Data Structures

Disjoint Sets

CS 225

Brad Solomon

October 16, 2024

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Department of Computer Science

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

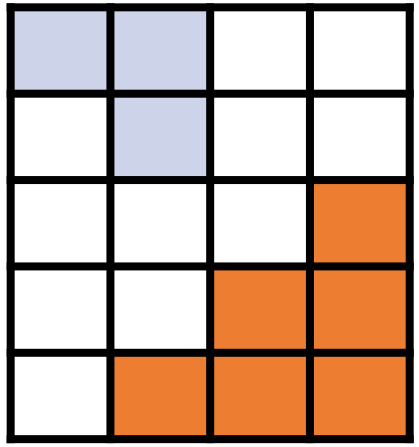
Introduce and implement disjoint sets

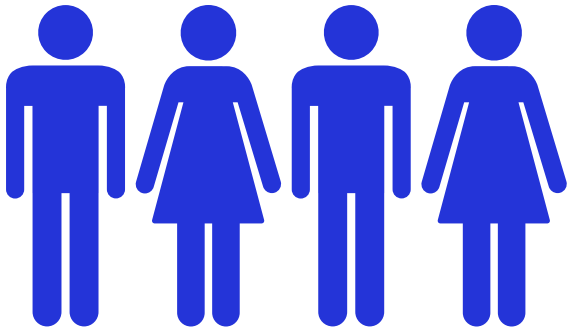
Discuss efficiency of disjoint sets

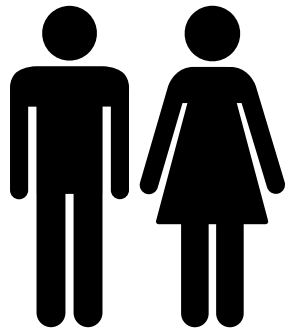
Identify improvements to implementation (and efficiency)

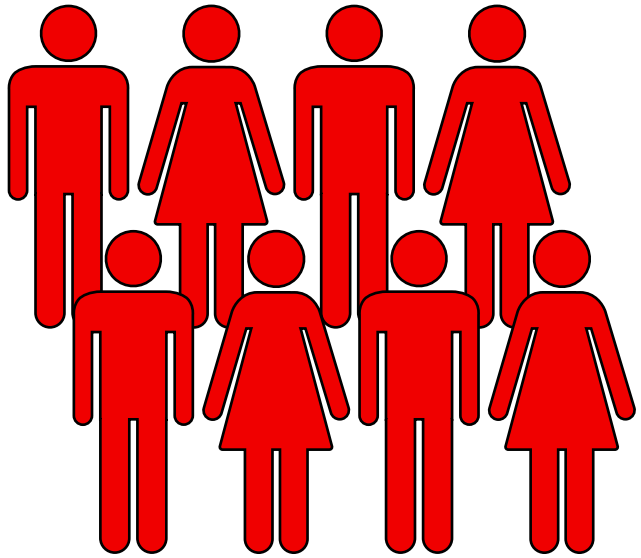
Storing and manipulating dynamic groups

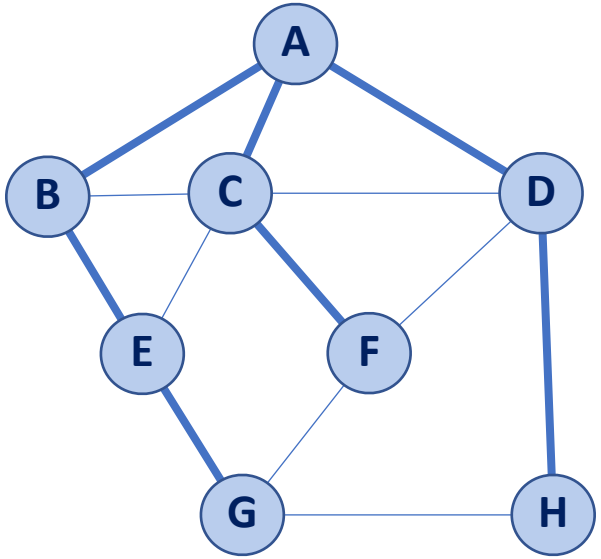
We need a data structure which can efficiently look up (and change) group dynamics





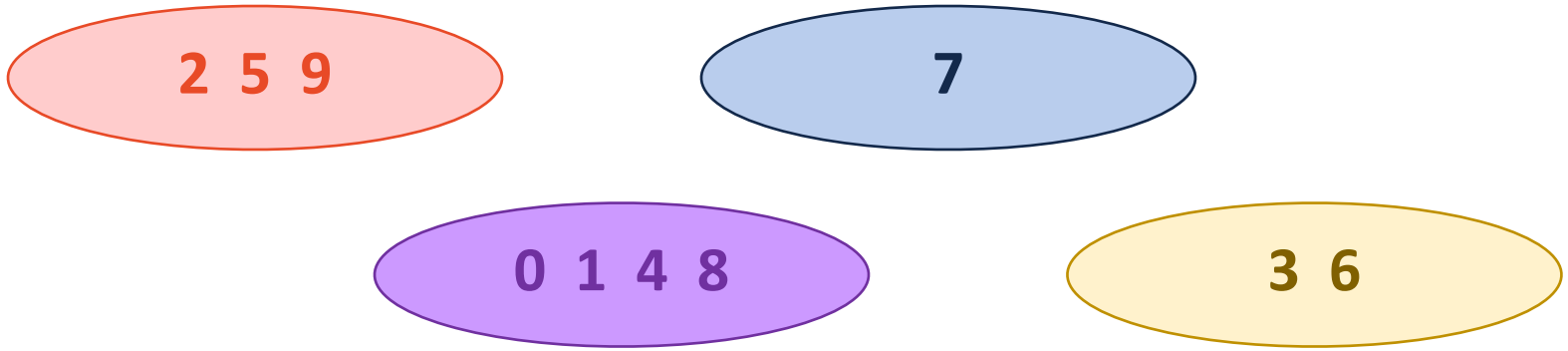






Disjoint Set ADT

A data structure designed to store relationships between items



Operations:

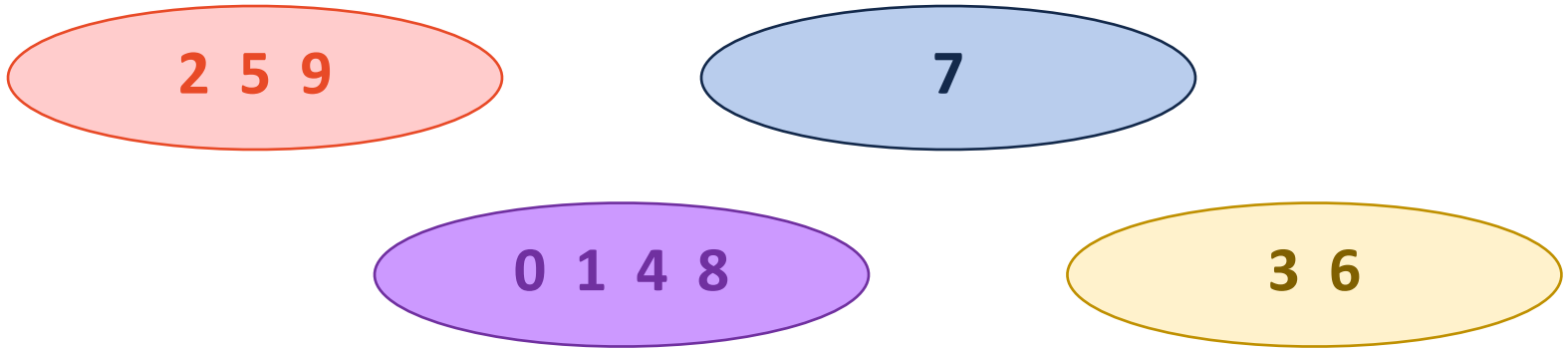
`find(k) – returns “set representation” for item x`

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

`Constructor – Make a new empty set`

Disjoint Sets ‘Set Representation’

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



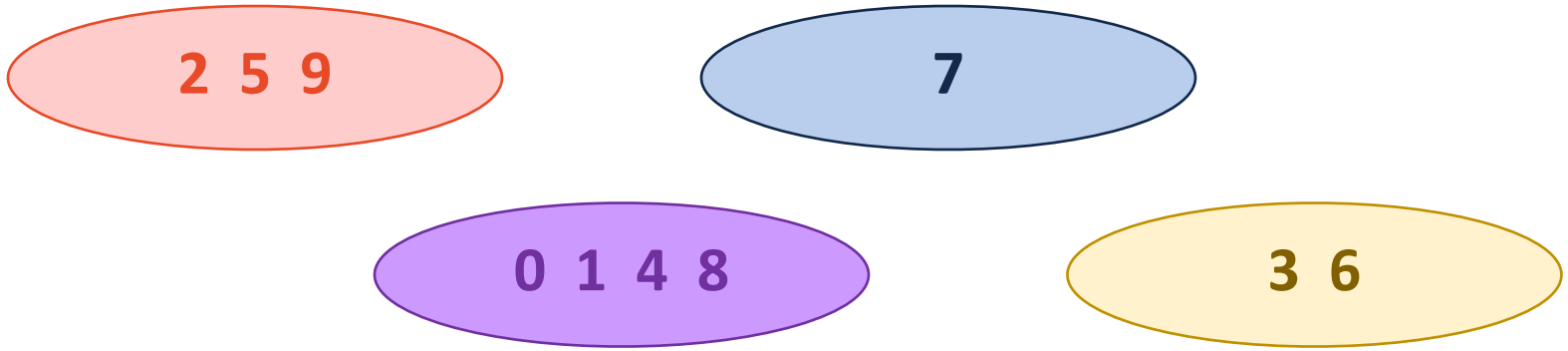
Operation:

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

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

Disjoint Sets ‘Set Representation’

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



Operation:

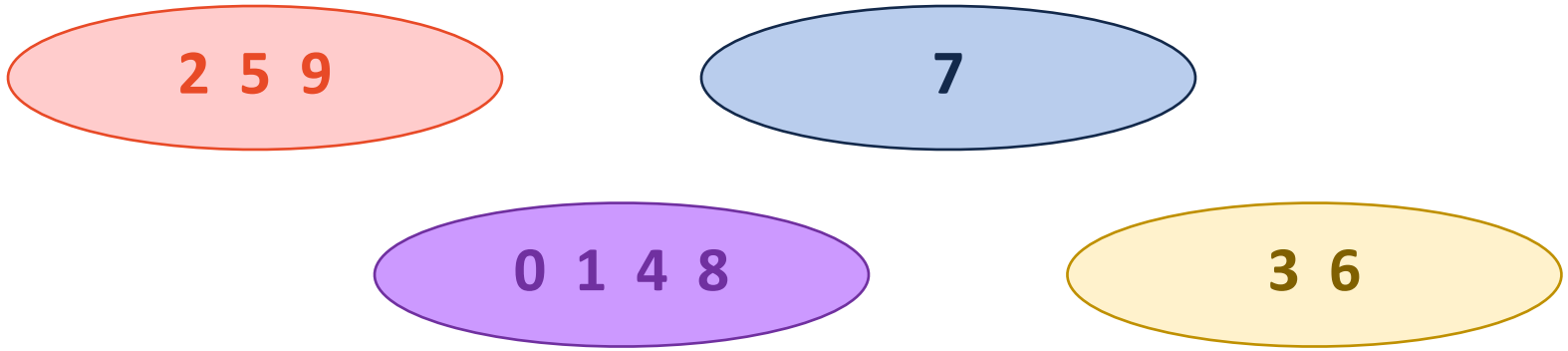
```
find(4) == find(8)
```

0	1	2	3	4	5	6	7	8	9

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

Disjoint Sets

The union operation combines two sets into one set.

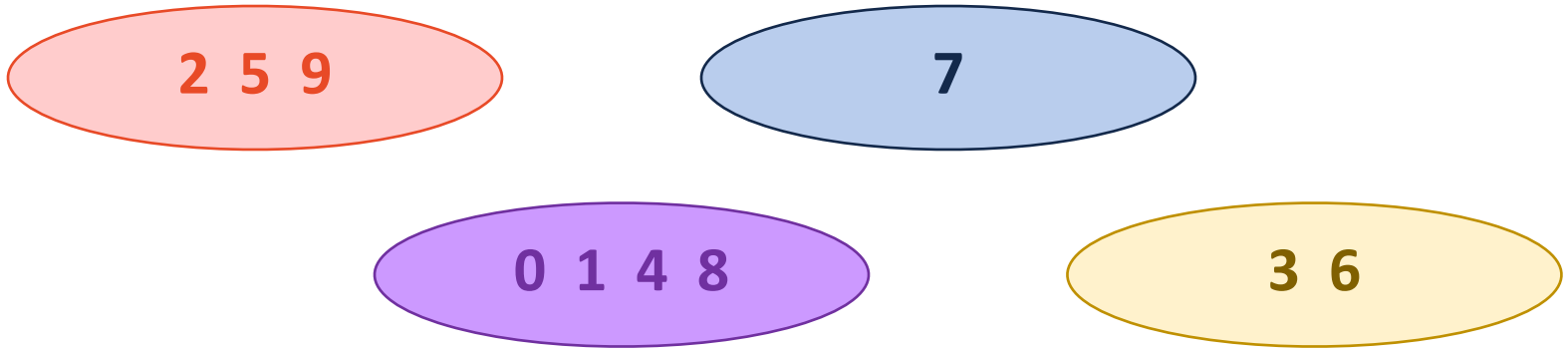


Operation:

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

Disjoint Sets

We add new items to our ‘universe’ by making new sets.



Operation:

```
makeSet(10);
```

Disjoint Sets ADT

Constructor

makeSet

Find

Union

Disjoint Sets

ADT:

```
makeSet(vector<T> items)

Find(T key)

Union(T k1, T k2)
```

Key Ideas:

Every item exists in exactly one set

Every item in each set has same representation

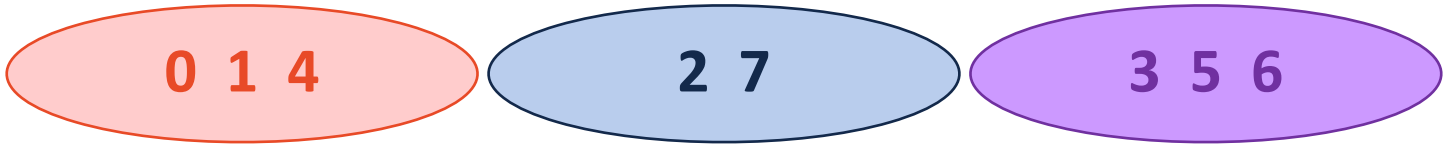
Every set has a different representation

Disjoint Sets

How might we implement a disjoint set?

Implementation #1

Allocate array for all keys, storing canonical key as index



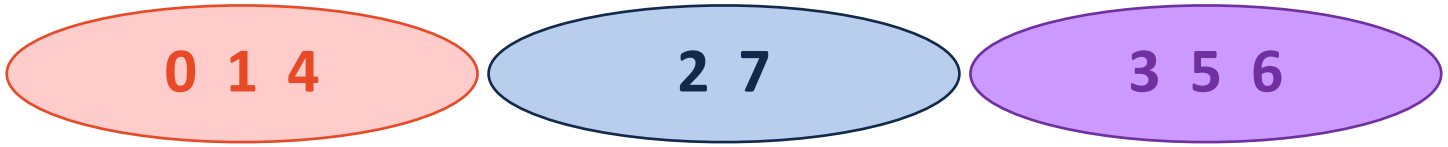
0	1	2	3	4	5	6	7

Find(k):

Union(k_1, k_2):

Implementation #1

Allocate array for all keys, storing canonical key as index



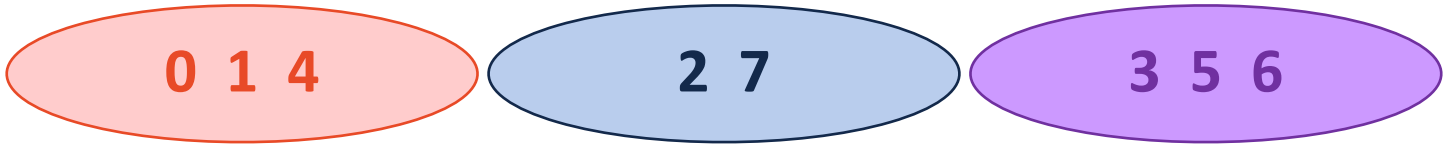
0	1	2	3	4	5	6	7
4	4	7	5	4	5	5	7

Find(k): Look up value in array

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

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

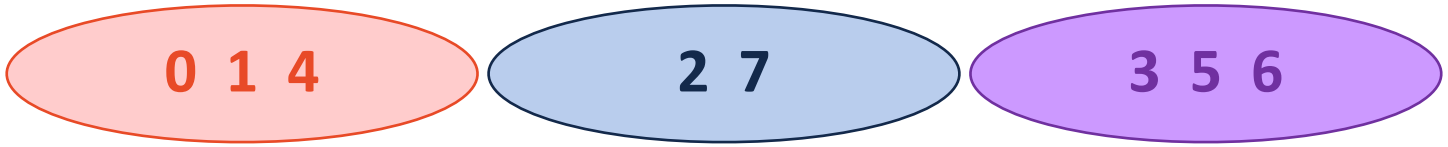
Find(k):

Union(k_1, k_2):

Implementation #2

Union(4, 7)

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



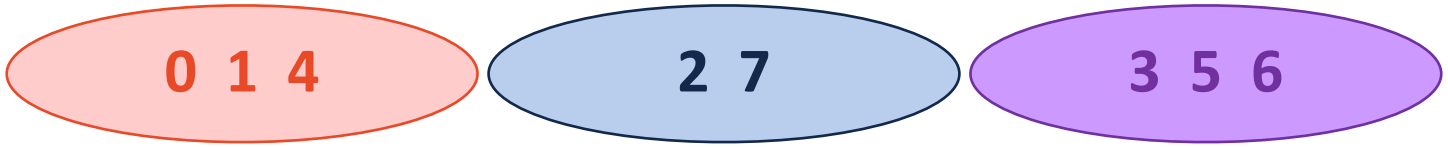
0	1	2	3	4	5	6	7
-1	0	-1	-1	0	3	3	2

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**



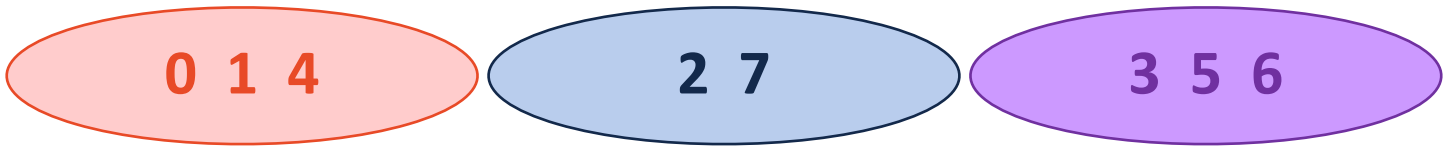
0	1	2	3	4	5	6	7
-1	0	-1	-1	0	3	3	2

```
Union(4, 7)
```

```
Find(7)
```

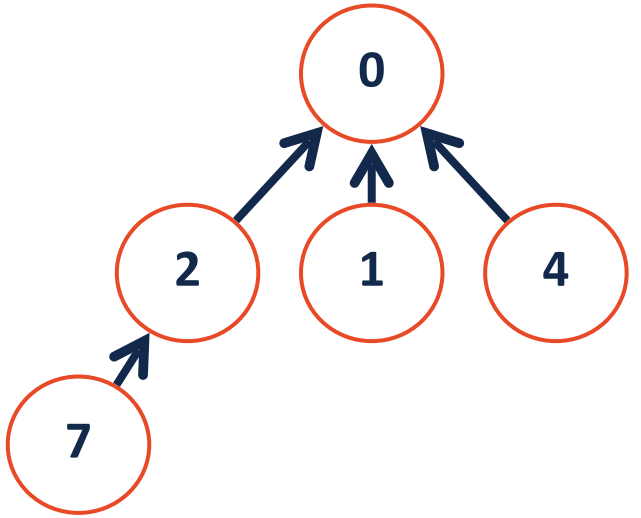
Implementation #2

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

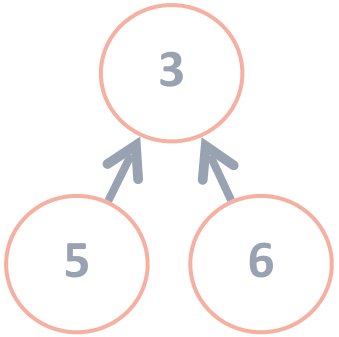


0	1	2	3	4	5	6	7
-1	0	**0**	-1	0	3	3	2

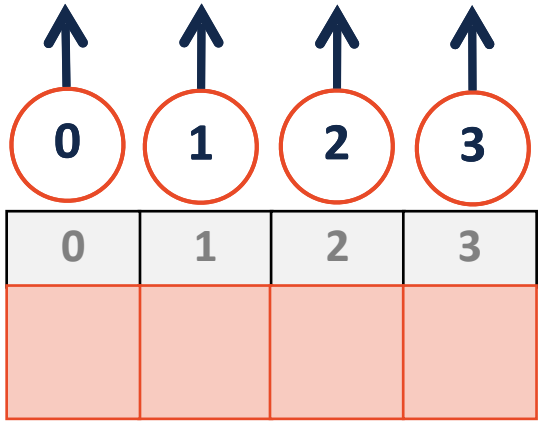
```
Union(4, 7): A[7]=2
```

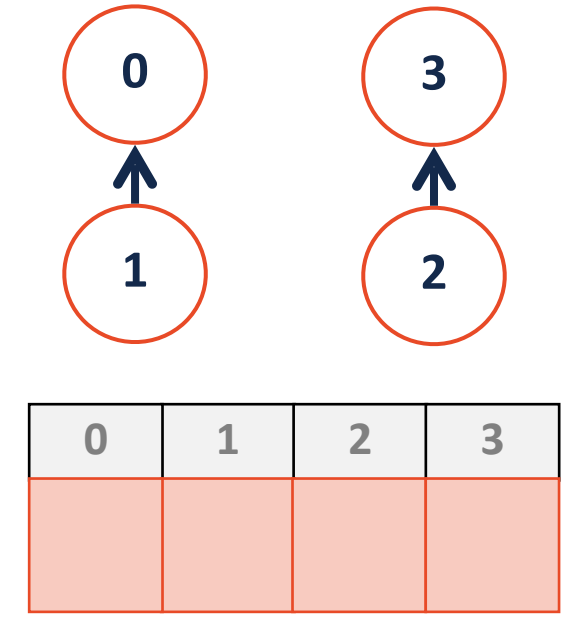


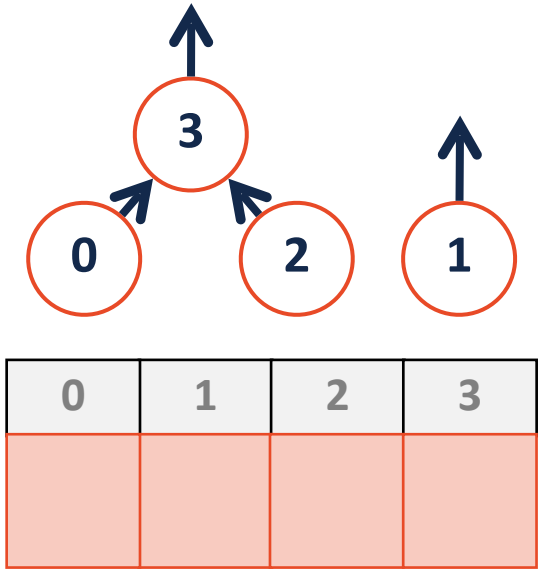

```
Find(7): A[7] -> A[2] -> A[0]
```

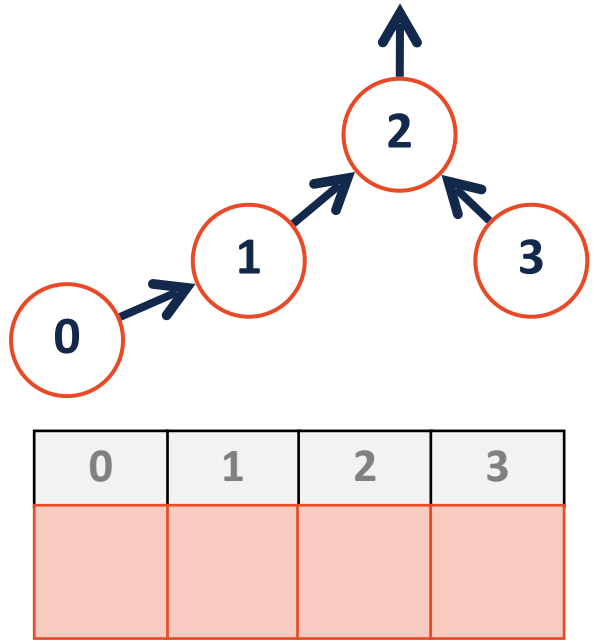


UpTrees

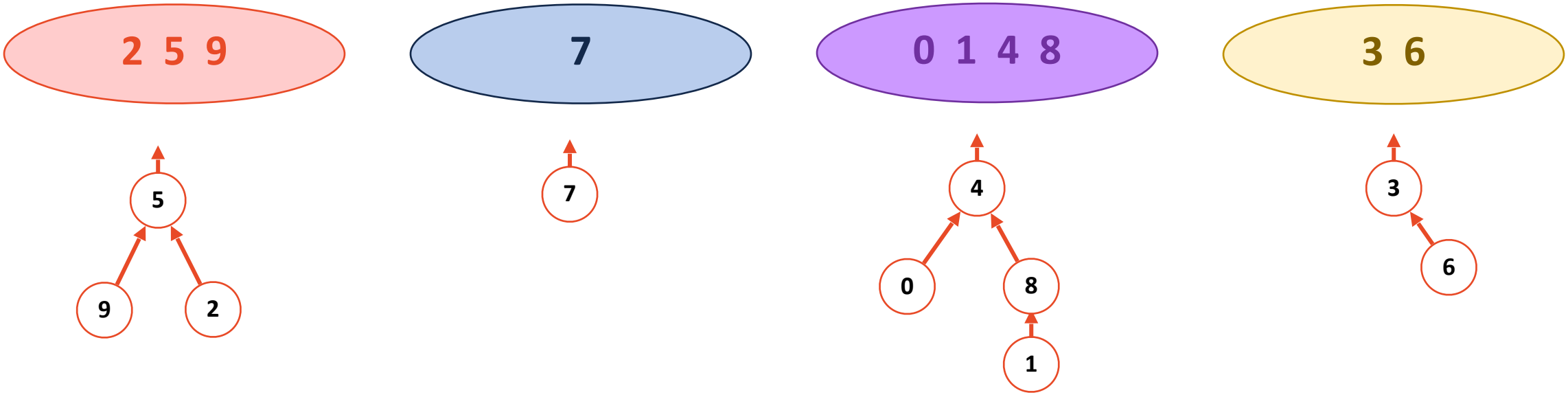






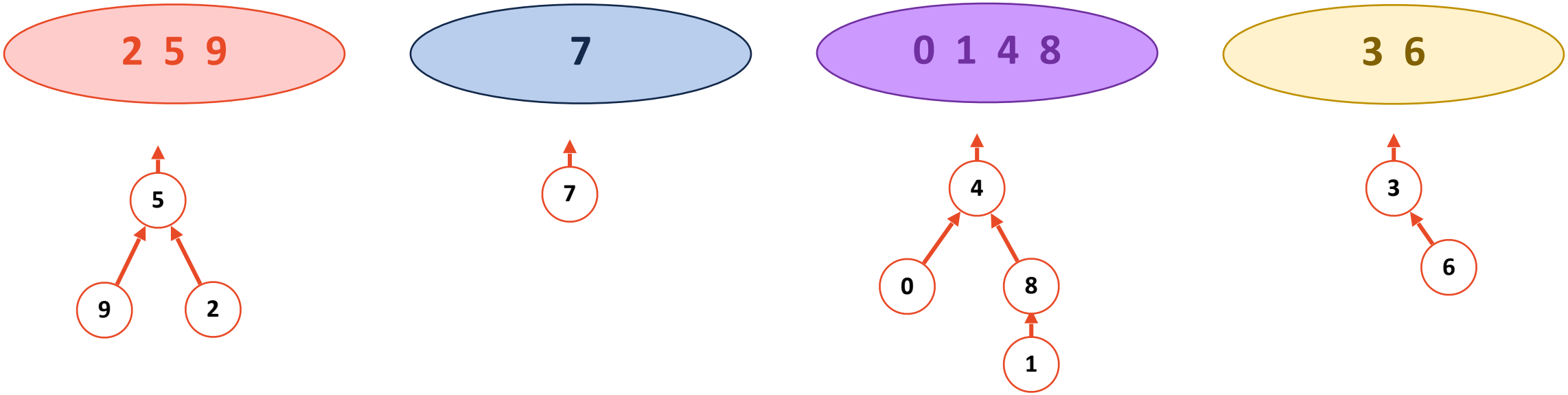


Disjoint Sets



0	1	2	3	4	5	6	7	8	9

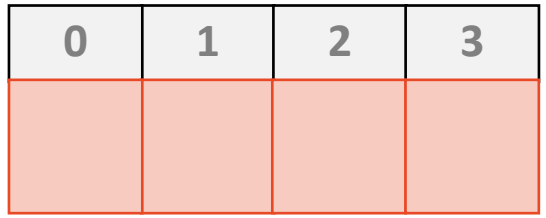
Disjoint Sets



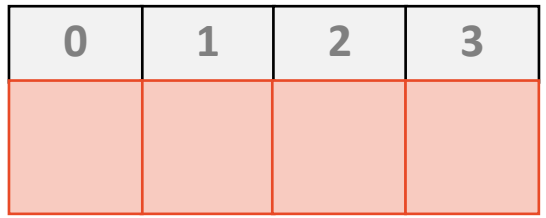
0	1	2	3	4	5	6	7	8	9
4	8	**5**	-1	-1	-1	3	-1	4	5

UpTrees Best and Worst Case

What does a best case UpTree look like?



What does a worst case UpTree look like?



Disjoint Sets Representation

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

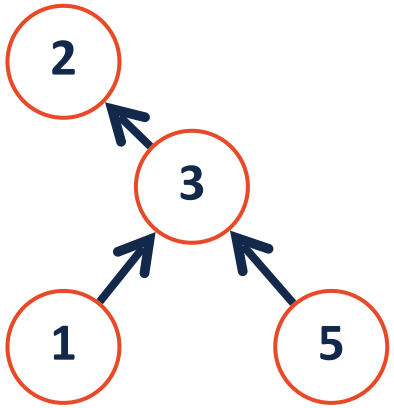
The values inside the array stores our sets as an **UpTree**

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:

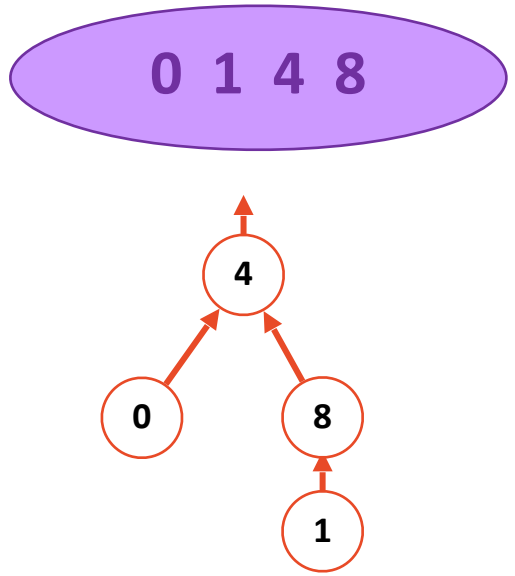
Big O for Union:



Disjoint Sets Find

Find(1)

```
1  int DisjointSets::find(int i) {
2    if ( s[i] < 0 ) { return i; }
3    else { return find( s[i] ); }
4  }
```



Running time?

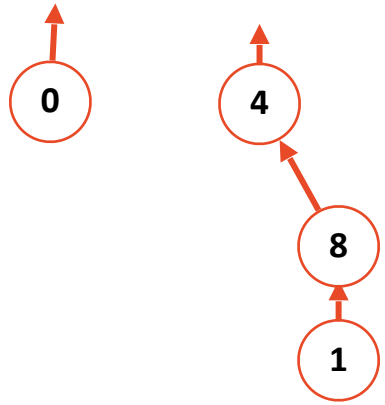
What is ideal UpTree?

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

Disjoint Sets Union

Union(0, 4)

```
1  int DisjointSets::union(int r1, int r2) {
2    // Naive Implementation
3
4    s[r2] = r1;
5  }
```

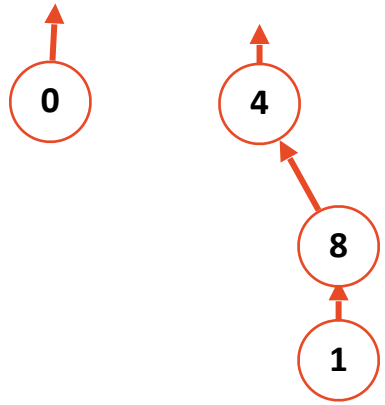


0	1	2	3	4	5	6	7	8	9
-1	8			-1				4	

Disjoint Sets Union

Union(4, 0)

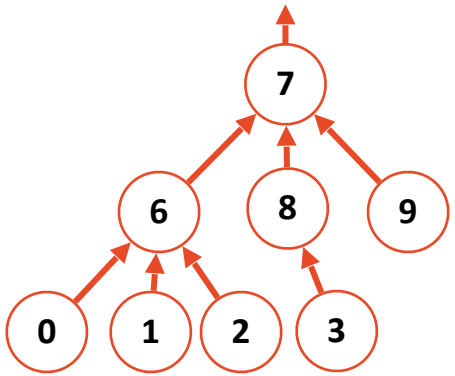
```
1  int DisjointSets::union(int r1, int r2) {
2    // Naive Implementation
3
4    s[r2] = r1;
5  }
```



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

Disjoint Sets – Union

How do I want to merge these sets?



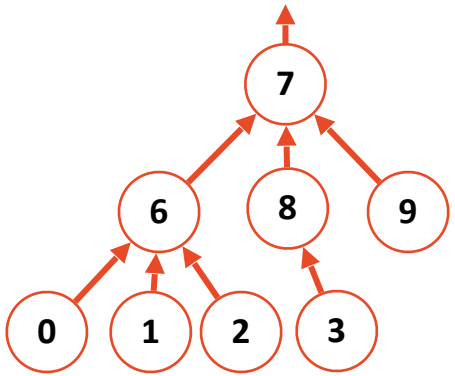
0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8	-1	10	7	-1	7	7	4	5

```
Union(4, 7)
```

```
Union(7, 4)
```

Disjoint Sets – Smart Union

Union(4, 7)



Union by height

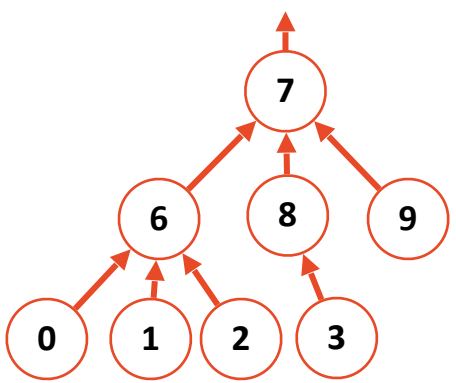
0	1	2	3	4	5	6	7	8	9	10	11
6	6	6	8		10	7		7	7	4	5

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

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

Disjoint Sets – Smart Union

`Union(7, 4)`



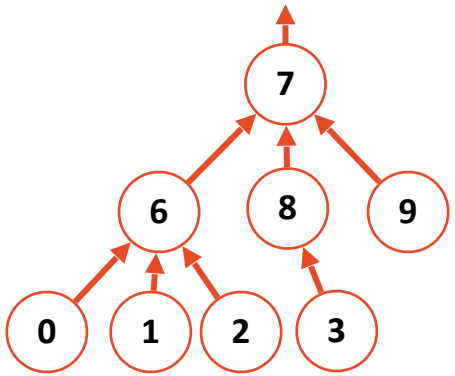
Union by size

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



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*

Both guarantee the height of the tree is: _____.