@happygirlzt

DISJOINT SETS

2024/04/13

Please subscribe to this channel ☺ @happygirlzt

Teaching Data Structure & Algorithms

Course Overview

- Introduction to Disjoint Sets
- Key Concepts and Operations
- Implementation Strategies
  - Code Demonstration
  - Time Complexity Analysis

Prerequisites

- Solid foundations in programming, including:
  - Object-oriented programming
  - Basic data structures, e.g. arrays, lists, sets and trees
  - Recursion

Learning Objective

Define disjoint sets

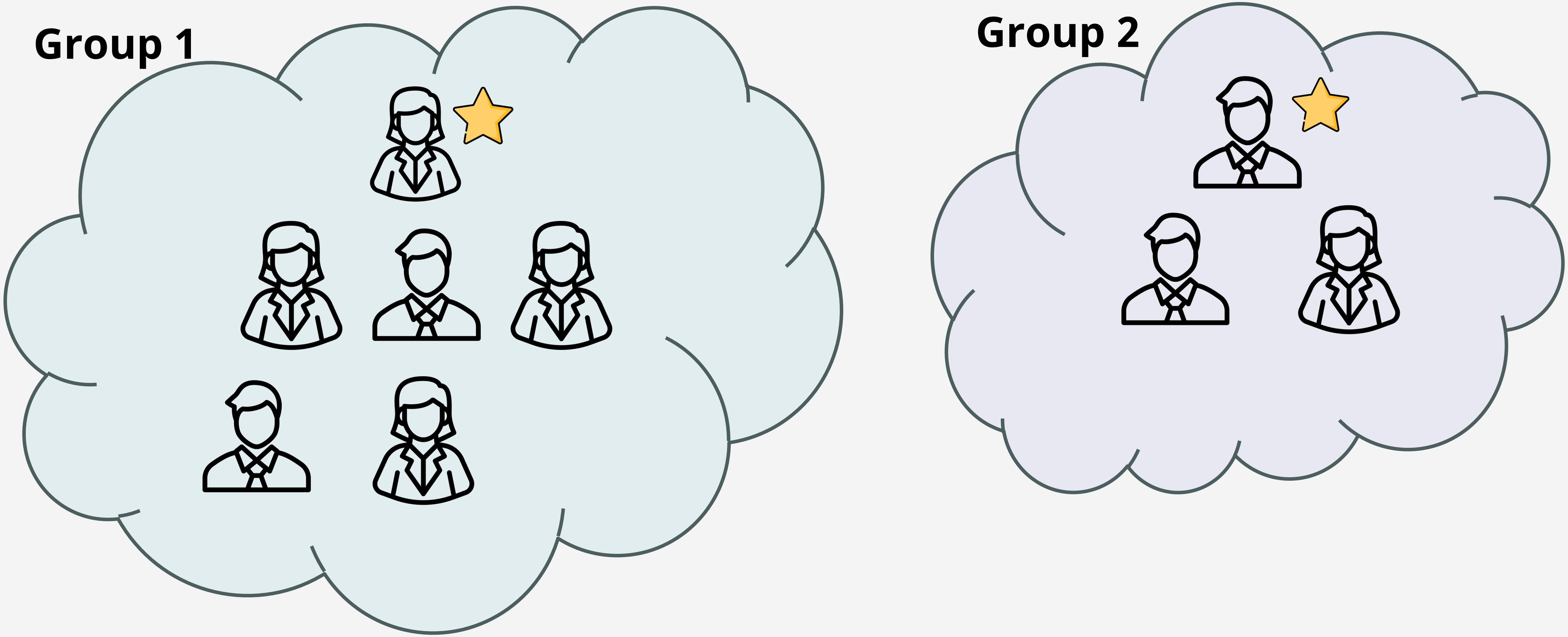
Explain the operations of disjoint sets

Implement disjoint sets with optimal strategies

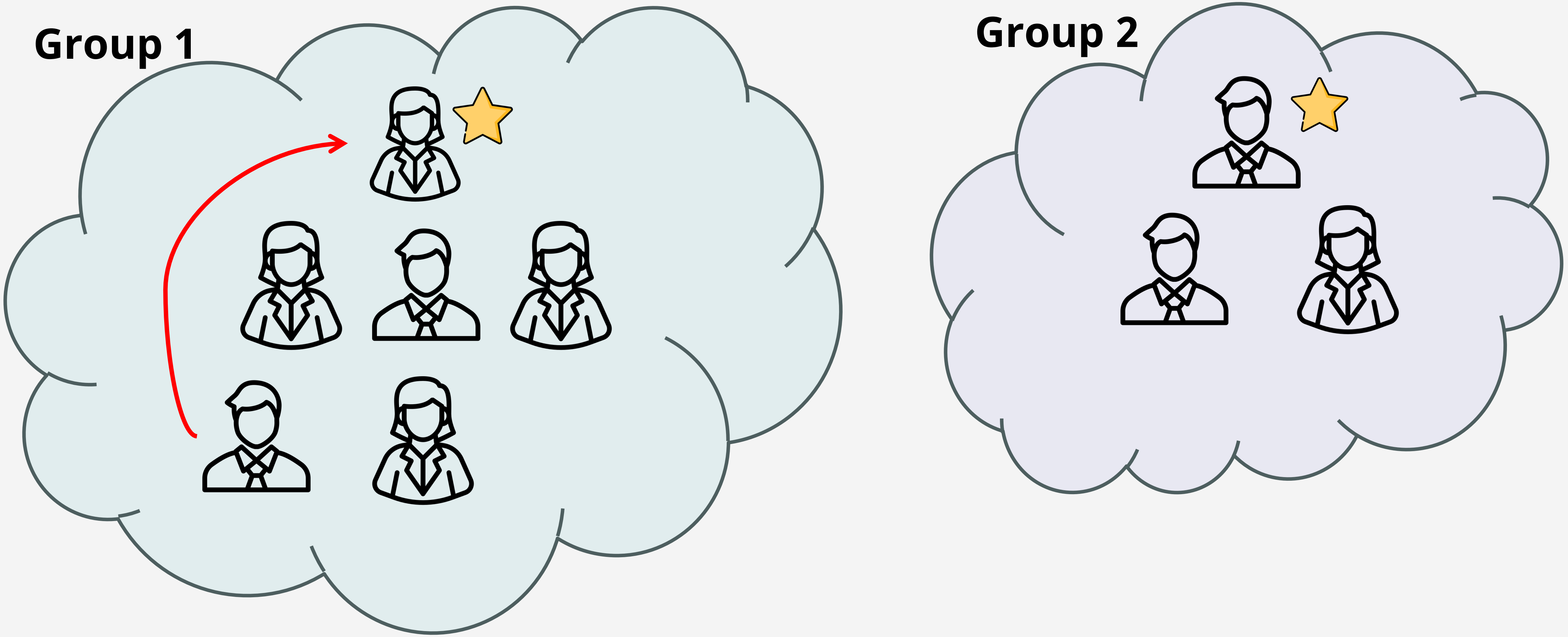
Analyze time complexity

Agenda

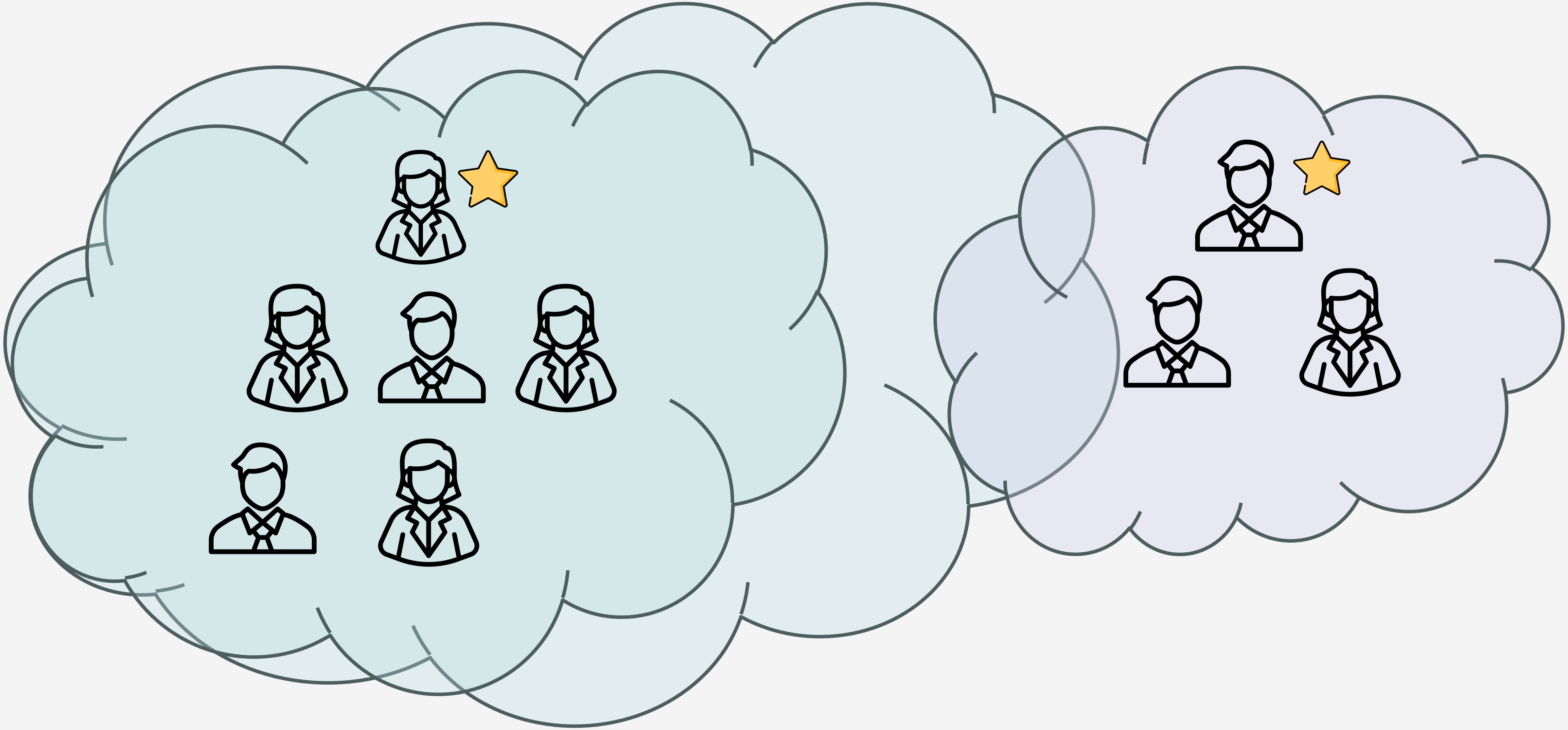
Two Groups of People



One Group has One Representative



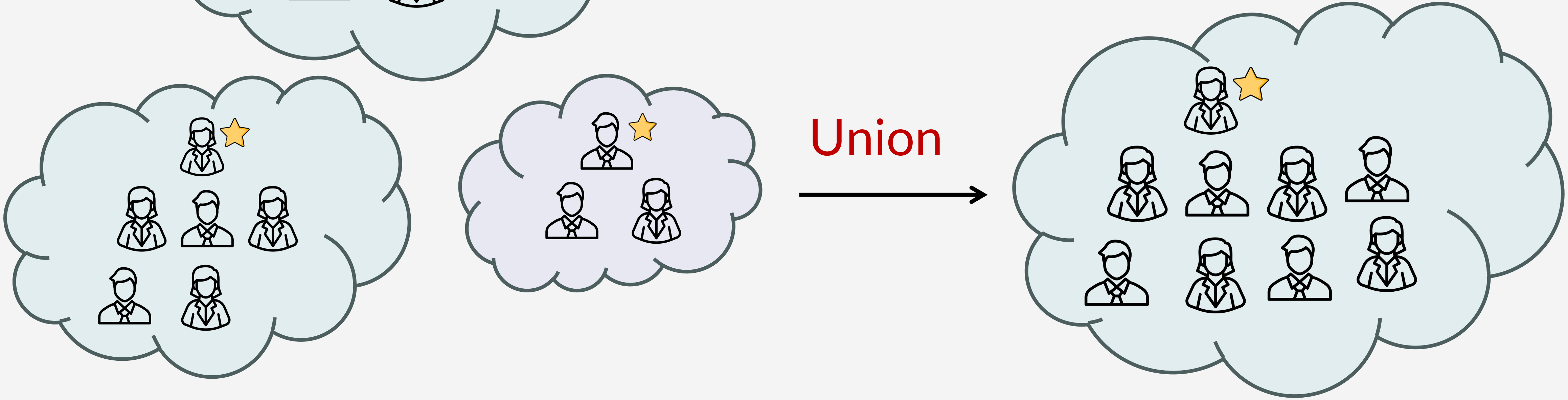
A Common Goal Emerges



Disjoint Set Data Structure

Find

`find(x):` look up the set containing x, and return its representative



`union(x, y):` look up the set containing x, and the set containing y, combine these two sets. Pick a representative for the new set

The Disjoint Set Interface

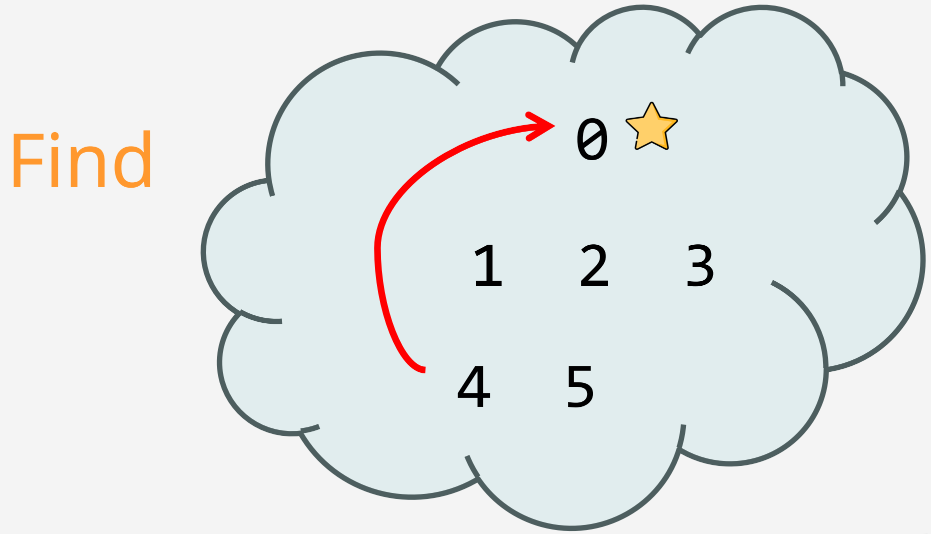
```java
public interface DisjointSet {
    /**
     * Returns the representative of the set containing x.
     * @param x Element whose set representative is to be found.
     * @return The representative of the set containing x.
     */
    int find(int x);
    /**
     * Merges the sets containing x and y.
     * @param x An element of the first set to merge.
     * @param y An element of the second set to merge.
     */
    void union(int x, int y);
}
```

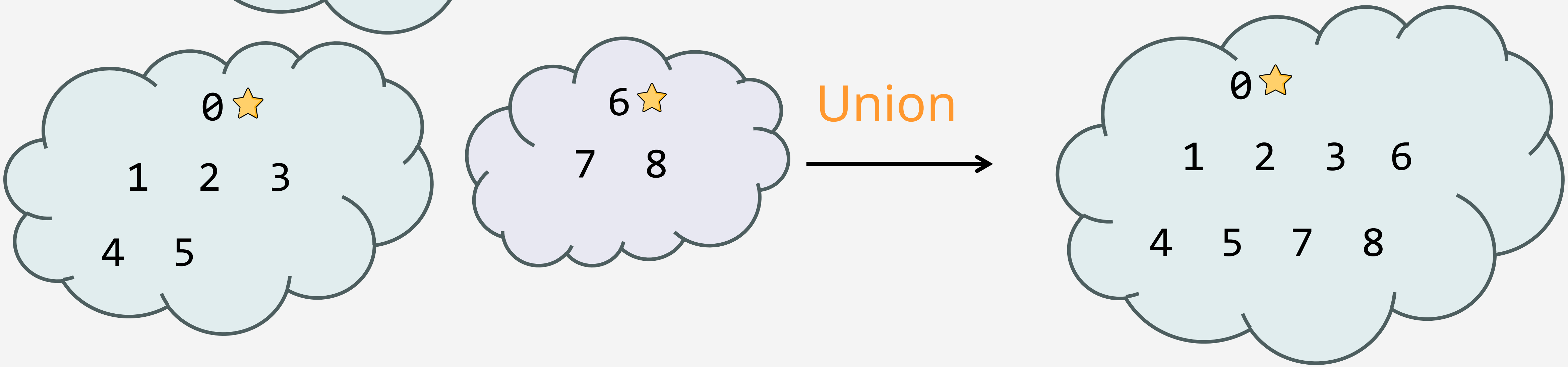
Disjoint Sets are useful

- Connected Components in Graphs
- **Kruskal's Minimum Spanning Tree Algorithm**
- Image Segmentation
- …

Disjoint Sets on Integers

- For simplicity, we use integers instead of arbitrary data





Implementing with Map<Integer, Set<Integer>>?

```
{0, 1, 2, 3, 4, 5}
{ 0:{0}, 1:{1}, 2:{2}, 3:{3}, 4:{4}, 5:{5} }
union(1, 2): { 0:{0}, 2:{1, 2}, 3:{3}, 4:{4}, 5:{5} }
union(0, 4): { 0:{0, 4}, 1:{1, 2}, 3:{3}, 5:{5} }
union(1, 3): { 0:{0, 4}, 3:{1, 2, 3}, 5:{5} }
union(0, 3): { 0:{0, 1, 2, 3, 4}, 5:{5} }
union(2, 5): { 0:{0, 1, 2, 3, 4, 5} }
```

The `find` operation: Looking for which set an item belongs to takes O(N) time!

The `union` operation: need to `find` first -> O(N)

Use another **Map<Integer, Integer>** to save root id?

Complicated! A simpler implementation?

Agenda

Implementing with Arrays?

	0	1	2	3	4	5	
root	0	1	2	3	4	5	0:{0}, 1:{1}, 2:{2}, 3:{3}, 4:{4}, 5:{5}
union index 1, 2:	0	2	2	3	4	5	0:{0}, **2:{1, 2}**, 3:{3}, 4:{4}, 5:{5}
union index 0, 4:	0	2	2	3	0	5	**0:{0, 4}**, 1:{1, 2}, 3:{3}, 5:{5}
union index 1, 3:	0	3	3	3	0	5	0:{0, 4}, **3:{1, 2, 3}**, 5:{5}
union index 2, 4:	0	0	0	0	0	5	**0:{0, 1, 2, 3, 4}**, 5:{5}
union index 2, 5:	0	0	0	0	0	0	**0:{0, 1, 2, 3, 4, 5}**

Quick Find Disjoint Set Implementation

```
public class QuickFindDisjointSet implements DisjointSet {
    private int[] root;

    @Override
    public int find(int x) {
        return root[x];
    }
    @Override
    public void union(int x, int y) {
        int xRoot = find(x);
        int yRoot = find(y);
        if (xRoot != yRoot) {
            for (int i = 0; i < root.length; i++) {
                if (root[i] == xRoot) {
                    root[i] = yRoot;
                }
            }
        }
    }
}
```

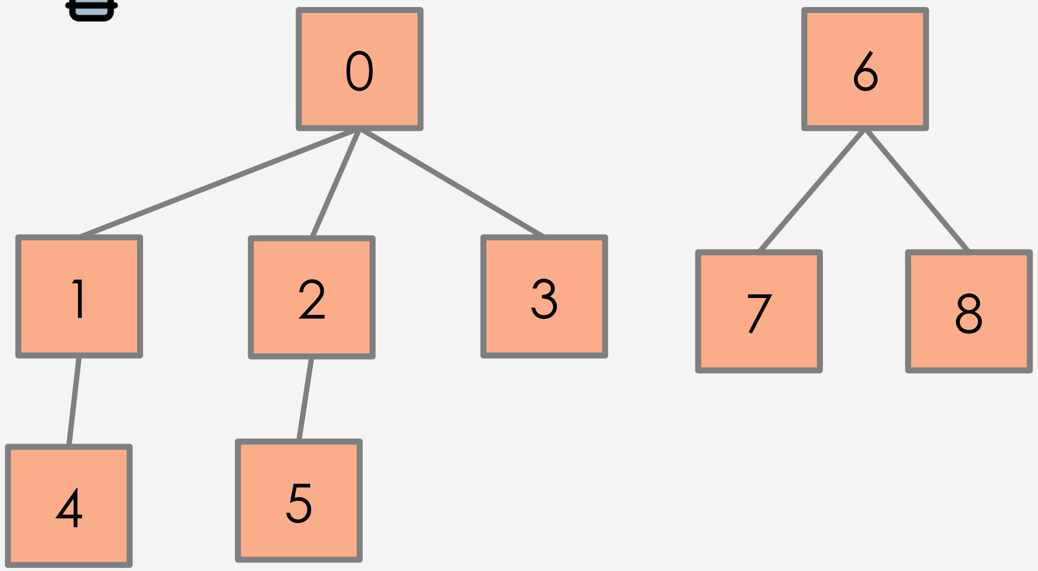
```
public QuickFindDisjointSet(int N) {
    root = new int[N];
    for (int i = 0; i < N; i++) {
        root[i] = i;
    }
}
```

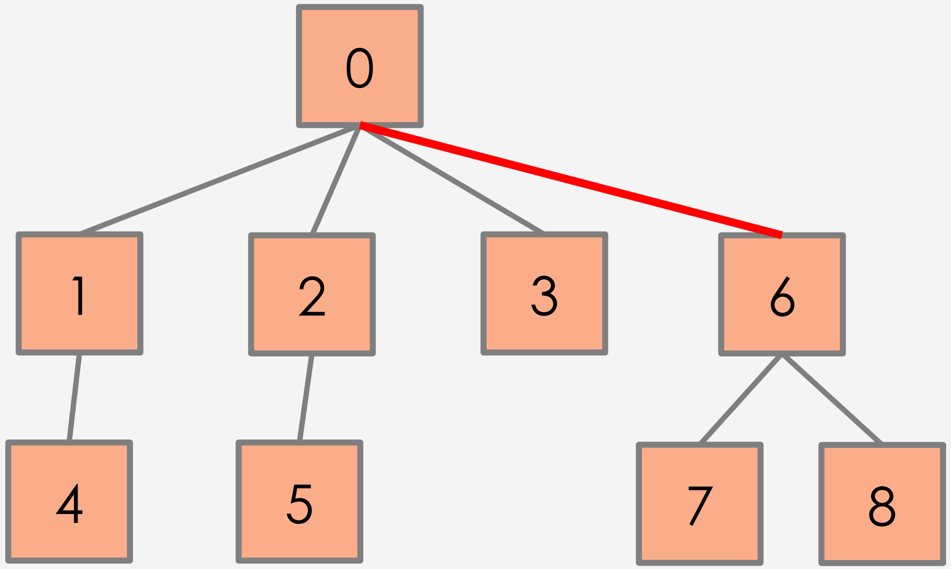
Worst-case Run time analysis

	Constructor	`find`	`union`
`QuickFind`	$\Theta(N)$	$\Theta(1)$	$\Theta(N)$

union is Slow

Think-pair-share: How can we only change 1 value when union?





index	0	1	2	3	4	5	6	7	8
root	0	0	0	0	0	0	6	6	6

index	0	1	2	3	4	5	6	7	8
root	0	0	0	0	0	0			

Agenda

1 Introduction to Disjoint Sets

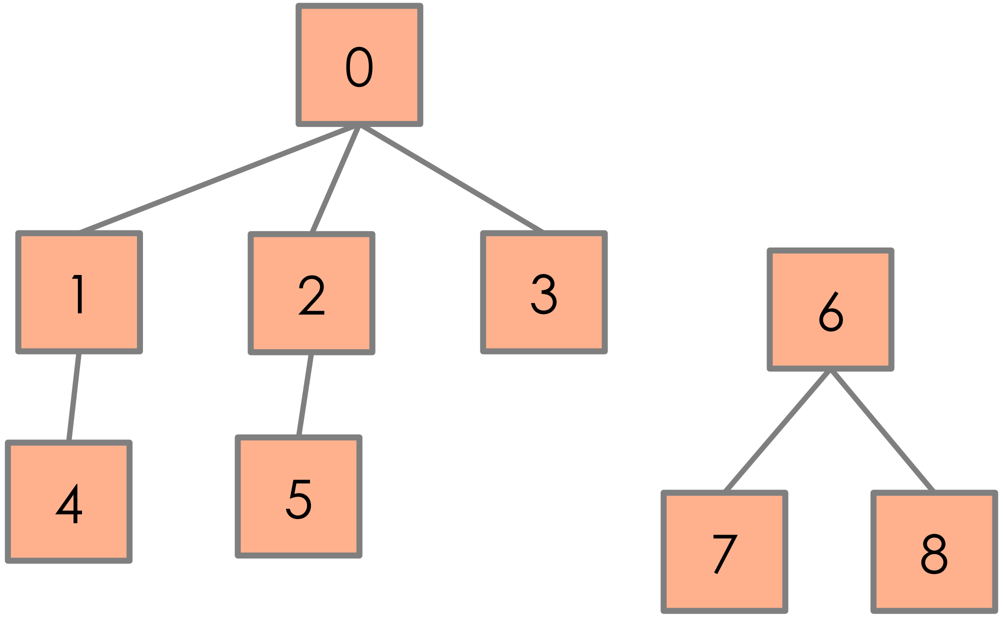
2 Quick Find

3 **Quick Union**

4 Weighted Quick Union

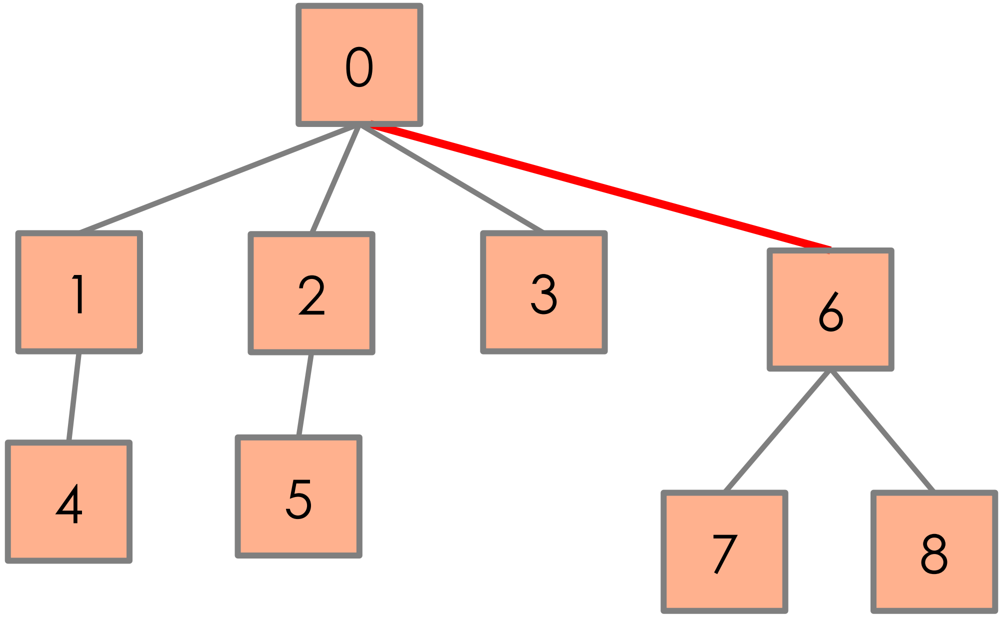
5 Weighted Quick Union with Path Compression

Tracking parent instead of root



index	0	1	2	3	4	5	6	7	8
parent	0	0	0	0	1	2	6	6	6

Tracking parent instead of root



index	0	1	2	3	4	5	6	7	8
parent	0	0	0	0	1	2	**0**	6	6

Quick Union Disjoint Set Implementation

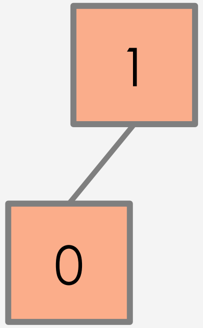
```java
public class QuickFindDisjointSet implements DisjointSet {
    private int[] parent;

    @Override
    public int find(int x) {
        while (x != parent[x]) {
            x = parent[x];
        }
        return x;
    }
    @Override
    public void union(int x, int y) {
        int xRoot = find(x);
        int yRoot = find(y);

        if (xRoot != yRoot) {
            parent[xRoot] = yRoot;
        }
    }
}
```

```java
public QuickUnionDisjointSet(int N) {
    parent = new int[N];
    for (int i = 0; i < N; i++) {
        parent[i] = i;
    }
}
```

The tree can be very tall

always make the 2nd node as new root

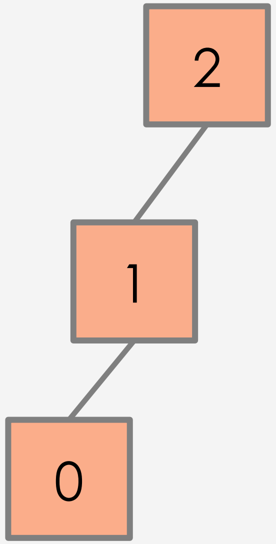
`union(0,1)`



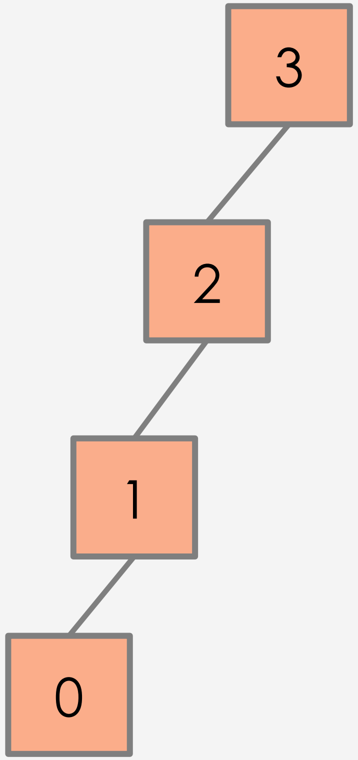
The tree can be very tall

always make the 2nd node as new root

```
union(0,1)
union(1,2)
```



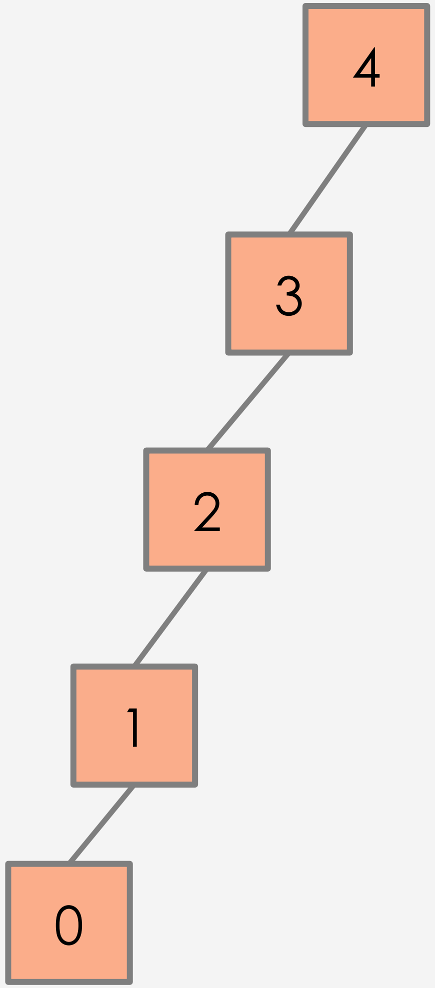
The tree can be very tall



always make the 2nd node as new root

```
union(0,1)
union(1,2)
union(2,3)
```

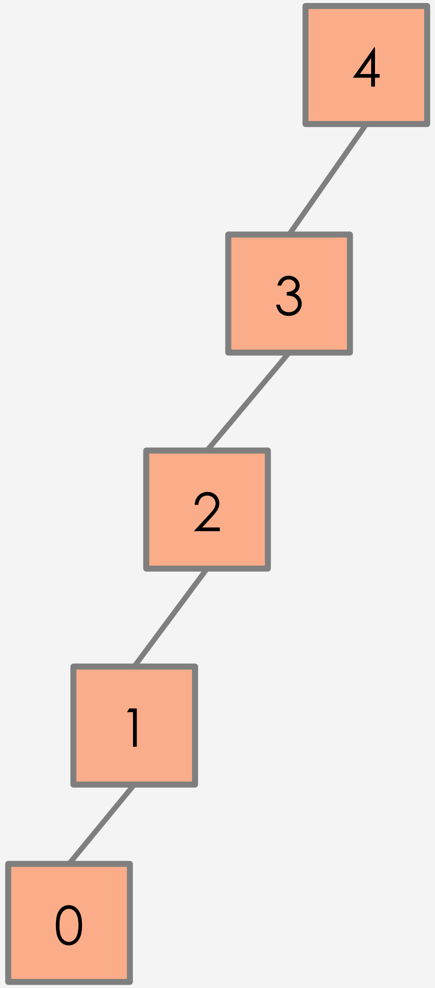
The tree can be very tall



always make the 2nd node as new root

```
union(0,1)
union(1,2)
union(2,3)
union(3,4)
```

The tree can be very tall



always make the 2nd node as new root

```
union(0,1)
union(1,2)
union(2,3)
union(3,4)
```

Worst-case, height is N

Worst-case Run Time Analysis

	Constructor	`find`	`union`
`QuickFind`	$\Theta(N)$	$\Theta(1)$	$\Theta(N)$
`QuickUnion`	$\Theta(N)$	$\Theta(N)$	$\Theta(N)$

Quiz 1

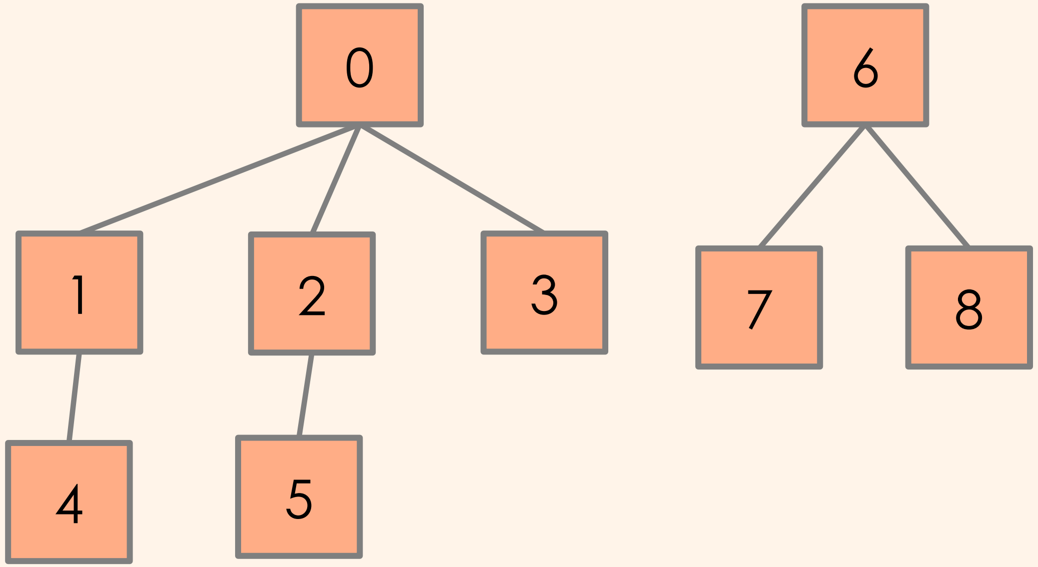
- In the ***Quick Union*** implementation, suppose we change the highlighted line to `parent[x] = yRoot`, is it still correct?
  - Yes
  - No

```
public void union(int x, int y) {
    int xRoot = find(x);
    int yRoot = find(y);

    if (xRoot != yRoot) {
        parent[xRoot] = yRoot;
    }
}
```

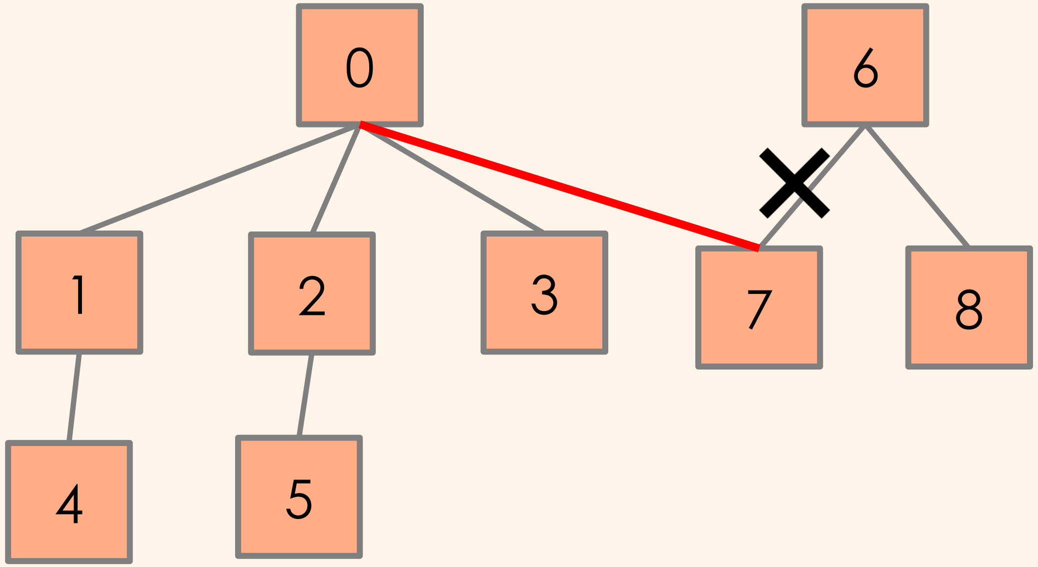
Quiz 1

```
public void union(int x, int y) {
    int xRoot = find(x);
    int yRoot = find(y);

    if (xRoot != yRoot) {
        parent[x] = yRoot;
    }
}
```

```
union(7, 5)
xRoot = 6, yRoot = 0
parent[7] = 0
```



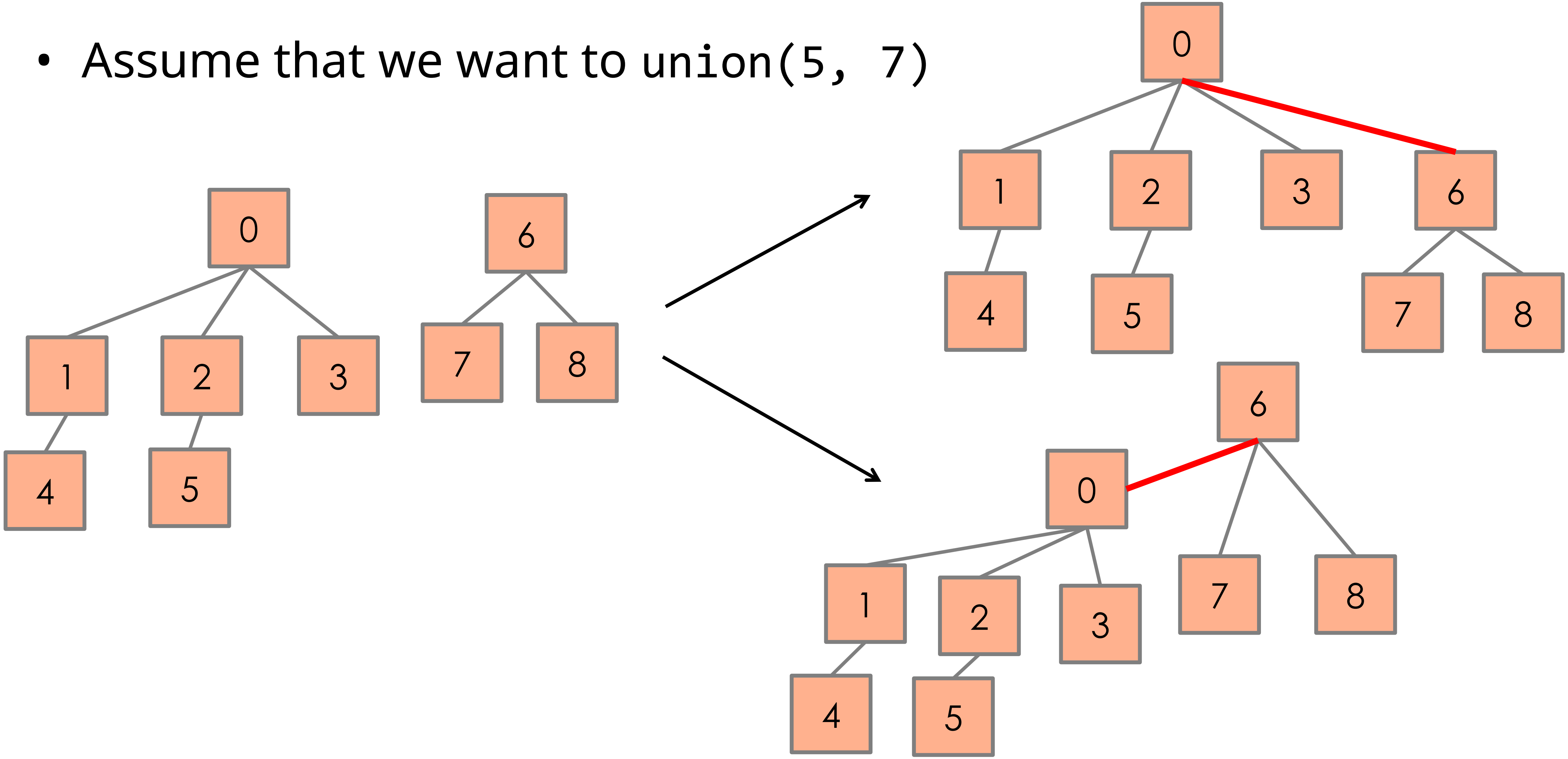
index	0	1	2	3	4	5	6	7	8
parent	0	0	0	0	1	2	6	6	6



index	0	1	2	3	4	5	6	7	8
parent	0	0	0	0	1	2	6	6	6

Which One is better?

- Assume that we want to `union(5, 7)`



Agenda

1 Introduction to Disjoint Sets

2 Quick Find

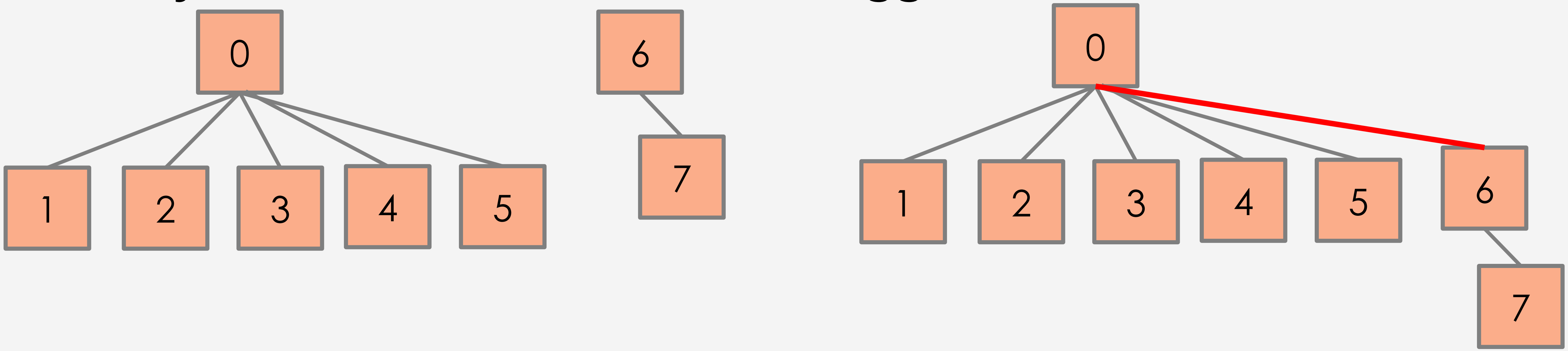
3 Quick Union

4 **Weighted Quick Union**

5 Weighted Quick Union with Path Compression

Weighted Quick-Union

- Track the ***size*** of the trees (how many nodes)
- Always link the ***smaller*** trees to ***bigger*** trees

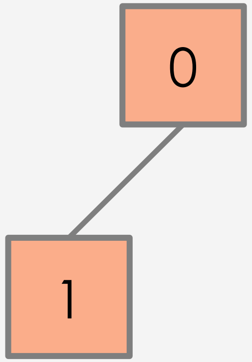


Weighted Quick Union Disjoint Set Implementation

```java
public class WeightedQuickUnionDisjointSet implements DisjointSet {
    private int[] parent;
    private int[] size;
    @Override
    public int find(int x) { ... }
    @Override
    public void union(int x, int y) {
        int xRoot = find(x);
        int yRoot = find(y);
        if (xRoot == yRoot) return;

        if (size[xRoot] < size[yRoot]) {
            parent[xRoot] = yRoot;
            size[yRoot] += size[xRoot];
        } else {
            parent[yRoot] = xRoot;
            size[xRoot] += size[yRoot];
        }
    }
}
```

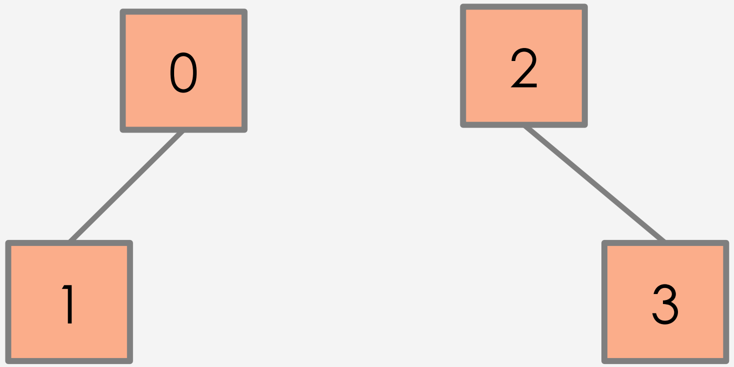
```java
public WeightedQuickUnionDisjointSet(int N) {
    parent = new int[N];
    size = new int[N];
    for (int i = 0; i < N; i++) {
        parent[i] = i;
        size[i] = 1;
    }
}
```

Still, the tree can be very tall



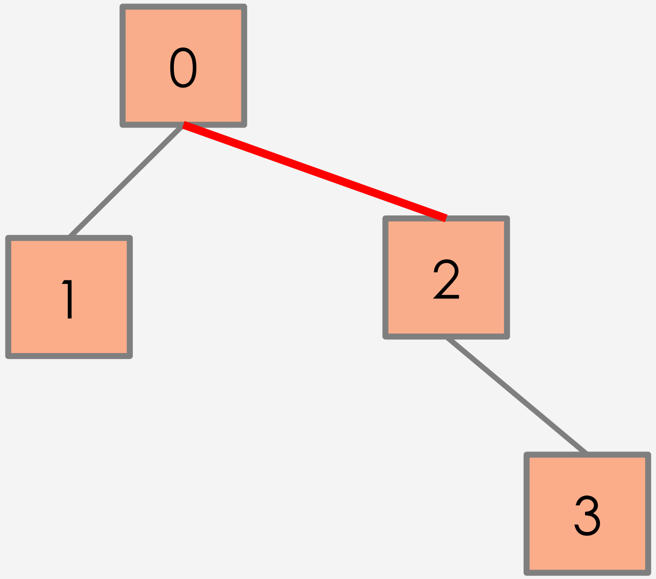
Number of Nodes	Height of the tree
2	1

Still, the tree can be very tall



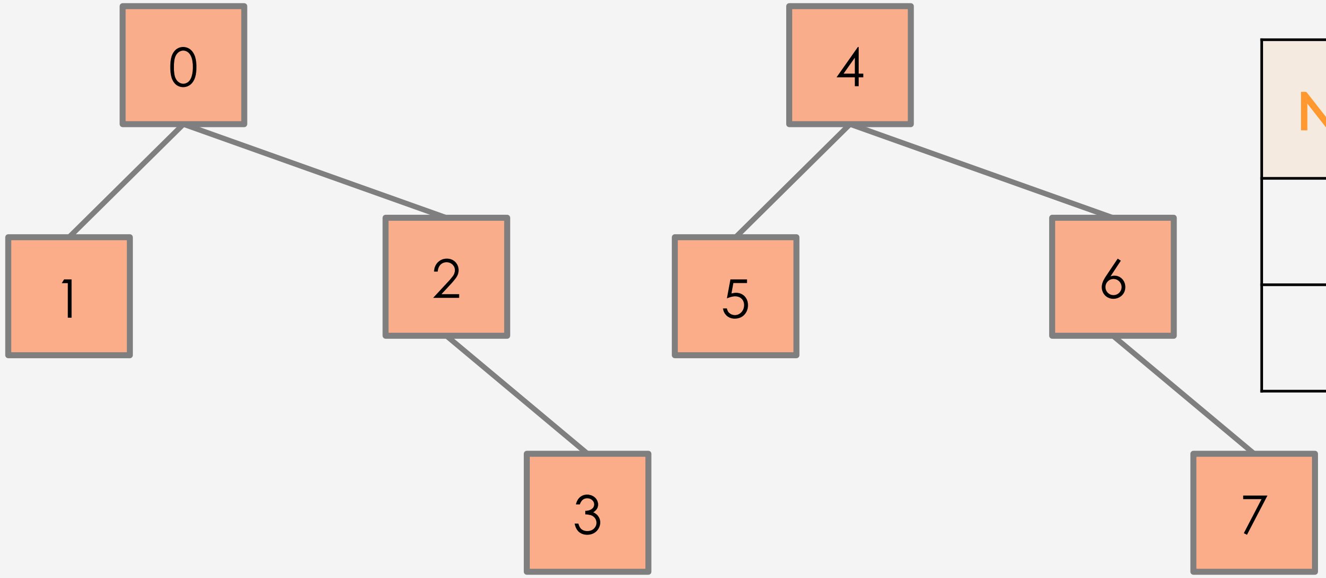
Number of Nodes	Height of the tree
2	1

Still, the tree can be very tall



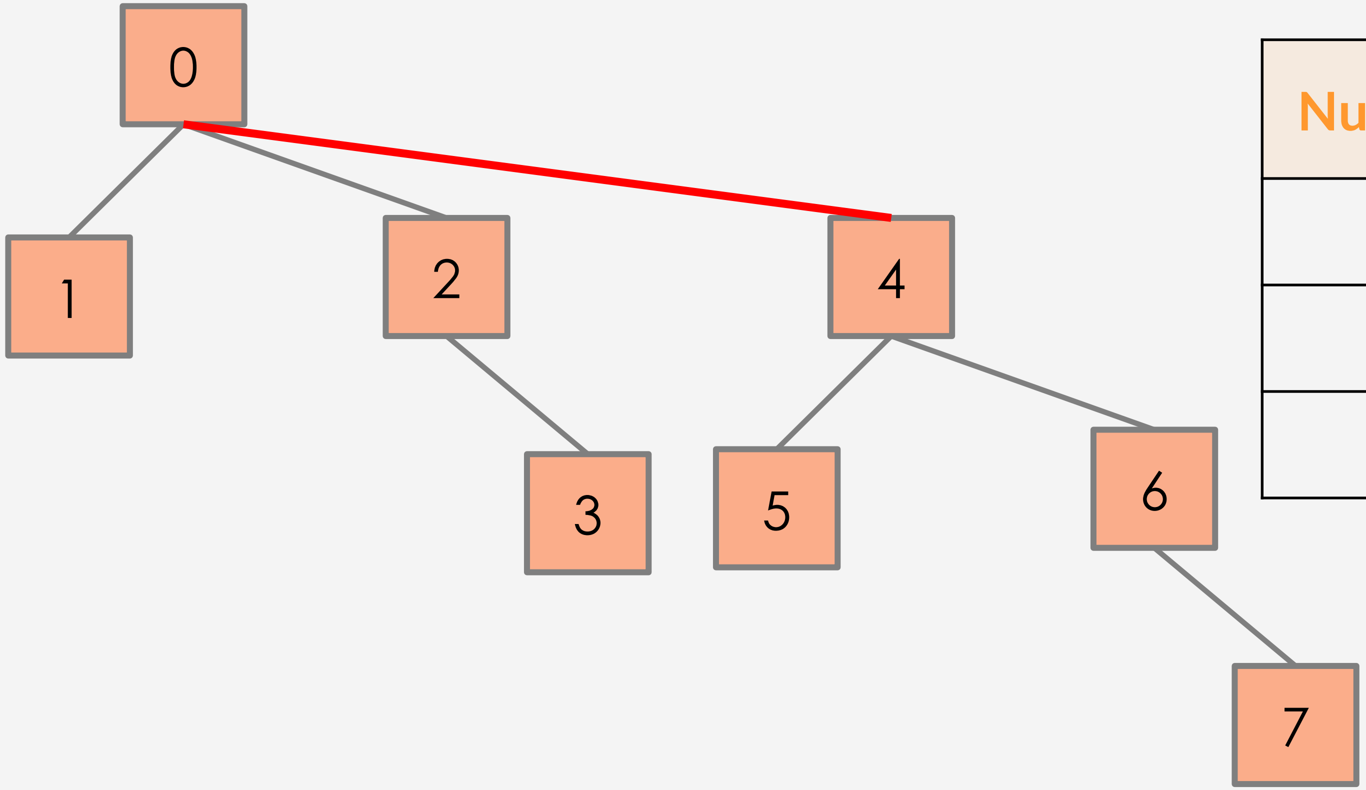
Number of Nodes	Height of the tree
2	1
4	2

Still, the tree can be very tall



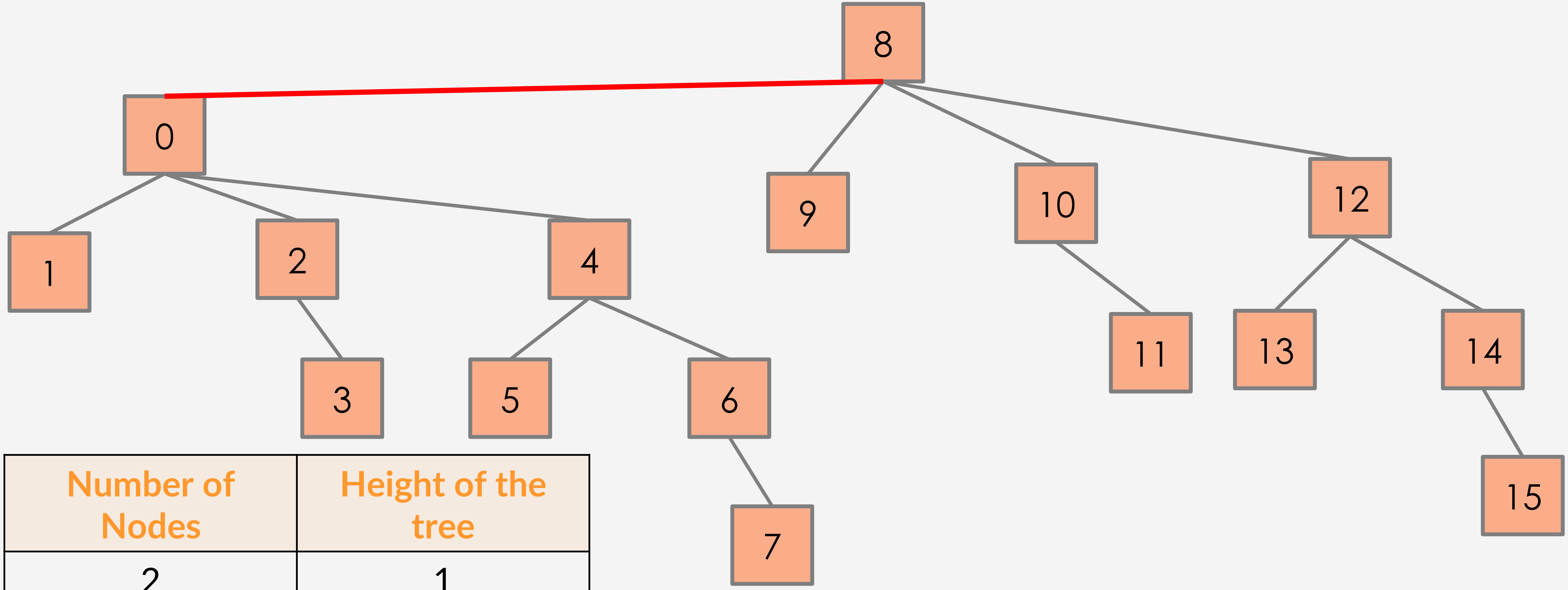
Number of Nodes	Height of the tree
2	1
4	2

Still, the tree can be very tall



Number of Nodes	Height of the tree
2	1
4	2
8	3

Still, the tree can be very tall



Number of Nodes	Height of the tree
2	1
4	2
8	3
16	4

Worst-case, height is logN

Worst-case Run Time Analysis

	Constructor	`find`	`union`
`QuickFind`	$O(N)$	$O(1)$	$O(N)$
`QuickUnion`	$O(N)$	$O(N)$	$O(N)$
`WeightedQuickUnion`	$O(N)$	$O(\log N)$	$O(\log N)$

Agenda

1 Introduction to Disjoint Sets

2 Quick Find

3 Quick Union

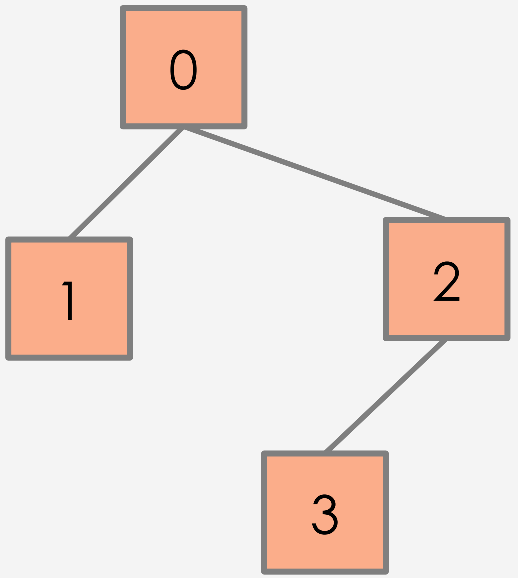
4 Weighted Quick Union

5 **Weighted Quick Union with Path Compression**

Weighted Quick Union with Path Compression

- When we do `find`, tie all the nodes seen to the root

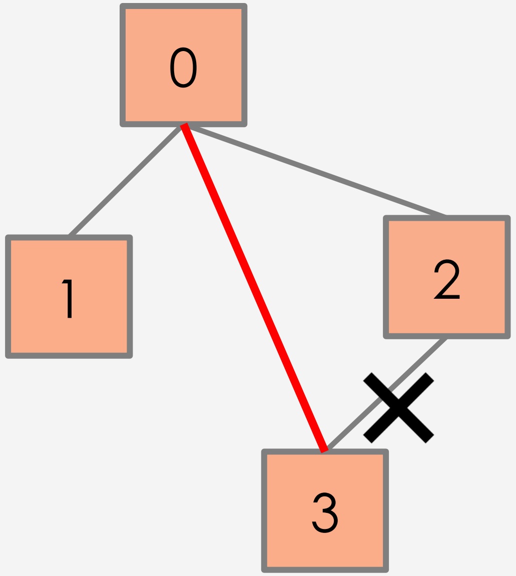
`find(3)`



Weighted Quick Union with Path Compression

- When we do `find`, tie all the nodes seen to the root

```
find(3)
```



Visualization: https://www.cs.usfca.edu/~galles/visualization/DisjointSets.html

Weighted Quick Union with Path Compression

- When we do `find`, tie all the nodes seen to the root

Quick Union

```
@Override
public int find(int x) {
      while (x != parent[x]) {
        x = parent[x];
      }
      return x;
}
```

```
@Override
public int find(int x) {
    int root = x;
    while (root != parent[root]) {
        root = parent[root];
    }
    while (x != root) {
        int newx = parent[x];
        parent[x] = root;
        x = newx;
    }
    return root;
}
```

Weighted Quick Union with Path Compression

- Recursion?

```
@Override
public int find(int x) {
    int root = x;
    while (root != parent[root]) {
        root = parent[root];
    }
    while (x != root) {
        int newx = parent[x];
        parent[x] = root;
        x = newx;
    }
    return root;
}
```

```
public int find(int x) {
    if (x != parent[x]) {
        parent[x] = find(parent[x]);
    }
    return parent[x];
}
```

Quiz 2

- In the ***Weighted Quick Union with Path Compression*** implementation (recursion), suppose we change the highlighted line to `return x`, is it still correct?
  - Yes
  - No

```
public int find(int x) {
    if (x != parent[x]) {
        parent[x] = find(parent[x]);
    }
    return parent[x];
}
```

Quiz 2

- Java Visualizer: https://pythontutor.com/render.html#mode=display

A simple example:

`union(0, 1)`

`union(1, 2)`

- `return parent[x]`
  - `parent: 0, 0, 0`
- `return x`
  - `parent: 0, 0, 1`

`Answer: No`

Amortized Analysis

- Each operation takes on average `lg*N` time
  - `lg* N` represents the iterated logarithm function, which is the number of times you need to take the logarithm of N before the result becomes less than or equal to 1
  - `lg*` is less than or equal to 5 for any realistic input
- A tighter upbound, each operation takes on average `α(N)` time
  - α is the inverse Ackermann function

N	lg* N
1	0
2	1
4	2
16	3
65536	4
2^{65536}	5

Run Time Analysis

- Assume we have N items and M operations (either `union` or `find`)

	Run Time
`QuickFind`	$O(NM)$
`QuickUnion`	$O(NM)$
`WeightedQuickUnion`	$O(N + M\ logN)$
`WeightedQuickUnionPathCompression`	$O(N + M\alpha(N))$

Summary

- Standard way how disjoint sets are implemented today:
  - weighted quick union + path compression
- All the implementations we have considered today:
  - Quick Find
  - Quick Union
    - Weighted Quick Union
      - Weighted Quick Union with Path Compression
- Run time analysis
- Exit ticket: https://forms.gle/DRoaUZ7ehKr9BUrG9

Acknowledgements

- The slides and quizzes got inspirations from
  - UC Berkeley CS 61B: Data Structures
  - University of Washington CSE 373: Data Structures and Algorithms
  - http://algs4.cs.princeton.edu
- Icons are from https://www.flaticon.com/