LEC 20

CSE 373

Disjoint Sets II

BEFORE WE START

Instructor **Hunter Schafer**

TAs
Ken Aragon
Khushi Chaudhari
Joyce Elauria
Santino Iannone
Leona Kazi
Nathan Lipiarski
Sam Long
Amanda Park
Paul Pham
Mitchell Szeto
Batina Shikhalieva
Ryan Siu
Elena Spasova
Alex Teng
Blarry Wang
Aileen Zeng

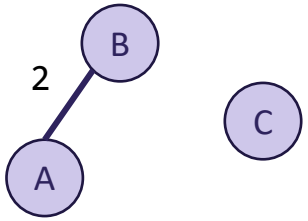
Learning Objectives

After this lecture, you should be able to...

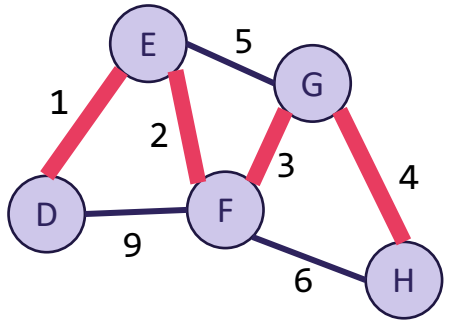
1. Implement WeightedQuickUnion and describe why making the change protects against the worst case find runtime
2. Implement path compression and argue why it improves runtimes, despite not following an invariant
3. Describe what contributes to the runtime of Prim's and Kruskal's, and compare/contrast the two algorithms
4. Implement WeightedQuickUnion using arrays and describe the benefits of doing so

Review MSTs

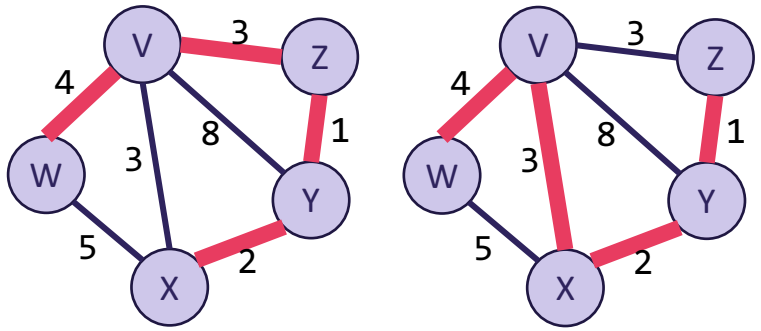
- Minimum (minimizes sum of edge weights) Spanning (connects all vertices) Tree (exactly one path between any two nodes)
  - Minimizing sum of edge weights is NOT the same as minimizing shortest paths!
- If a graph is **connected**, has at least one MST
- If a graph is connected and has all unique edges, has exactly one MST
- If a graph is connected and has duplicate edges, it *may* have multiple valid MSTs
  - Which one we pick is down to arbitrary order we visit duplicates: Prim's & Kruskal's could potentially differ, but both MSTs would still be valid.



No MSTs

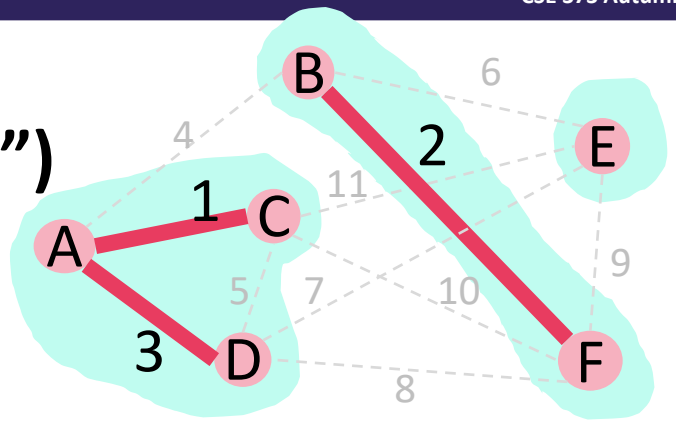


Exactly 1 MST



Multiple Valid MSTs

Review Disjoint Sets ADT (aka "Union-Find")



- Kruskal's MST algorithm goes edge-by-edge, but it needs a Disjoint Sets ADT under the hood to check whether vertices are already connected!
  - Conceptually, a single instance of this ADT contains a "family" of sets that are disjoint (no element belongs to multiple sets)

```
kruskalMST(G graph)
  DisjointSets<V> msts; Set finalMST;
  initialize msts with each vertex as single-element MST
  sort all edges by weight (smallest to largest)

  for each edge (u,v) in ascending order:
    uMST = msts.find(u)
    vMST = msts.find(v)
    if (uMST != vMST):
      finalMST.add(edge (u, v))
      msts.union(uMST, vMST)
```

DISJOINT SETS ADT

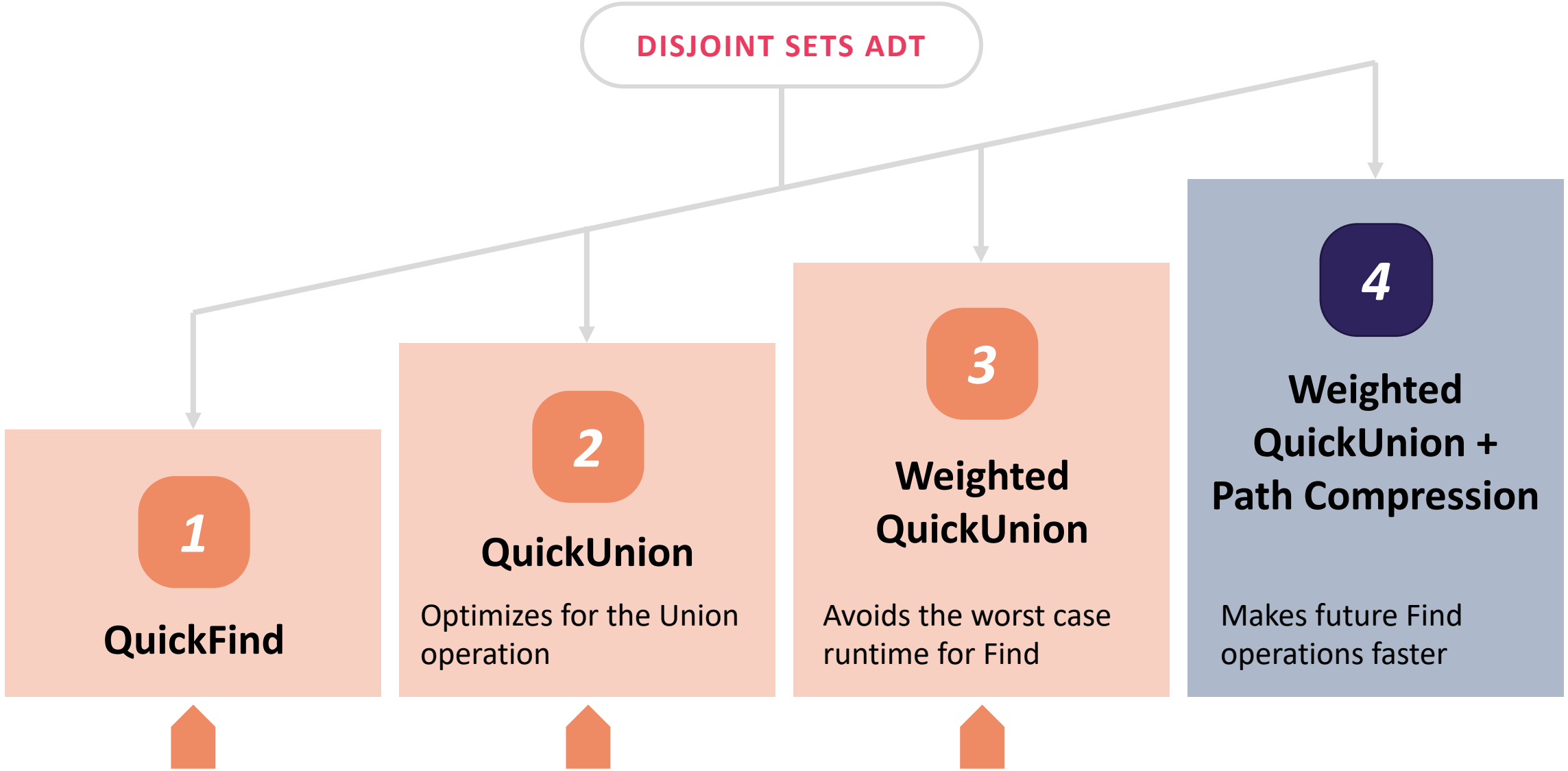
State

Family of Sets
- disjoint: no shared elements
- each set has a representative (either a member or a unique ID)

Behavior

makeSet(value) – new set with value as only member (and representative)
find(value) – return representative of the set containing value
union(x, y) – combine sets containing x and y into one set with all elements, choose single new representative

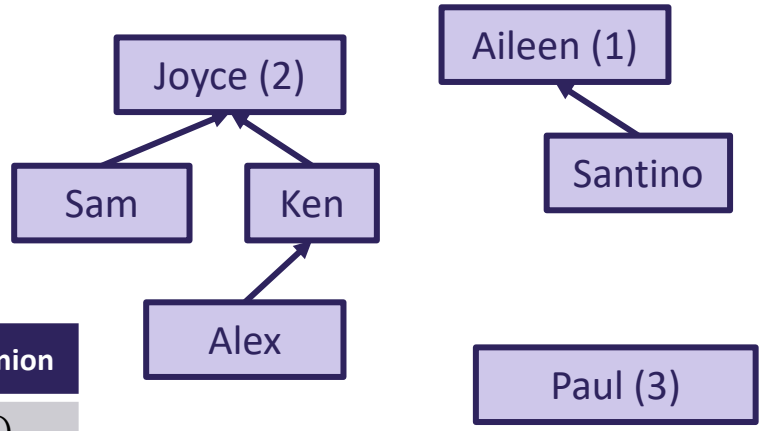
Lecture Outline



Review QuickFind vs. QuickUnion

map from value to representative ID

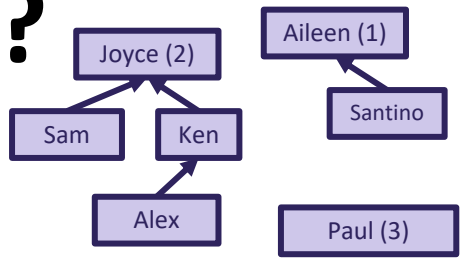
trees of values with representative ID at each root



Could also use one element from each set (e.g. the root) as its representative: only uniqueness matters

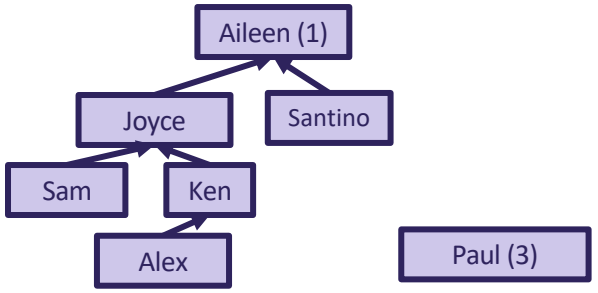
	(Baseline)	QuickFind	QuickUnion
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
`union(x, y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

Review QuickUnion: Why Use Both Roots?



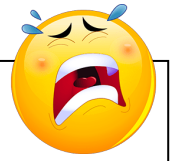
Example: result of `union(Ken, Santino)` on these Disjoint Sets given three possible implementations:

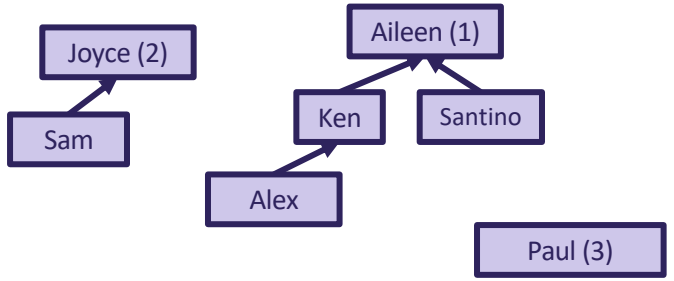
```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  set rootA to point to rootB
```



Correct: Everything in Ken's set now connected to everything in Santino's set!

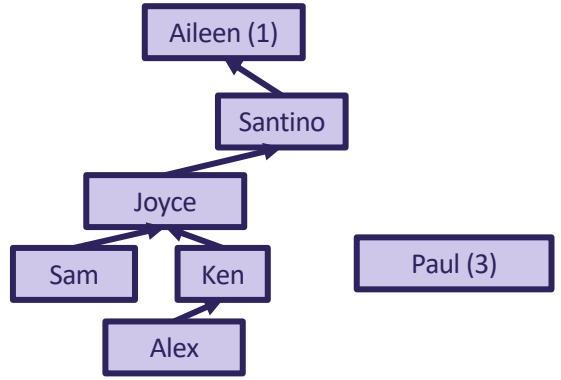
```
union(A, B):
  rootB = find(B)
  set A to point to rootB
```





Incorrect: Ken and Joyce were connected before; the union operation shouldn't remove connections.

```
union(A, B):
  rootA = find(A)
  set rootA to point to B
```



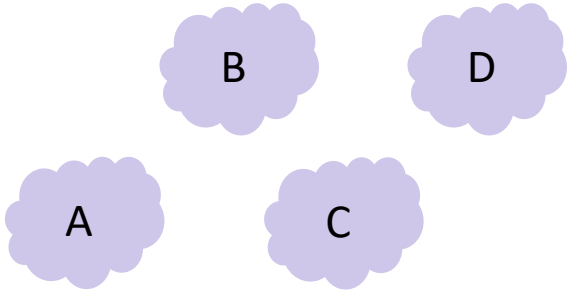
Inefficient: Technically correct, but increases height of the up-tree so makes

Review WeightedQuickUnion

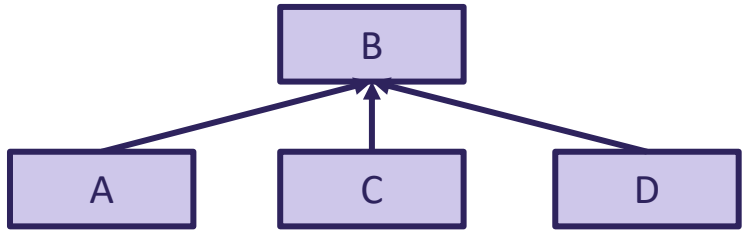
- Goal: Always pick the smaller tree to go under the larger tree
- Implementation: Store the number of nodes (or "weight") of each tree in the root
  - Constant-time lookup instead of having to traverse the entire tree to count

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  put lighter root under heavier root
```

```
union(A, B)
union(B, C)
union(C, D)
find(A)
```



Now what happens?



Perfect! Best runtime we can get.

Review WeightedQuickUnion: Performance

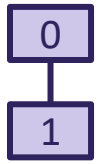
- Consider the worst case where the tree height grows as fast as possible

N	H
1	0

Review WeightedQuickUnion: Performance

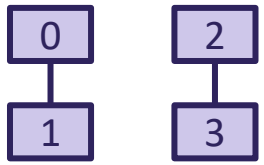
- Consider the worst case where the tree height grows as fast as possible

N	H
1	0
2	1



Review WeightedQuickUnion: Performance

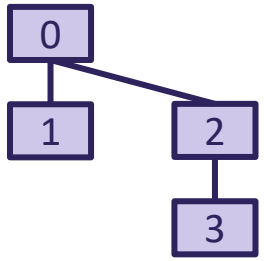
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	?

Review WeightedQuickUnion: Performance

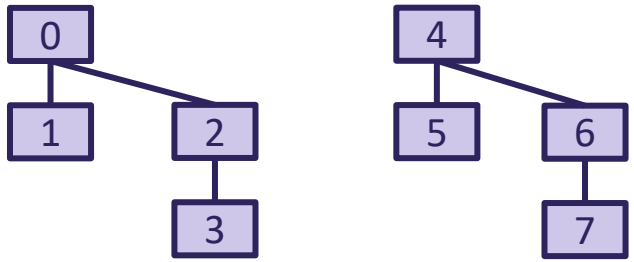
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	2

Review WeightedQuickUnion: Performance

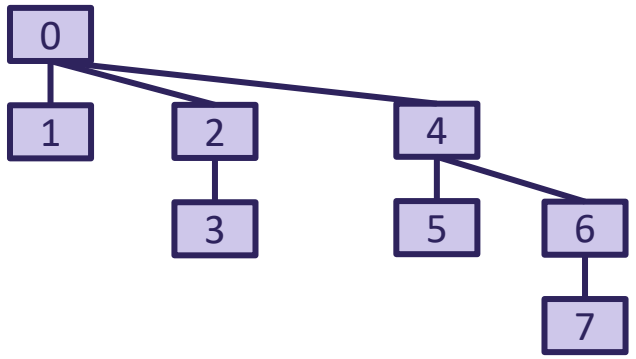
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	2
8	?

Review WeightedQuickUnion: Performance

- Consider the worst case where the tree height grows as fast as possible

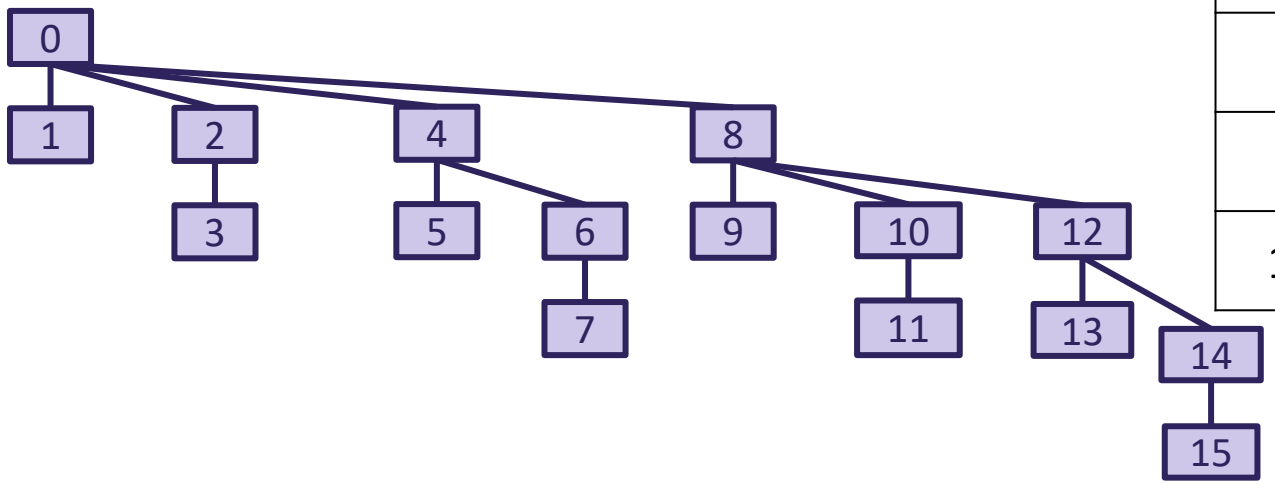


N	H
1	0
2	1
4	2
8	3

Review WeightedQuickUnion: Performance

- Consider the worst case where the tree height grows as fast as possible
- Worst case tree height is Θ(log N)

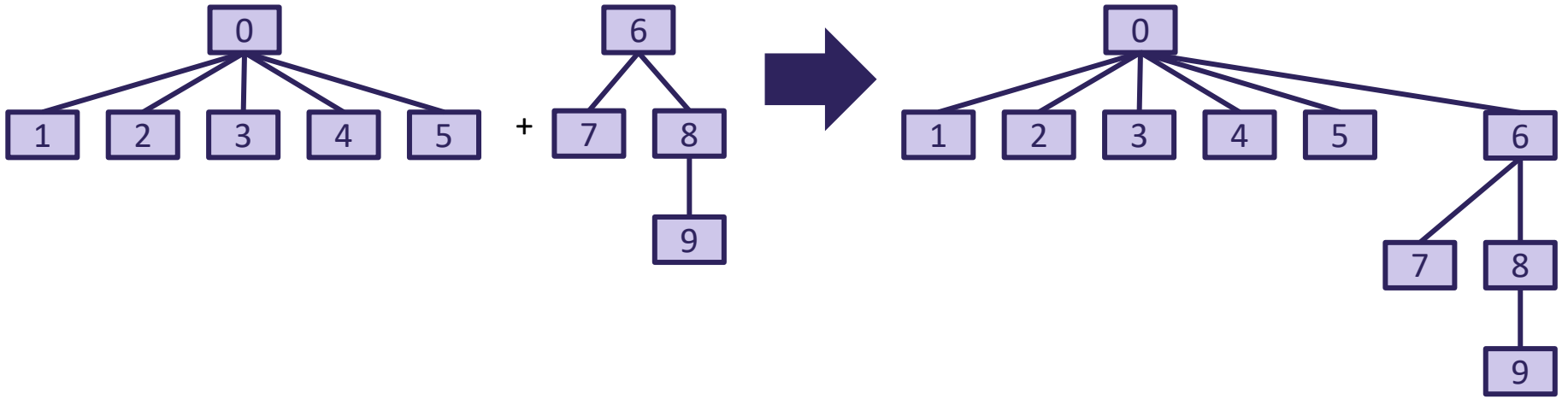
N	H
1	0
2	1
4	2
8	3
16	4



Review Why Weights Instead of Heights?

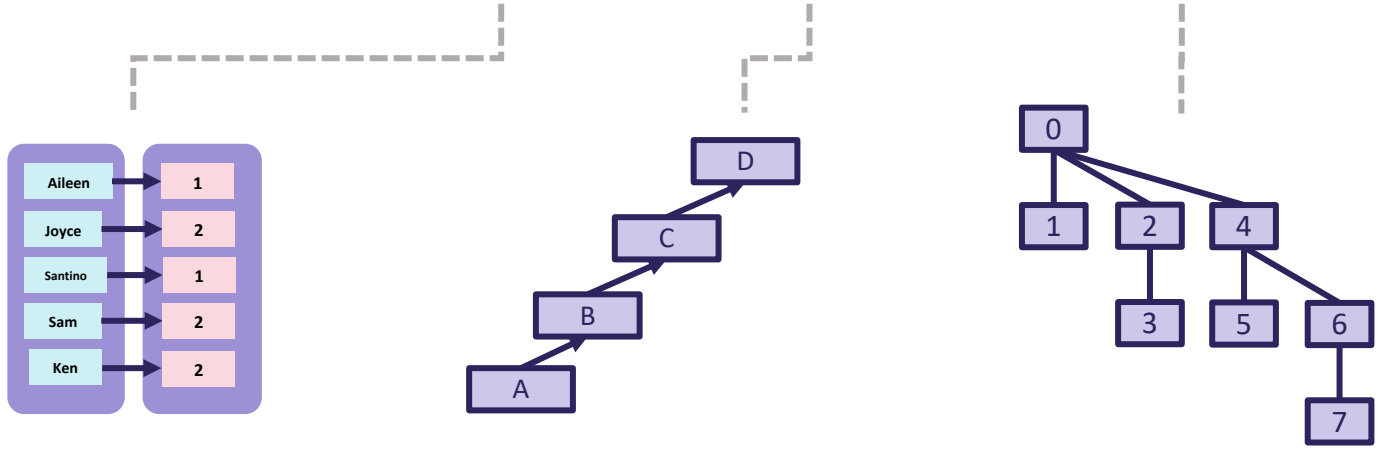
- We used the number of items in a tree to decide upon the root

- Why not use the height of the tree?
  - HeightedQuickUnion's runtime is asymptotically the same: Θ(log(n))
  - It's easier to track weights than heights, even though WeightedQuickUnion can lead to some suboptimal structures like this one:



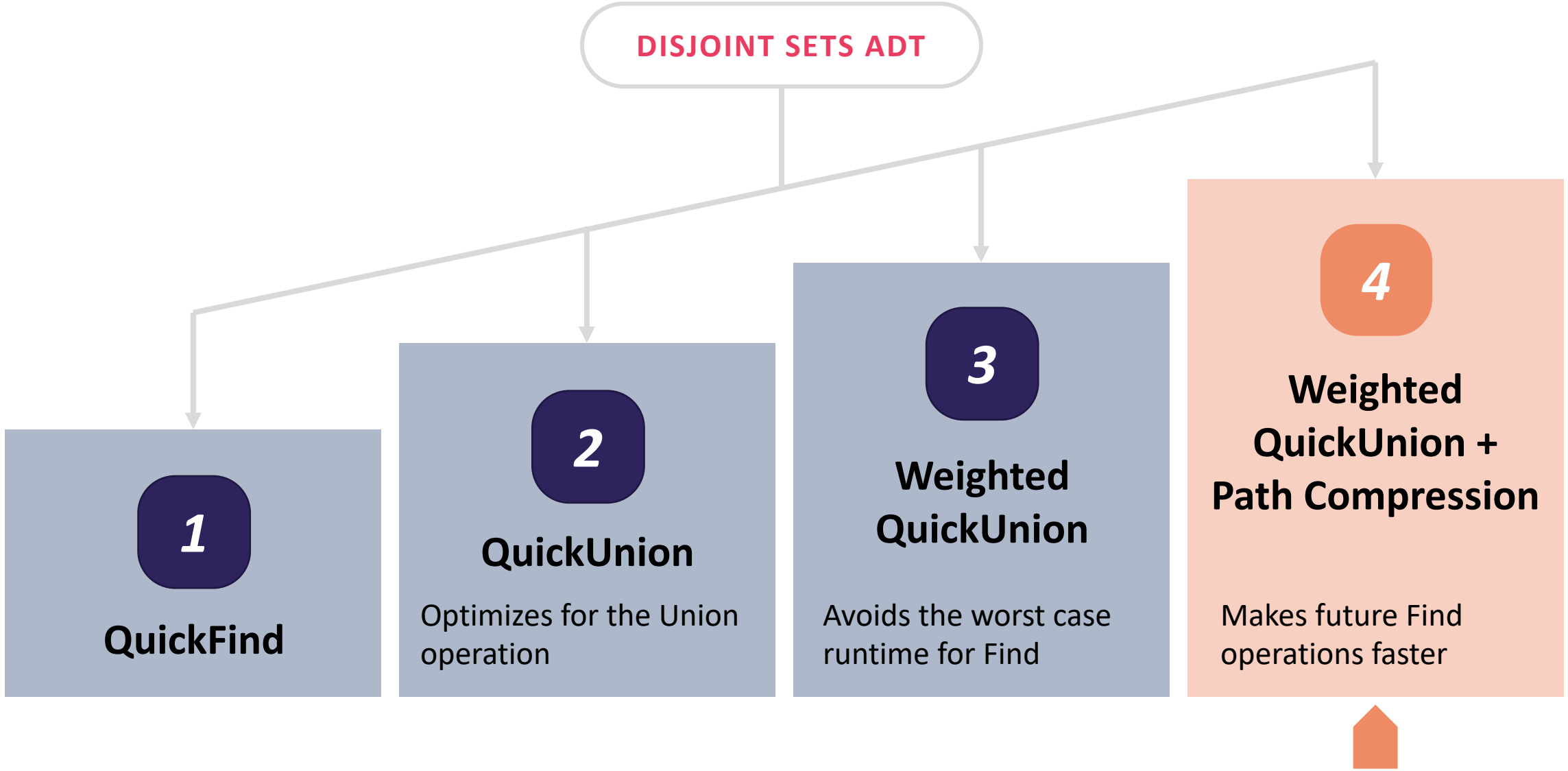
Review WeightedQuickUnion Runtime

	(Baseline)	QuickFind	QuickUnion	WeightedQuickUnion
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
`union(x, y)` assuming root args	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
`union(x, y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$



- This is pretty good! But there's one final optimization we can make: **path compression**

Lecture Outline



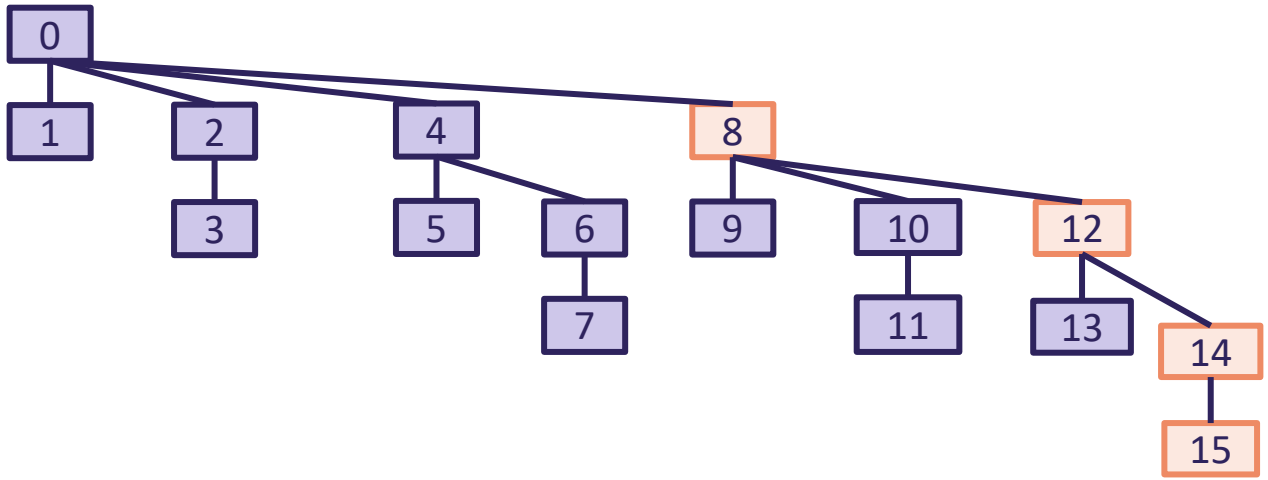
Modifying Data Structures for Future Gains

- Thus far, the modifications we've studied are designed to *preserve invariants*
  - E.g. Performing rotations to preserve the AVL invariant
  - We rely on those invariants always being true so every call is fast

- Path compression is entirely different: we are modifying the tree structure to *improve future performance*
  - Not adhering to a specific invariant
  - The first call may be slow, but will optimize so future calls can be fast

Path Compression: Idea

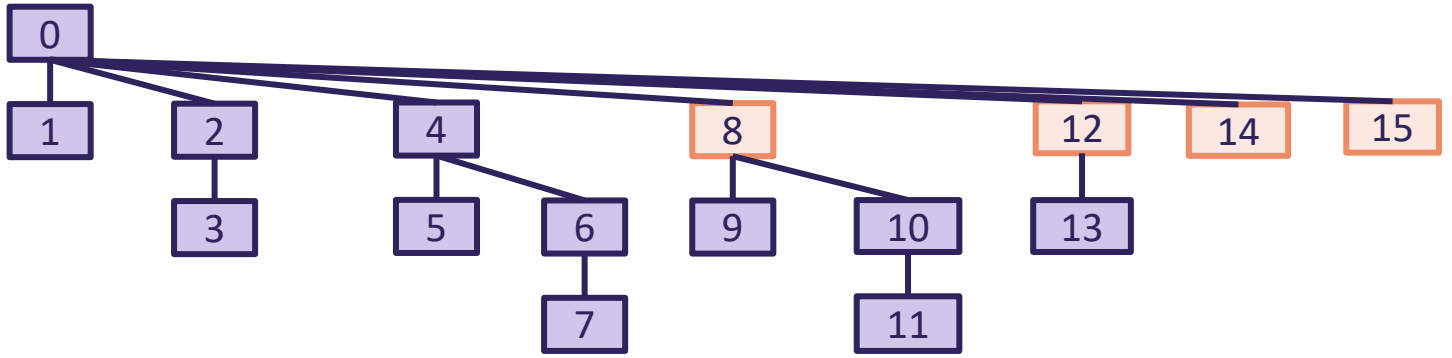
- This is the worst-case topology if we use WeightedQuickUnion



- Idea: When we do find(15), move all **visited nodes** under the root
  - Additional cost is insignificant (we already have to visit those nodes, just constant time work to point to root too)

Path Compression: Idea

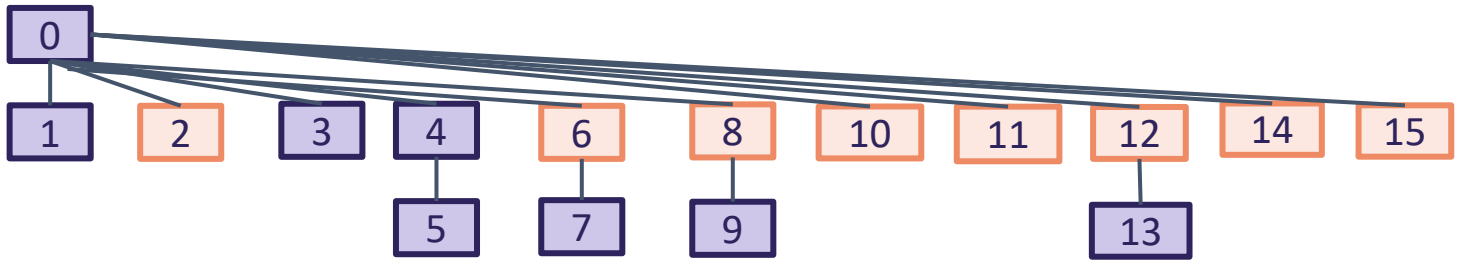
- This is the worst-case topology if we use WeightedQuickUnion



- Idea: When we do find(15), move all **visited nodes** under the root
  - Additional cost is insignificant (we already have to visit those nodes, just constant time work to point to root too)
- Perform Path Compression on every find(), so future calls to find() are faster!

Path Compression: Details and Runtime

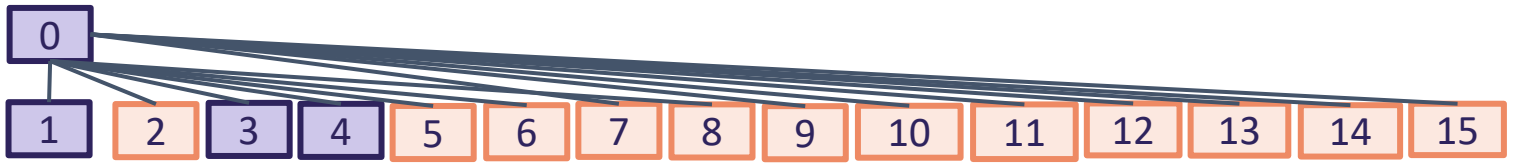
- Run path compression on every find()!
  - Including the find()s that are invoked as part of a union()



- Understanding the performance of M>1 operations requires **amortized analysis**
  - Effectively averaging out rare events over many common ones
  - Typically used for "In-Practice" case
    - E.g. when we assume an array doesn't resize "in practice", we can do that because the rare resizing calls are *amortized* over many faster calls
  - In 373 we don't go in-depth on amortized analysis

Path Compression: Runtime

- M find()s on WeightedQuickUnion requires takes Θ(M log N)



- … but M find()s on WeightedQuickUnionWithPathCompression takes O(M log*N)!
  - log*n is the "iterated log": the number of times you need to apply log to n before it's <= 1
  - Note: log* is a loose bound

Path Compression: Runtime

- Path compression results in find()s and union()s that are very very close to (amortized) constant time
  - log* is less than 5 for any realistic input
  - If M find()s/union()s on N nodes is O(M log*N) and log*N ≈ 5, then find()/union()s amortizes to O(1)! 🤯

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

2^{16}

Number of atoms in the known universe is 2^{256}ish

WQU + Path Compression Runtime

In-Practice Runtimes:

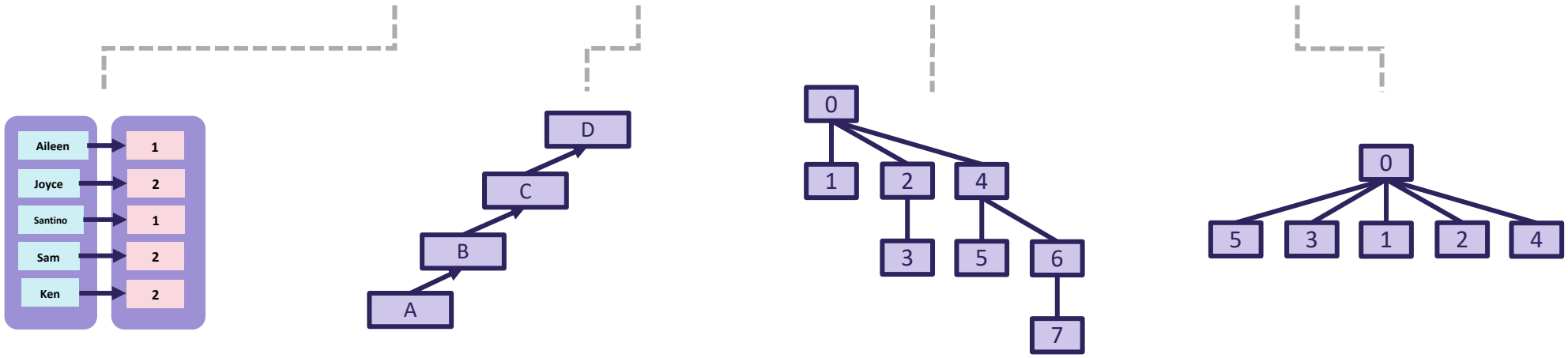
	(Baseline)	QuickFind	QuickUnion	WeightedQuickUnion	WQU + Path Compression
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$
`union(x, y)` assuming root args	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`union(x, y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$

- And if log* n <= 5 for any reasonable input…
  - We've just witnessed an incredible feat of data structure engineering: every operation is constant!?*
  - *Caveat: *amortized* constant, in the "in-practice" case; still logarithmic in the worst case!

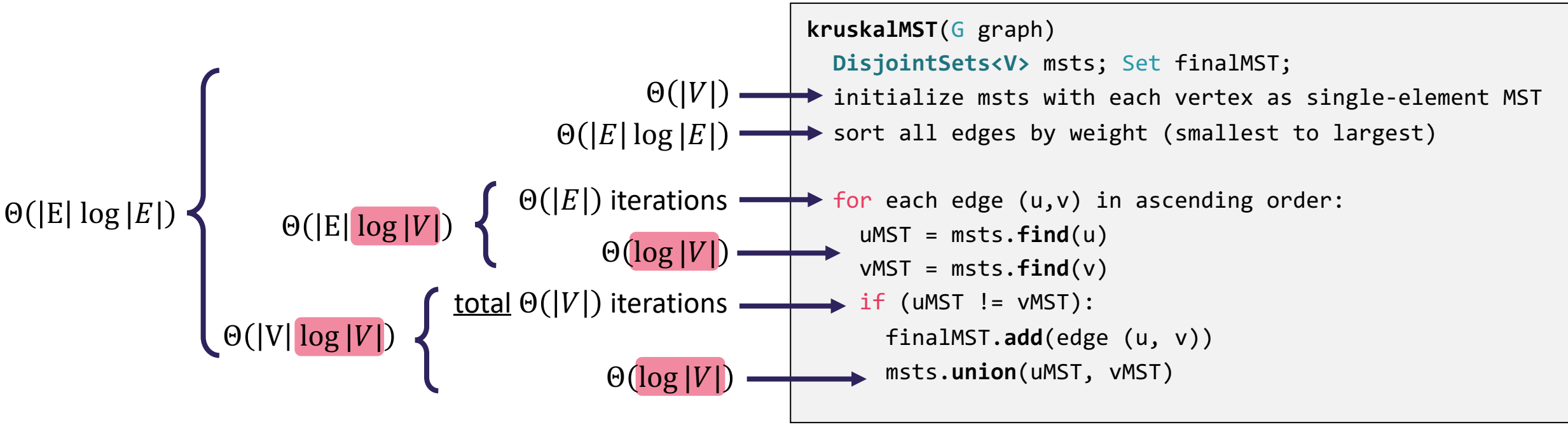
Disjoint Sets Implementation

In-Practice Runtimes:

	(Baseline)	QuickFind	QuickUnion	WeightedQuickUnion	WQU + Path Compression
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$
`union(x, y)` assuming root args	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`union(x, y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$



Kruskal's Runtime



- find and union are log|V| in worst case, but amortized constant "in practice"
- Either way, dominated by time to sort the edges ☹
  - For an MST to exist, E can't be smaller than V, so assume it dominates
  - Note: some people write |E|log|V|, which is the same (within a constant factor)

TRAVERSAL
(COMMONLY SHORTEST PATHS)

MINIMUM SPANNING TREES

Dijkstra's

$O(|V| \log |V| + |E| \log |V|)$

- Goes in order of shortest-path-so-far
- Choose when:
  - Want shortest path on *weighted* graph

Prim's

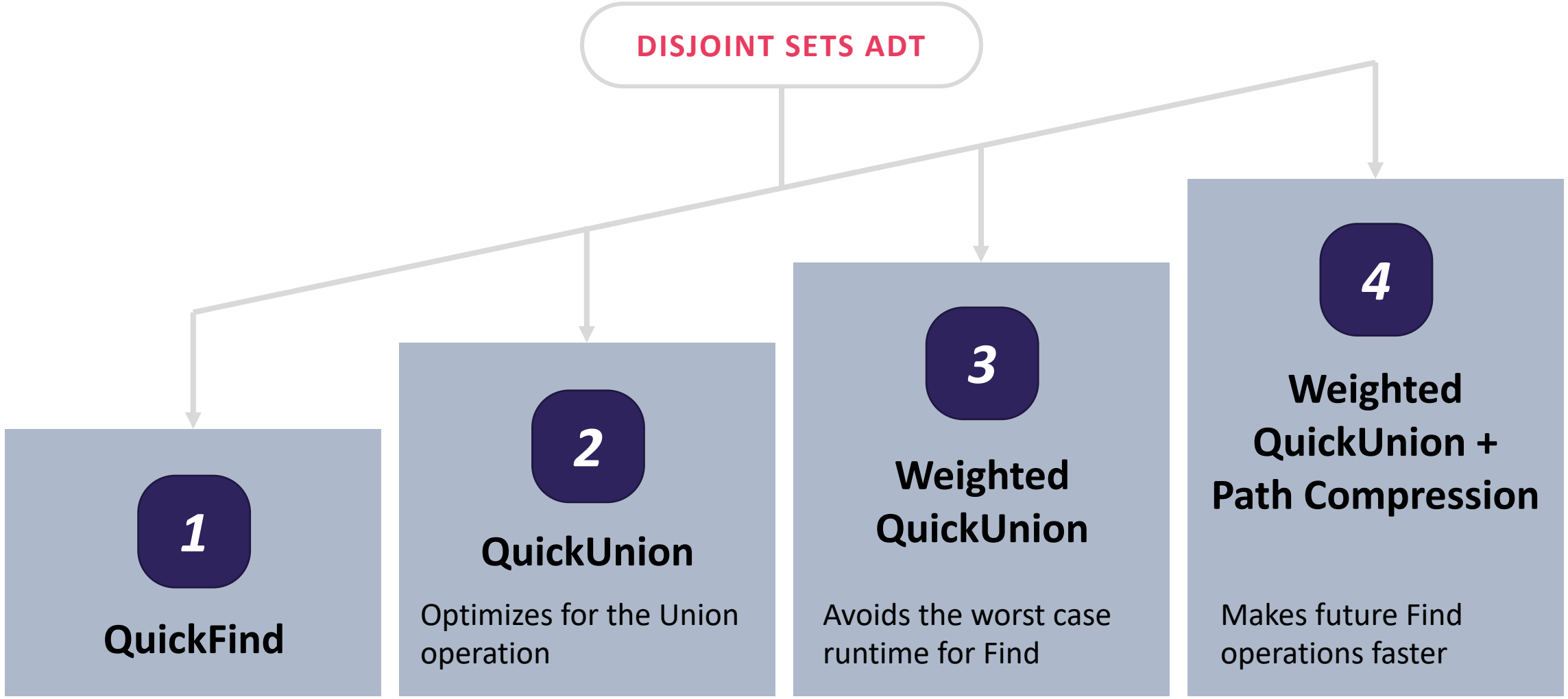
$O(|E| \log |V|)$

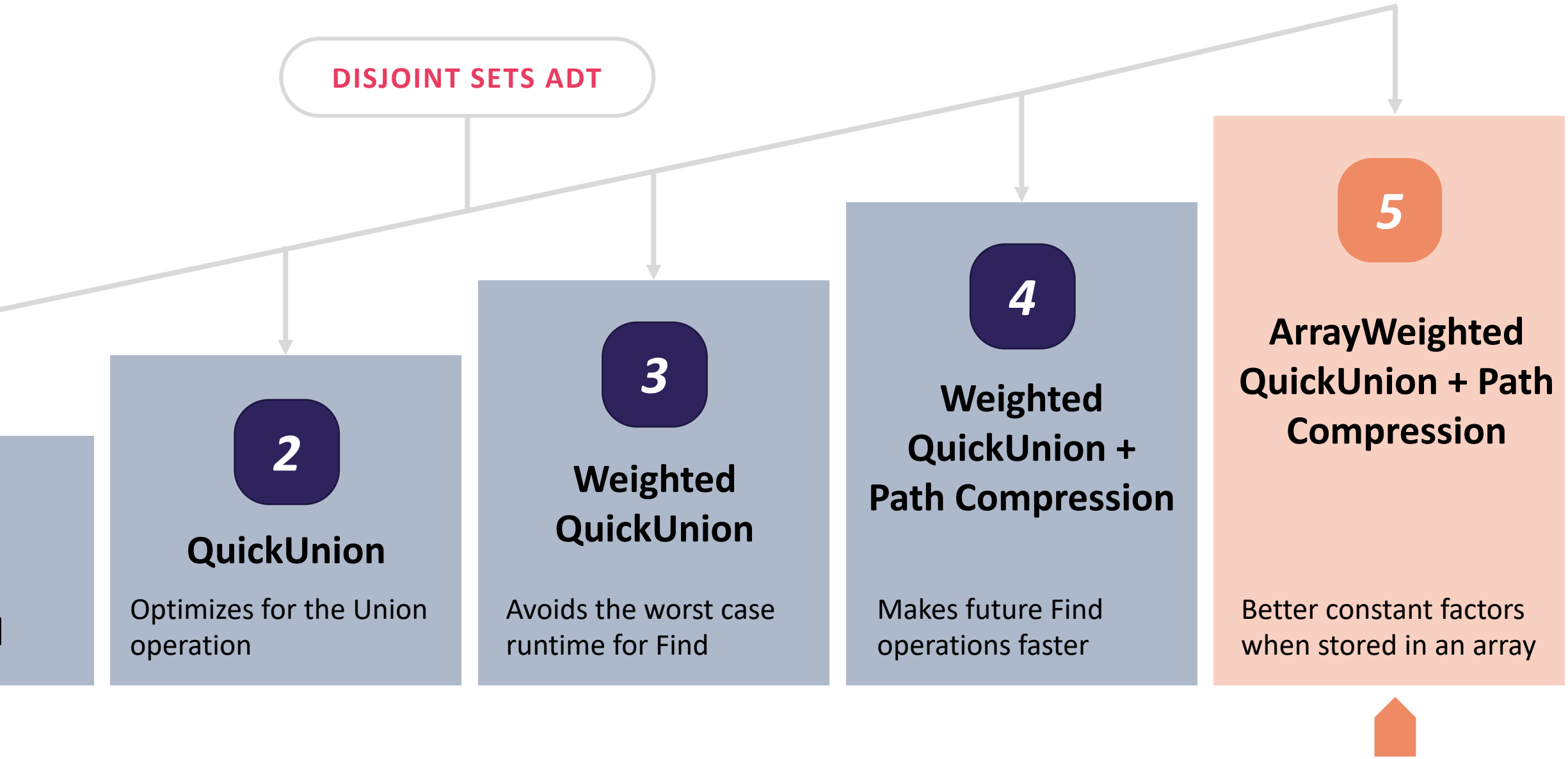
- Goes vertex-by-vertex
- Choose when:
  - Want MST
  - Graph is dense (more edges)

Kruskal's

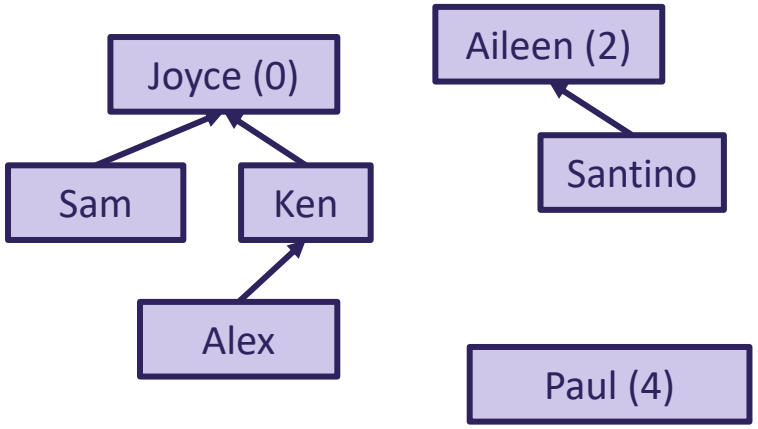
$O(|E| \log |V|)$

- Goes edge-by-edge
- Choose when:
  - Want MST
  - Graph is sparse (fewer edges)
  - Edges already sorted


DISJOINT SETS ADT
1
QuickFind
2
QuickUnion
Optimizes for the Union operation
3
Weighted QuickUnion
Avoids the worst case runtime for Find
4
Weighted QuickUnion + Path Compression
Makes future Find operations faster


DISJOINT SETS ADT
2
QuickUnion
Optimizes for the Union operation
3
Weighted QuickUnion
Avoids the worst case runtime for Find
4
Weighted QuickUnion + Path Compression
Makes future Find operations faster
5
ArrayWeighted QuickUnion + Path Compression
Better constant factors when stored in an array

Using Arrays for Up-Trees



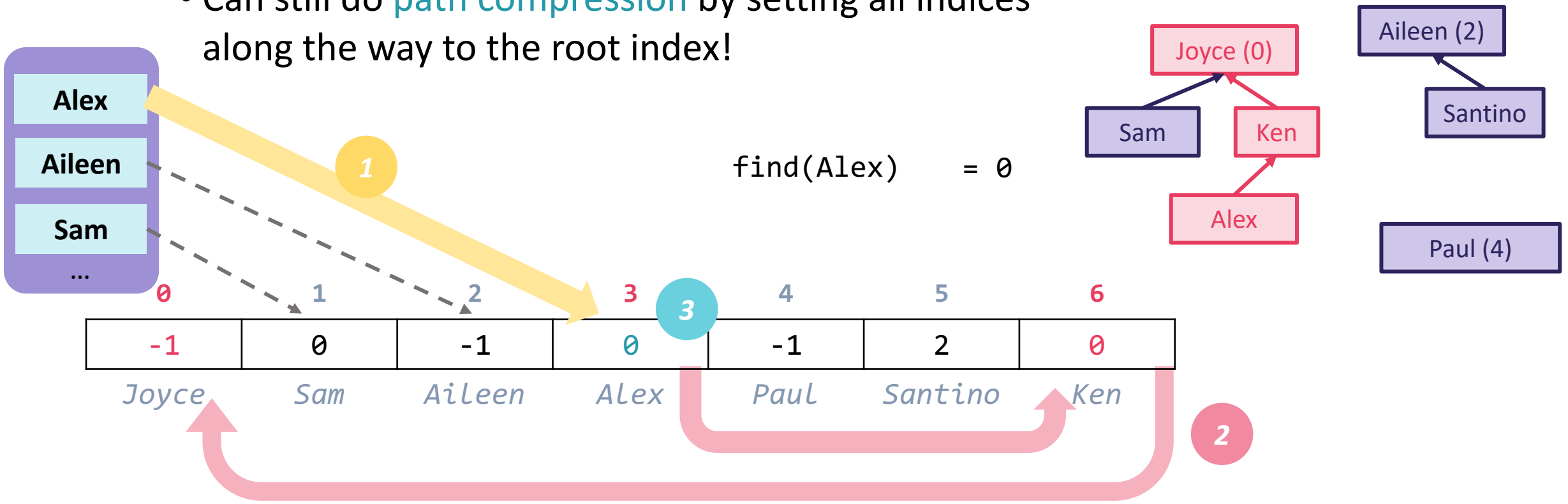
- Since every node can have at most one parent, what if we use an array to store the parent relationships?
- Proposal: each node corresponds to an index, where we store the index of the parent (or –1 for roots). Use the root index as the representative ID!
- Just like with heaps, tree picture still conceptually correct, but exists in our minds!

0	1	2	3	4	5	6
-1	0	-1	6	-1	2	0
Joyce	*Sam*	*Aileen*	*Alex*	*Paul*	*Santino*	*Ken*

Using Arrays: Find

- Initial jump to element still done with extra Map
- But traversing up the tree can be done purely within the array!
  - Can still do path compression by setting all indices along the way to the root index!

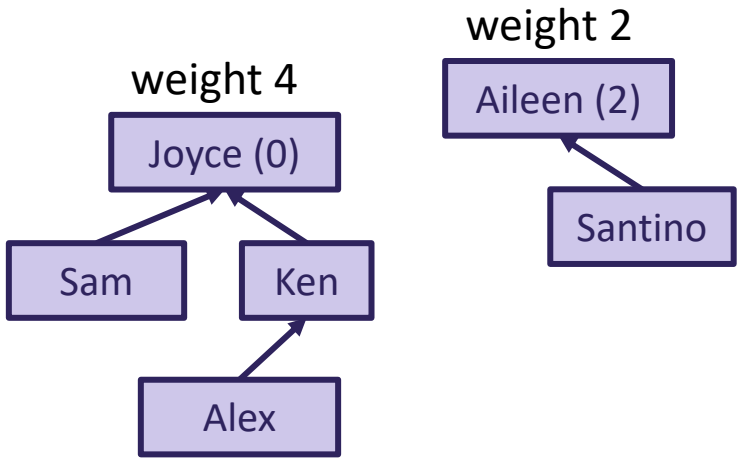
```
find(A):
  index = jump to A node's index
  while array[index] > 0:
    index = array[index]
  path compression
  return index
```



Using Arrays: Union

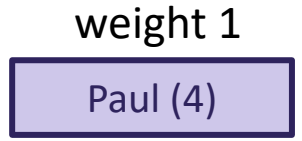
- For WeightedQuickUnion, we need to store the number of nodes in each tree (the weight)
- Instead of just storing -1 to indicate a root, we can store -1 * weight!

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  use -1 * array[rootA] and -1 *
       array[rootB] to determine weights
  put lighter root under heavier root
```



union(Ken, Santino)

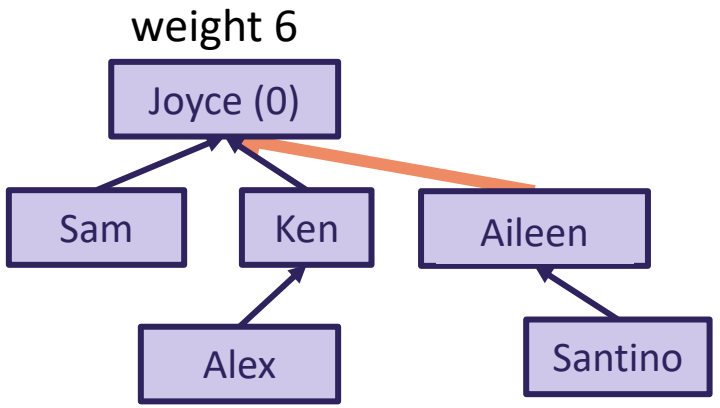
0	1	2	3	4	5	6
-4	0	-2	6	-1	2	0
Joyce	*Sam*	*Aileen*	*Alex*	*Paul*	*Santino*	*Ken*



Using Arrays: Union

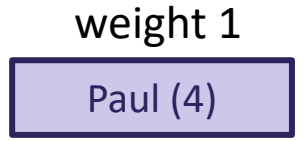
- For WeightedQuickUnion, we need to store the number of nodes in each tree (the weight)
- Instead of just storing -1 to indicate a root, we can store -1 * weight!

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  use -1 * array[rootA] and -1 *
       array[rootB] to determine weights
  put lighter root under heavier root
```



union(Ken, Santino)

0	1	2	3	4	5	6
-6	0	0	6	-1	2	0
Joyce	*Sam*	*Aileen*	*Alex*	*Paul*	*Santino*	*Ken*



Using Arrays for WQU+PC

- Same asymptotic runtime as using tree nodes, but check out all these other benefits:
  - More compact in memory
  - Better spatial locality, leading to better constant factors from cache usage
  - Simplify the implementation!

	(Baseline)	QuickFind	QuickUnion	WeightedQuickUnion	WQU + Path Compression	ArrayWQU+PC
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$	$O(\log^* n)$
`union(x, y)` `assuming root args`	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`union(x, y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$	$O(\log^* n)$