Data Structures
Disjoint Sets

CS 225
Brad Solomon

October 16, 2024

Nobody panic

There are no batteries to start class

IT on the way!

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science

Survey EC

Current EC at semesters end: +12

Credit for stickers, lists, and IEF

Bonus video!

Great work!

Exam 3 (10/23 — 10/25)

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam on PL

Topics covered can be found on website

Registration started October 10

https://courses.engr.illinois.edu/cs225/fa2024/exams/

Learning Objectives

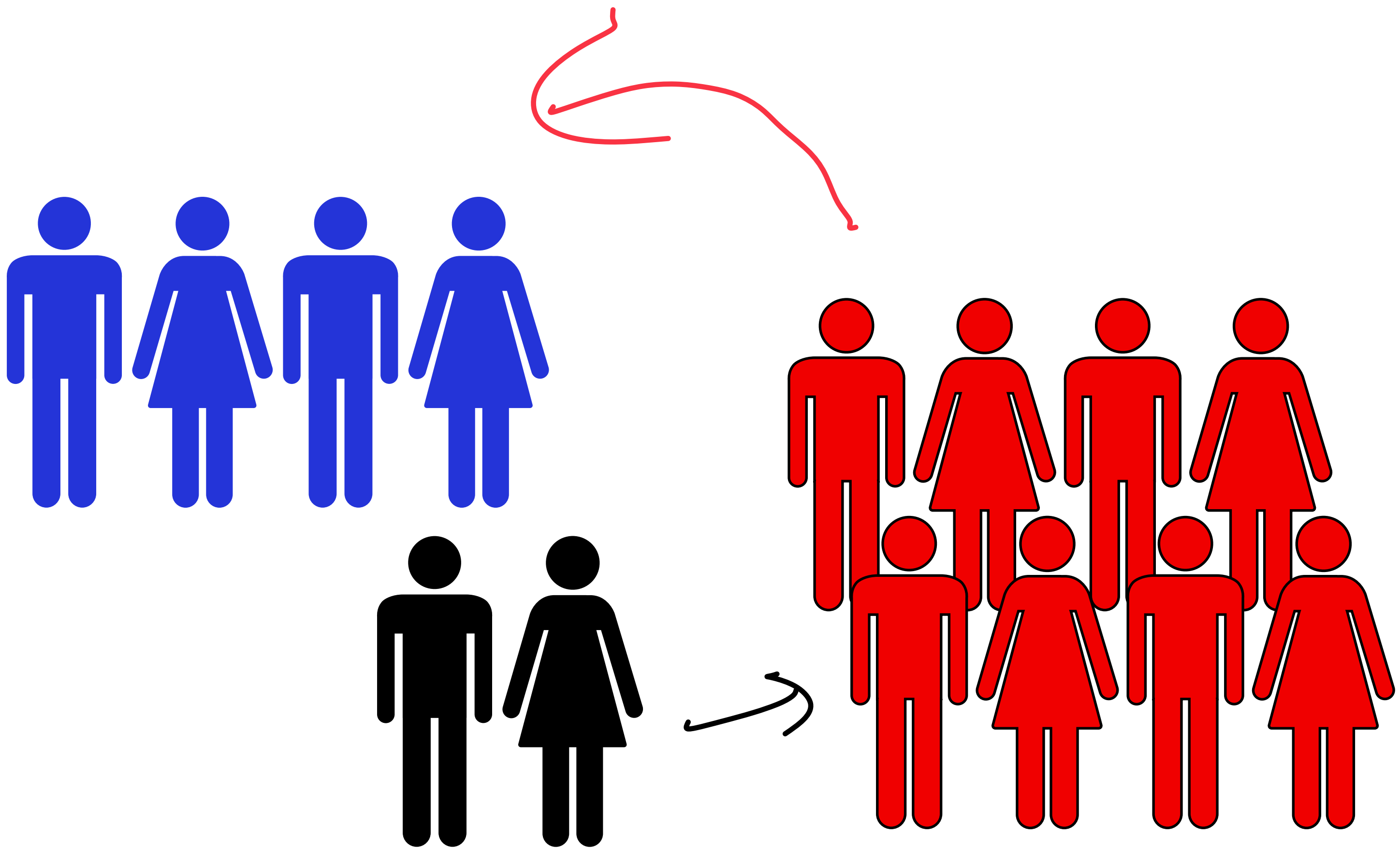
Introduce and implement disjoint sets

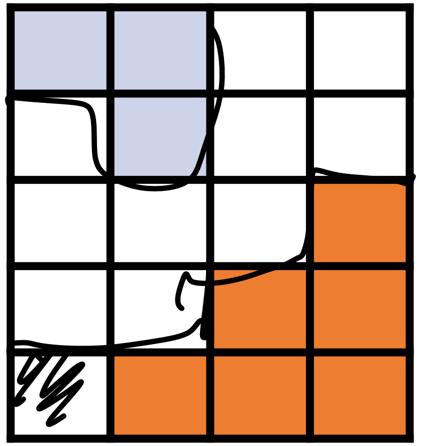
Discuss efficiency of disjoint sets

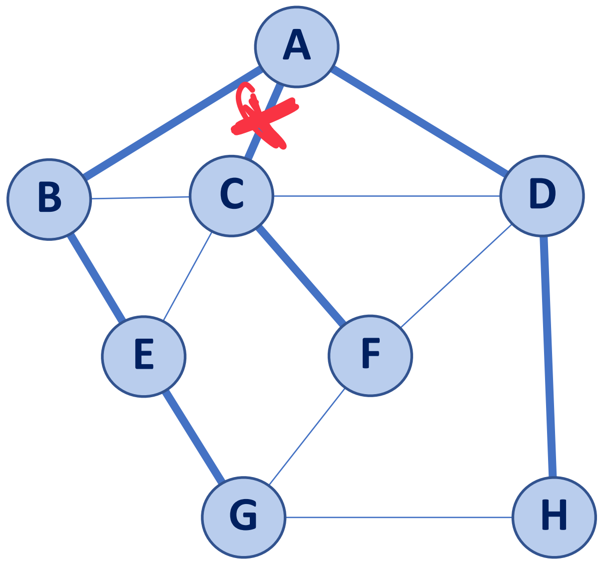
Identify improvements to implementation (and efficiency)

Storing and manipulating dynamic groups

We need a data structure which can efficiently look up (and change) group dynamics

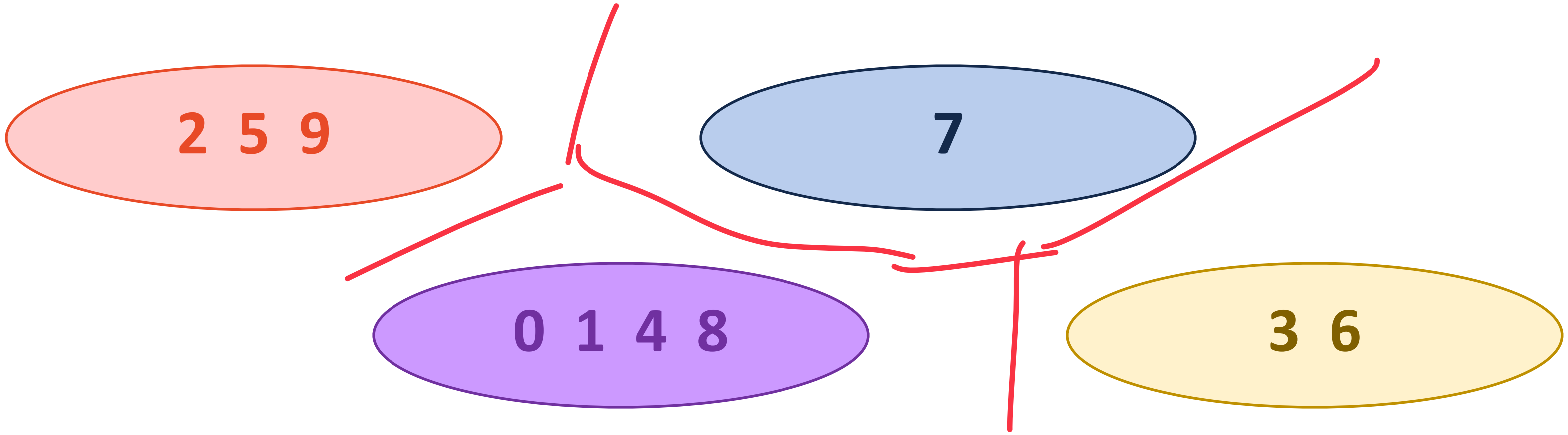






Disjoint Set ADT

A data structure designed to store relationships between items



Operations:

`find(k)` — returns "set representation" for item x

`union(s1, s2)` — Merge s1 and s2 into one set

`Constructor` — Make a new set

Disjoint Sets ‘Set Representation’

All items in a set have the same ‘Set Representation’

2 5 9

7

0 1 4 8

3 6

Operation:

`find(4) == find(8)`

↳ x ↳ x

`find(4) != find(3)`

↳ x ↳ y

How to store? or label

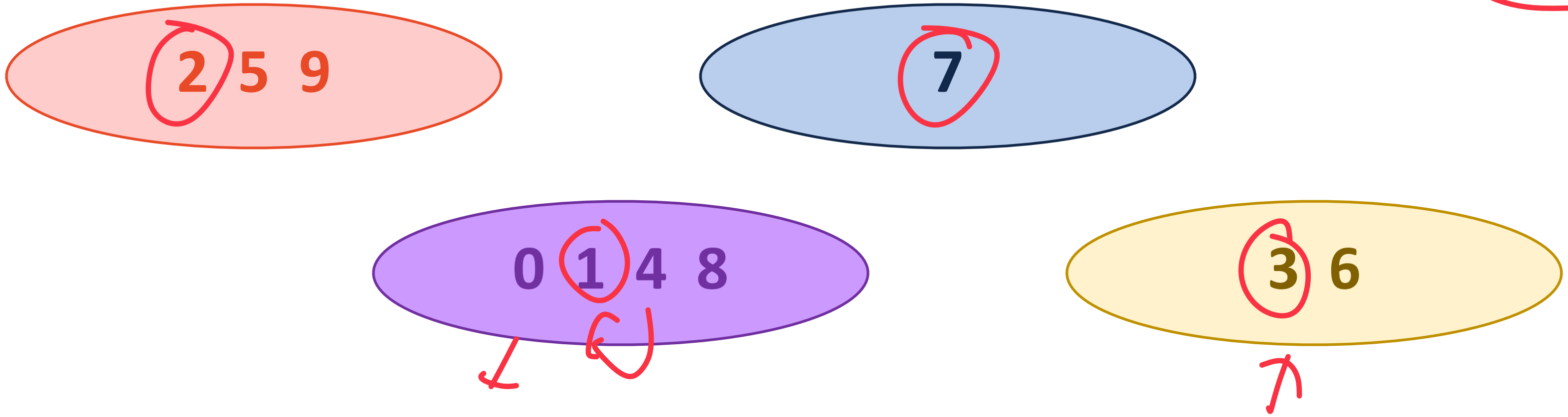
↳ Store address or index of a set

↳ Key, value pairs

↳ item, value is set

Disjoint Sets 'Set Representation'

Each set is represented by a **canonical element** (internally defined)



Operation:

`find(4) == find(8)`

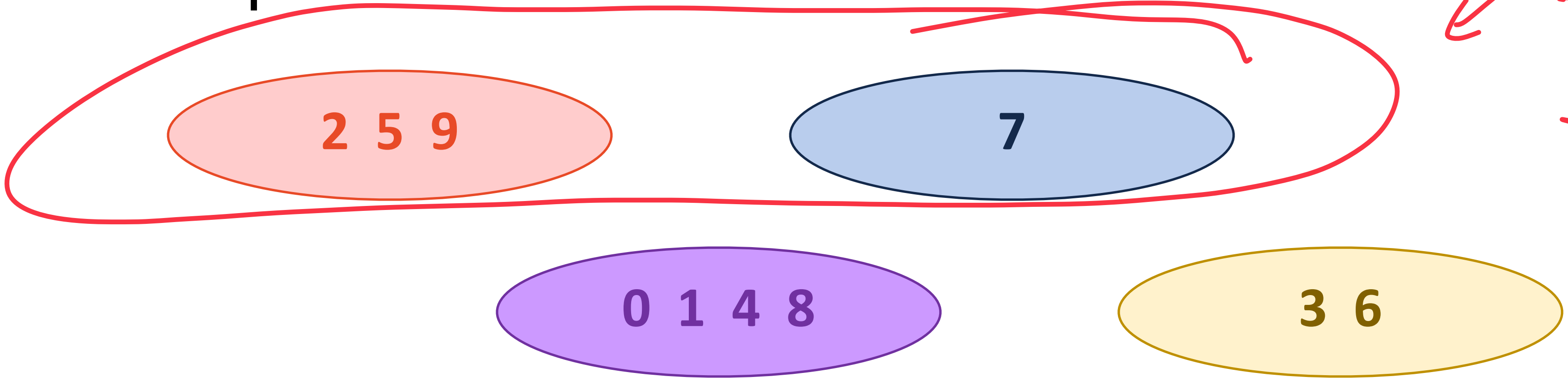
1 1

`find(4) != find(3)`

1 3

Disjoint Sets

The union operation combines two sets into one set.



1 canonical element from either set

Operation:

```
if find(2) != find(7){
   union( 2, 7 );
}
```

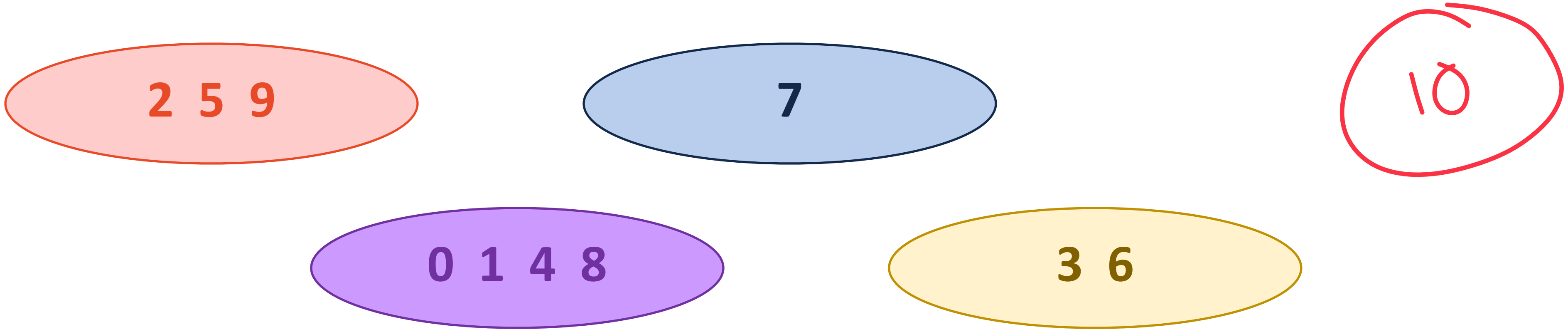
key 1 key 2

find(2) to get set

find(7) to get set

Disjoint Sets

We add new items to our 'universe' by making new sets.



Operation:

```
makeSet(10);
```

Disjoint Sets

ADT:

`makeSet(vector<T> items)`

`Find(T key)` – my group

`Union(T k1, T k2)` – 2 sets together

Key Ideas:

Every item exists in exactly one set

representative

Every item in each set has same representation – canonical element

Every set has a different representation

Disjoint Sets

How might we implement a disjoint set?

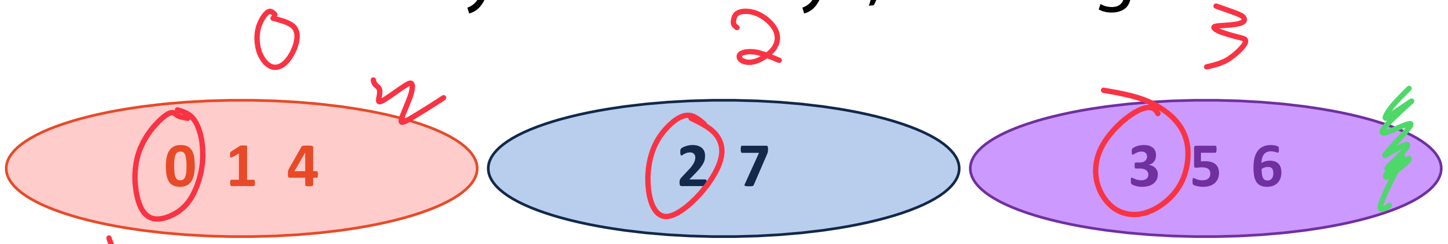
↳ Map / Dictionary

Implementation #1

unsigned integers

max size is max element

Allocate array for all keys, storing canonical key as index



1) We pick canonical element

0	1	2	3	4	5	6	7
0	0	2	3	0	3	3	2

Says what set item i is in

Find(k): Look up index k O(1)

tradeoff

fast

slow

Union(k_1, k_2): Walk across array & update every 2 → 0

0 7

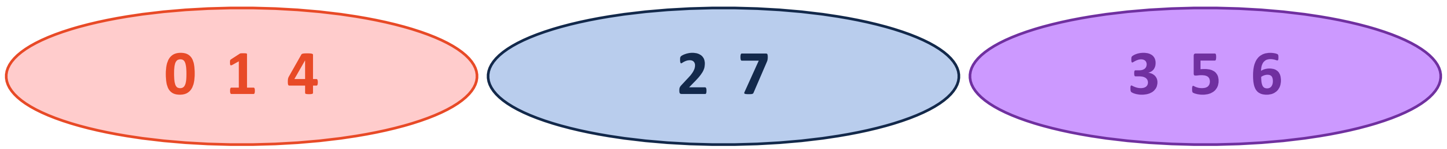
or

0 → 2

O(n)

Implementation #1

Allocate array for all keys, storing canonical key as index



0	1	2	3	4	5	6	7
4	4	7	5	4	5	5	7

Find(k): Look up value in array $O(1)$

tradeoff

Union(k_1, k_2): Update **every item** in one set with new representation

$O(n)$

Implementation #2

Same idea but store canonical elements as **-1**

0 1 4 | 2 7 | 3 5 6

Find(1) Find(7)

0	1	2	3	4	5	6	7
-1	0	~~-1~~ 0	-1	0	3	3	2

Find(k): Repeat lookups until -1

↳ O(n)

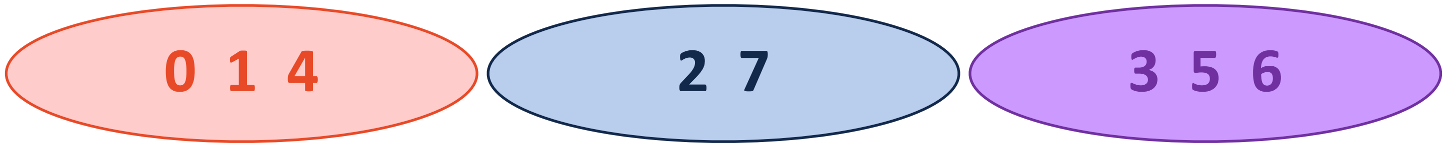
Union(k_1, k_2): Update one canonical element to point to other

2 7

O(1) :)

Implementation #2

Same idea but store canonical elements as **-1**



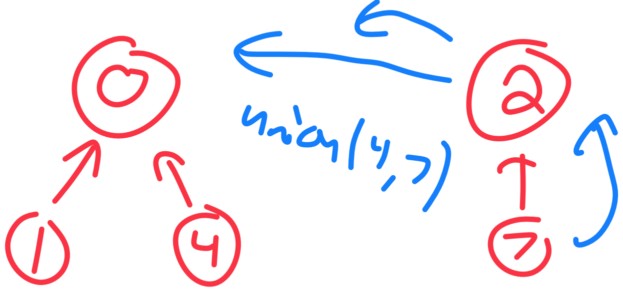
0	1	2	3	4	5	6	7
-1	0	-1	-1	0	3	3	2

Find(k): Repeatedly look up values until **-1**

Union(k_1, k_2): Update one canonical item to point at the other

Implementation #2

Same idea but store canonical elements as **-1**



0 1 4

2 7

3 5 6

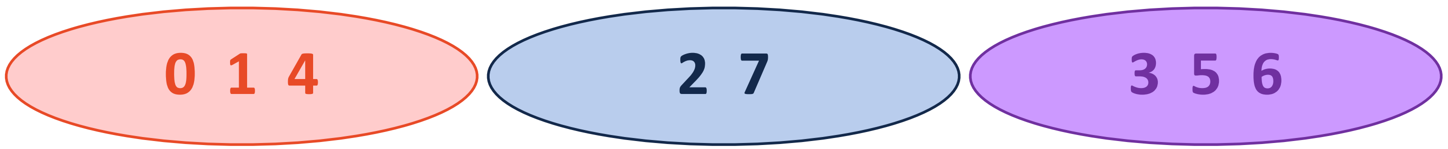
0	1	2	3	4	5	6	7
-1	0	~~-1~~ 0	-1	0	3	3	2

Union(4, 7) - setting 2 to have value 0

Find(7) - walk up tree → set 0

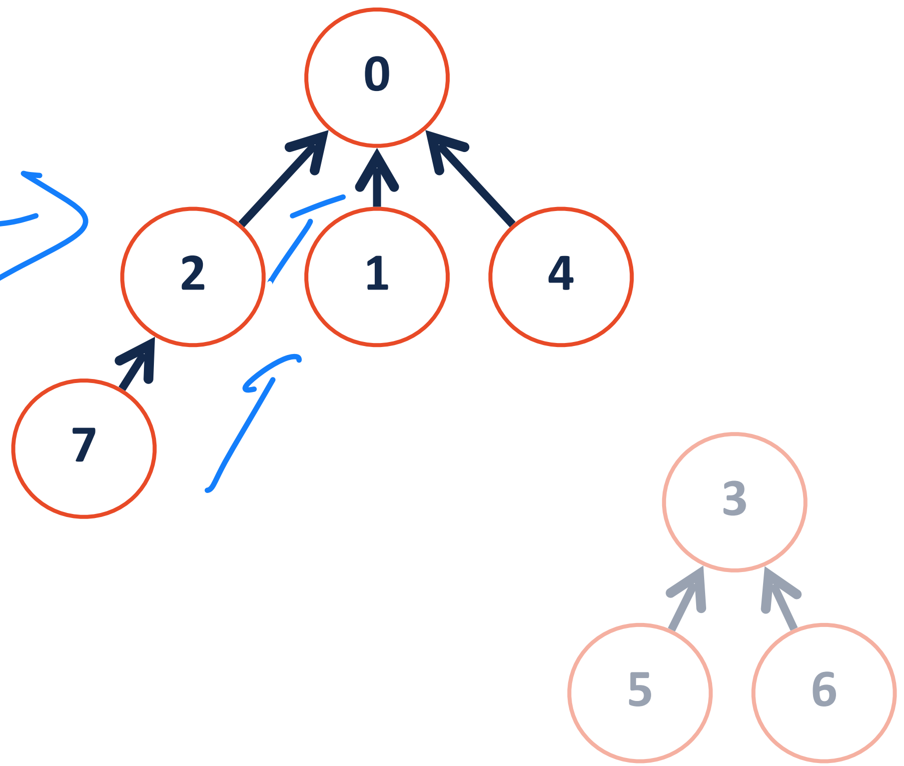
Implementation #2

Same idea but store canonical elements as **-1**



0	1	2	3	4	5	6	7
-1	0	**0**	-1	0	3	3	2

union

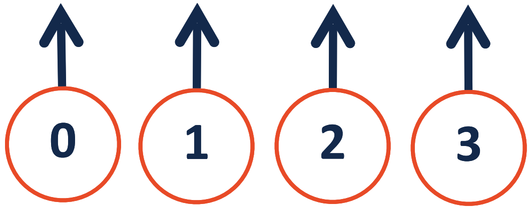


Union(4, 7): A[7]=2

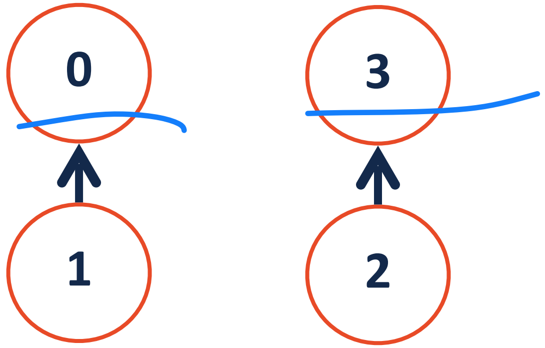
↳ Find(7) → 2 ↳

Find(7): A[7] -> A[2] -> A[0]

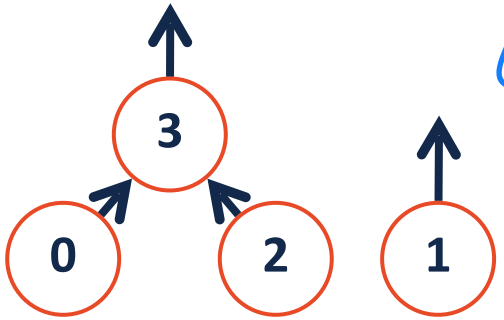
UpTrees



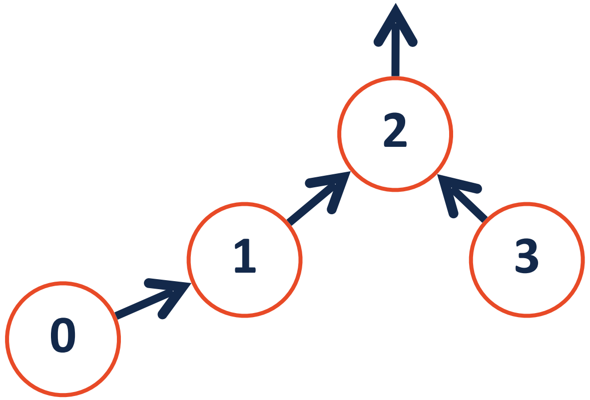
0	1	2	3
-1	-1	-1	-1



0	1	2	3
-1	0	3	-1

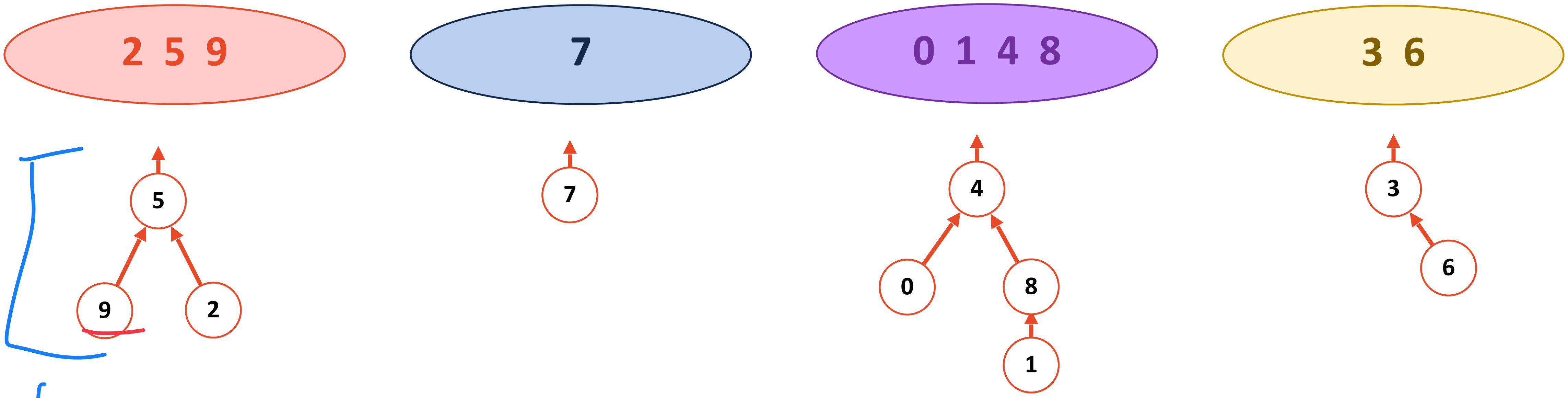


0	1	2	3
3	-1	3	-1



0	1	2	3
1	2	-1	2

Disjoint Sets

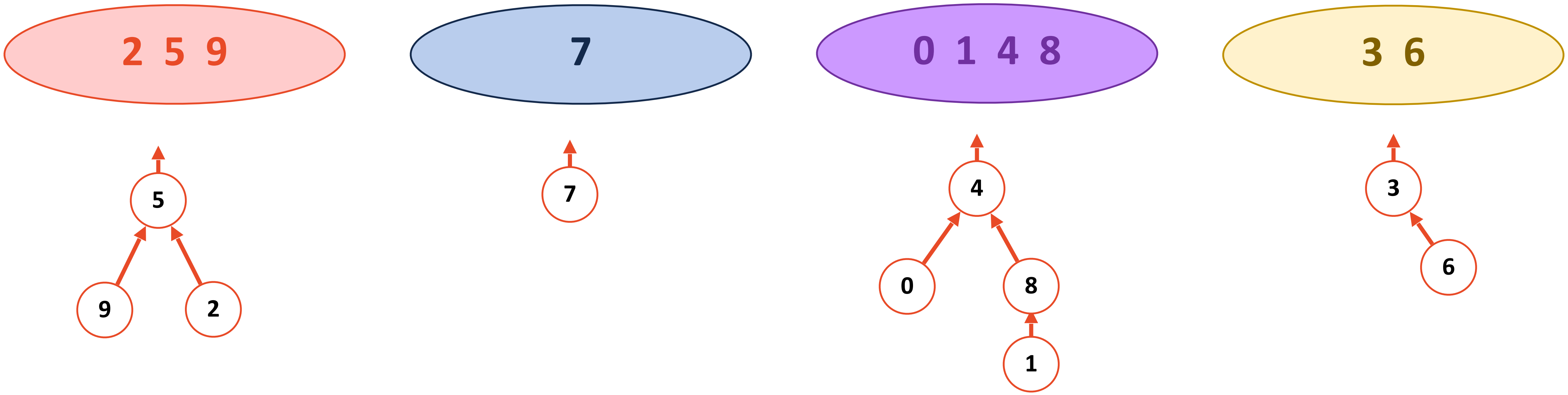


uptrees

rep as array

0	1	2	3	4	5	6	7	8	9
		5			-1				5

Disjoint Sets

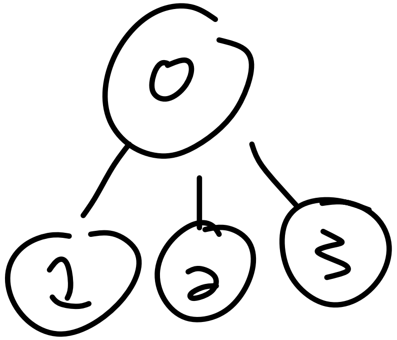


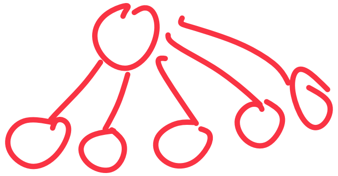
0	1	2	3	4	5	6	7	8	9
4	8	5	-1	-1	-1	3	-1	4	5

UpTrees Best and Worst Case

What does a best case UpTree look like?

$O(1)$





0	1	2	3
-1	0	0	0

Alt correct answer: every item own set (-1, -1, ...)

What does a worst case UpTree look like?

$0 \to 1 \to 2 \to 3$

$O(n)$

0	1	2	3
1	2	3	-1

Disjoint Sets Representation

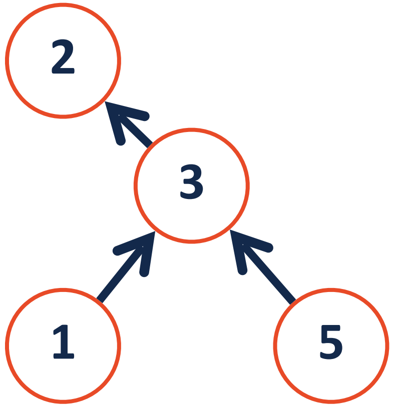
Implemented as an array where the value of key is index in array

The values inside the array stores our sets as an **UpTree**

- The value **-1** is our representative element (the root)
- All other set members store the index to a parent of the UpTree

Big O for Find: $O(h)$ $\quad 2 \leq h \leq n$

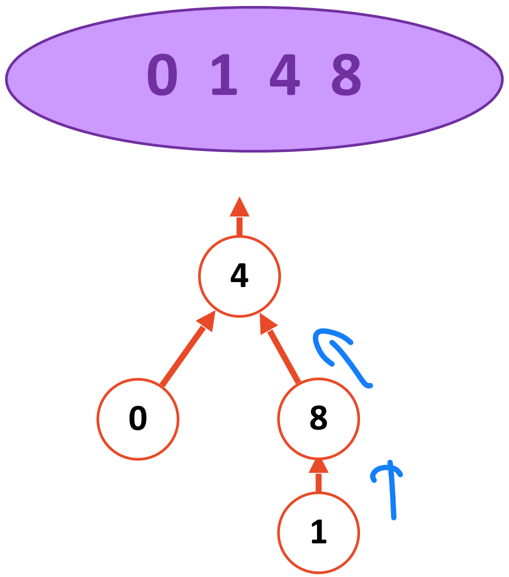
Big O for Union: $O(1)^*$ * ignoring find to get canonical



Disjoint Sets Find

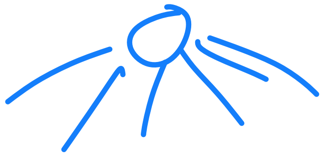
Find(1)

```
int DisjointSets::find(int i) {
  if ( s[i] < 0 ) { return i; }
  else { return find( s[i] ); }
}
```



Running time? $O(h) \approx O(n)$

What is ideal UpTree?



$O(1)$

$i=4$

0	1	2	3	4	5	6	7	8	9
4	8			-1				4	

Disjoint Sets Union

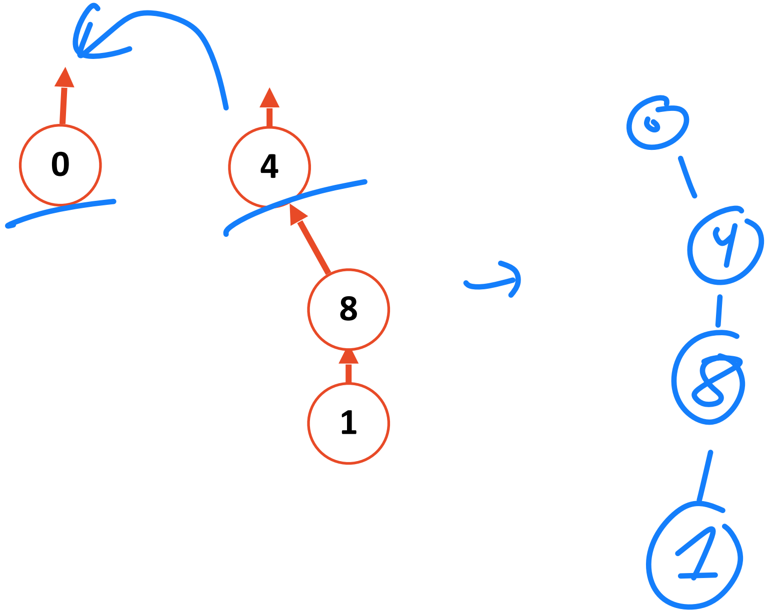
O(1)

Union(0, 4)

```
int DisjointSets::union(int r1, int r2) {
  // Naive Implementation

  s[r2] = r1;
}
```

r2 & r1 must be canonical



tangent

Implied we did find ahead of time (if necessary)

0	1	2	3	4	5	6	7	8	9
-1	8			~~-1~~ 0				4	

Disjoint Sets Union

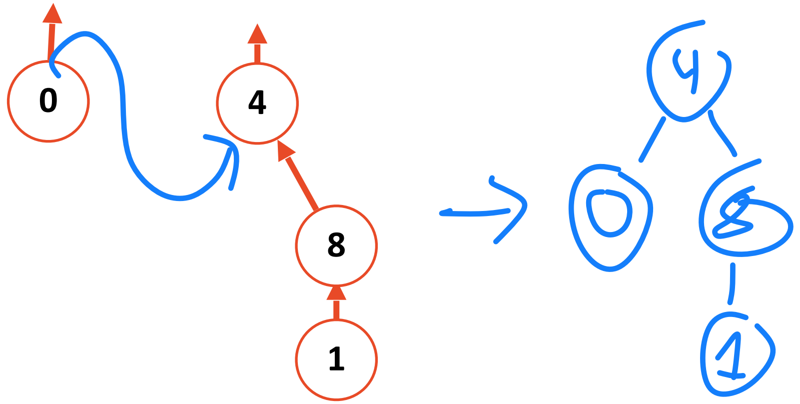
Union(4, 0)

```
int DisjointSets::union(int r1, int r2) {
  // Naive Implementation

  s[r2] = r1;
}
```

More balanced

Less height (changes)

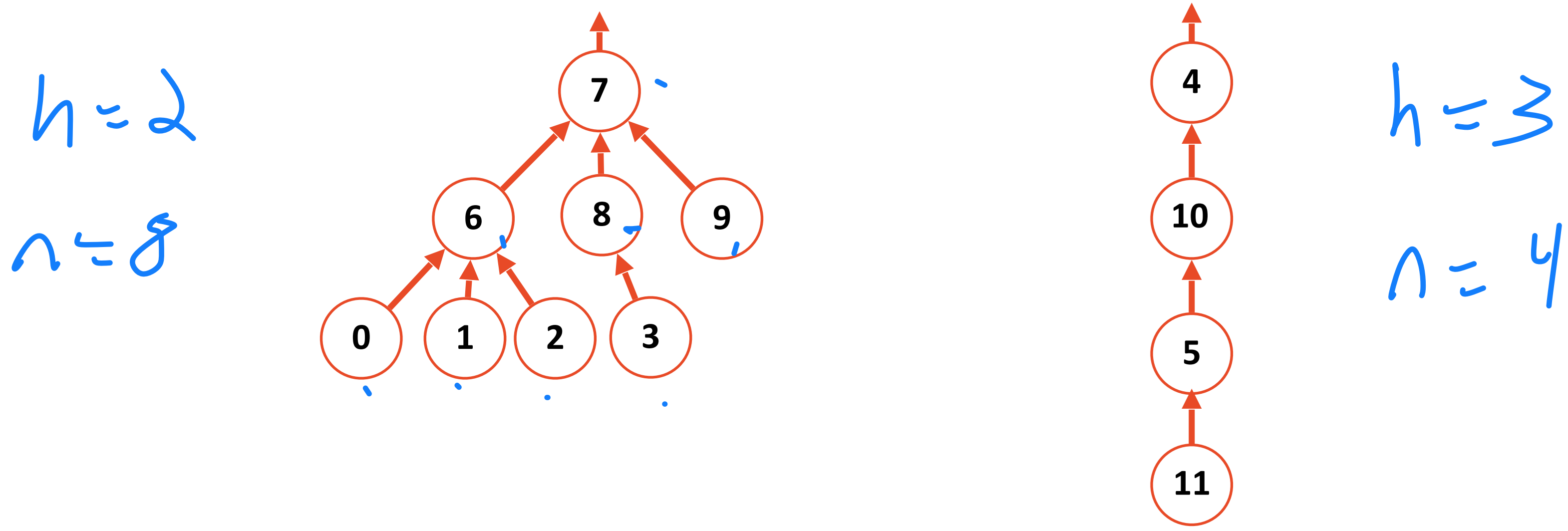


0	1	2	3	4	5	6	7	8	9
~~-1~~	8			-1				4	

4

Disjoint Sets – Union

How do I want to merge these sets?



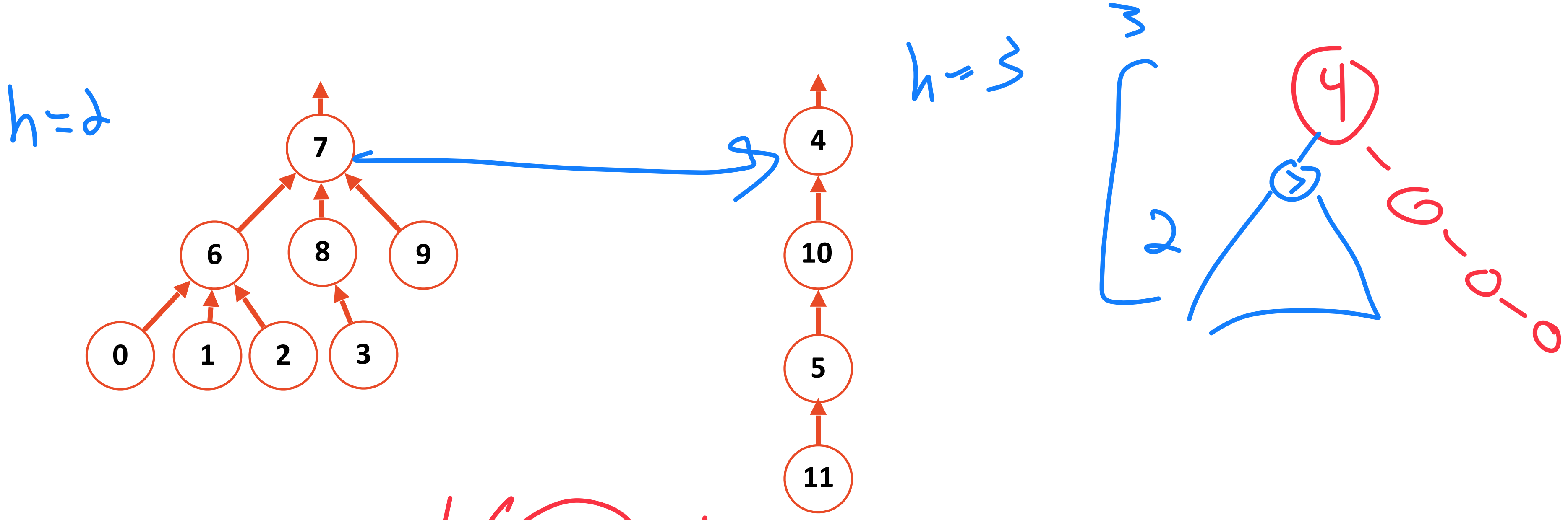
0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-1	10	7	-1	7	7	4	5

Union(4, 7)

Union(7, 4)

Disjoint Sets – Smart Union

Union(4, 7)



Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-4	10	7	~~-3~~ 4	7	7	4	5

***Idea**: Keep the height of the tree as small as possible.*

Base case is single node

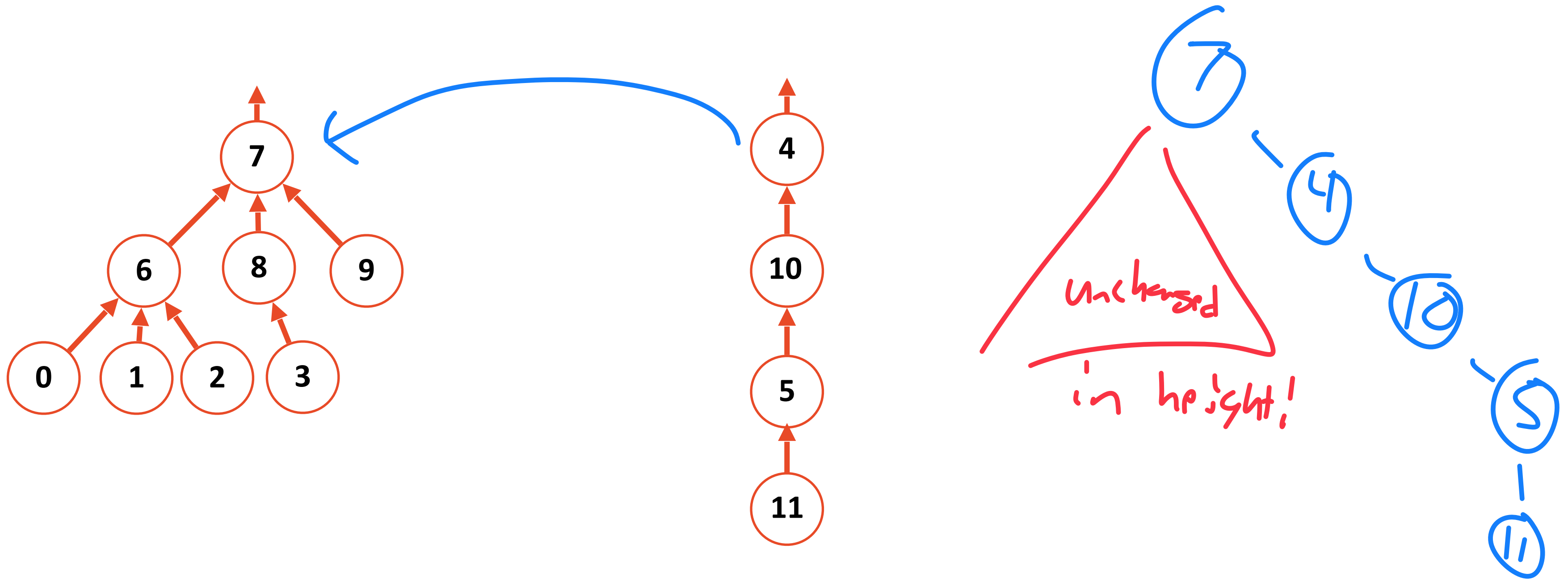
$h = 0 \to -1$

???

Clever Trick: If we union by height, store -1*(height+1) in canonical!

Disjoint Sets – Smart Union

Union(7, 4)



Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	~~-1~~ 7	10	7	-8	7	7	4	5

***Idea**: Minimize the number of nodes that increase in height*

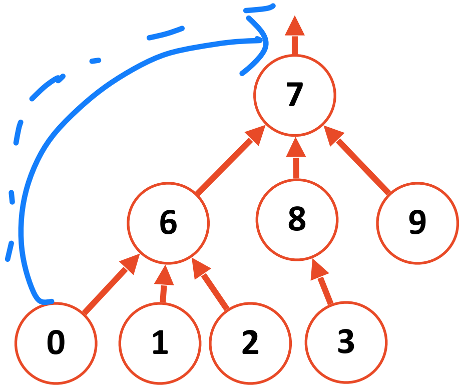
Clever Trick: If we union by size, store -1*(size) in canonical!

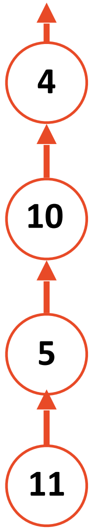
Disjoint Sets – Smart Union

Path compression (Friday)

why not set this?

Find(0)=7





−(height+1)

Union by height

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	**-4**	10	7	**4**	7	7	4	5

***Idea**: Keep the height of the tree as small as possible.*

Union by size

0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	**7**	10	7	**-12**	7	7	4	5

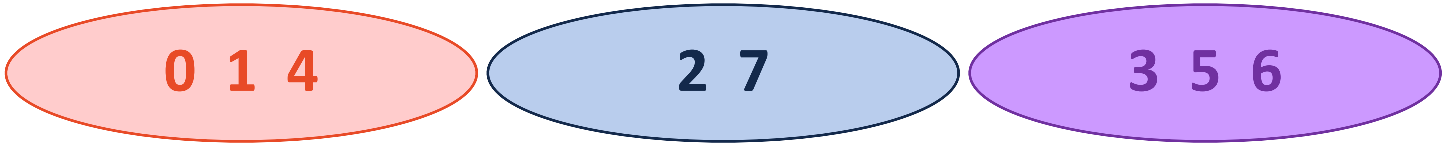
***Idea**: Minimize the number of nodes that increase in height*

−(size)

Both guarantee the height of the tree is: O(log n).

Disjoint Set Implementation

Store an UpTree as an array, canonical items store **height / size**



0	1	2	3	4	5	6	7
	0			0	3	3	2

Find(k): Repeatedly look up values until **negative value**

Union(k_1, k_2): Update ***smaller*** canonical item to point to larger

Update value of remaining canonical item