Lecture 27: Array Disjoint Sets

CSE 373: Data Structures and Algorithms

Warmup

DisjointSet tree implementation 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)
   - if treeA's rank == treeB's rank:
     - It doesn't matter which is the parent so choose arbitrarily. Increase the rank by one.
   - otherwise:
     - Choose the larger rank tree to become the parent. The rank is just the rank of the parent.

Optimized Disjoint Set Runtime

makeSet(x)

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

findSet(x)

Without Optimizations	**O(n)**
With Optimizations	**on average : O(1)**

union(x, y)

Without Optimizations	**O(n)**
With Optimizations	**on average : O(1)**

Announcements

- hw7 out (due next Friday)
- hw5 p2 feedback went out last night / this morning
- final topics out Monday
- final exam review session next Wed (4pm – 5:50pm)
- please fill out course evals when they come out

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.).

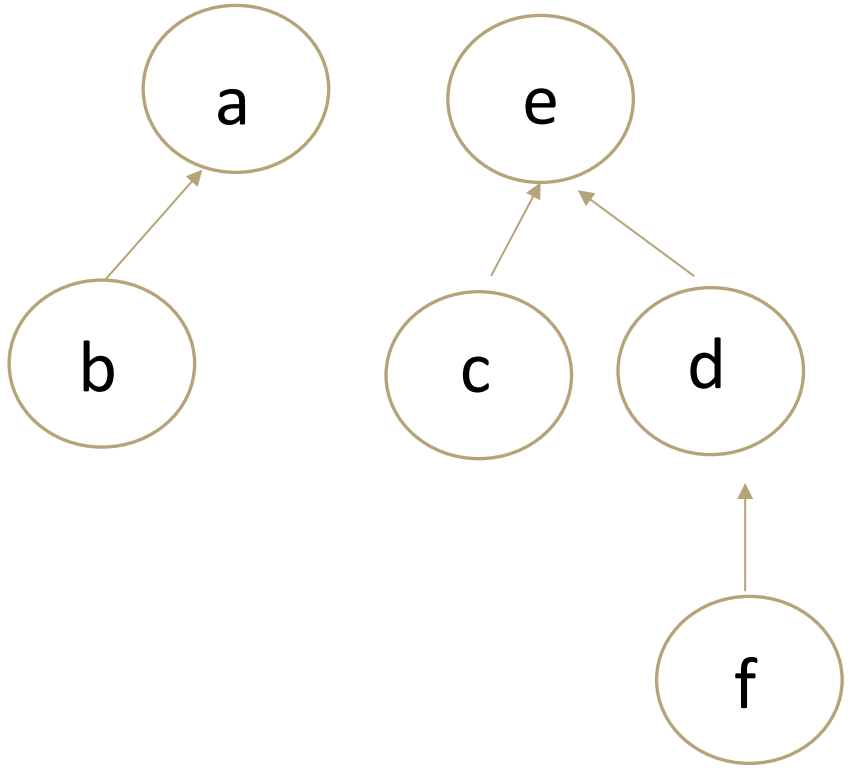
A couple of ideas:

- 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

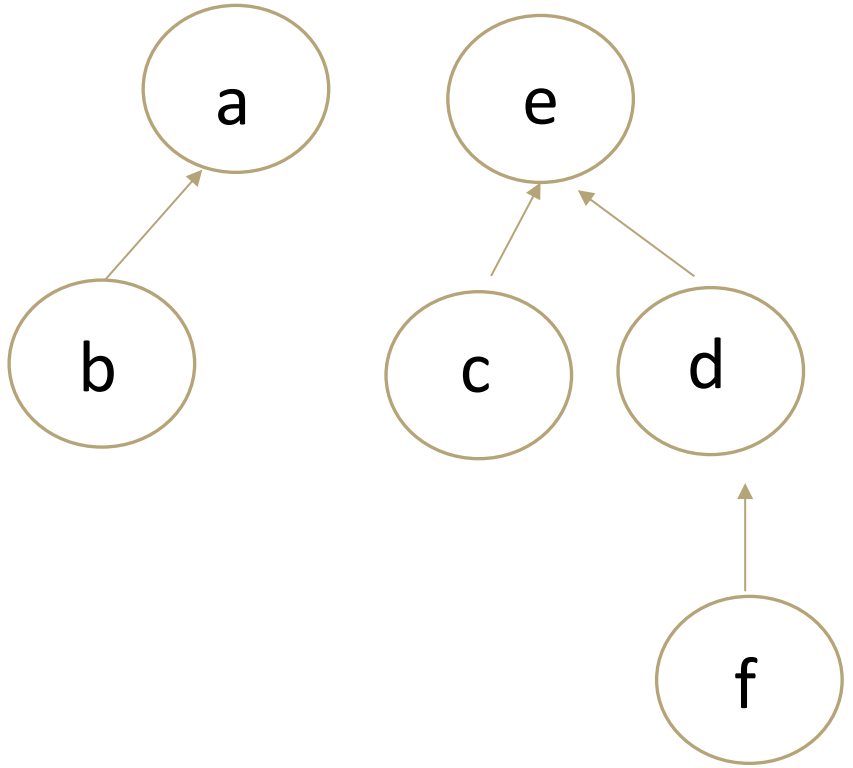
at each node's position, store the index of the parent node

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



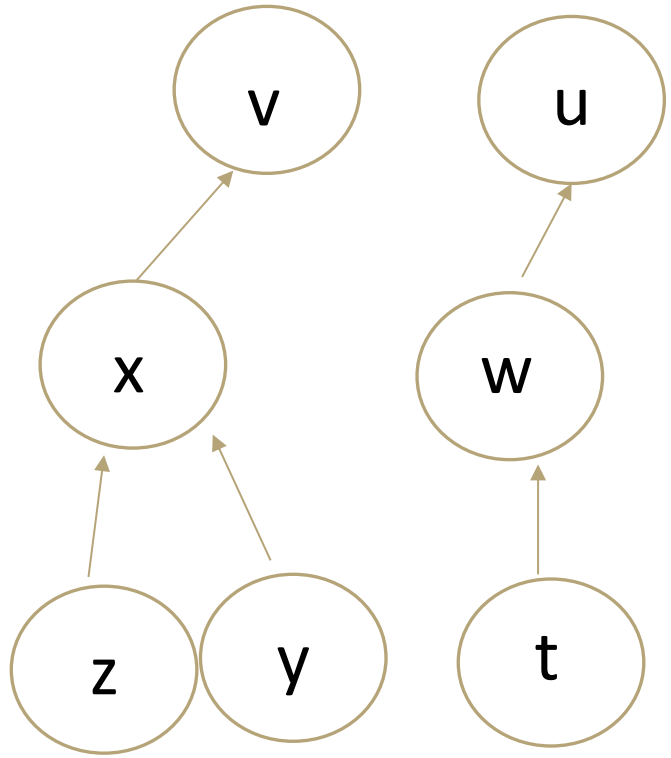
at each node's position, store the index of the parent node

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



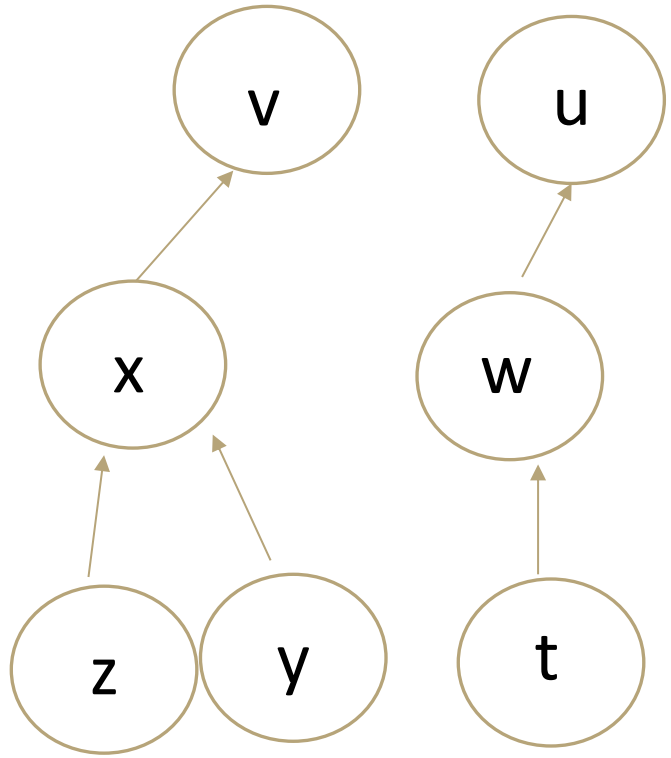
Exercise (1 min)
at each node's position, store the index of the parent node

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



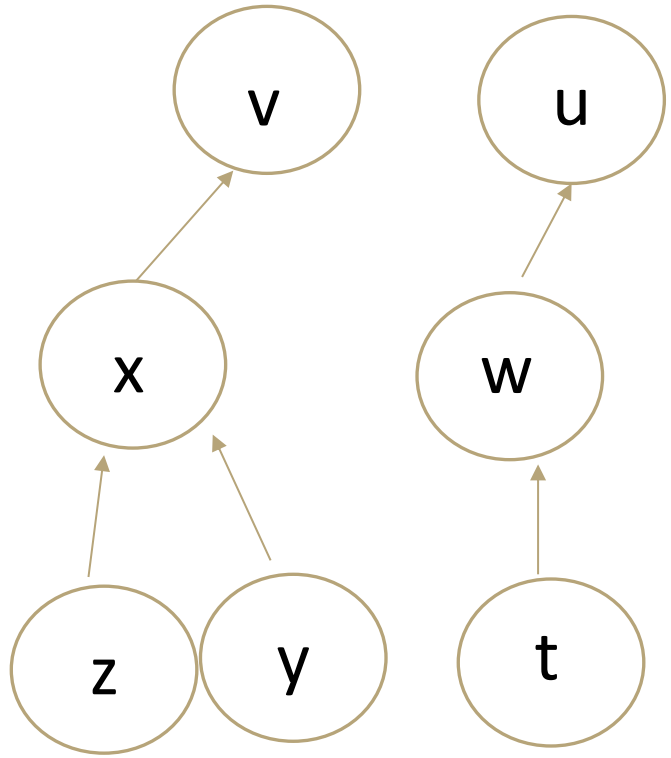
Exercise (1 min)
at each node's position, store the index of the parent node

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



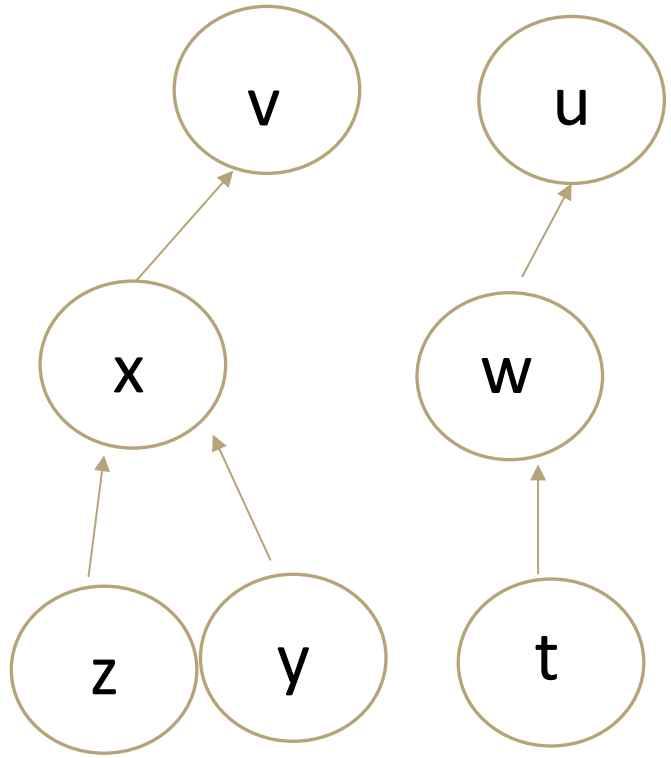
Exercise (1 min)
at each node's position, store the index of the parent node

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



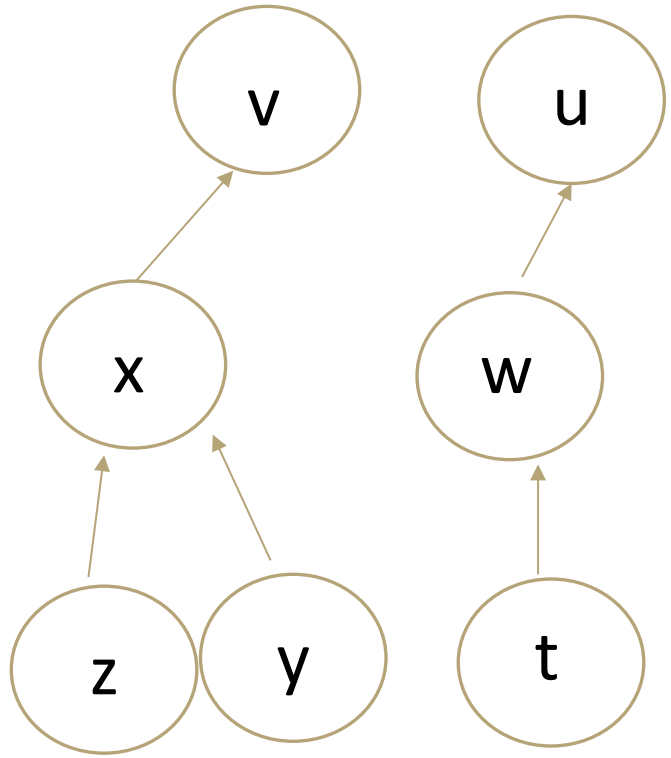
Exercise (1 min)
at each node's position, store the index of the parent node

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



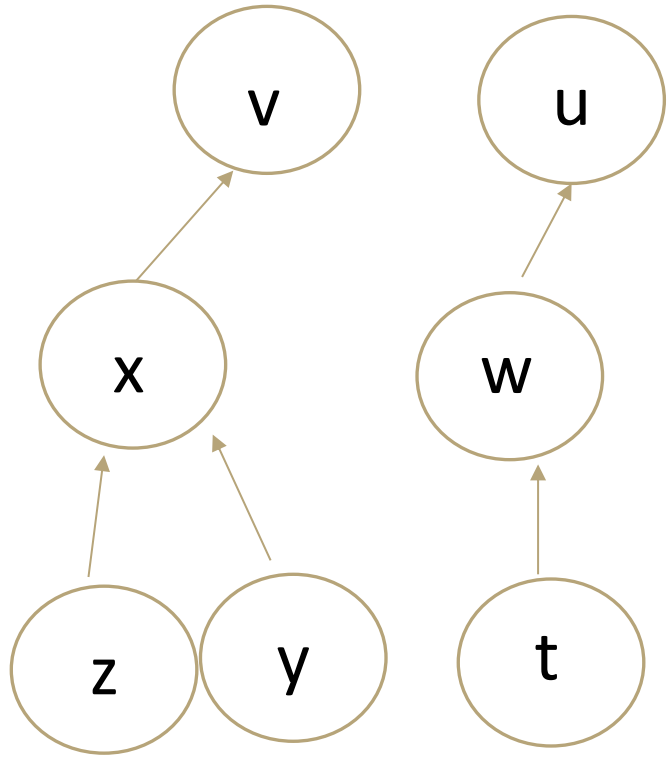
Exercise (1 min)
at each node's position, store the index of the parent node

	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	-	-

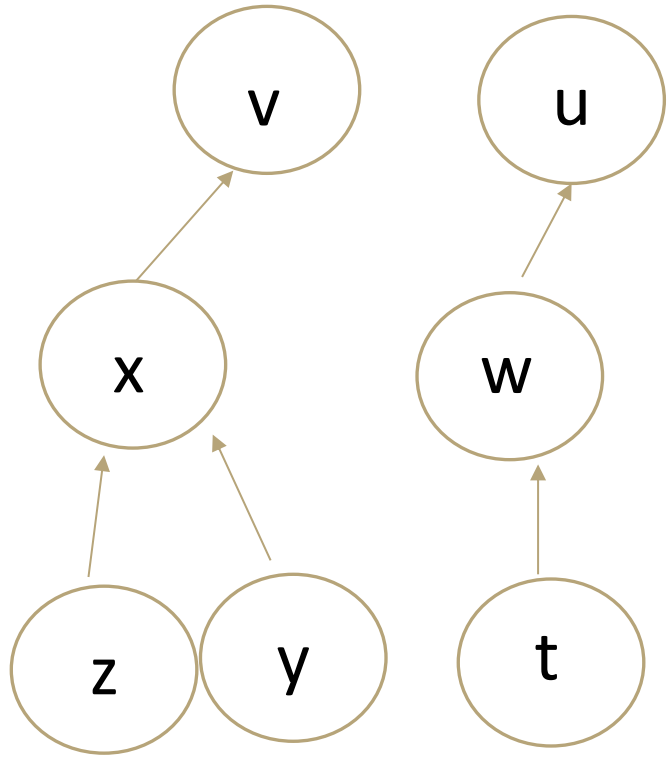


How would findSet work for array implementation?

	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)

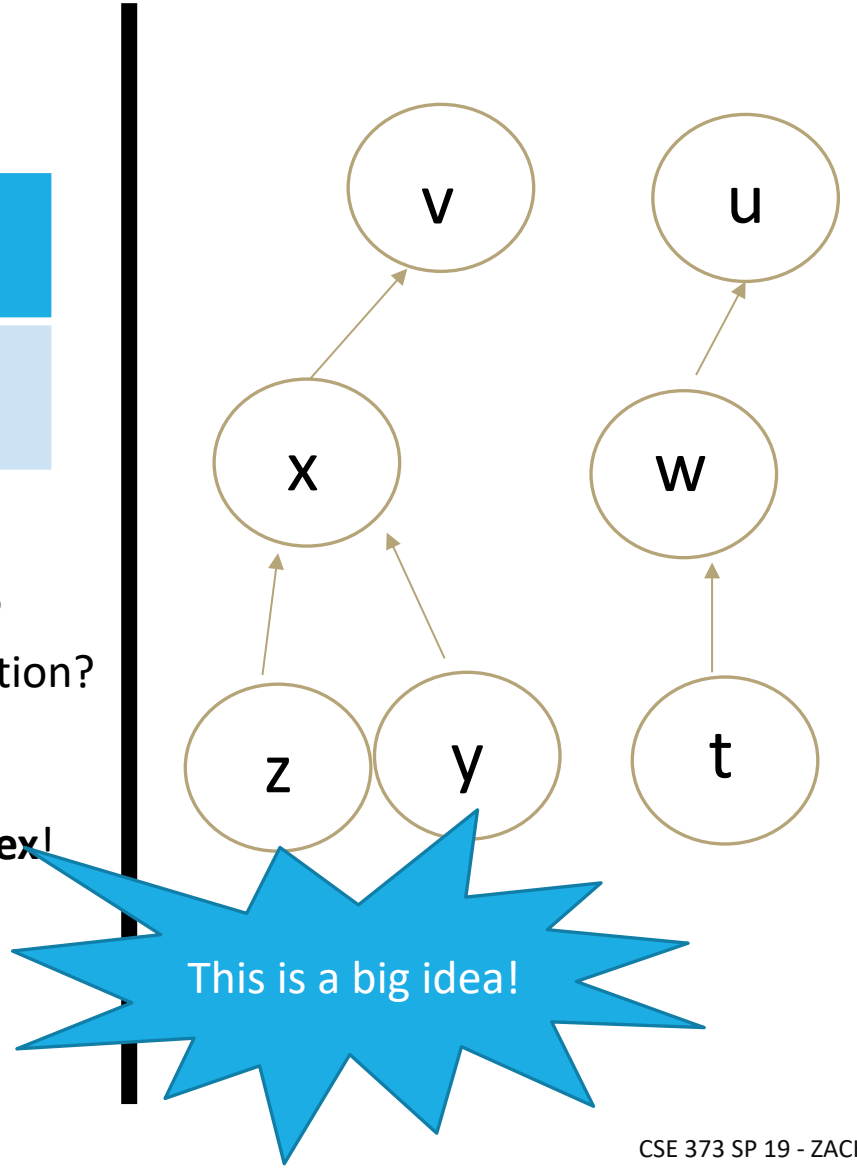


How would findSet work for array implementation?
(Looking up the index for a given value)

	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?

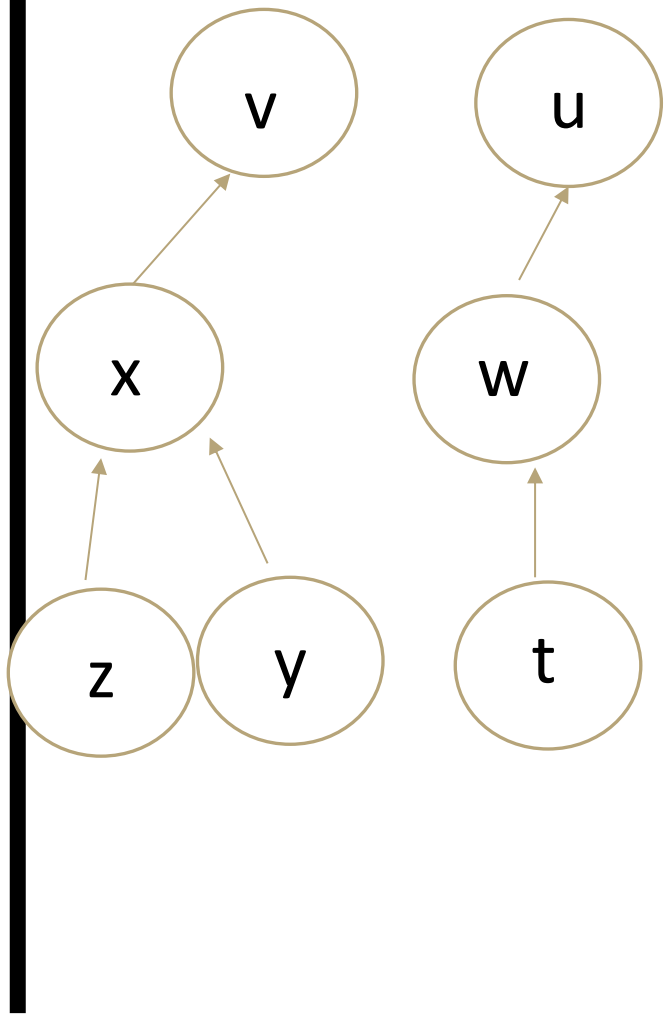
Whenever you add new values into your disjoint set,
keep track of what index you stored it at with a **dictionary of value to index**!
This is similar to the thing as what we did in our ArrayHeap.



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?)

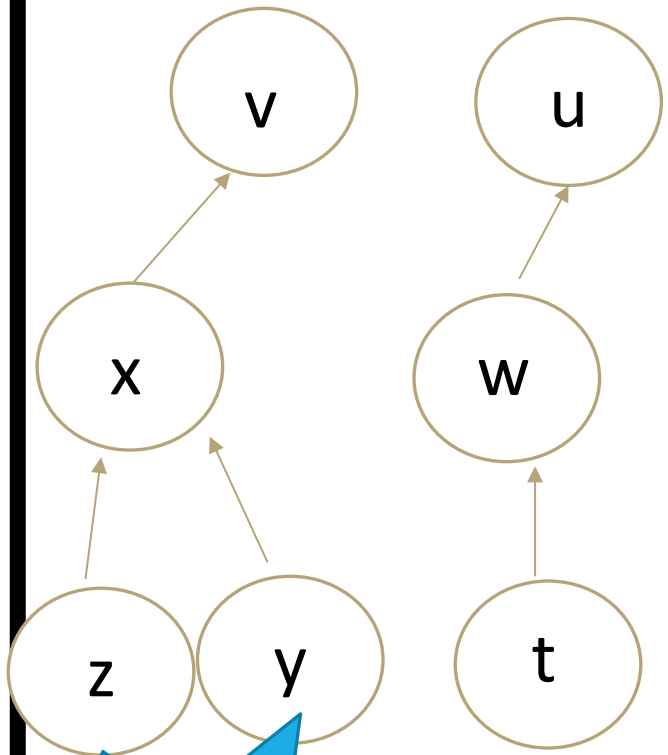
	z	y	t	x	w	v	u
index	0	1	2	3	4	5	6
value	3	3	4	5	6	-3	-3

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 ra… for our root nodes, we'll store (-1 * rank) - 1.

Note: You can basically count how many levels of nodes there are and just tack on a negative sign.



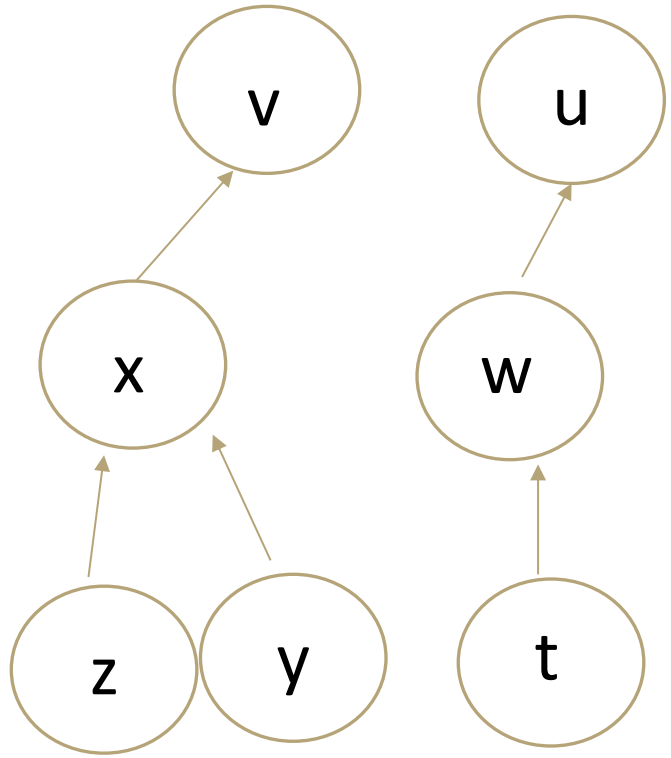
This is a big idea!

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	-3	-3

example : findSet(y)

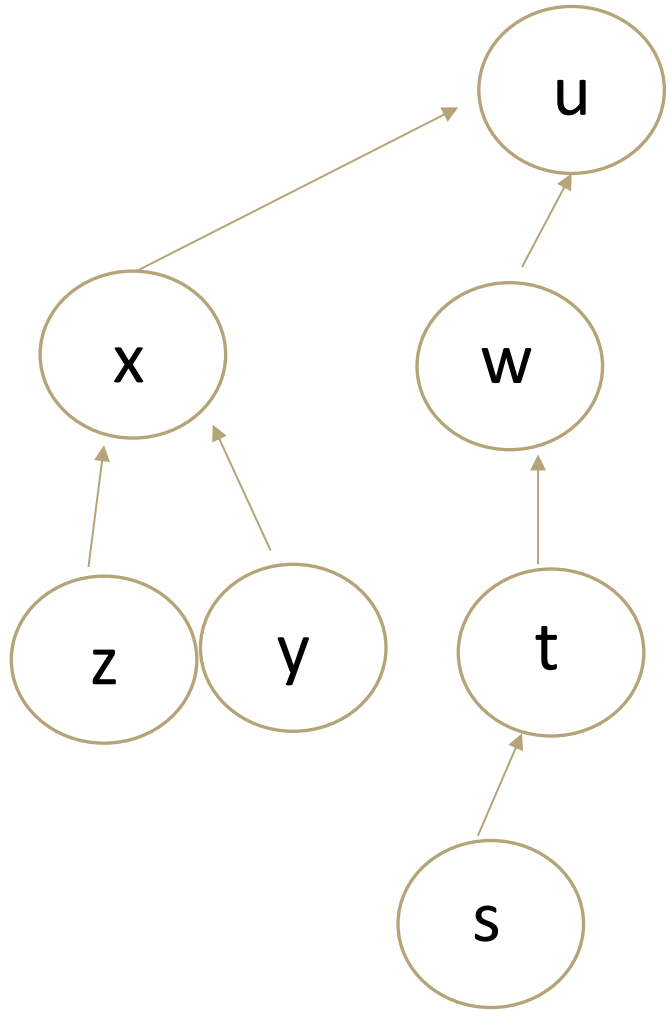
- look up the index of y in our array with index dictionary (index 1)
- keep traversing till we get to the root, signified by negative numbers
- path compression (set everything to point to the index of the root - in this case set everything on the path to 5)
- return the **index** of the root (in this case return 5)



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	-4	2

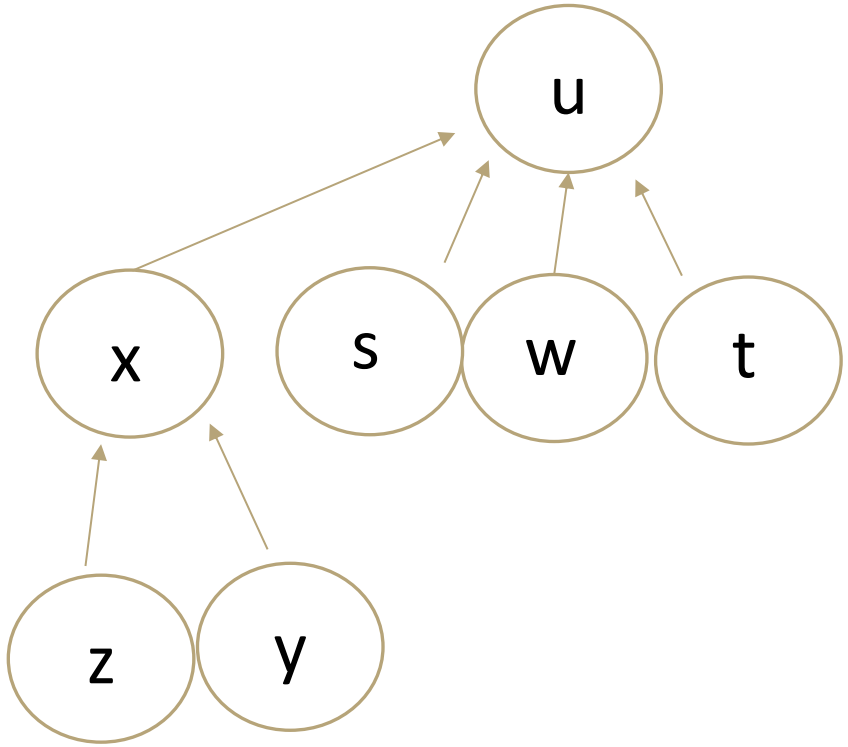
- look up the index of value in our array with index dictionary keep traversing till we get to the root, signified by negative numbers
- path compression (set everything to point to the index of the root)
- return the **index** of the root



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	-4	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



returns 5

How would union work for array implementation?

note: formula to store in root notes is (-1 * rank) - 1

index	0	1	2	3
value	/	/	/	/

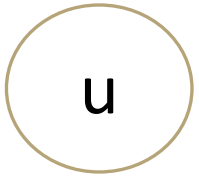
makeSet(u)
makeSet(v)
union(u, v)

How would union work for array implementation?

note: formula to store in root notes is (-1 * rank) - 1

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 notes is (-1 * rank) - 1

u v

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

~~makeSet(u)~~

~~makeSet(v)~~

union(u, v)

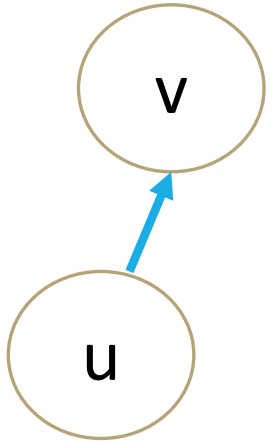
How would union work for array implementation?

note: formula to store in root notes is (-1 * rank) - 1

	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)~~

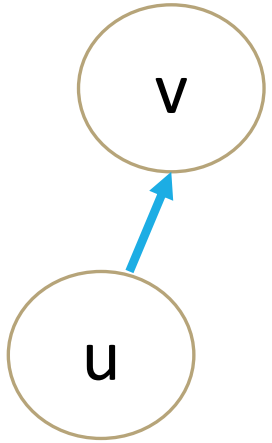
How would union work for array implementation?

note: formula to store in root notes is (-1 * rank) - 1

	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: since this was a tie, we update the rank to be 1 bigger than before. Because we store (-1 * rank) – 1 in our array, this actually just the same as subtracting 1. Before we stored -1 because the rank was 0, and now when the rank is 1 we'll store -2.



~~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 * (rank + 1), 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 (rank of 0) 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 ranks and indices of valueA and valueB's sets. Compare the ranks like in the tree representation. You'll have to be careful when you look up the rank, as the formula stored is (-1 * rank) – 1. If you need to increase the rank because of a tie in ranks, you can just subtract 1 from the current value stored (see previous slide).