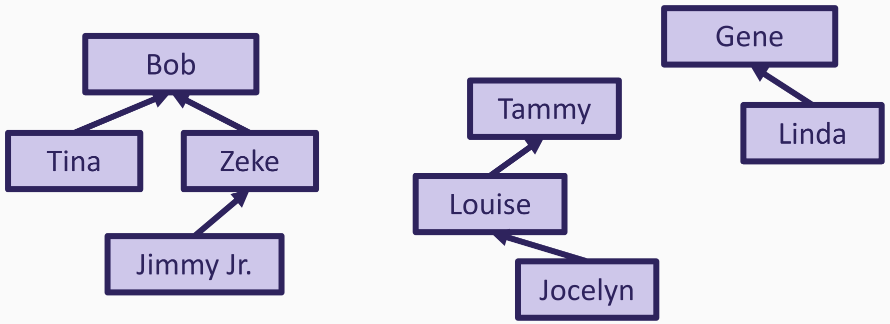
LEC 19

CSE 373

Disjoint Sets II

BEFORE WE START

Consider the following WeightedQuickUnion structure. What's the result of calling `union(Louise, Linda)` and then `union(Louise, Tina)`?



pollev.com/uwcse373

Instructor: Aaron Johnston

TAs: Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

Announcements

- EX3 due **TONIGHT** 11:59pm PDT
- P4 (Mazes) has been released
  - Please start early! Truly 2 weeks worth of work, and some of the coolest work too so we don't want anyone to miss out!
  - Not sure how to start writing code? That's okay! Reading and integrating with substantial starter code is an objective for this assignment.
    - Remember to read the instructions and
- EX4 will be released Monday
  - You'll need Monday's lecture for a good portion of it
  - Still due Monday, 8/17 (week 9)
  - Need something else fun to do this weekend? Consider solving a maze or two!

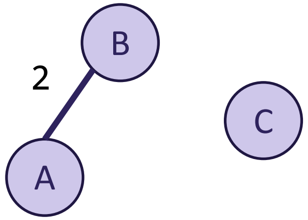
Learning Objectives

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

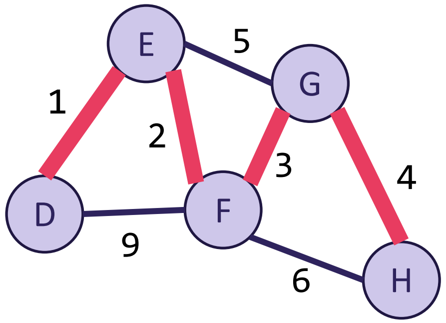
1. Implement WeightedQuickUnion and describe why making the change protects against the worst case find runtime
2. Implement path compression and argue why it improves runtimes, despite not following an invariant
3. Describe what contributes to the runtime of Prim's and Kruskal's, and compare/contrast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

Review MSTs

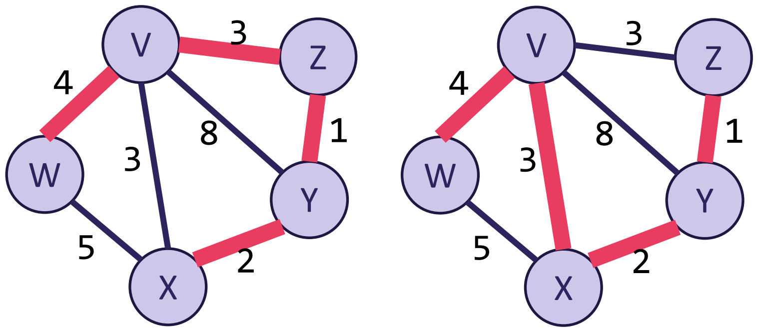
- Minimum (minimizes sum of edge weights) Spanning (connects all vertices) Tree (exactly one path between any two nodes)
  - Minimizing sum of edge weights is NOT the same as minimizing shortest paths!
- If a graph is **connected**, has at least one MST
- If a graph is connected and has all unique edges, 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

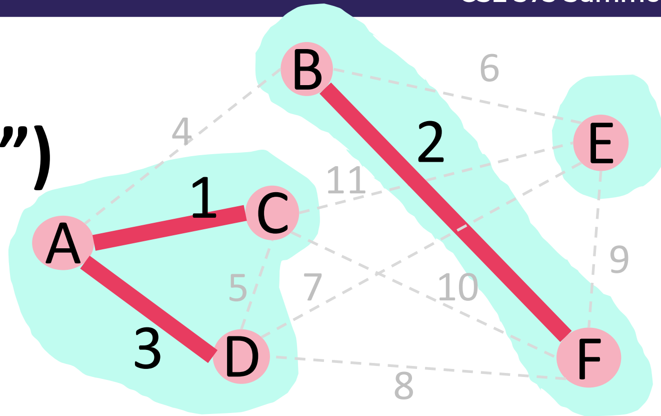


Exactly 1 MST



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

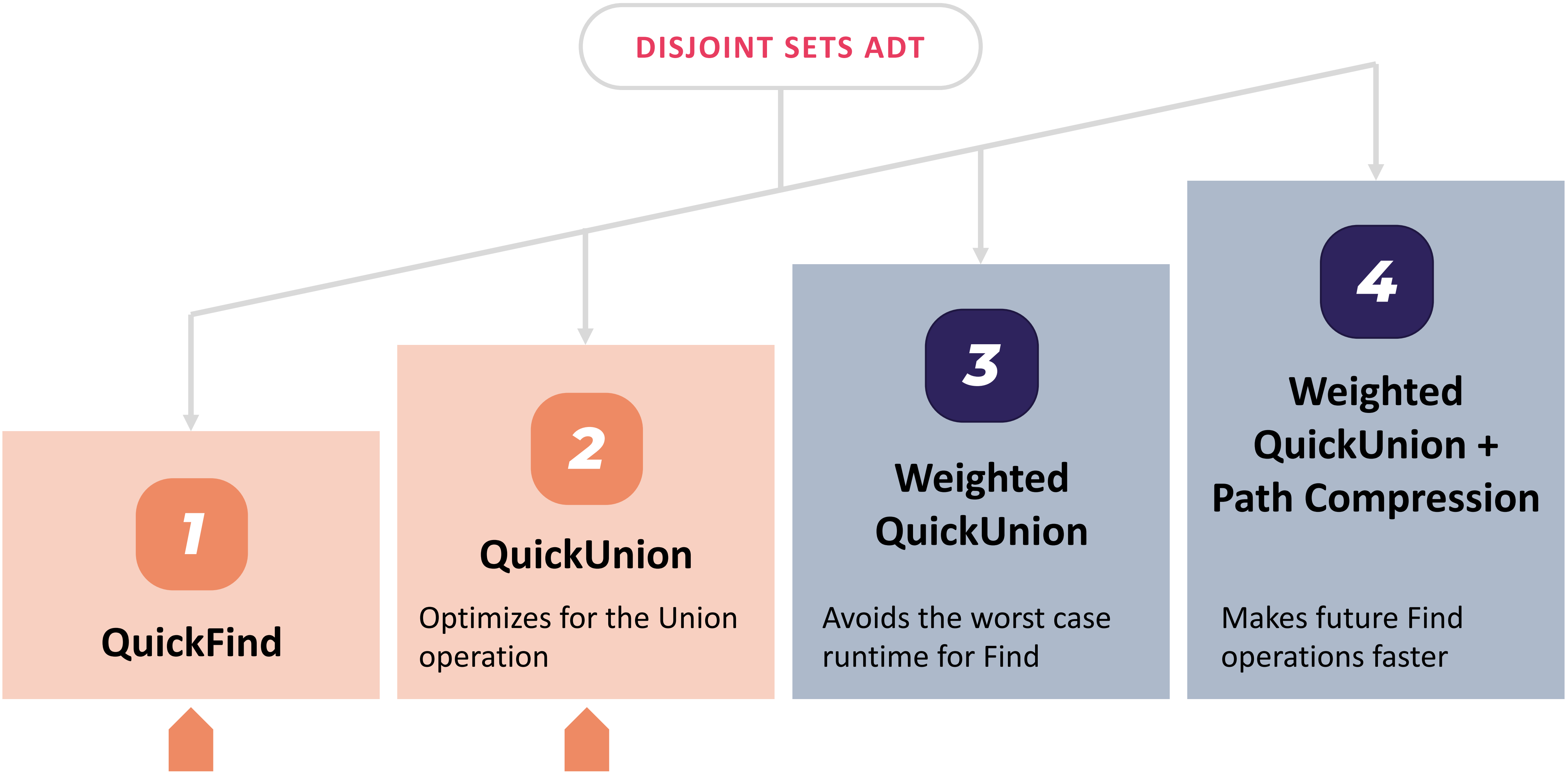
State

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

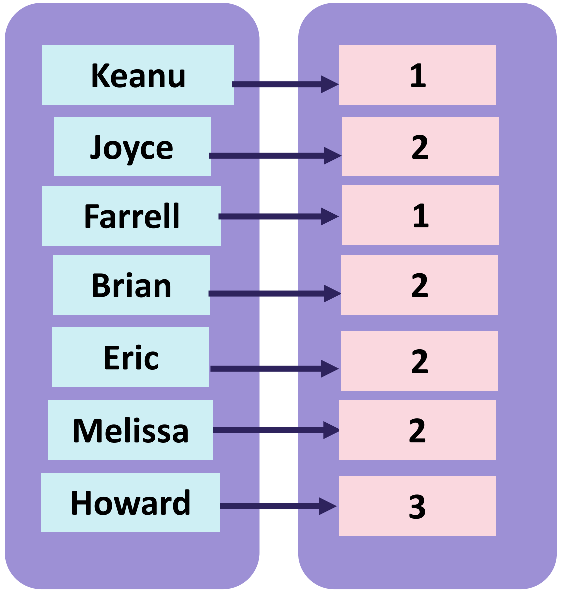


Review QuickFind vs. QuickUnion

DISJOINT SETS ADT

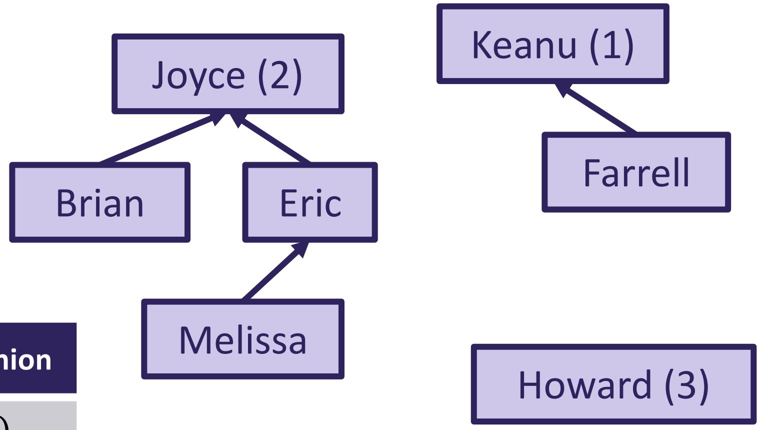
QuickFind

map from value to representative ID



QuickUnion

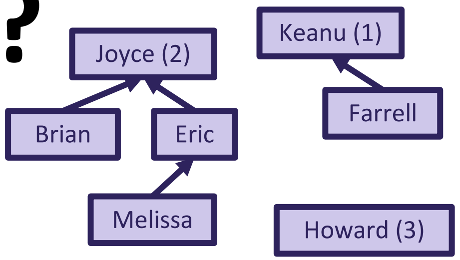
trees of values with representative ID at each root



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

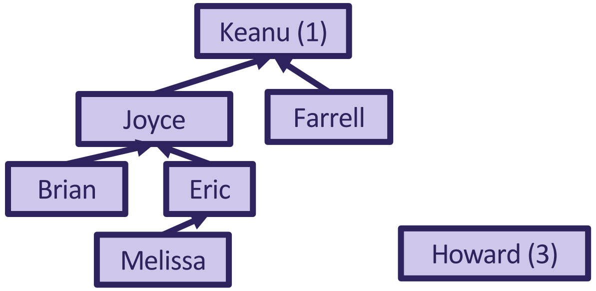
	(Baseline)	QuickFind	QuickUnion
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
`union(x, y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$

Review QuickUnion: Why Use Both Roots?



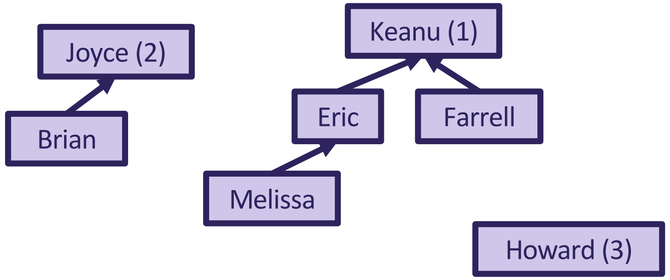
Example: result of `union(Eric, Farrell)` on these Disjoint Sets given three possible implementations:

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



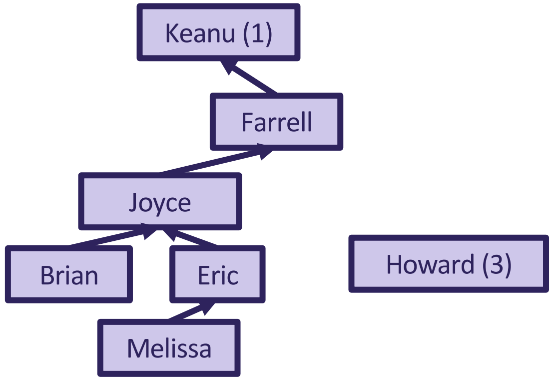
Correct: Everything in Eric's set now connected to everything in Farrell's set!

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



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

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



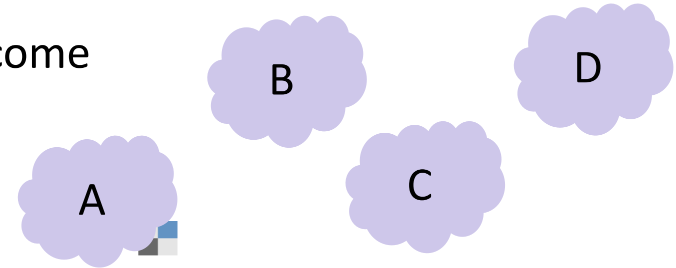
Inefficient: Technically correct, but increases height of the up-tree so makes

Poll Everywhere

pollev.com/uwcse373

Review QuickUnion: Let's Build a Worst Case

Even with the "use-the-roots" implementation of union, try to come up with a series of calls to union that would create a worst-case runtime for find on these Disjoint Sets:



QuickUnion: Let's Build a Worst Case

Top

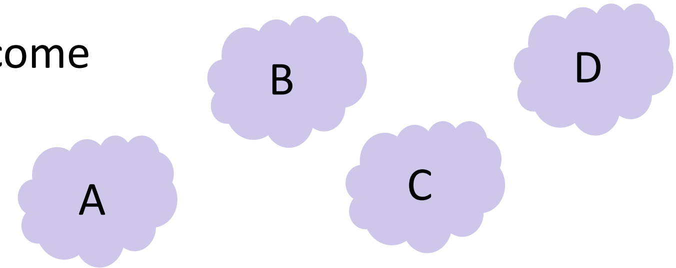
```
find(A):
  jump to A node
  travel upward until root
  return ID
```

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

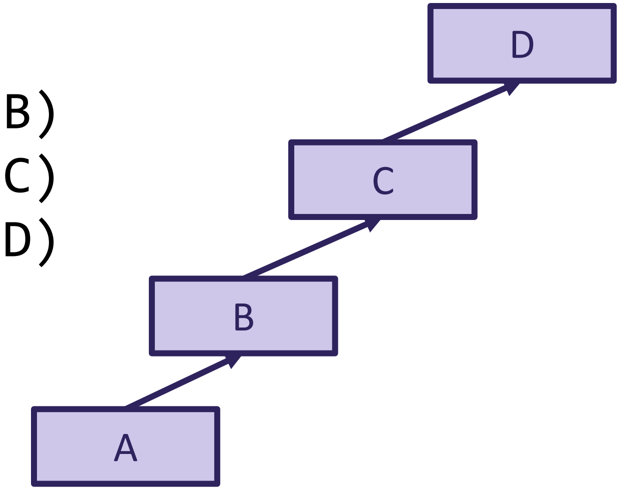
Poll Everywhere pollev.com/uwcse373

Review QuickUnion: Let's Build a Worst Case

Even with the "use-the-roots" implementation of union, try to come up with a series of calls to union that would create a worst-case runtime for find on these Disjoint Sets:




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




```
find(A):
  jump to A node
  travel upward until root
  return ID
```

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

Lecture Outline

DISJOINT SETS ADT

1. **QuickFind**

2. **QuickUnion**
Optimizes for the Union operation

3. **Weighted QuickUnion**
Avoids the worst case runtime for Find

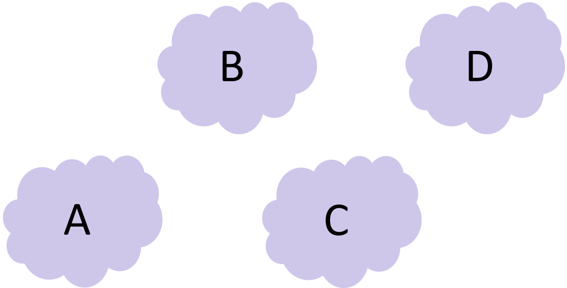
4. **Weighted QuickUnion + Path Compression**
Makes future Find operations faster

Review WeightedQuickUnion

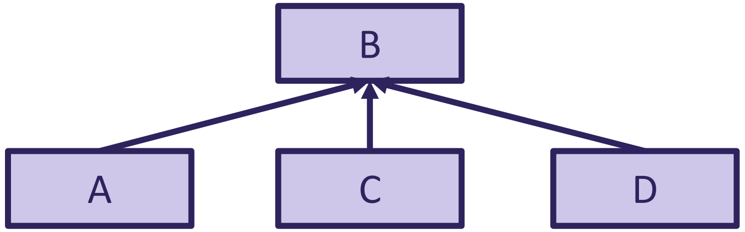
- Goal: Always pick the smaller tree to go under the larger tree
- Implementation: Store the number of nodes (or "weight") of each tree in the root
  - Constant-time lookup instead of having to traverse the entire tree to count

```
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.

WeightedQuickUnion: Performance

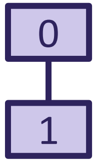
- Consider the worst case where the tree height grows as fast as possible

N	H
1	0

WeightedQuickUnion: Performance

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

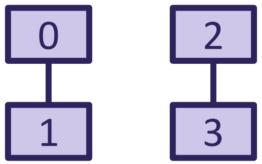
N	H
1	0
2	1



WeightedQuickUnion: Performance

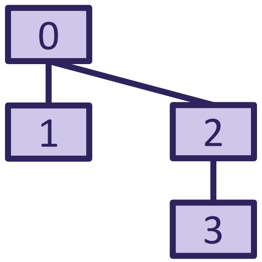
- Consider the worst case where the tree height grows as fast as possible

N	H
1	0
2	1
4	?



WeightedQuickUnion: Performance

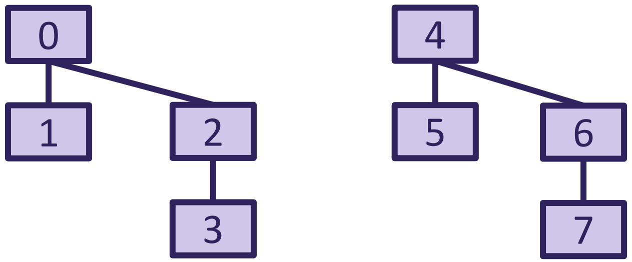
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	2

WeightedQuickUnion: Performance

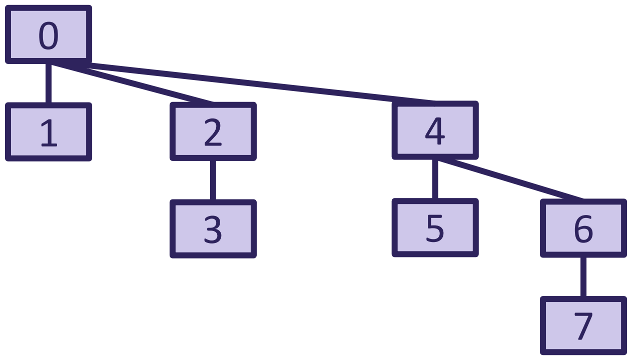
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	2
8	?

WeightedQuickUnion: Performance

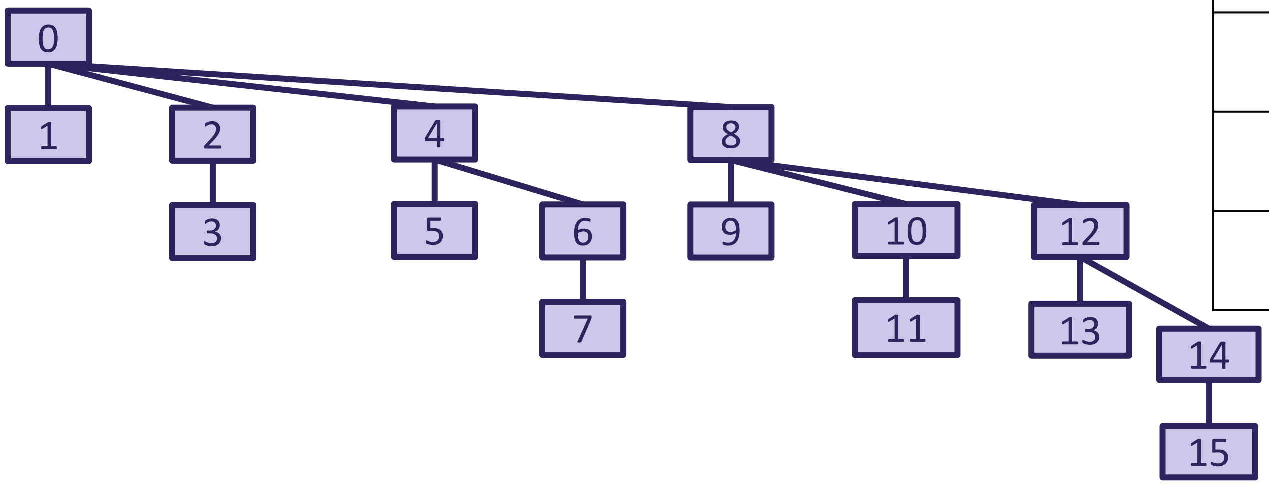
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	2
8	3

WeightedQuickUnion: Performance

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

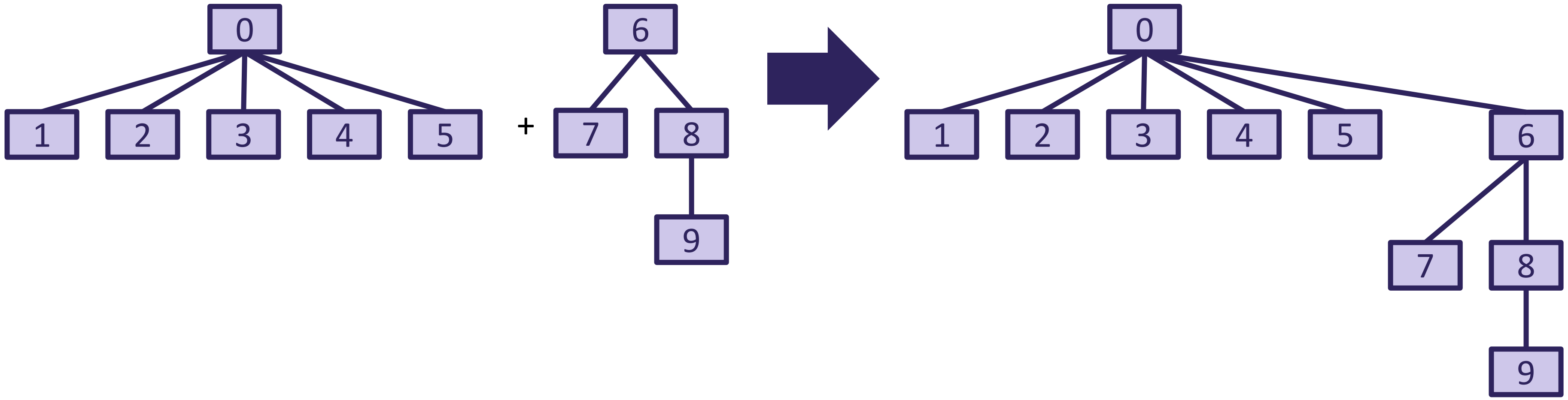


N	H
1	0
2	1
4	2
8	3
16	4

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:



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

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

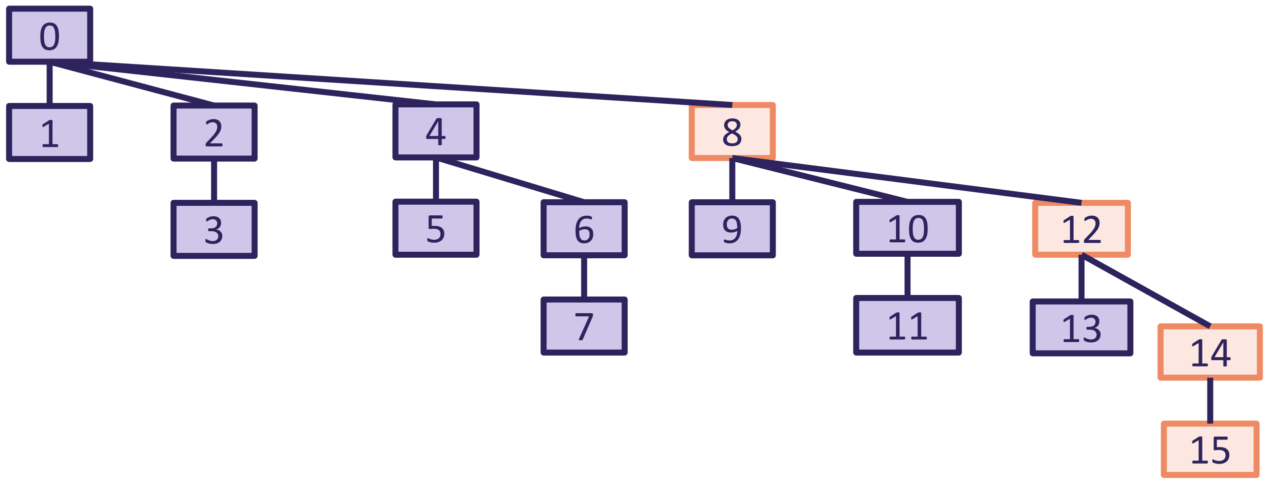
Modifying Data Structures for Future Gains

- 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

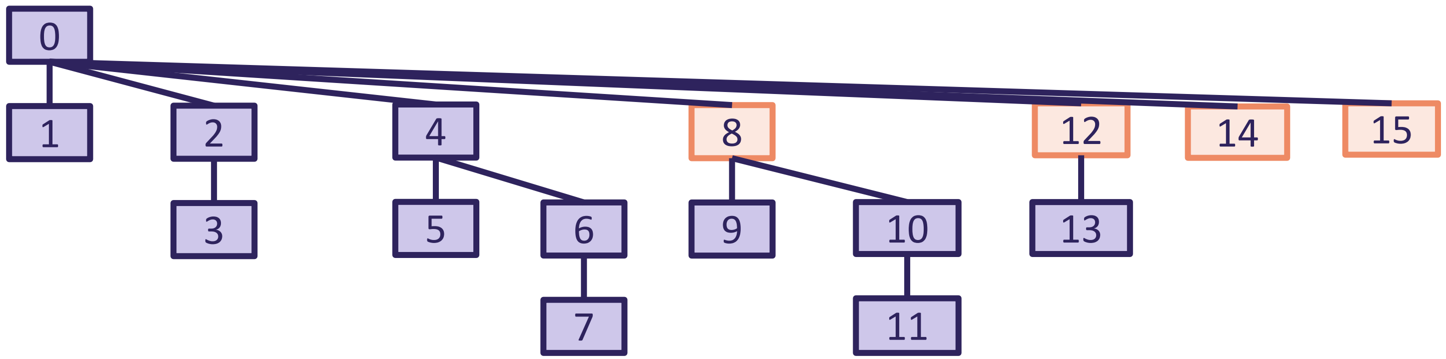
- This is the worst-case topology if we use WeightedQuickUnion



- 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

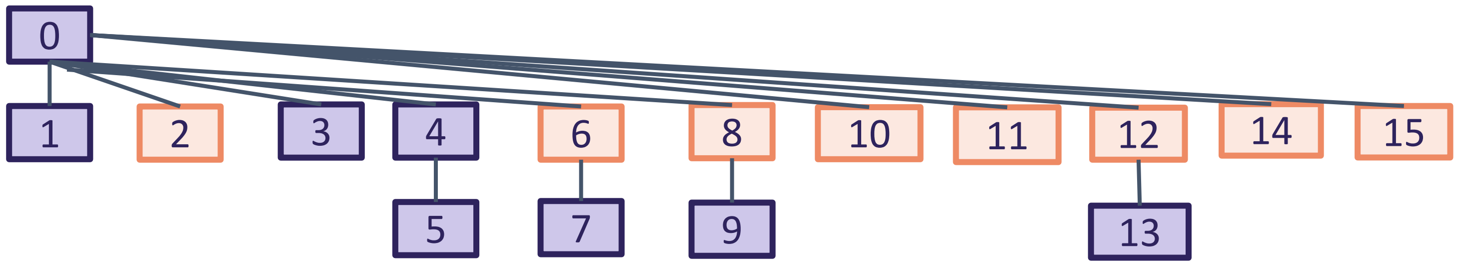
- This is the worst-case topology if we use WeightedQuickUnion



- 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

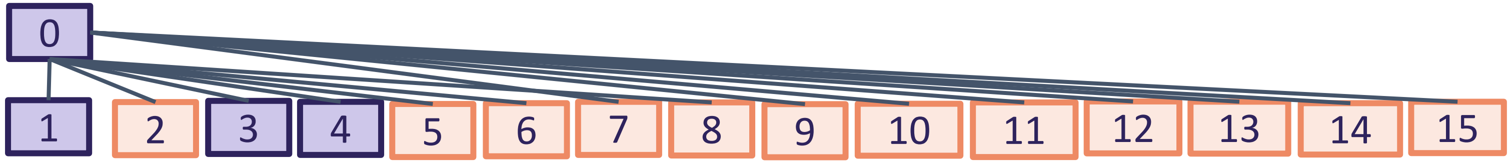
- Run path compression on every find()!
  - Including the find()s that are invoked as part of a union()



- 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!

Kruskal's Runtime

$\Theta(|E| \log |E|)$

$\Theta(|V|)$

$\Theta(|E| \log |E|)$

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

$\Theta(|E|)$ iterations

$\Theta(\log |V|)$

$\Theta(|V| \log |V|)$

<u>total</u> $\Theta(|V|)$ iterations

$\Theta(\log |V|)$

```
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)
```

- 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

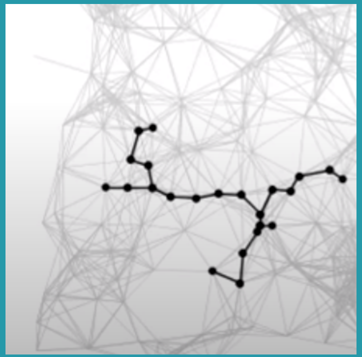
$\Theta(|V|\log|V| + |E|\log|V|)$

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

Prim's

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

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



Kruskal's

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

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

Lecture Outline

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

Lecture Outline

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

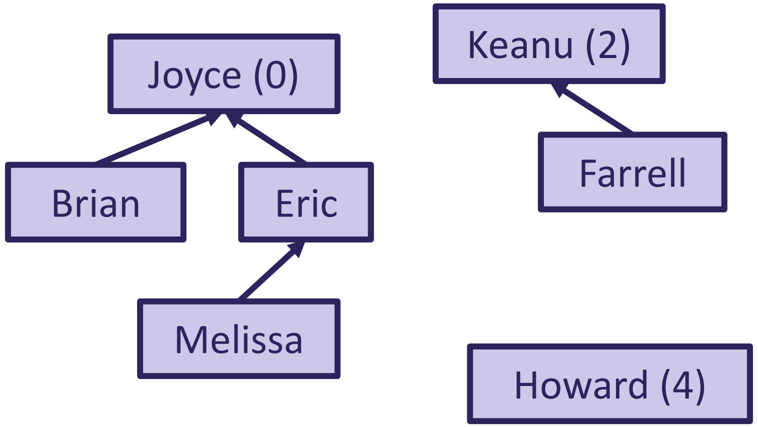
Makes future Find operations faster

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	*Brian*	*Keanu*	*Melissa*	*Howard*	*Farrell*	*Eric*

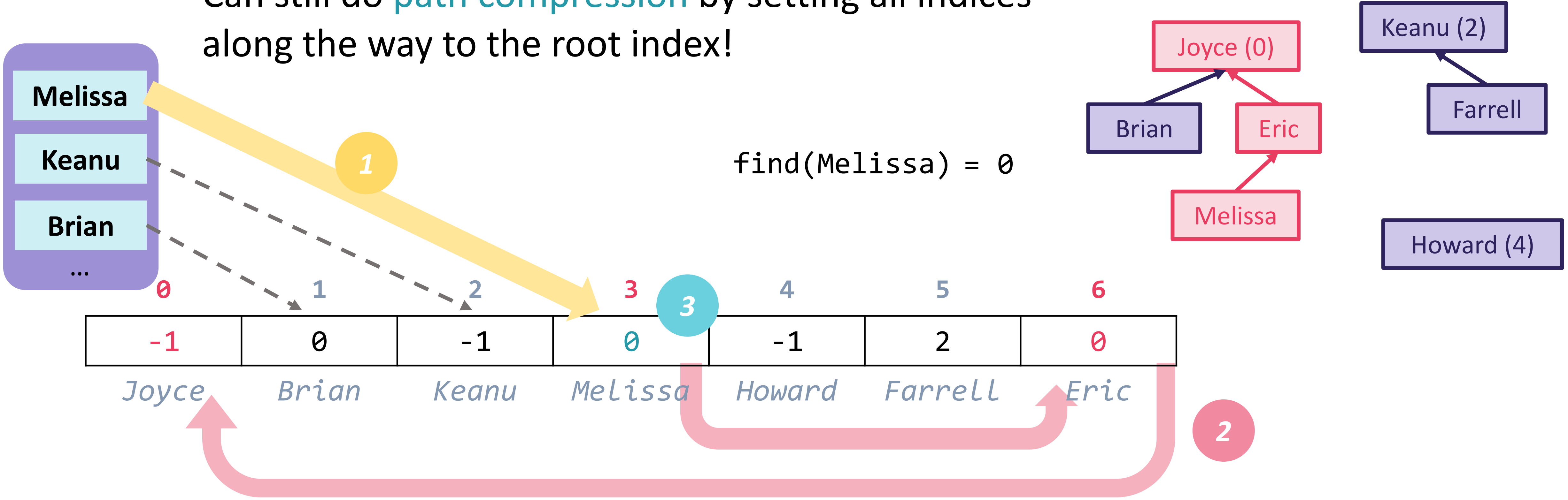
Using Arrays: Find

- 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):
1  index = jump to A node's index
2  while array[index] >= 0:
     index = array[index]
3  path compression
   return index
```

CORRECTED AFTER LECTURE

find(Melissa) = 0

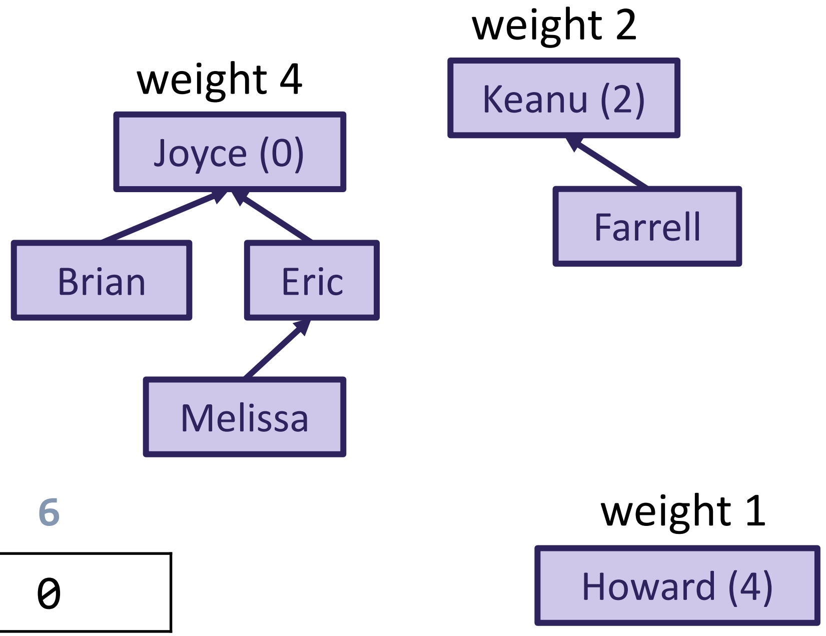


0	1	2	3	4	5	6
-1	0	-1	0	-1	2	0
Joyce	Brian	Keanu	Melissa	Howard	Farrell	Eric

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
```



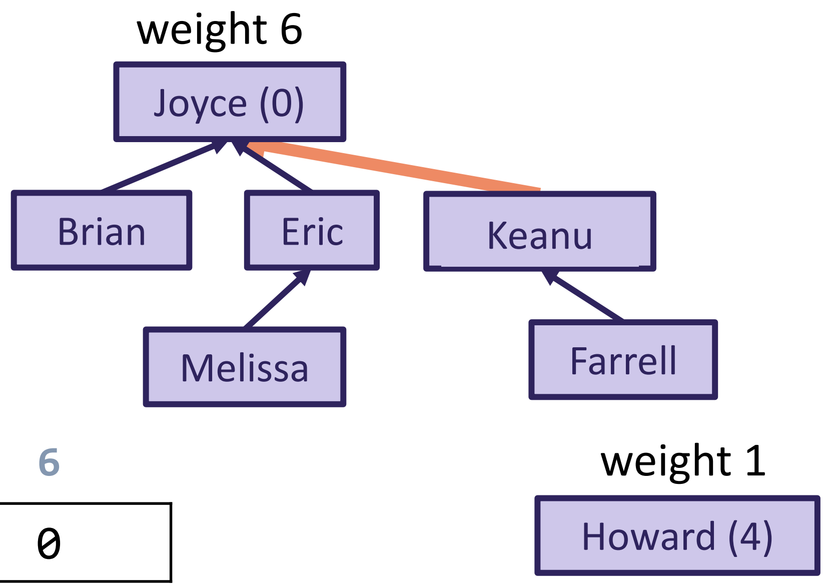
`union(Eric, Farrell)`

0	1	2	3	4	5	6
-4	0	-2	6	-1	2	0
Joyce	*Brian*	*Keanu*	*Melissa*	*Howard*	*Farrell*	*Eric*

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(Eric, Farrell)`

0	1	2	3	4	5	6
-6	0	0	6	-1	2	0
Joyce	*Brian*	*Keanu*	*Melissa*	*Howard*	*Farrell*	*Eric*

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)`	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$O(\log^* n)$	$O(\log^* n)$

Recap: Graph Modeling

Often need to refine original model as you work through details of algorithm

SCENARIO & QUESTION TO ANSWER

Many ways to model any scenario with a graph, but question motivates which data is important

MODEL AS A GRAPH

- Choose vertices
- Choose edges
- Directed/Undirected
- Weighted/Unweighted
- Cyclic/Acyclic

…

RUN ALGORITHM

- Just visit every node?
  - BFS or DFS
- s-t Connectivity?
  - BFS or DFS
- Unweighted shortest path?
  - BFS
- Weighted shortest path?
  - Dijkstra's
- Minimum Spanning Tree?
  - Prim's or Kruskal's

ANSWER!

Appendix

Another Graph Modeling Practice Problem

Graph Modeling Activity

Note Passing - Part I

Imagine you are an American High School student. You have a very important note to pass to your crush, but the two of you do not share a class so you need to rely on a chain of friends to pass the note along for you. A note can only be passed from one student to another when they share a class, meaning when two students have the same teacher during the same class period.

Unfortunately, the school administration is not as romantic as you, and passing notes is against the rules. If a teacher sees a note, they will take it and destroy it. Figure out if there is a sequence of handoffs to enable you to get your note to your crush.

How could you model this situation as a graph?

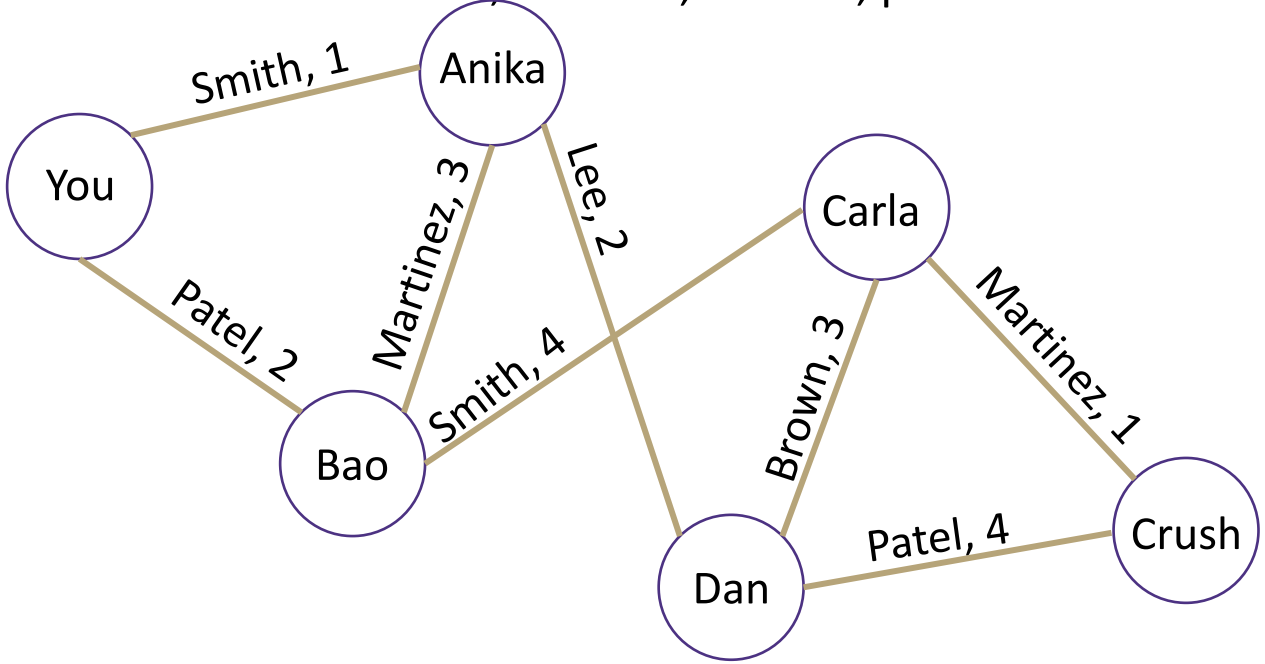
	Period 1	Period 2	Period 3	Period 4
You	Smith	Patel	Lee	Brown
Anika	Smith	Lee	Martinez	Brown
Bao	Brown	Patel	Martinez	Smith
Carla	Martinez	Jones	Brown	Smith
Dan	Lee	Lee	Brown	Patel
Crush	Martinez	Brown	Smith	Patel

Possible Design

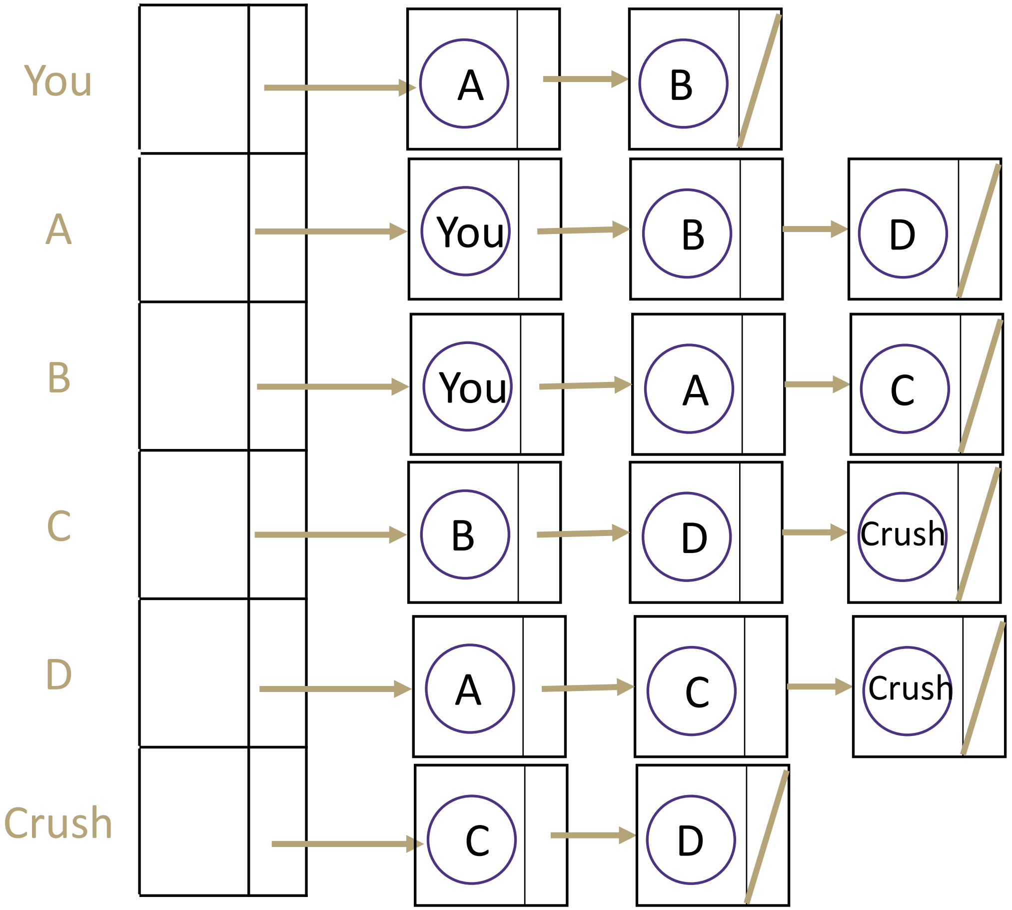
Algorithm

BFS or DFS to see if you and your Crush are connected

- **Vertices**
  - Students
  - Fields: Name, have note
- **Edges**
  - Classes shared by students
  - Not directed
  - Could be left without weights
  - Fields: vertex 1, vertex 2, teacher, period



Adjacency List



More Design

Note Passing - Part II

Now that you know there exists a way to get your note to your crush, we can work on picking the best hand off path possible.

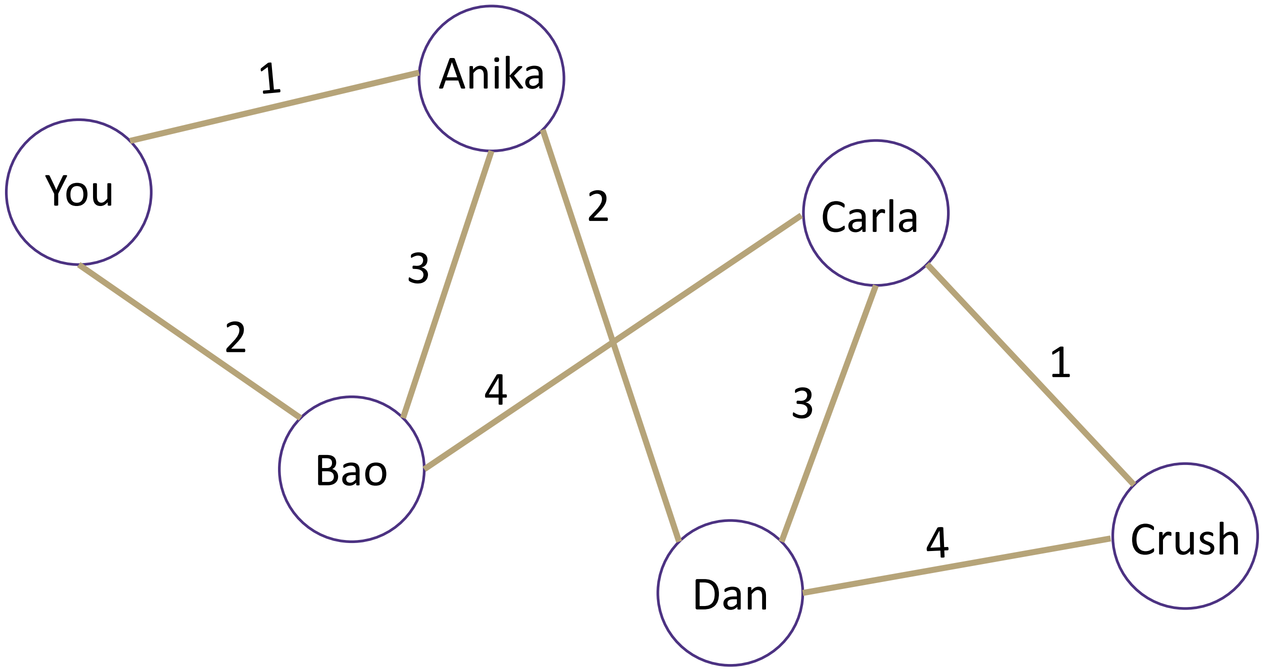
Thought Experiments:

1. What if you want to optimize for time to get your crush the note as early in the day as possible?
   - How can we use our knowledge of which period students share to calculate for time knowing that period 1 is earliest in the day and period 4 is later in the day?
   - How can we account for the possibility that it might take more than a single school day to deliver the note?

2. What if you want to optimize for rick avoidance to make sure your note only gets passed in classes least likely for it to get intercepted?
   - Some teachers are better at intercepting notes than others. The more notes a teacher has intercepted, the more likely it is they will take yours and it will never get to your crush. If we knew how many notes each teacher has intercepted how might we incorporate that into our graph to find the least risky route?

Optimize for Time

“Distance” will represent the sum of which periods the note is passed in, because smaller period values are earlier in the day the smaller the sum the earlier the note gets there except in the case of a “wrap around”



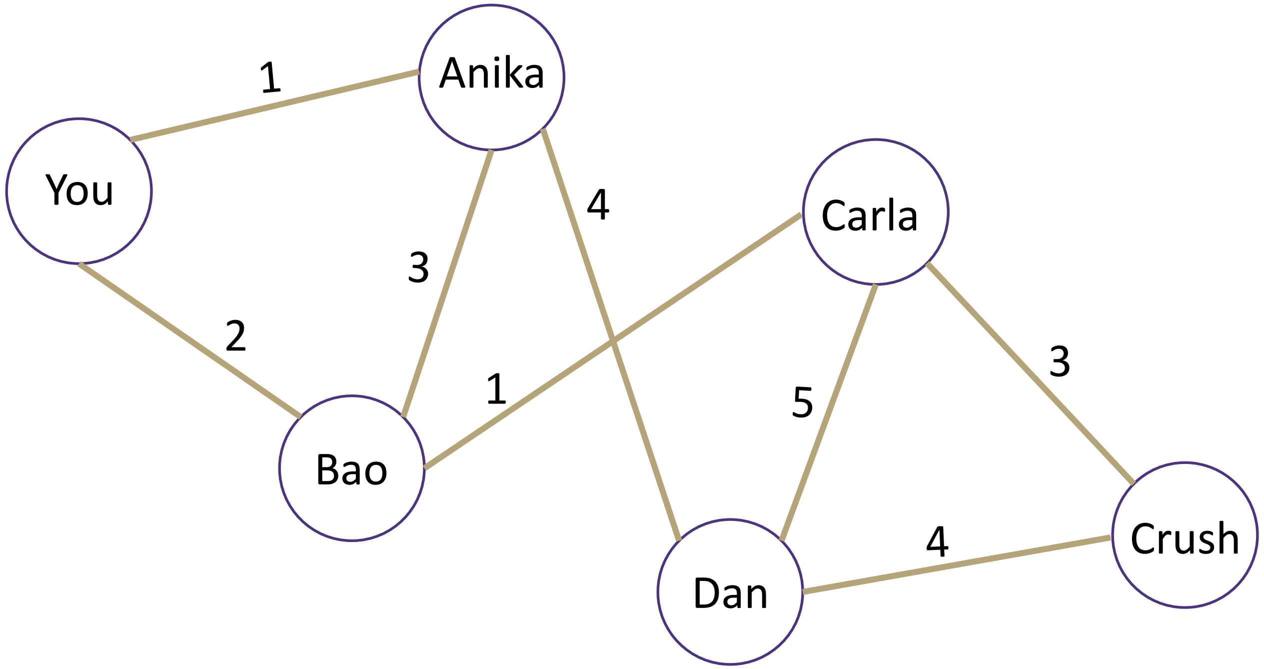
1. Add the period number to each edge as its weight
2. Run Dijkstra’s from You to Crush

Vertex	Distance	Predecessor	Process Order
You	0	--	0
Anika	1	You	1
Bao	2	You	5
Carla	6	Dan	3
Dan	3	Anika	2
Crush	7	Carla	4*

*The path found wraps around to a new school day because the path moves from a later period to an earlier one
- We can change our algorithm to check for wrap arounds and try other routes

Optimize for Risk

"Distance" will represent the sum of notes intercepted across the teachers in your passing route. The smaller the sum of notes the "safer" the path.



1. Add the number of letters intercepted by the teacher to each edge as its weight
2. Run Dijkstra's from You to Crush

Teacher	Notes Intercepted
Smith	1
Martinez	3
Lee	4
Brown	5
Patel	2

Vertex	Distance	Predecessor	Process Order
You	0	--	0
Anika	1	You	1
Bao	4	Anika	2
Carla	5	Bao	3
Dan	10	Carla	5
Crush	8	Carla	4