUnionBySizeCompressing
big idea: at each node's position, store the index of the parent node

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



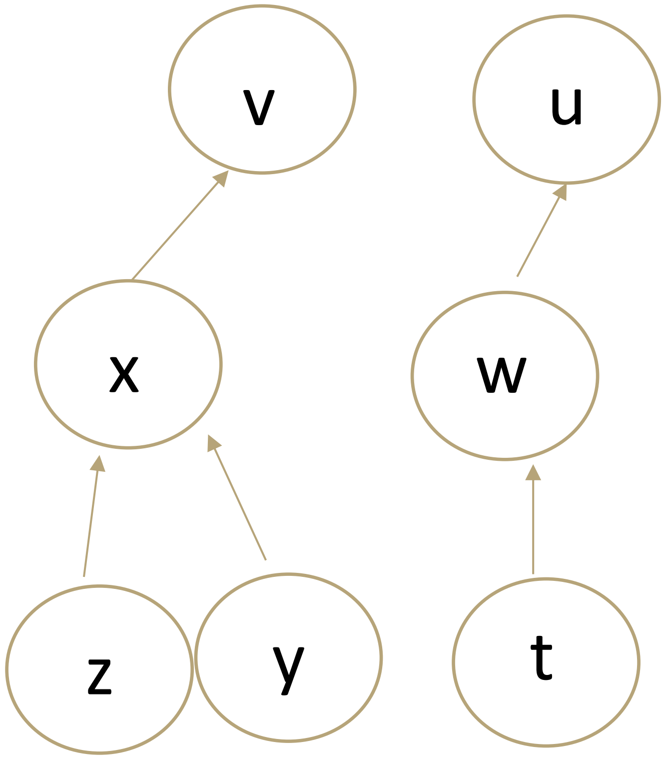
5. ArrayQuickUnionBySizeCompressing
big idea: at each node's position, store the index of the parent node

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



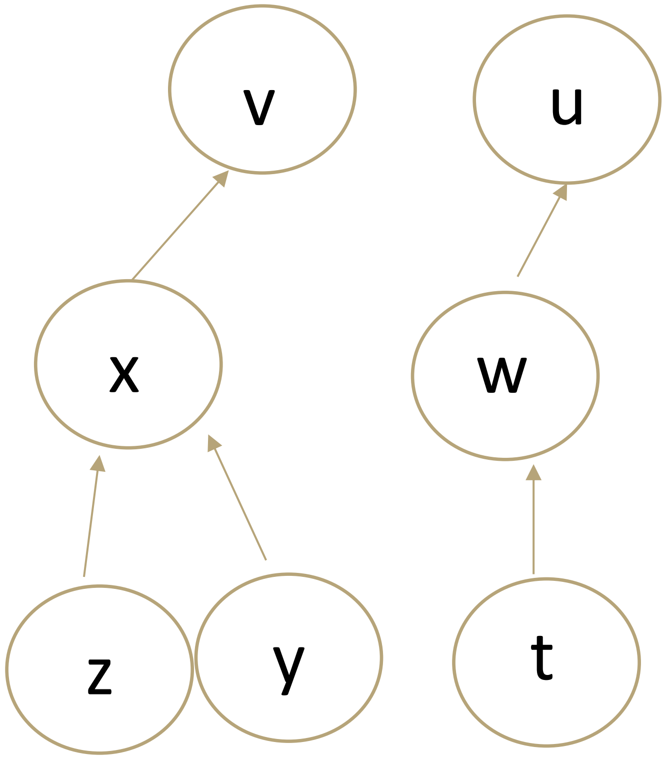
5. ArrayQuickUnionBySizeCompressing
big idea: at each node's position, store the index of the parent node (practice)

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



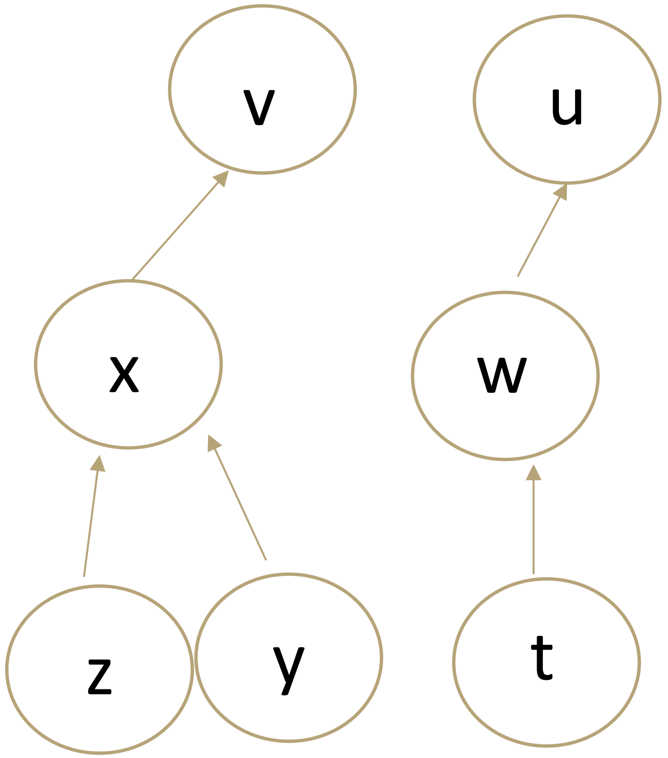
5. ArrayQuickUnionBySizeCompressing
big idea: at each node's position, store the index of the parent node

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



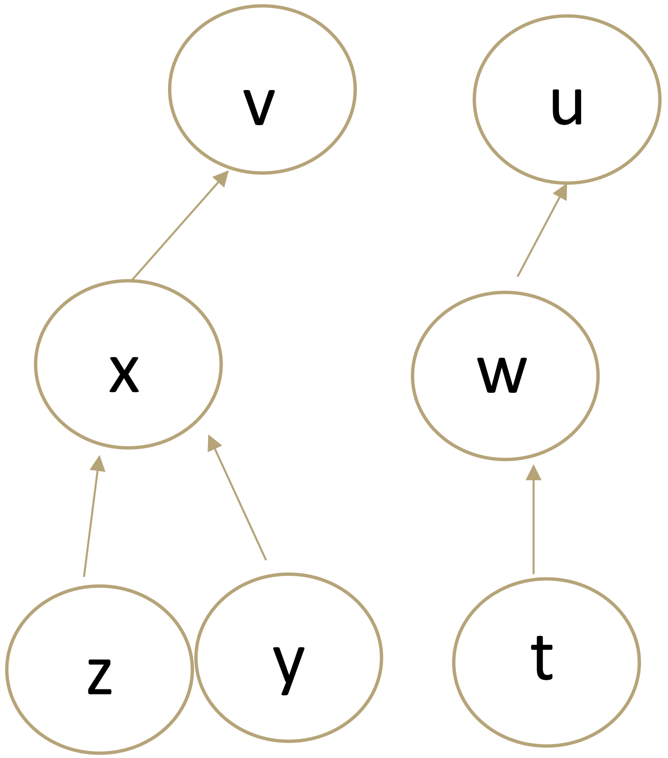
5. ArrayQuickUnionBySizeCompressing
big idea: at each node's position, store the index of the parent node

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



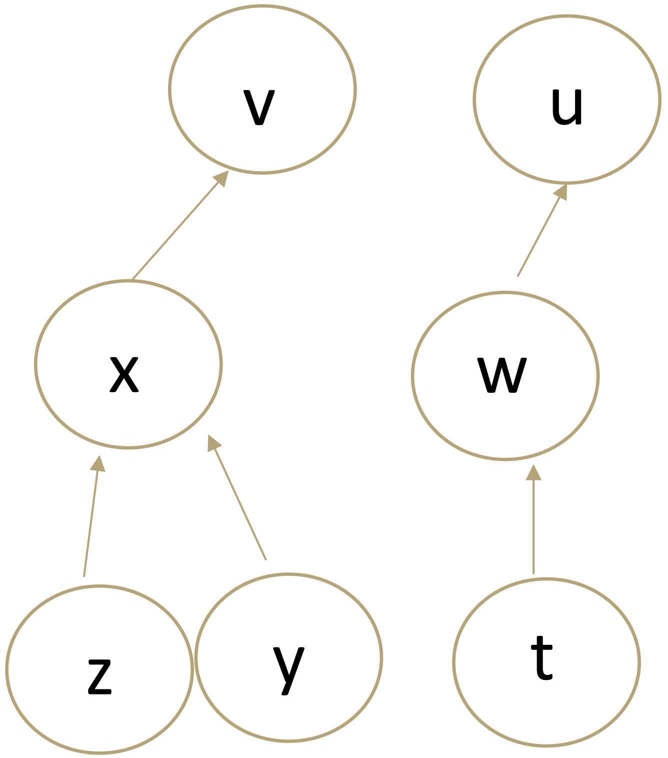
5. ArrayQuickUnionBySizeCompressing
big idea: at each node's position, store the index of the parent node

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



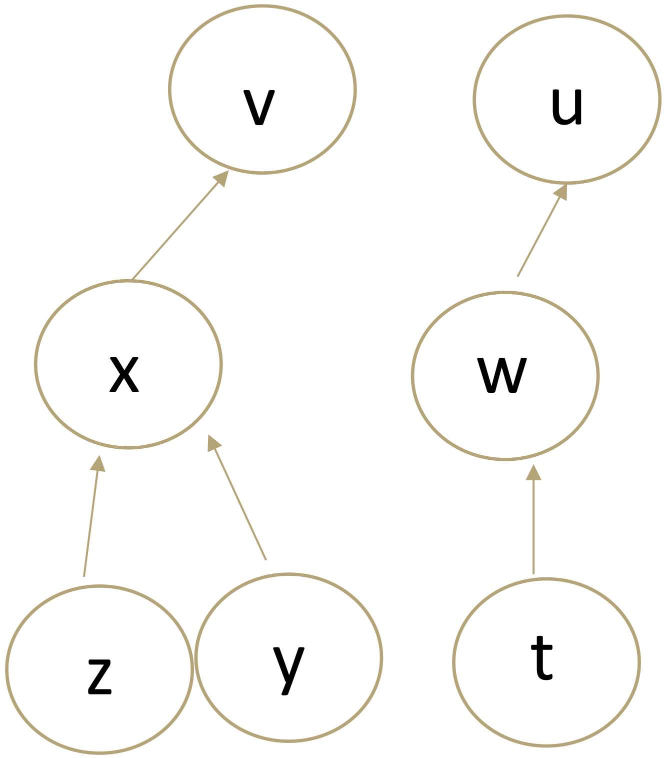
5. ArrayQuickUnionBySizeCompressing
big idea: at each node's position, store the index of the parent node

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



5. ArrayQuickUnionBySizeCompressing
big idea: 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	-	-

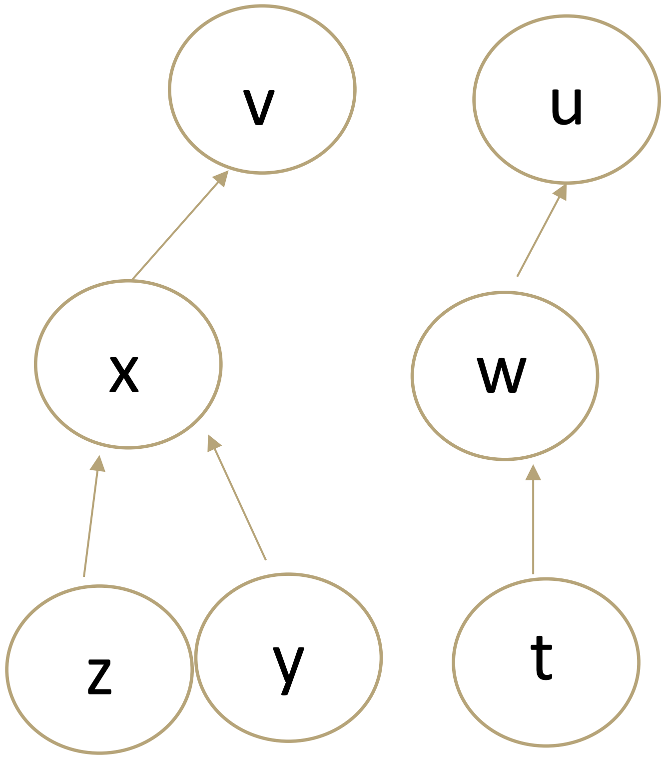


5. ArrayQuickUnionBySizeCompressing findSet()

	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

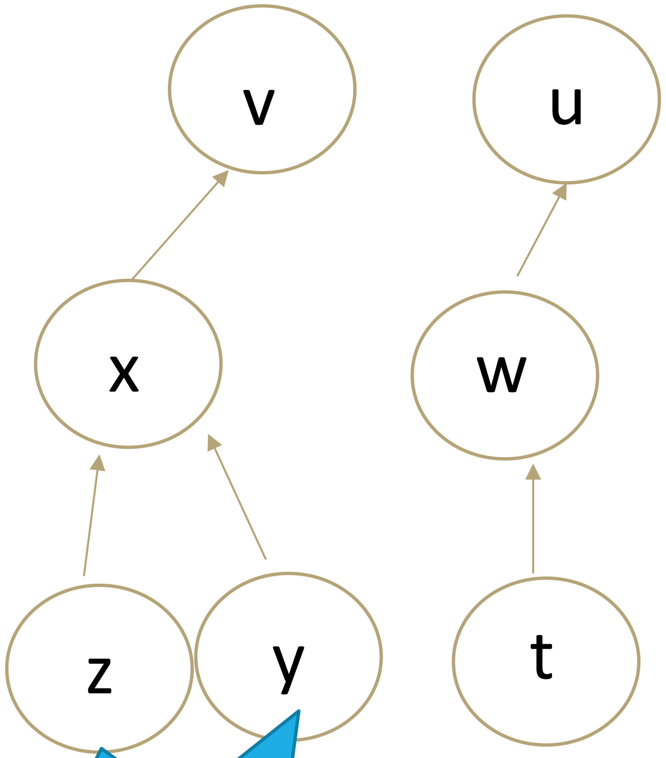


5. ArrayQuickUnionBySizeCompressing findSet(): (Looking up the index for a given value)

	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 heap project.

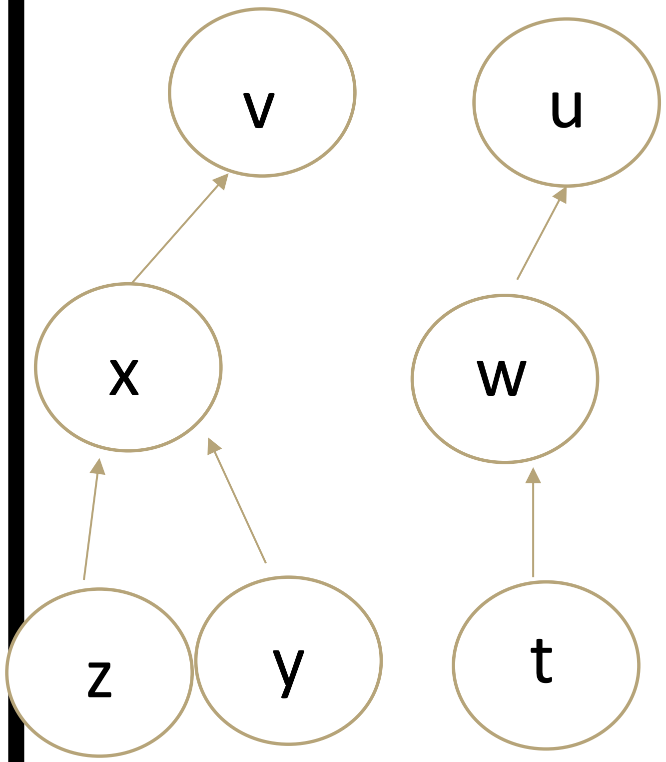


This is a big idea!

5. ArrayQuickUnionBySizeCompressing findSet(): (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?



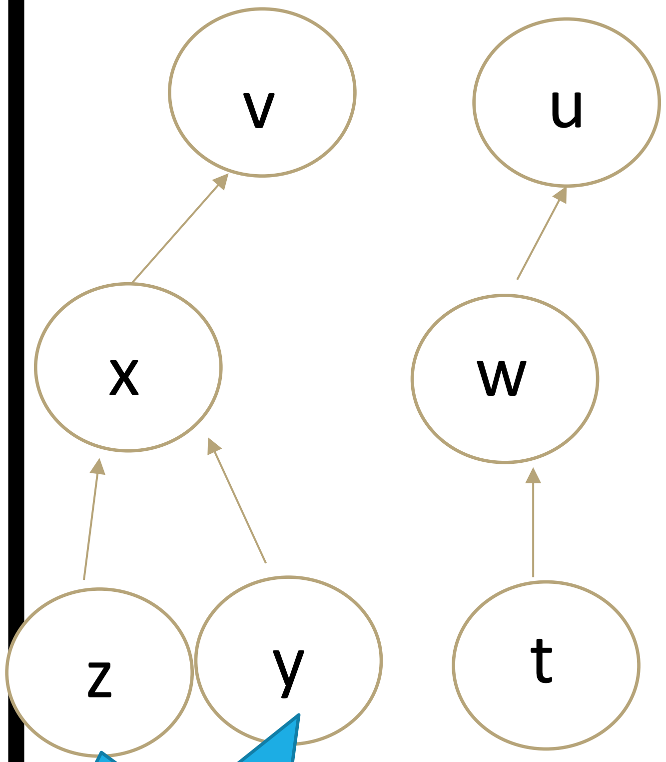
5. ArrayQuickUnionBySizeCompressing findSet(): (What do we store at the root position so we know when to stop?)

	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 the size for our root nodes.



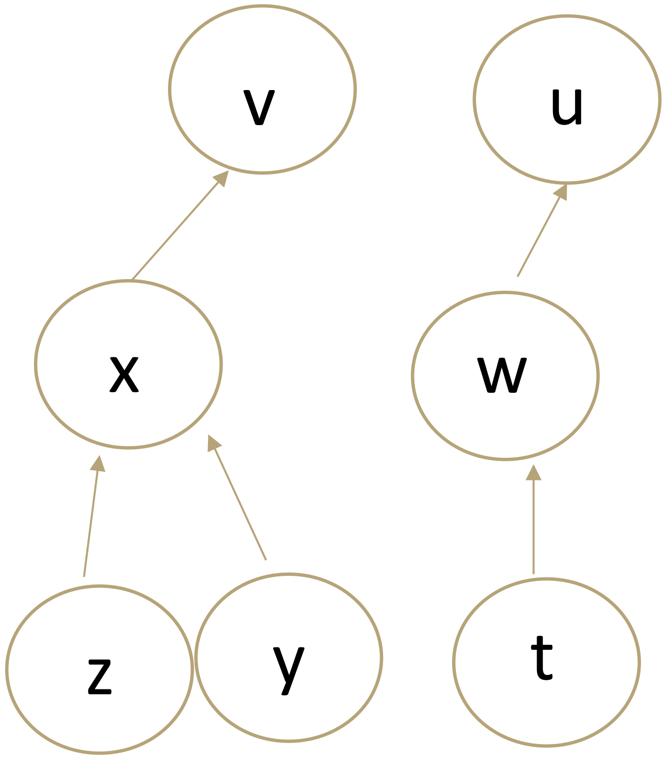
This is a big idea!

5. ArrayQuickUnionBySizeCompressing findSet(): full 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)

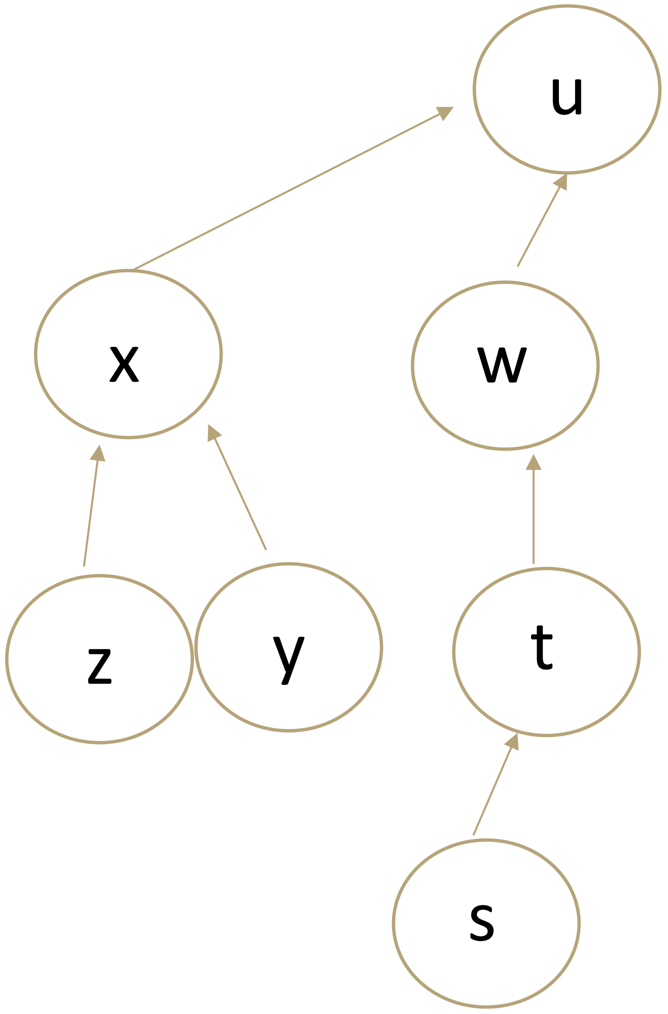
- look up the index of y in our array with index dictionary (index 1)
- keep traversing till we get to the root, signified by negative numbers
- path compression (set everything to point to the index of the root - in this case set everything on the path to 5)
- return the **index** of the root (in this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



5. ArrayQuickUnionBySizeCompressing practice: `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

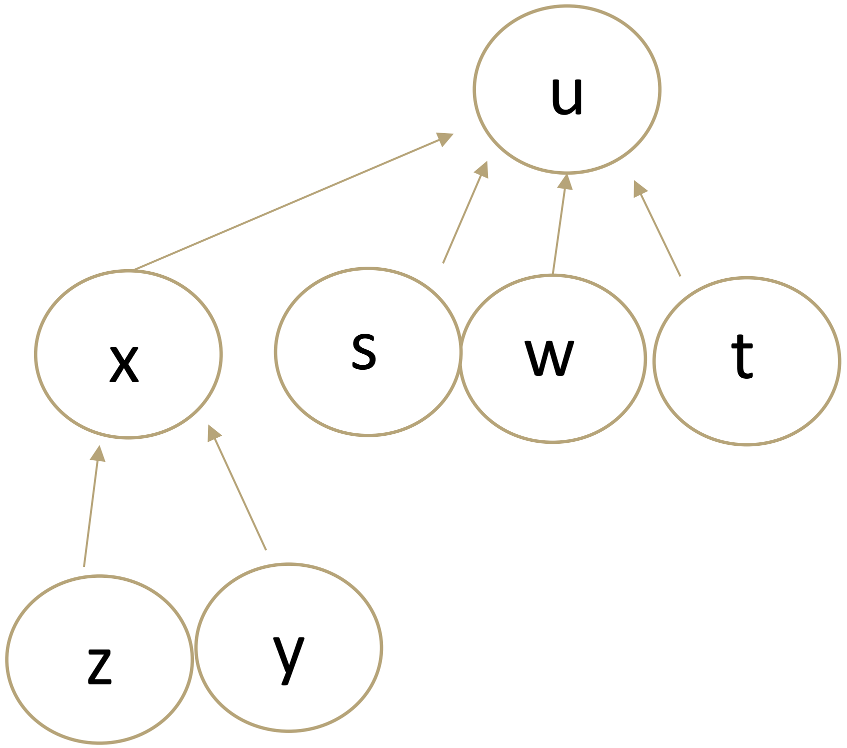
- look up the index of value in our array with index dictionary keep traversing till we get to the root, signified by negative numbers
- path compression (set everything to point to the index of the root)
- return the **index** of the root (in this case return 5). Instead of the actual node itself, we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



5. ArrayQuickUnionBySizeCompressing practice: `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

5. ArrayQuickUnionBySizeCompressing union

note: formula to store in root nodes is negative size

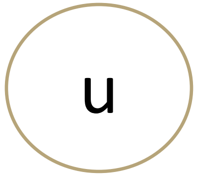
index	0	1	2	3
value	/	/	/	/

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

5. ArrayQuickUnionBySizeCompressing union

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)

5. ArrayQuickUnionBySizeCompressing union

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)

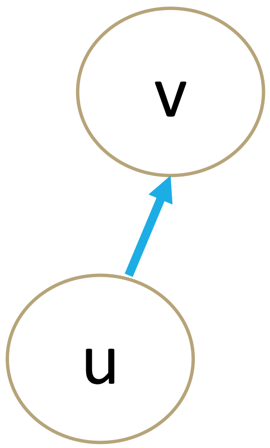
5. ArrayQuickUnionBySizeCompressing union

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)~~

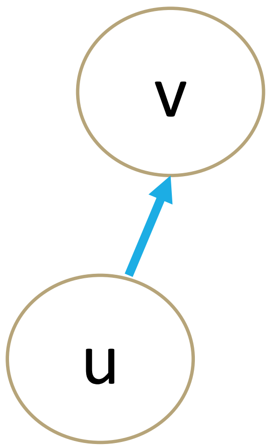
5. ArrayQuickUnionBySizeCompressing union

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)~~

5. ArrayQuickUnionBySizeCompressing union (practice)

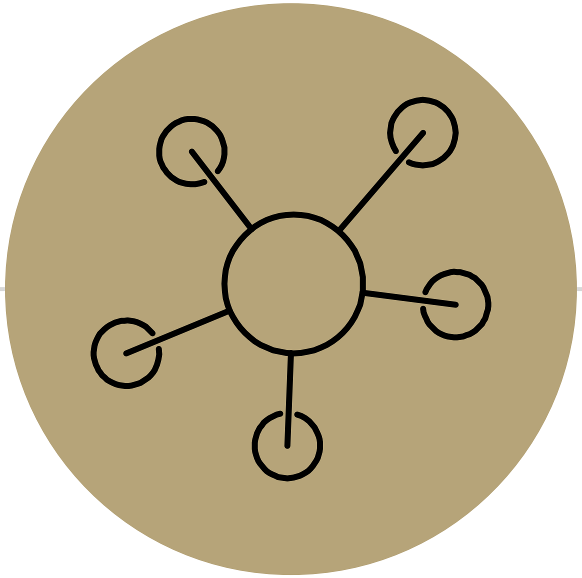
a	b	c	d	e	f	g
0	1	2	3	4	5	6
-1	-1	-1	-1	-1	-1	-1

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)

5. ArrayQuickUnionBySizeCompressing big ideas summary

- 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.



Questions?

Graph Algorithms Review

Breadth First Search (BFS)

Good for:
- establishing connectivity between set of vertices
- Traversing the graph
- Breadth pattern can be leveraged to answer other questions

Algorithm:
- Pick starting vertex
- Add all direct neighbors to queue
- add next in queue to processed list and repeat

Depth First Search (DFS)

Good for:
- establishing connectivity between set of vertices
- Traversing the graph
- Can "stop early" when looking for connectivity between two points

Algorithm:
- Pick starting vertex
- Add all direct neighbors to queue
- add next in stack to processed list and repeat

Dijkstra's

Good for:
- Minimum weight path from source to destination
- Requires weighted edges, no negatives

Algorithm:
- Start at source
- Select next closest neighbor
- Update selected neighbor to sum distance from original source
- Repeat for selected vertex
- Repeat until all vertices processed
- Backtrack from destination vertex to determine path

Prim's

Good for:
- Finding minimum weight set of vertices for complete connectivity
- Requires weighted edges

Algorithm:
- Pick starting vertex
- Add neighbor with smallest weight edge
- Consider all neighbors to current spanning tree
- Select closest neighbor
- Repeat until all vertices are connected

Kruskal's

Good for:
- Finding minimum weight set of vertices for complete connectivity
- Requires weighted edges

Algorithm:
- Sort all edges
- Add lightest edge to solution and place two vertices it connects into a single component
- Add next lightest edge and combine two connected vertices
- Repeat until all vertices are connected

BFS/DFS runtime

Queue for BFS
Stack for DFS

```
perimeter.add(start);
discovered.add(start);
start's distance = 0;
Θ(V)while (!perimeter.isEmpty()) {
    Vertex from = perimeter.remove();
Θ(E of Vx) for (E edge : graph.outgoingEdgesFrom(from)) {
        Vertex to = edge.to();
        if (!discovered.contains(to)) {
            to's distance = from.distance + 1;
            to's predecessorEdge = edge;
      Θ(1) perimeter.add(to);
      Θ(1) discovered.add(to)
        }
    }
}
```

O(E + V)

Dijkstra’s Runtime

```
Dijkstra(Graph G, Vertex source)
   for (Vertex v : G.getVertices()) { v.dist = INFINITY; } Θ(V)
   G.getVertex(source).dist = 0;
   initialize MPQ as a Min Priority Queue, add source
Θ(V) while(MPQ is not empty){
      u = MPQ.removeMin();Θ(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) } Θ(logV)
            else { MPQ.updatePriority(v, newDist) }
         }
      }
   }
```

This actually doesn’t run E times for every iteration of the outer loop. It actually will run E 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(V log V + E log V)

Prim’s Runtime

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

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

$\Theta(V \log V + E \log V)$

$\Theta(V \log V + E \log V)$

*Prim’s is the same algorithm as Dijkstra’s with a different if check which doesn’t impact the runtime

Kruskal's Runtime

For MST algorithms, assume that E dominates V
(if it doesn't, there is no spanning tree to find)

```
KruskalMST(Graph G)
    initialize new DisjointSets DS
    for(v : G.vertices) { DS.makeSet(v) }
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(DS.findSet(u) != DS.findSet(v)){
            add (u,v) to the MST
            DS.union(u,v)
        }
    }
```

$\Theta(V)$

$\Theta(E \log E)$

E calls

V calls, do

Intuition: We could make the $\log V$ running time happen once…but not really more than that.
Since we're counting total operations, we're actually going to see the "in-practice" behavior

Whether we hit worst-case or not: $\Theta(E \log E)$ is dominating term.

Graph problems

What algorithms would you use to solve each of the following?

1. s-t Path. Is there a path between vertices s and t? BFS or DFS
2. Connectivity. Is the graph connected? BFS or DFS
3. Biconnectivity. Is there a vertex whose removal disconnects the graph? BFS or DFS
4. Shortest s-t Path. What is the shortest path between vertices s and t? Dijkstra's
5. Cycle Detection. Does the graph contain any cycles? Prim's or Kruskal's