Disjoint Sets

CSE 373 Winter 2020

Instructor: Hannah C. Tang

Teaching Assistants:

Aaron Johnston
Amanda Park
Anish Velagapudi
Brian Chan
Elena Spasova
Ethan Knutson
Farrell Fileas
Howard Xiao
Jade Watkins
Lea Quan
Nathan Lipiarski
Sam Long
Yifan Bai
Yuma Tou

Announcements

- HW7 released; due Fri, Feb 28
  - HW7 exercises a different set of skills: reading/understanding a large codebase and figuring out where to plug in
  - **Read the spec carefully**; HW7 substantially longer than last quarter's because we added **a ton of hints**

Feedback from Reading Quiz

- When do we use Disjoint Sets?
- Can we make union() constant time?
- How did you choose the values for the ids?

Lecture Outline

- **Disjoint Set ADT**
- QuickFind Data Structure
- QuickUnion Data Structure
- WeightedQuickUnion Data Structure
  - Path Compression

Disjoint Sets ADT

Disjoint Sets ADT. A collection of elements and sets of those elements.

- An element can only belong to a single set.
- Each set is identified by a unique id.
- Sets can be combined/ connected/ unioned.

- The Disjoint Sets ADT has two operations:
  - find(e): gets the id of the element's set
  - union(e1, e2): combines the set containing e1 with the set containing e2
- Example: ability to travel to drive to a country
  - union(france, germany)
  - union(spain, france)
  - find(spain) == find(germany)?
  - union(england, france)

Disjoint Sets ADT

- The Disjoint Sets ADT has two operations:
  - find(e): gets the id of the element's set
  - union(e1, e2): combines the set containing e1 with the set containing e2
- Applications include percolation theory (computational chemistry) and …. Kruskal's algorithm
- Simplifying assumptions
  - We can map elements to indices quickly (see reading)
  - We know all the items in advance; they're all disconnected initially

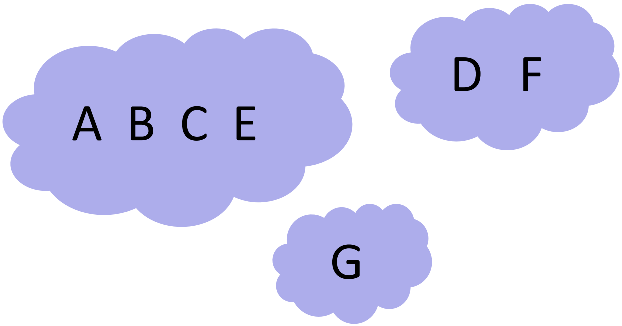
An Observation …

- Today's lecture on the data structures which implement the Disjoint Sets ADT is an interesting case study in data structure design and iterative design improvements
  - Dust off your metacognitive skills and pay attention to what stays the same and what changes between our 3 options

Lecture Outline

- Disjoint Set ADT
- **QuickFind Data Structure**
- QuickUnion Data Structure
- WeightedQuickUnion Data Structure
  - Path Compression

QuickFind (review)



A	0
B	1
C	2
D	3
E	4
F	5
G	6

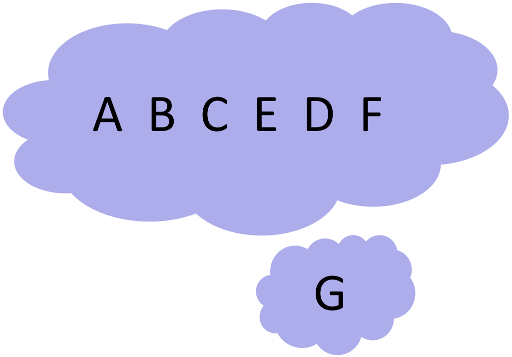
```
find(A) == 123
find(B) == 123
find(A) == find(B)
find(C) != find(D)
```

int[] ids

0	1	2	3	4	5	6
123	123	123	456	123	456	789

```
union(C, D)
```

QuickFind (review)



A	0
B	1
C	2
D	3
E	4
F	5
G	6

```
find(A) == 123
find(B) == 123
find(A) == find(B)
find(C) != find(D)
```

int[] ids

0	1	2	3	4	5	6
123	123	123	456	123	456	789

```
union(C, D)
```

int[] ids

0	1	2	3	4	5	6
456	456	456	456	456	456	789

Disjoint Sets: Runtime

- Feedback from reading quiz: "can we make union() constant time?"

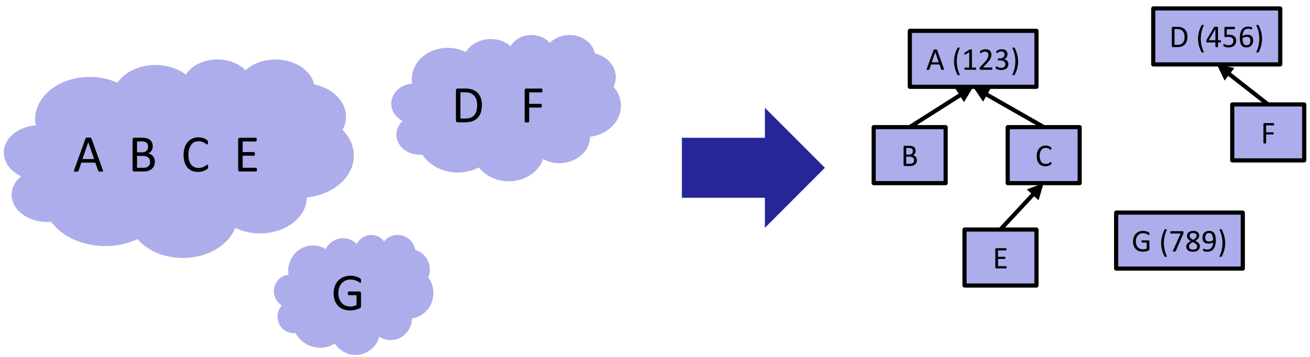
	find	union
QuickFind	Θ(1)	Θ(N)

Lecture Outline

- Disjoint Set ADT
- QuickFind Data Structure
- **QuickUnion Data Structure**
- WeightedQuickUnion Data Structure
  - Path Compression

QuickUnion Data Structure

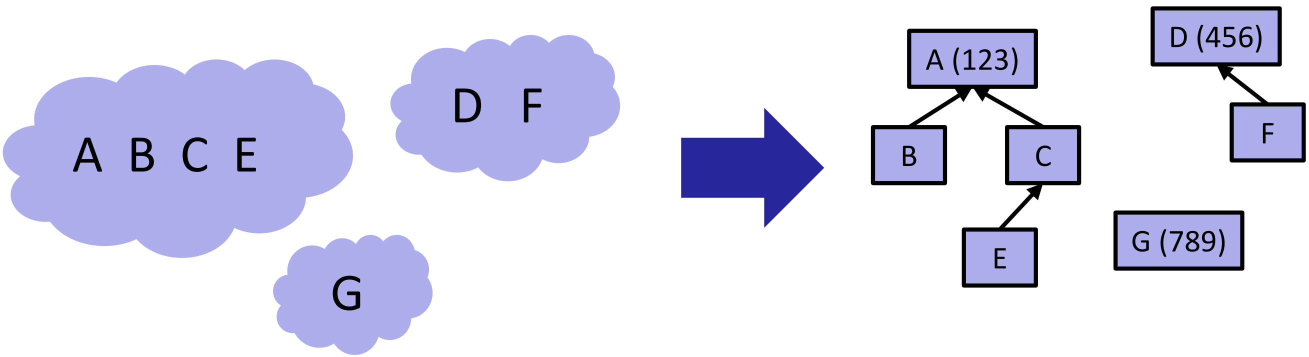
- Fundamental idea:
  - QuickFind tracks each element's ID
  - QuickFind tracks each element's *parent*. Only the root has an ID




```
find(A) == 123
find(B) == 123
find(A) == find(B)
find(C) != find(D)
```

QuickUnion: Representation

- Like the binary heap, we can represent QuickUnion as an array
  - Note: we represent ids as negative numbers to clarify that they're not indices

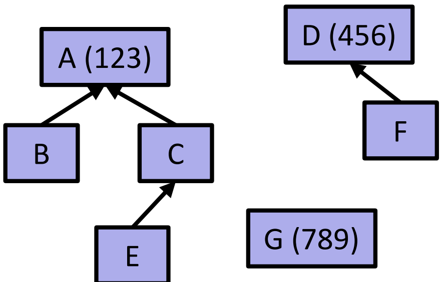


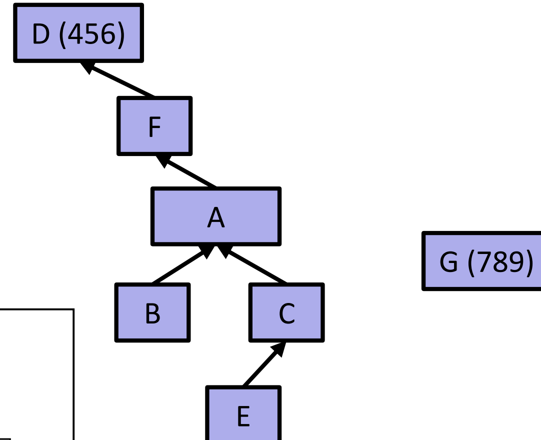
int[] parents

int[] parents	-123	0	0	-456	2	3	-789
	0	1	2	3	4	5	6

QuickUnion: Union

- How does this data structure implement the union operation?





Note! One find() inside union()

```
union(F, C):
  find(C)
  move under F
```

int[] parents	-123	0	0	-456	2	3	-789
	0	1	2	3	4	5	6

int[] parents	3	0	0	-456	2	3	-789
	0	1	2	3	4	5	6

Poll Everywhere

pollev.com/uwcse373

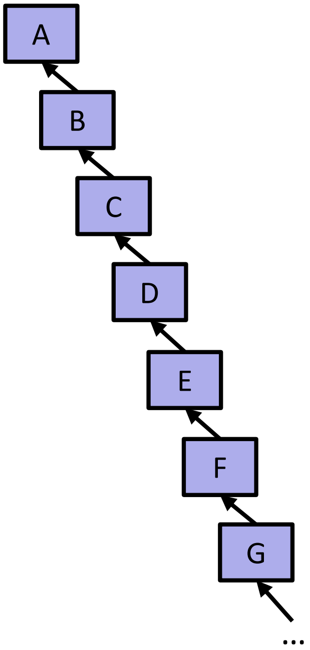
- What are QuickUnion's runtimes?
  - (do not include the runtime for the find() call that union() requires)

	find	union *excludes find*
QuickFind	Θ(1)	Θ(N)
QuickUnion		

A. Θ(N) / Θ(1)
B. Θ(N) / O(1)
C. O(N) / Θ(1)
D. O(N) / O(1)
E. I'm not sure …

Worst-case QuickUnion

```
union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)
union(F, G)
…
```



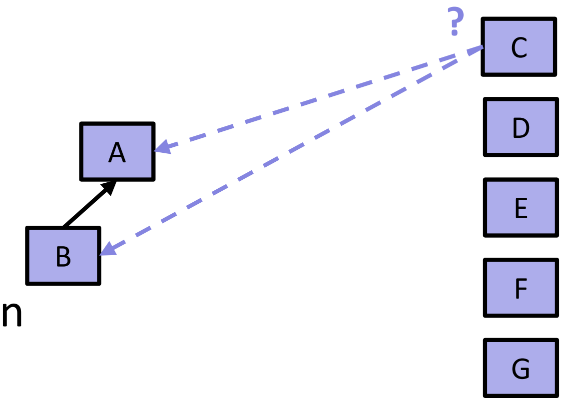
Worst-case Structure

If only I could keep these trees (semi-?)balanced

Lecture Outline

- Disjoint Set ADT
- QuickFind Data Structure
- QuickUnion Data Structure
- **WeightedQuickUnion Data Structure**
  - Path Compression

WeightedQuickUnion



- QuickUnion always picked the same argument (the second argument) to become the child in the unioned structure

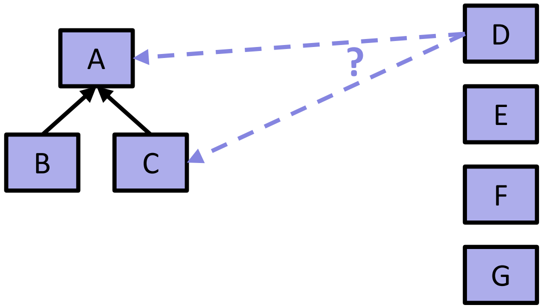
- QuickUnion only found the root of the second argument

- Instead, let's:
  - Pick the smaller tree to be the new child
  - Add the new child to the root

```
union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)
union(F, G)
…
```

WeightedQuickUnion: Union

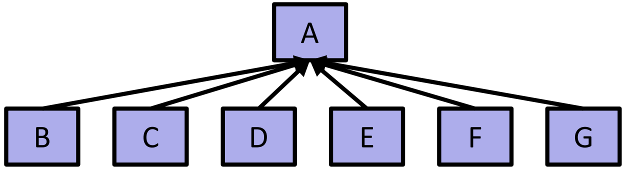
- Pick the smaller tree to be the new child
- Add the new child to the root




```
union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)
union(F, G)
…
```

WeightedQuickUnion: Union

- Pick the smaller tree to be the new child
- Add the new child to the root



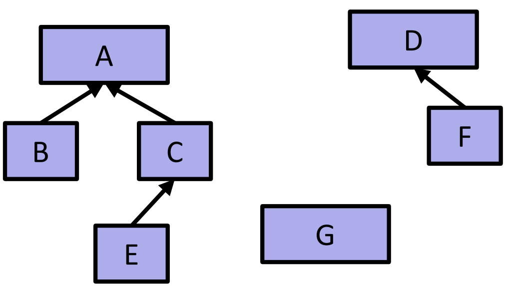

```
union(A, B)
union(B, C)
union(C, D)
union(D, E)
union(E, F)
union(F, G)
…
```

WeightedQuickUnion: Representation

- Need to store the *number of nodes* (or "weight") of each tree

- Don't need to store the root's ID; we can hash the element as needed
- Now we can store the weight there instead!
  - However, we still use negative values to indicate they're not indices

A	0
B	1
C	2
D	3
E	4
F	5
G	6



int[] parents

-4	0	0	-2	2	3	-1
0	1	2	3	4	5	6

WeightedQuickUnion: Performance

- union()'s runtime is still dependent on find()'s runtime, which is a function of the tree's height

```
union(e1, e2):
  find(e1)
  find(e2)
  move lighter root under heavier root
```

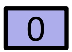
Note! Now have two find() calls inside union()!

- What's the worst-case height for WeightedQuickUnion?

WeightedQuickUnion: Performance

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

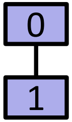
N	H
1	0



WeightedQuickUnion: Performance

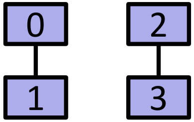
- Consider the worst case where the tree height grows as fast as possible

N	H
1	0
2	1



WeightedQuickUnion: Performance

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

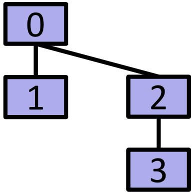


N	H
1	0
2	1
4	?

WeightedQuickUnion: Performance

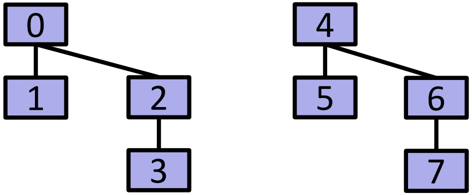
- Consider the worst case where the tree height grows as fast as possible

N	H
1	0
2	1
4	2



WeightedQuickUnion: Performance

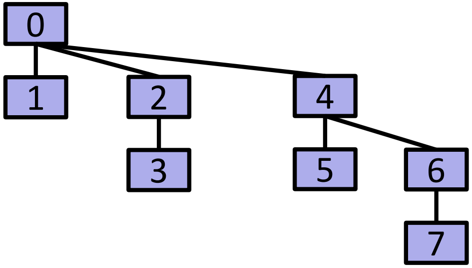
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	2
8	?

WeightedQuickUnion: Performance

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



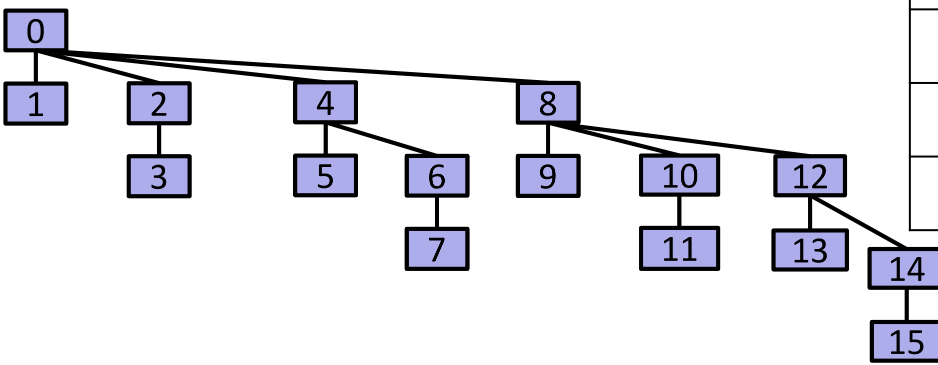
N	H
1	0
2	1
4	2
8	3

WeightedQuickUnion: Performance

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

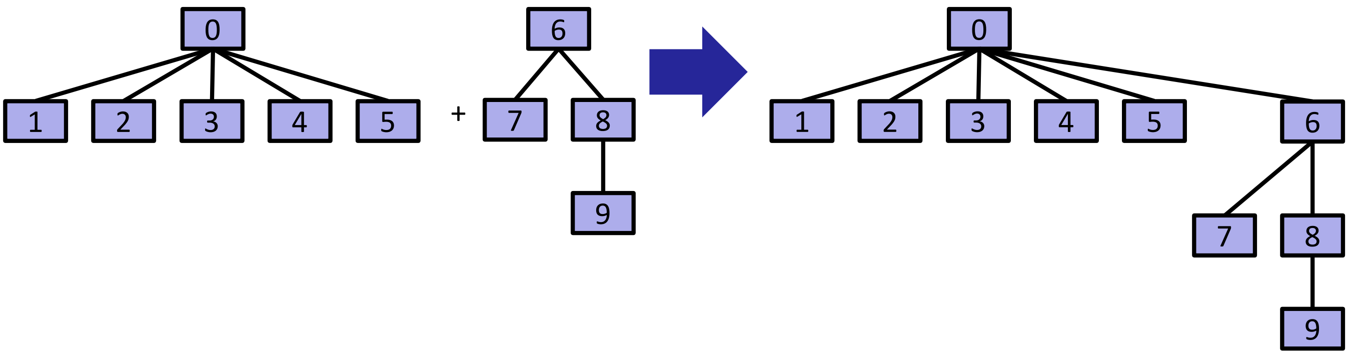
- Worst case tree height is Θ(log N)

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



WeightedQuickUnion Runtime

	find	union *excludes find(s)*	union *includes find(s)*
QuickFind	Θ(1)	Θ(N)	N/A
QuickUnion	h = O(N)	Θ(1)	O(N)
WeightedQuickUnion	h = Θ(log N)	Θ(1)	Θ(log N)

- There's one final optimization we can make: path compression

Lecture Outline

- Disjoint Set ADT
- QuickFind Data Structure
- QuickUnion Data Structure
- WeightedQuickUnion Data Structure
  - **Path Compression**

Modifying Data Structures To Preserve Invariants

- Thus far, the modifications we've studied are designed to *preserve invariants* (aka "repair the data structure")
  - **Tree rotations**: preserve LLRB tree invariants (eg, a right-leaning red edge)
  - **Promoting keys / splitting leaves**: preserve B-tree invariants (eg, L+1 keys stored in a leaf node)

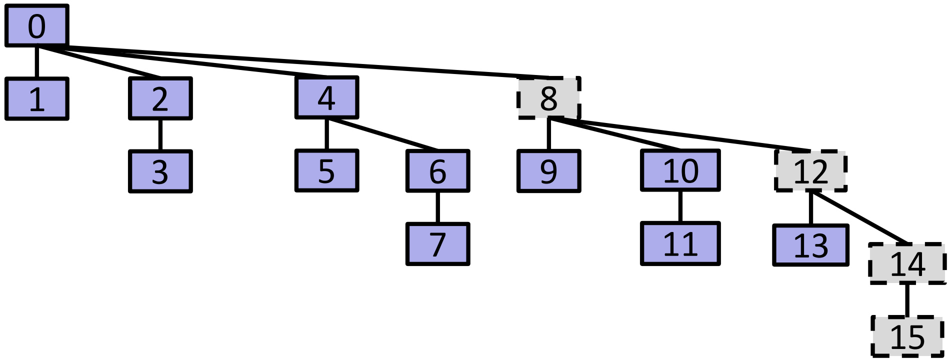
- Notably, the modifications don't improve runtime between identical method calls
  - If bst.find(x) takes 2 µs, we expect future calls to take ~2 µs
  - If we call bst.find(x) M times, the total runtime should be 2*M µs

Modifying Data Structures for Future Gains

- Path compression is entirely different: we are modifying the tree structure to *improve future performance*
  - If wquWithPathCompression.find(x) takes 2 μs, we expect future calls to take <2 μs
  - If we call wquWithPathCompression.find(x) M times, the total runtime should be <2*M μs (and possibly even << 2*M μs)

Path Compression: Idea

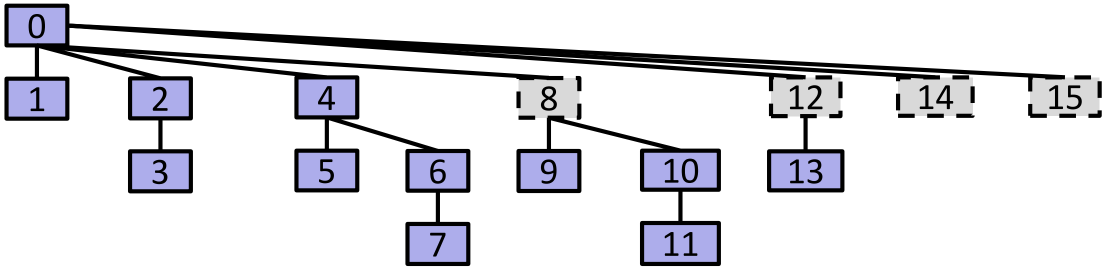
- This is the worst-case topology if we use WeightedQuickUnion



- Idea: When we do find(15), move all visited nodes under the root
  - Additional cost is insignificant (same order of growth)

Path Compression: Example

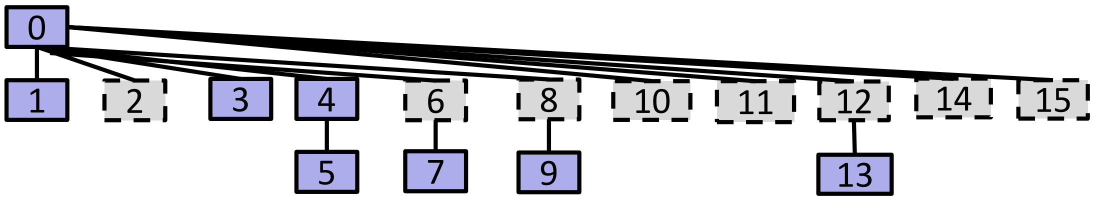
- This is the worst-case topology if we use WeightedQuickUnion



- Idea: When we do find(15), move all visited nodes under the root
  - Doesn't meaningfully change runtime for *this* invocation of find(15), but subsequent find(15)s (and subsequent find(14)s and find(12)s and …) will be faster

Path Compression: Details and Runtime

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

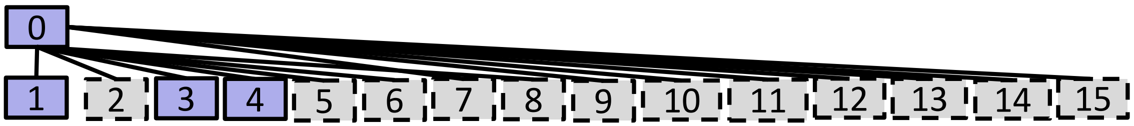


- Understanding the performance of M>1 operations requires *amortized analysis*

- We won't go into it here, but we've seen it before
  - It's how we assert that appending to an array is "O(1) on average" if we double whenever we resize

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

tl;dr

- Disjoint Sets ADT implementations:

	find	union *excludes find(s)*	union *includes find(s)*
QuickFind	$\Theta(1)$	$\Theta(N)$	N/A
QuickUnion	h = $O(N)$	$\Theta(1)$	$O(N)$
WeightedQuickUnion	h = $\Theta(\log N)$	$\Theta(1)$	$\Theta(\log N)$
WQU + Path Compression	h = $O(1)$*	$O(1)$*	$O(1)$*

* amortized

- Kruskal's Algorithm: $O(V * \text{union} + E * \text{find}) = O(V + E \log V)$