CompSci 201, L27: Minimum Spanning Tree (MST) and Disjoint Sets (Continued)

Logistics, coming up

- Today, Monday, 4/24
  - Project P6: Route (last project) due
- End of Semester Survey
  - Due by LDOC this Wednesday, April 26
  - Required class survey, run by the course staff
- Course Evaluations
  - Due by this Saturday, April 29
  - Run by Duke, anonymous to us

Final Exam Logistics Reminder

- 3 final sections: F1, F2, F3 corresponding to 3 midterms M1, M2, M3.
  - Exams grade = Avg(Max(M1, F1), Max(M2, F2), Max(M3, F3))
  - If happy with grades? Don't need to take it.
  - If you missed a midterm? Make sure to take at least that part.

- Monday 5/1, 9 am - noon. You will have 50 minutes to complete each part.
  - 9-9:50 am. F1 (corresponding to M1)
  - 10-10:50 am. F2 (corresponding to M2)
  - 11-11:50 am. F3 (corresponding to M3)

Final Grade Estimates

- Everything should be live and updated in the sakai gradebook *except*:
  - To add today or tomorrow:
    - P5
  - To add by this weekend
    - P6
    - APT9
    - Last couple weeks WOTOs

Today's Agenda

1. Review Minimum Spanning Tree (MST) problem and Kruskal's Algorithm

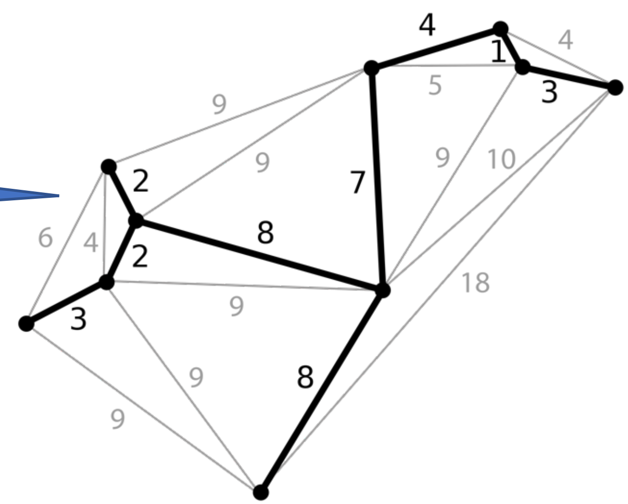
2. Investigate efficient disjoint sets / union find data structure

3. (time permitting) Solve an MST problem together live

Minimum Spanning Tree (MST) Problem

- Given N nodes and M edges, each with a weight/cost…
- Find a set of edges that connect *all* the nodes with minimum total cost. (will be a tree)

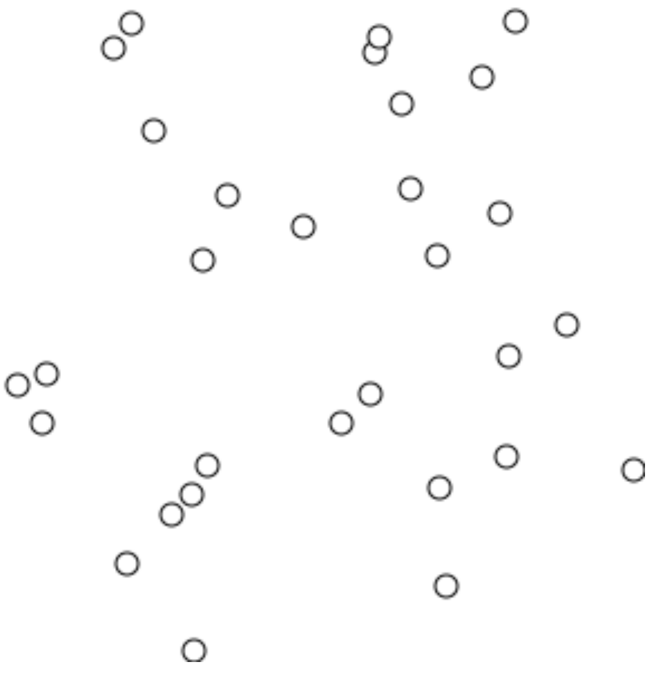
Weighted undirected graph with:
- Edges labeled with weights/costs
- Minimum spanning tree highlighted



Visualizing Kruskal's Algorithm

In the visualization:

- Edges between all pairs of vertices
- Weights are implicit by distances
- Algorithm greedily grows by cheapest edge that connects disjoint sets/trees.



By Shiyu Ji - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=54420894

Kruskal's Algorithm in *Pseudocode*

Input: N node, M edges, M edge weights

- Let MST to an empty set
- Let S be a collection of N **disjoint sets**, one per node
- While S has more than 1 set:
  - Let (u, v) be the minimum cost remaining edge
  - **Find** which sets u and v are in. If not equal:
    - **Union** the sets
    - Add (u, v) to MST
- Return MST

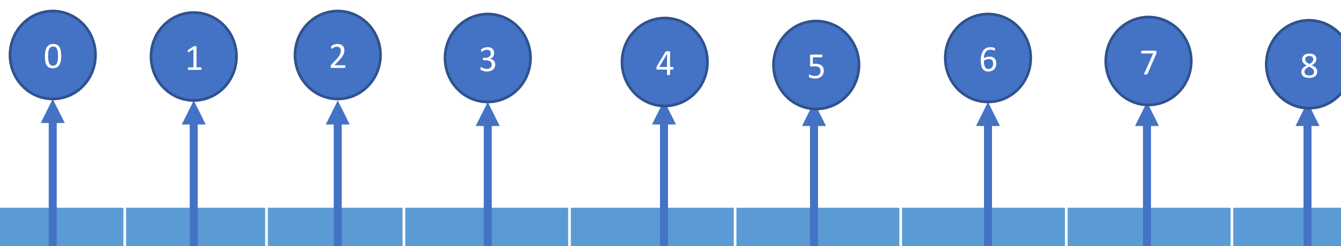
Disjoint Sets and Union-Find

Union Find Data Structure

- Aka Disjoint Set Data Structure
- Start with N distinct (disjoint) sets
  - consider them labeled by integers: 0, 1, …
- ***Union*** two sets: create set containing both
  - label with one of the numbers
- ***Find*** the label of the set containing a number
  - Initially self, but changes after unions

Disjoint-Set Forest Implementation

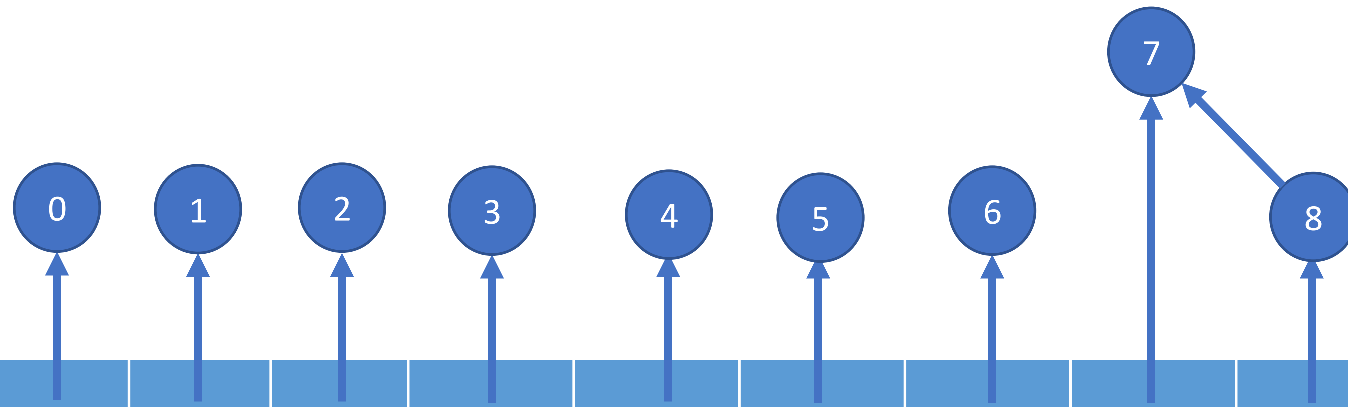
- Each set will be represented by a parent "tree": Instead of child pointers, nodes have a parent "pointer".
- Everything starts as its own tree: a single node



parent									
itemID	0	1	2	3	4	5	6	7	8

Disjoint-Set Forest Union

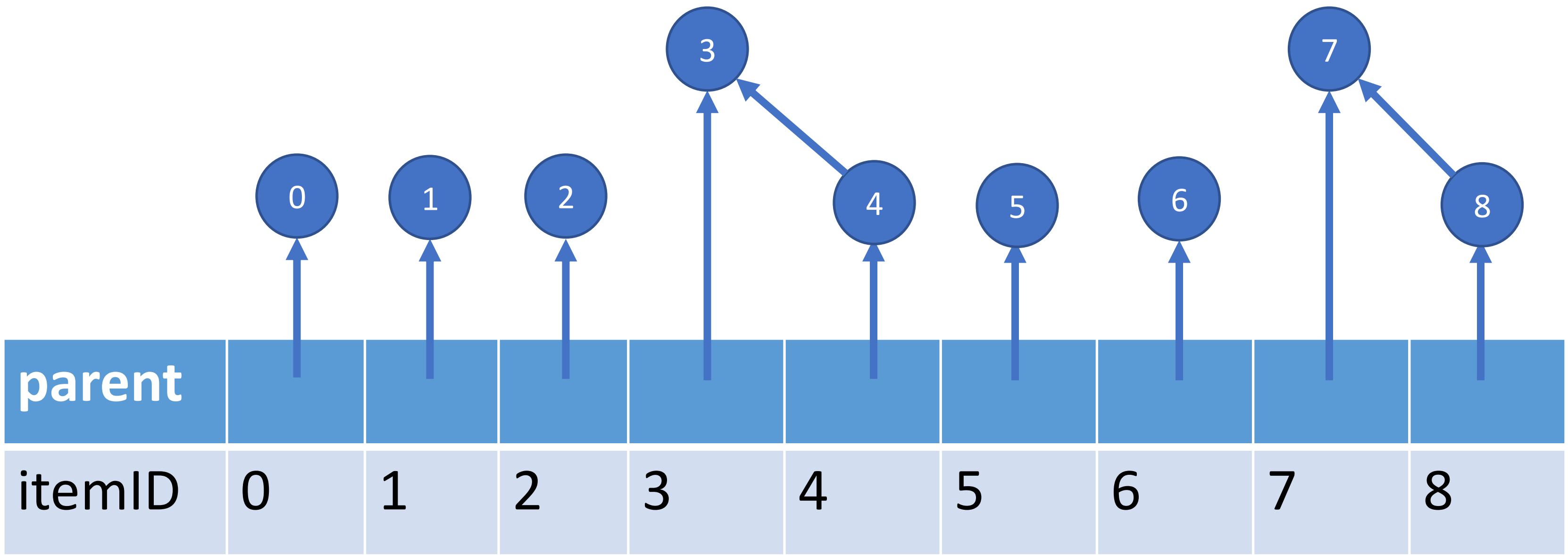
- Union(7,8)
- Just make leaf/root point to parent[7]



parent									
itemID	0	1	2	3	4	5	6	7	8

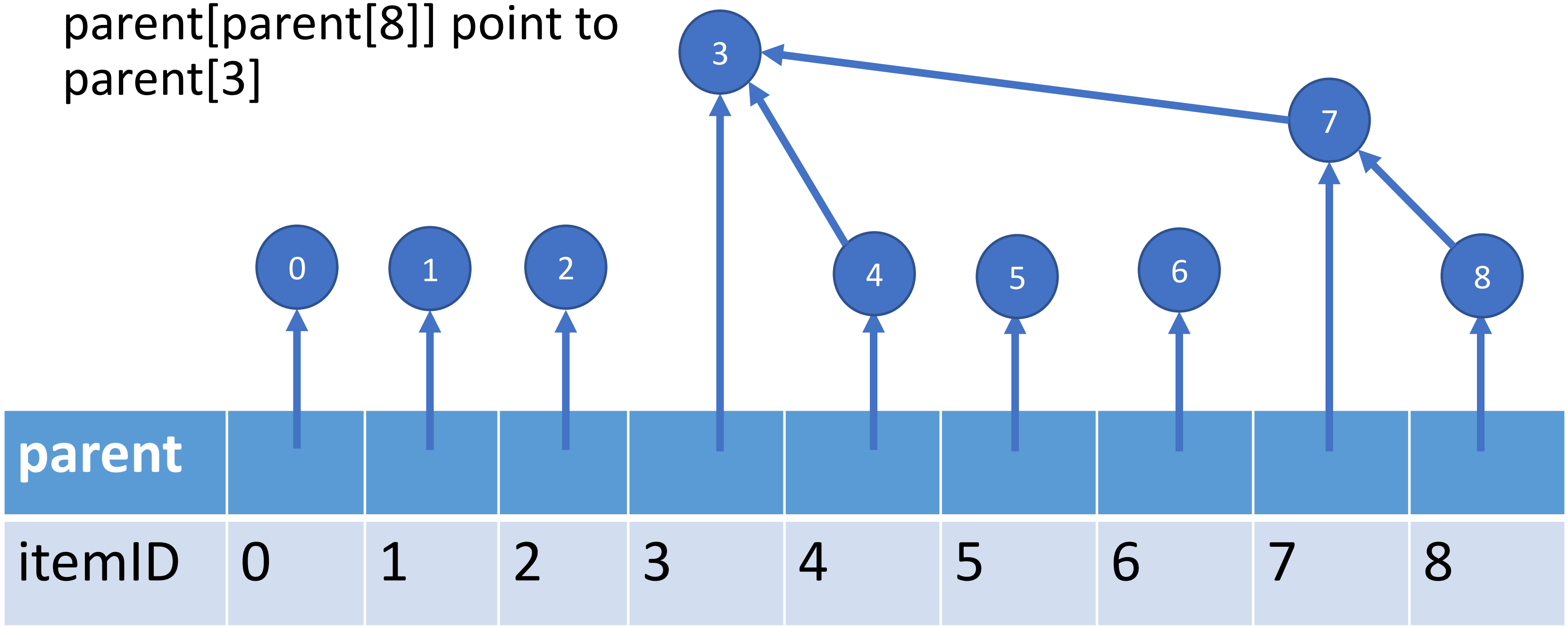
Disjoint-Set Forest Union

- Union(3,4)
- Just make parent[4] point to parent[3]



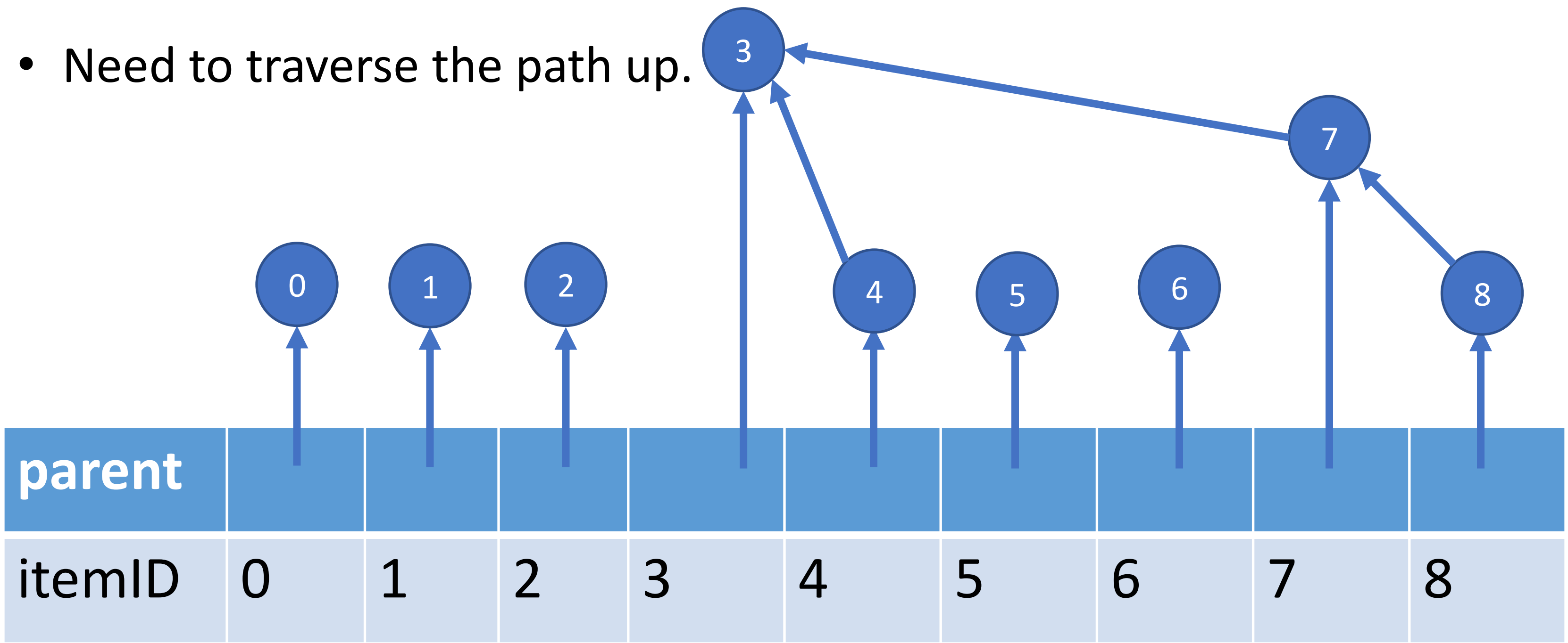
Disjoint-Set Forest Union

- Union(3,8)
- Multi-level, make parent[parent[8]] point to parent[3]



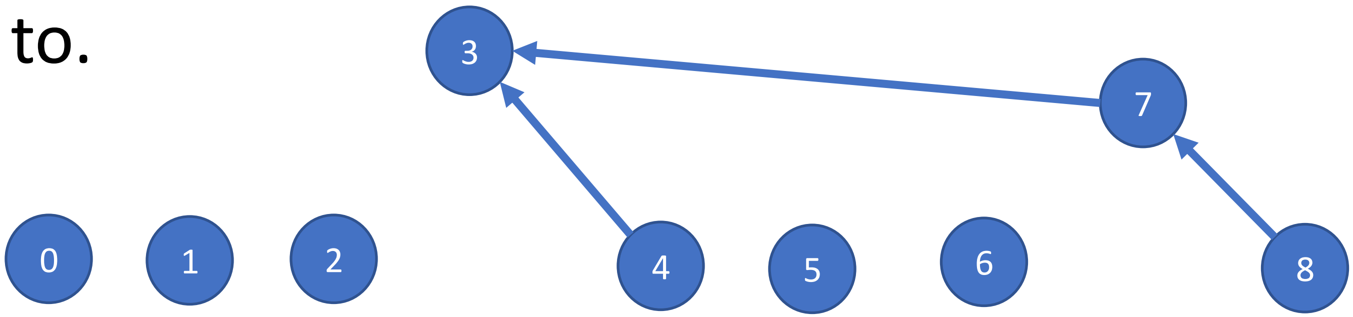
Disjoint-Set Forest Find

- Find(8)
- Return last ancestor of 8.
- Need to traverse the path up.



Disjoint-Set Forest Array Representation

- The “nodes” and “pointers” are just conceptual – can represent with a simple array, like binary heap.
- Parent array just stores what the itemID node points to.



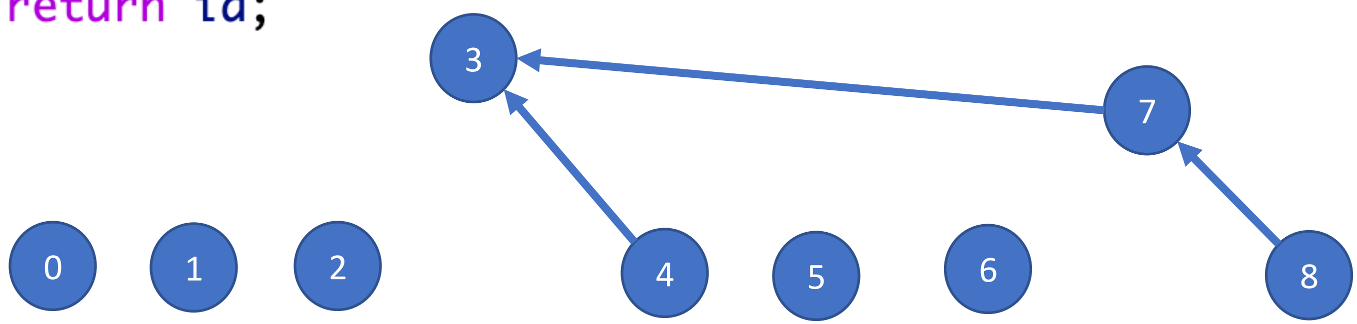
parent	0	1	2	3	3	5	6	3	7
itemID	0	1	2	3	4	5	6	7	8

Disjoint-Set Forest Find

```java
public int find(int id) {
    while (id != parent[id]) {
        id = parent[id];
    }
    return id;
}
```

"last ancestor" is just when parent[i] = i

Else go to next "node up"



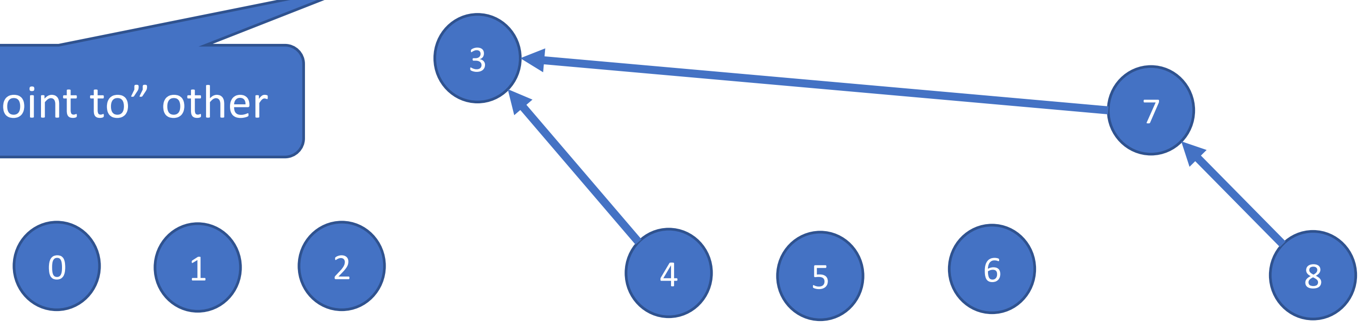
parent	0	1	2	3	3	5	6	3	7
itemID	0	1	2	3	4	5	6	7	8

Disjoint-Set Forest Union Revisited

```java
public void union(int set1, int set2) {
    int root1 = find(set1);
    int root2 = find(set2);
    parent[root2] = root1;
```

"last ancestors" from initial set1 and initial set2 "nodes"

Make one "point to" other



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

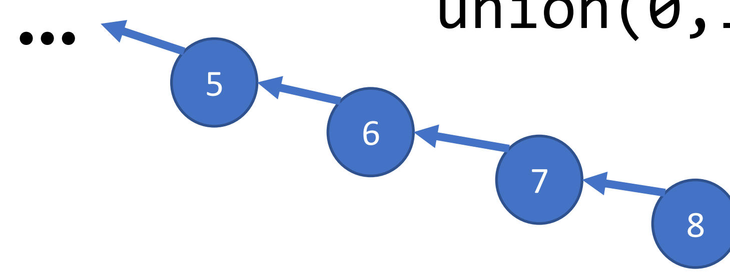
Worst-Case Runtime Complexity?

```java
public void union(int set1, int set2) {
    int root1 = find(set1);
    int root2 = find(set2);
    parent[root2] = root1;
```

What if we…

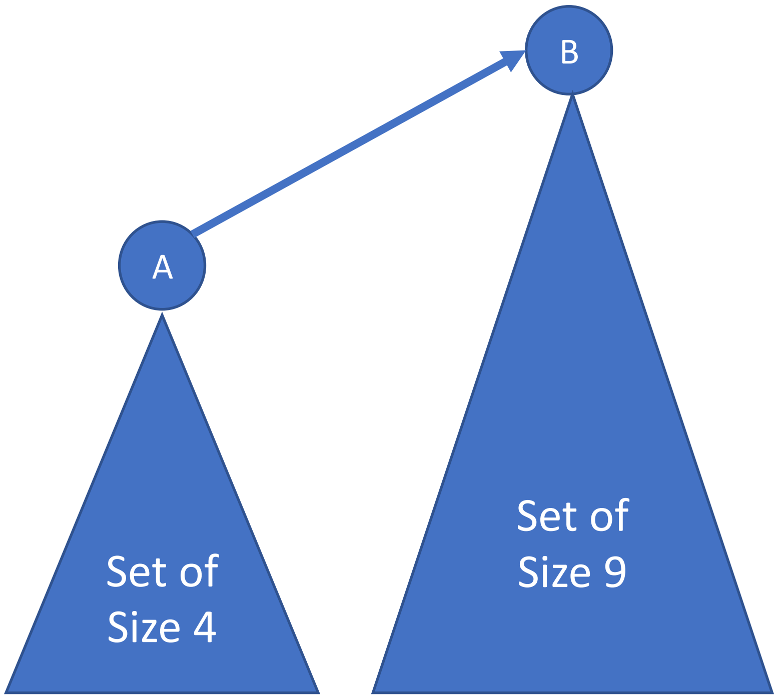
```
union(7,8)
union(6,7)
union(5,6)
…
union(0,1)
```

Now `find(8)` would have linear runtime complexity!!



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

Optimization 1: Union by Size



Be careful in how you union. Always make the "root" for the set with *fewer* elements point to the "root" for the set with *more* elements.

Sufficient for worst case logarithmic efficiency.

Optimization 1: Union by Size

```java
public void union(int set1, int set2) {
    int root1 = find(set1);
    int root2 = find(set2);
    if (root1 == root2) { return; }
    if (setSizes[root1] < setSizes[root2]) {
        parent[root1] = root2;
        setSizes[root2] += setSizes[root1];
    }
    else {
        parent[root2] = root1;
        setSizes[root1] += setSizes[root2];
    }
    size--;
}
```

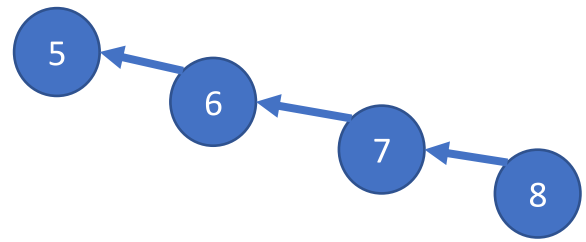
If already in same set, nothing to do.

Make the smaller set “point to” the bigger set.

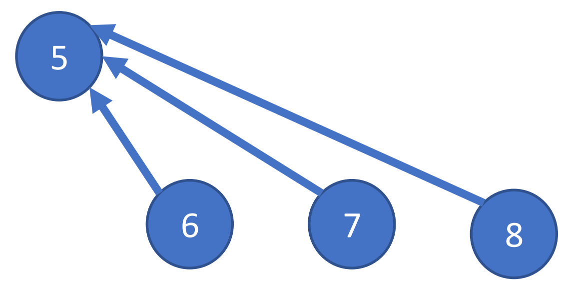
Lazy Path Compression

- **Lazy path compression**: When ever you traverse a path in `find`, connect all the pointers to the top.

- Sufficient for **amortized logarithmic** runtime complexity for union/find operations.



`find(5)`



Disjoint Set Forest Path Compression

```java
public int find(int id) {
    int idCopy = id;
    while (id != parent[id]) {
        id = parent[id];
    }
    int root = id;
    id = idCopy;
    while(id != parent[id]) {
        parent[idCopy] = root;
        id = parent[id];
        idCopy = id;
    }
    return id;
}
```

Get the “last ancestor” as before

Traverse path again, assigning everything to the “last ancestor”

Optimized Runtime Complexity

- Optimizations considered separately:
  - Union by size: Worst case logarithmic
  - Path compression: Amortized logarithmic

- Considered together…?
  - Worst case logarithmic, and *amortized inverse Ackermann function a(n)*.
  - $a(n) < 5$ for $n < 2^{2^{2^{2^{16}}}} = 2^{2^{2^{65536}}}$
  - Practically constant for any n you can write down

Remember Kruskal's Algorithm Runtime?

Input: N node, M edges, M edge weights

- Let MST to an empty set
- Let S be a collection of N **disjoint sets**, one per node
- While S has more than 1 set:
  - Let (u, v) be the minimum cost remaining edge
  - **Find** which sets u and v are in. If not equal:
    - **Union** the sets
    - Add (u, v) to MST
- Return MST

Looping over (worst case) all M edges

Remove from binary heap, O(log(M))

O(M(log(M)~~+C~~) because C < log(M) for our optimized union find

Solving Example MST Problem

leetcode.com/problems/min-cost-to-connect-all-points

Live Coding