CompSci 201, L25: Minimum Spanning Tree (MST) and Disjoint Sets

Logistics, coming up

- This Wednesday, 4/19
  - Midterm exam 3
  - APT 9 (last APTs) - extended to Thursday 4/20

- Next Monday, 4/24
  - Project P6: Route (last project) due

- Monday after next, 5/1
  - Final exam, 9 am

Midterm Exam 3

- Logistics:
  - 60 minutes, in-person, short answer
  - Can bring 1 reference/notes page
- Topics could include:
  - Trees, binary search trees, binary heaps, recursion
  - Red-black trees: Properties and implications, yes, details of rebalance algorithm, no.
  - Greedy, Huffman
  - Graphs, DFS, BFS, Dijkstra's
- Practice exams will release by the weekend

Final Exam Details – Parts & Grading

- 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 Exam Details - Format

- Topics/questions same as for corresponding midterms.

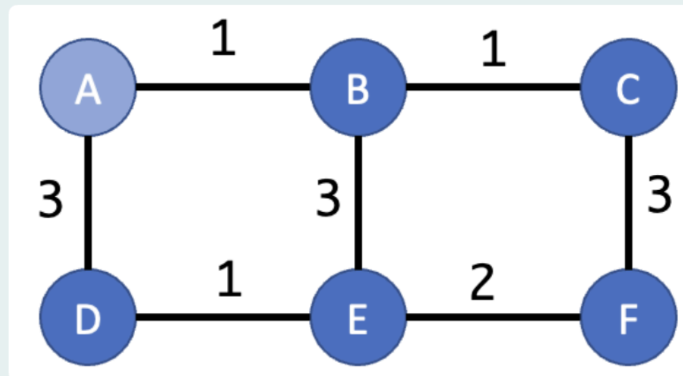
- 20-25 multiple choice questions per part.
  - Questions like those on midterms, just multiple choice.

- Previous practice midterm exams and actual midterm exams will be the most closely aligned practices to review.
  - Examples in multiple choice format in discussion review this Friday.

Reviewing WOTO from last class

6

Again starting from A in the same graph, which path from A to E will Dijkstra's algorithm record as the shortest path? Hint: See the if statement above in problem 4. *




```
for (char neighbor : aList.get(current)) {
    int weight = getWeight(current, neighbor);
    if (!distance.containsKey(neighbor)
    || distance.get(neighbor) > distance.get(current) + weight) {
```

- A, B, E
- A, D, E
- Might be either, depends on tie breaking in the priority queue

- Explore A
- B is closer than D, so we explore B after A
- Find path A -> B -> E of length 4
- Will later find path A -> D -> E of length 4, but the code updates if finding a strictly shorter path

Runtime Complexity of Dijkstra's Algorithm (with N nodes, M edges)

N iterations?

O(log(N)), heap

```
while (toExplore.size() > 0) {
    char current = toExplore.remove();
    for (char neighbor : aList.get(current)) {⋯
}
return distance;
```

Iterations over neighbors

O(1) in HashMap, O(log(N)) in TreeMap

Like BFS, consider each node once and each edge twice, log(N) operations for each: **O((N+M)log(N))**

Problem with Heap Duplicates

May actually loop more than N times

```java
while (toExplore.size() > 0) {
    char current = toExplore.remove();
    for (char neighbor : aList.get(current)) {…
}
return distance;
```

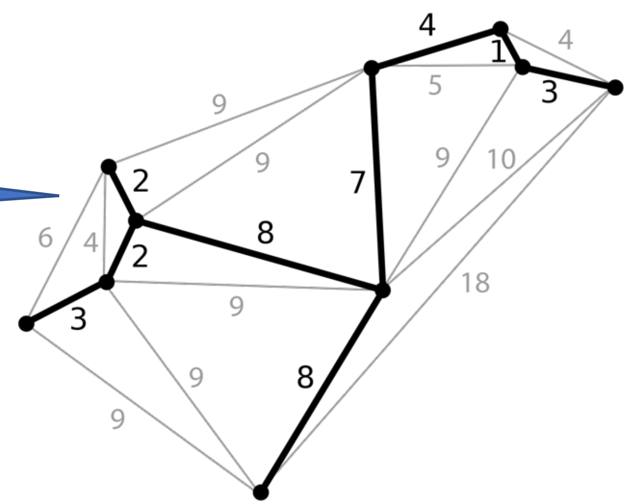
- In graphs with *constant degree* (where each node has at most a constant number of neighbors), will still just be O(N) iterations, maybe not N.
- For general graphs worst-case provable O((N+M) log(N)) need an efficient priority queue update.

Minimum Spanning Tree (MST) and Greedy Graph Algorithms

Minimum Spanning Tree (MST) Problem

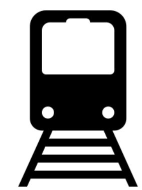
- Given N nodes and M edges, each with a weight/cost…
- Find a set of edges that connect *all* the nodes with minimum total cost. (will be a tree)

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



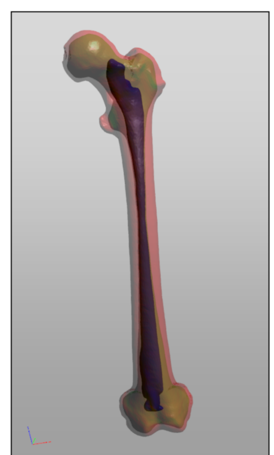
Motivating/Applying Minimum Spanning Tree

- You want to created a connected cable/data network with the least cable/cost/energy possible.



- City planning: Connect several metro stops with least tunneling

- Image Segmentation



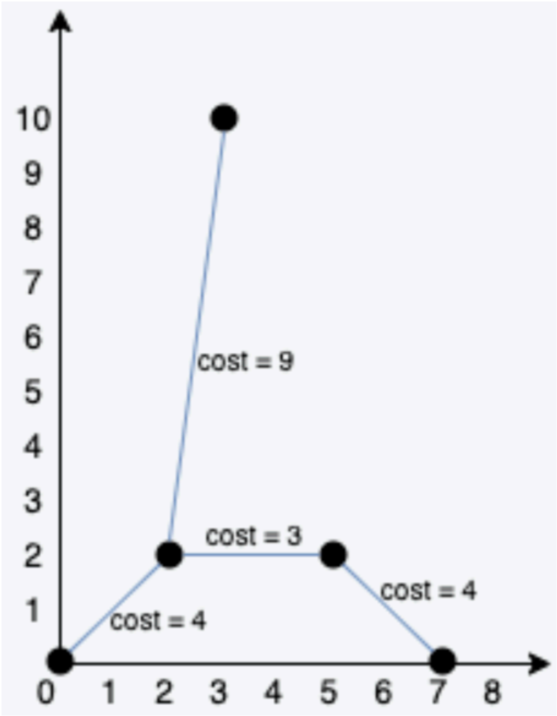
Example MST Problem

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

You are given an array points representing integer coordinates of some points on a 2D-plane, where points[i] = $[x_i, y_i]$.

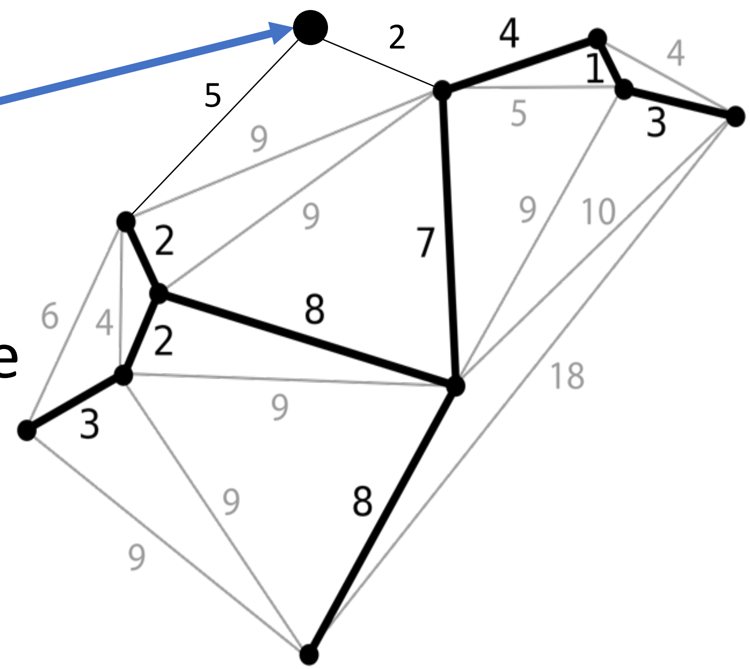
The cost of connecting two points $[x_i, y_i]$ and $[x_j, y_j]$ is the **manhattan distance** between them: $|x_i - x_j| + |y_i - y_j|$, where |val| denotes the absolute value of val.

Return *the minimum cost to make all points connected.* All points are connected if there is **exactly one** simple path between any two points.



Intuitive Inductive Reasoning

- Suppose we have the MST on N-1 vertices.
- We consider the next vertex to get the MST on N vertices.
  - Must use the cost 2 or the cost 5 edge *regardless* of the rest of the MST
  - Might as well use the cheaper cost 2 edge



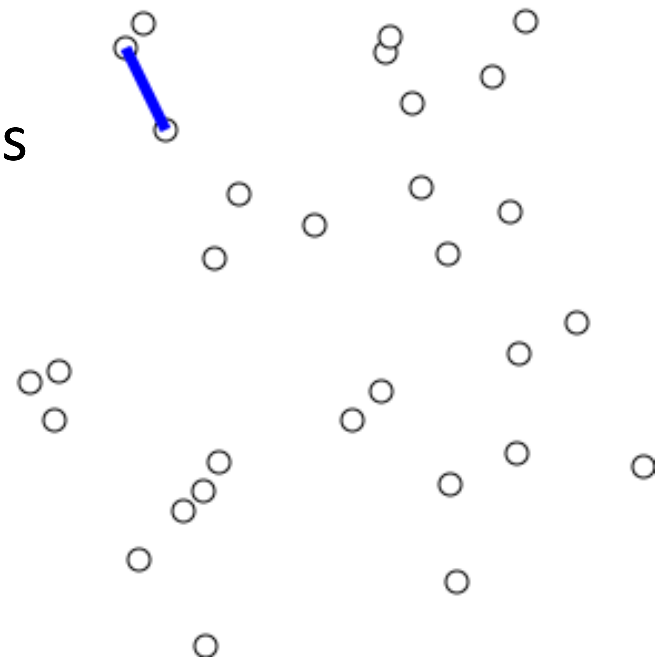
Greedy Optimization: Prim's Algorithm

- Initialize?
  - Choose an arbitrary vertex
- Partial solution?
  - MST connecting *subset* of the vertices.
- Greedy step?
  - Choose the cheapest / least weight edge that connects a new vertex to the partial solution.

Visualizing Prim's Algorithm

In the visualization:

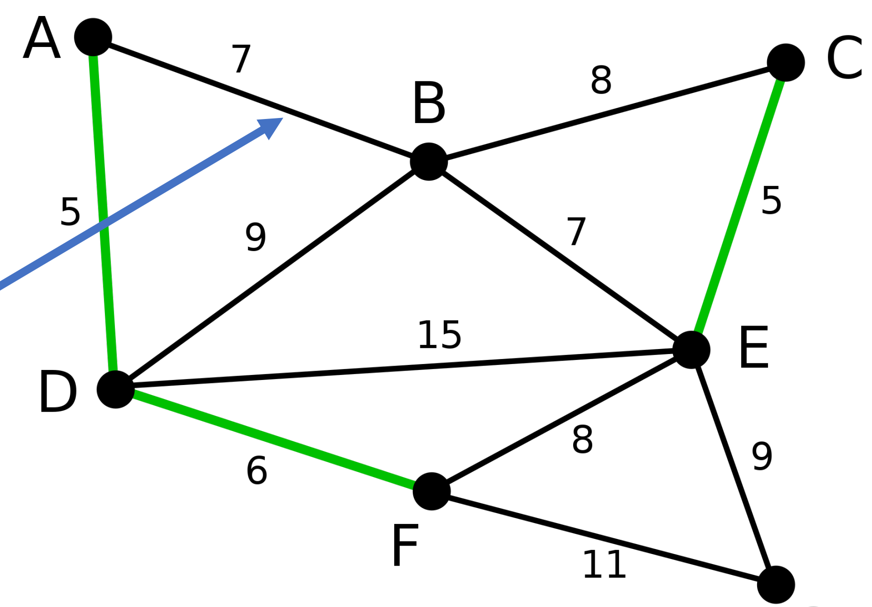
- Edges between all pairs of vertices
- Weights are implicit by distances
- Algorithm greedily grows by choosing closest unconnected vertex



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

More Intuitive Inductive Reasoning

- Suppose we have chosen some spanning trees so far.
- Must connect all of them, might as well choose the *cheapest* edge connecting two trees.



Bold green edges = chosen

Public Domain, https://commons.wikimedia.org/w/index.php?curid=644030

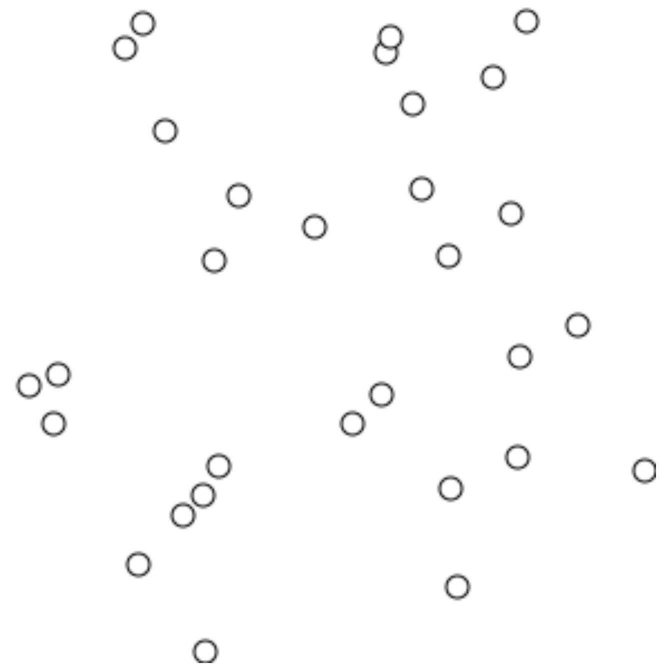
Greedy Optimization Again: Kruskal's Algorithm

- Initialize?
  - All nodes in ***disjoint sets***
- Partial solution?
  - Forest of spanning trees in disjoint sets
- Greedy step?
  - Choose the cheapest / least weight edge that connects two disjoint sets / trees, connect them.

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.



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

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)) where C is time for Union/Find

Disjoint Sets and Union-Find

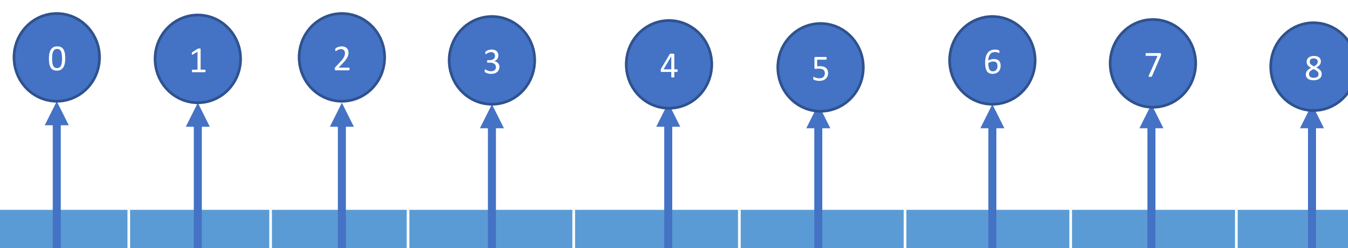
DIYDisjointSets implementation viewable here:
coursework.cs.duke.edu/cs-201-spring-23/diydisjointsets

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 set containing a number
  - Initially self, but changes after unions

Disjoint-Set Forest Implementation

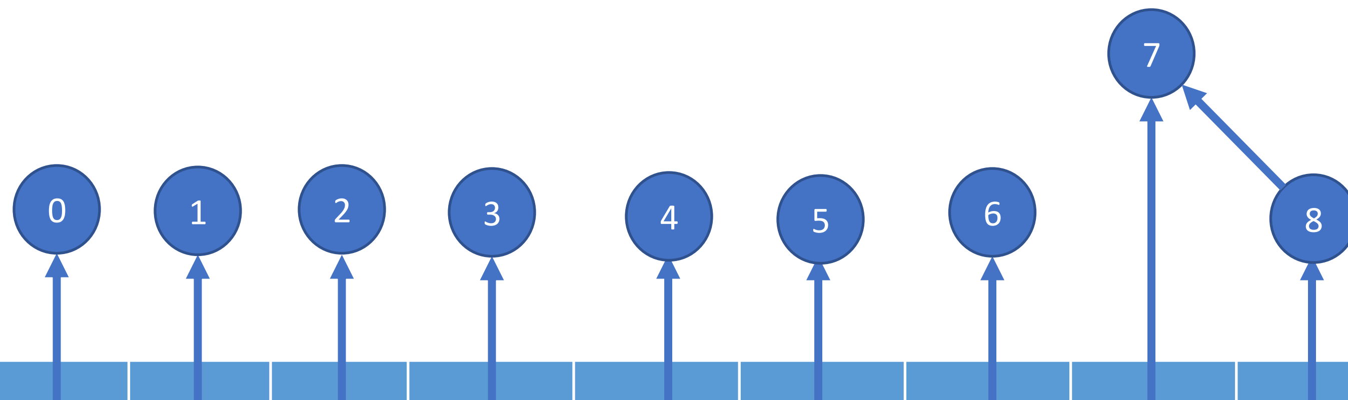
- Each set will be represented by a parent “tree”: Instead of child pointers, nodes have a parent “pointer”.
- Everything starts as its own tree: a single node



parent									
itemID	0	1	2	3	4	5	6	7	8

Disjoint-Set Forest Union

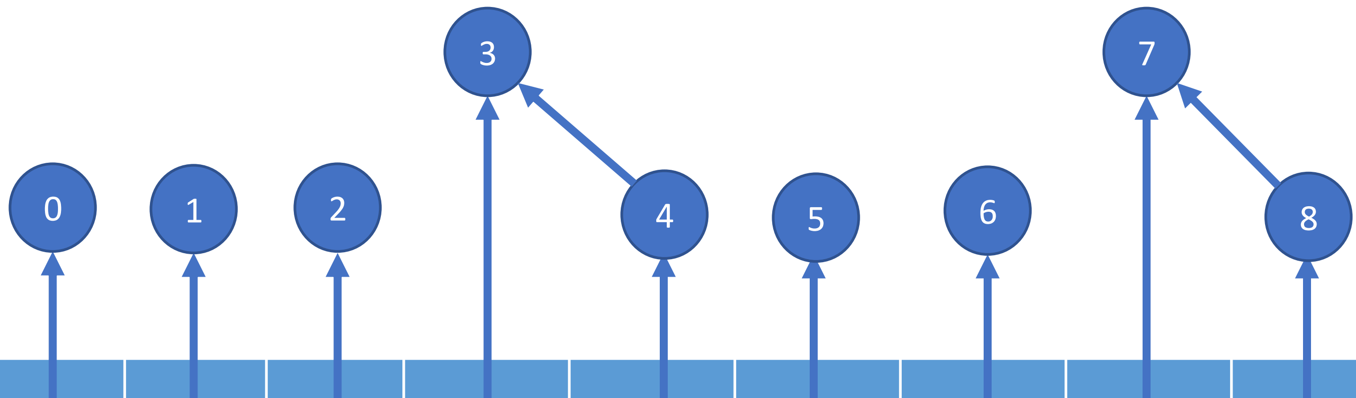
- Union(7,8)
- Just make leaf/root point to parent[7]



parent									
itemID	0	1	2	3	4	5	6	7	8

Disjoint-Set Forest Union

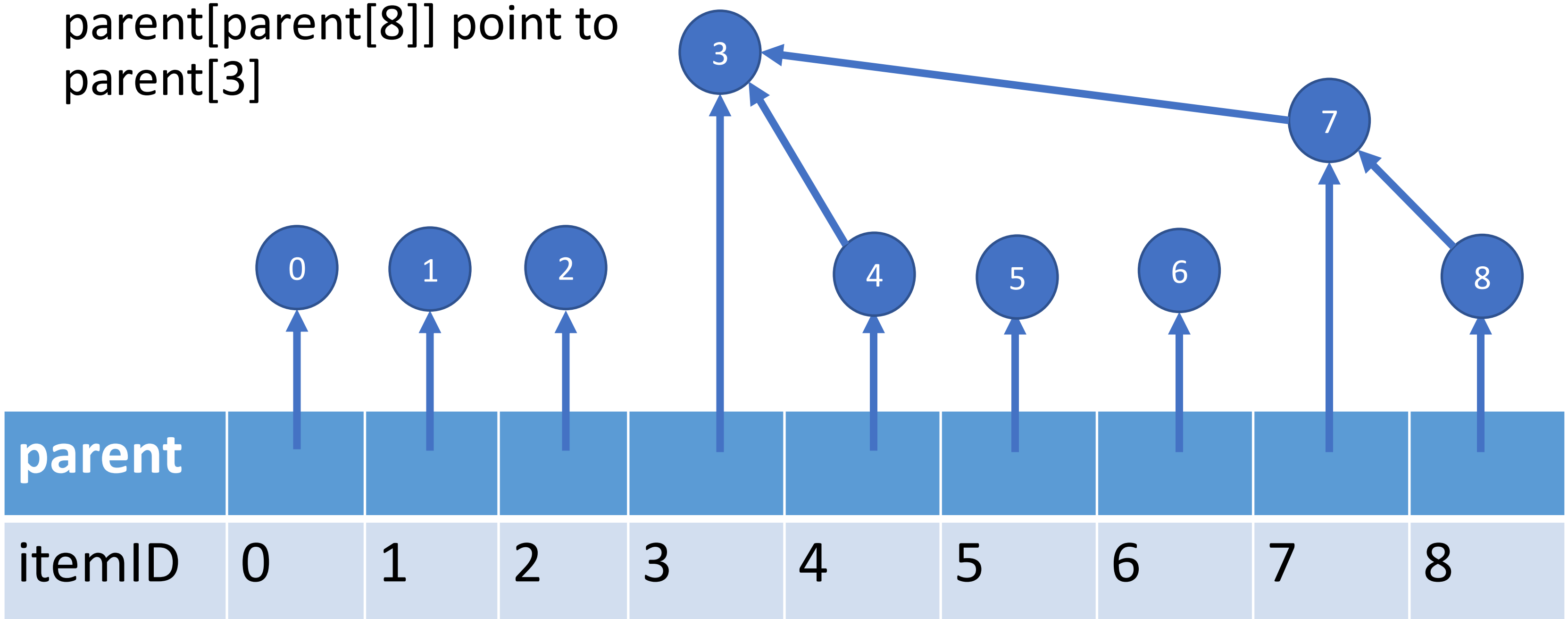
- Union(3,4)
- Just make parent[4] point to parent[3]



parent									
itemID	0	1	2	3	4	5	6	7	8

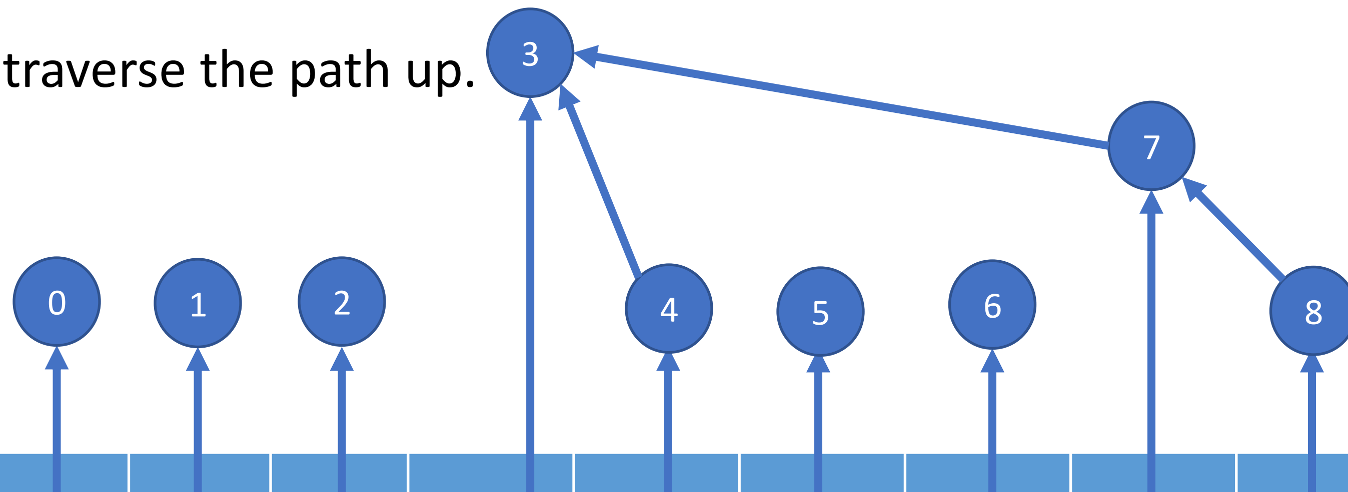
Disjoint-Set Forest Union

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



Disjoint-Set Forest Find

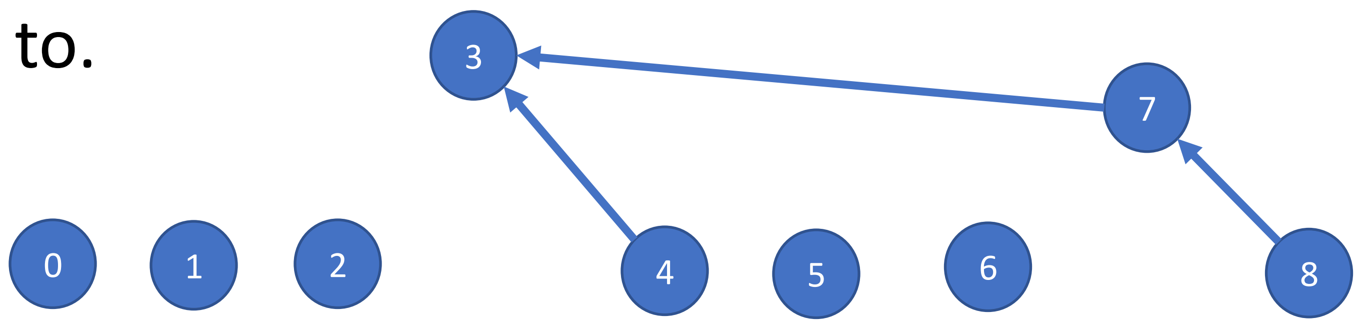
- Find(8)
- Return last ancestor of 8.
- Need to traverse the path up.



parent									
itemID	0	1	2	3	4	5	6	7	8

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.



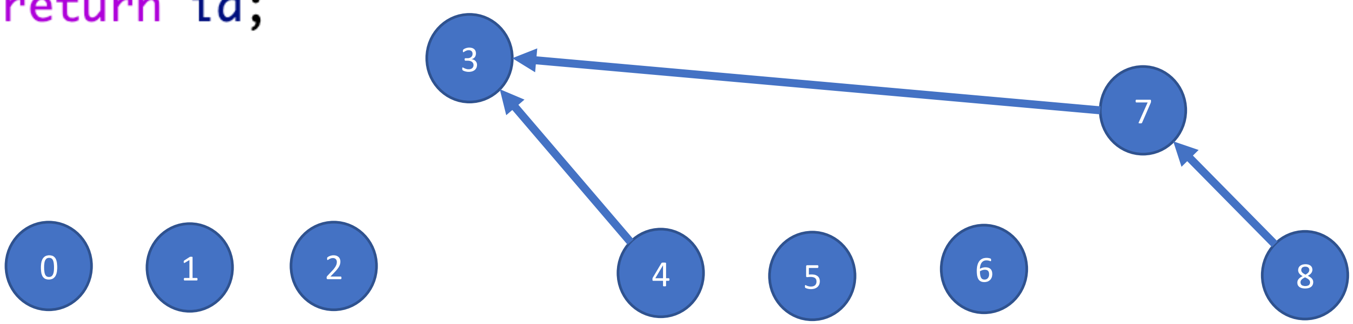
parent	**0**	**1**	**2**	**3**	**3**	**5**	**6**	**3**	**7**
itemID	0	1	2	3	4	5	6	7	8

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"



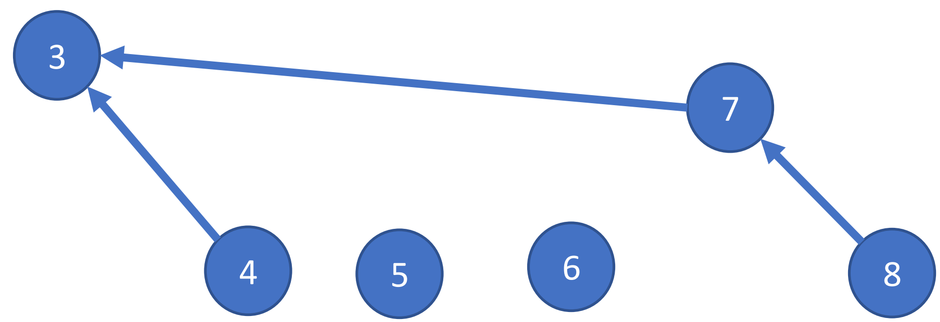
parent	0	1	2	3	3	5	6	3	7
itemID	0	1	2	3	4	5	6	7	8

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

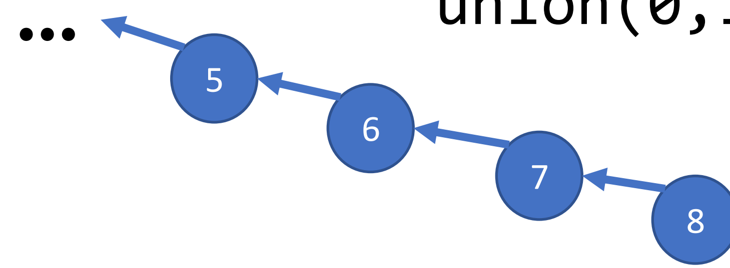
Worst-Case Runtime Complexity?

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

What if we…

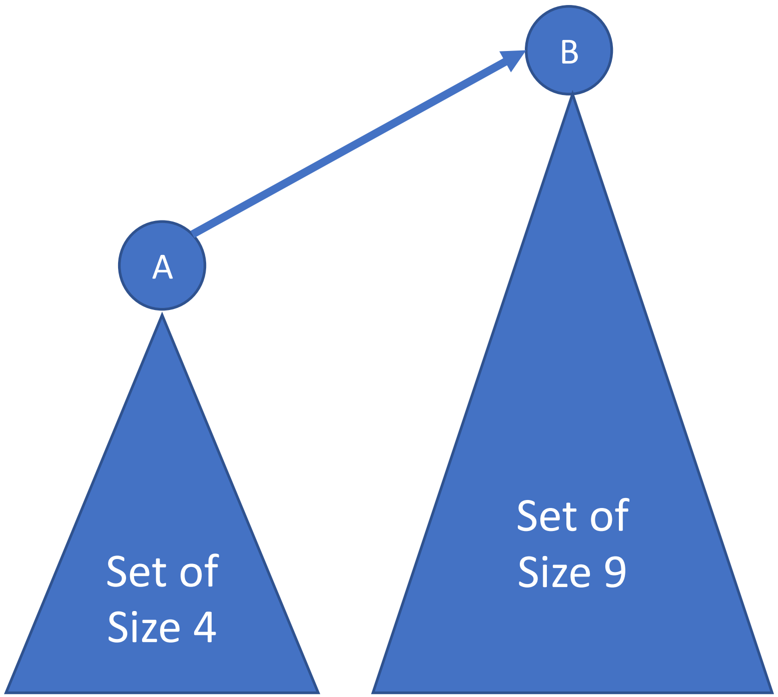
`union(7,8)`
`union(6,7)`
`union(5,6)`
…
`union(0,1)`

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

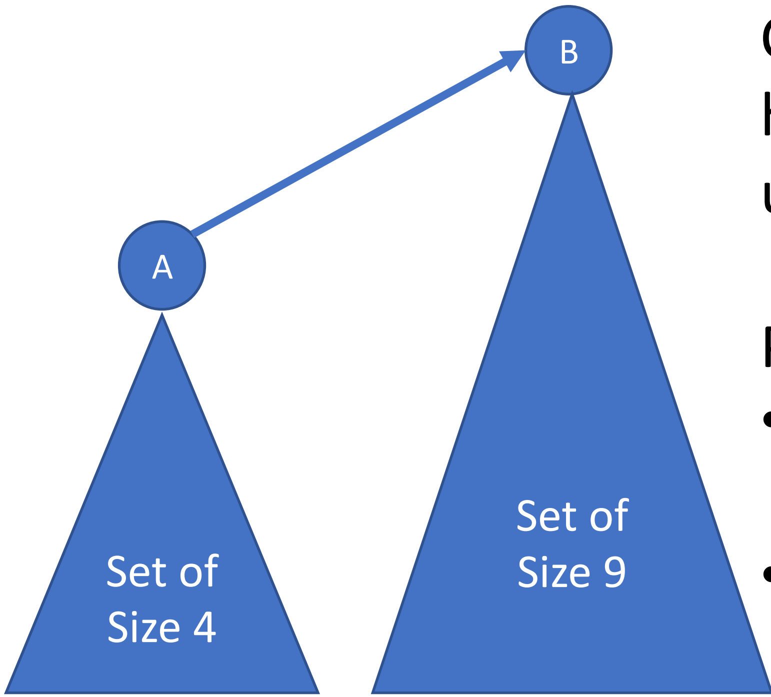
Optimization 1: Union by Size



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



Claim. Each element to root path has length at most O(log(N)) with union by size optimization.

Proof.
- Consider an element a, initially a set of size 1.
- Each time the path length increases, the size of the set must at least double.
- Can happen at most O(log(N)) times with N initial sets.

Optimization 1: Union by Size

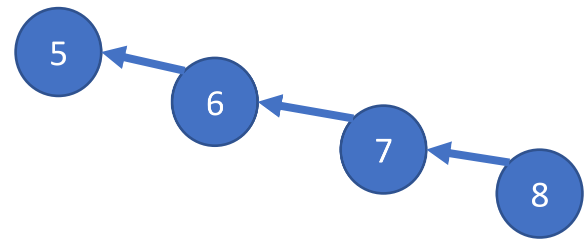
```
public void union(int set1, int set2) {
    int root1 = find(set1);
    int root2 = find(set2);
    if (root1 == root2) { return; }
    if (setSizes[root1] < setSizes[root2]) {
        parent[root1] = root2;
        setSizes[root2] += setSizes[root1];
    }
    else {
        parent[root2] = root1;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

If already in same set, nothing to do.

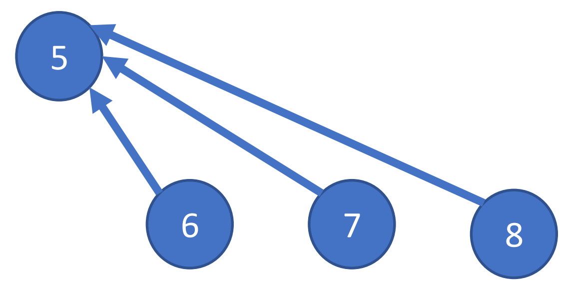
Make the smaller set “point to” the bigger set.

Lazy Path Compression

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



`find(5)`



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

Disjoint Set Forest Path Compression

```
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