Disjoint Sets

CSE 332 Spring 2020

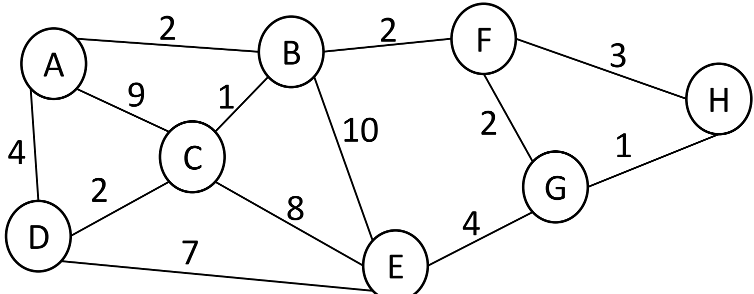
Instructor: Hannah C. Tang

Teaching Assistants:

Annie Mao	Howard Xiao	Lucy Jiang
Chris Choi	Jade Watkins	Nathan Lipiarski
Ethan Knutson	Khushi Chaudhari	Richard Jiang

Warm-up

- Write your answers to the following questions on a piece of scratch paper:
  - How many times do you need to take the log of 20,035,299,304,068,464,649,790,723,515,602,557,504,478,254,755, 697,514,192,650,169,737 to get to a value <= 1?
    - Hint: that value is equal to 2^{65536}, and 65536 = 2^{16}
  - Find an MST using Kruskal's
    - If you have extra time, find it using Prim's. Under what conditions will these algorithms give the same result?



Announcements

- Q4 feedback released later today
- No changes to this week's schedule, but we'll try to make quiz 5 easier than quiz 4. Finalized schedule is:
  - *Wed, Jun 3*: Project 3 due
  - *Wed, Jun 3 - Fri, Jun 5*: Quiz 5
  - *Fri, Jun 5*: Exercises 14-15 due

Lecture questions: pollev.com/cse332

Learning Objectives

- Understand the Disjoint Sets ADT
- Implement Up-trees with Weighted Union and Path Compression
  - Describe its amortized and worst-case runtimes

Lecture Outline

- **Review: Disjoint Sets ADT**

- Up-Tree Data Structure
  - Representation
  - Optimization: Weighted Union
  - Optimization: Path Compression

Lecture questions: pollev.com/cse332

Review: Disjoint Sets ADT (1 of 2)

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)

Review: Disjoint Sets ADT (2 of 2)

- 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
  - We know all the items in advance; they're all disconnected initially
- In the *this* lecture, we'll see:
  - We can do union() in constant time
  - We can get find() to be ***amortized*** constant time
    - Worst case O(log n) for an individual find operation

Example Application: Kruskal's Algorithm

```
kruskals(Graph g) {
  mst = {}
  forests = buildDisjointSets(g.vertices)
  numforests = g.vertices
  edges = buildHeap(g.edges)

  while (numforests > 1):
    e = edges.deleteMin()
    u_id = forests.find(e.u)
    v_id = forests.find(e.v)
    if (u_id != v_id):
      mst.addEdge(e)
      forests.union(e.u, e.v)
      numforests--
}
```

|E| times

|V| times

} |E| deleteMin()s

} 2|E| find()s

} |V| union()s

Runtime: $|E|(\log|E| + 2\log|V| + 1) + |V|(1 + 1 + 1) \in O(|E|\log|V| + |V|\log|V|)$

However, since we know E $\in O(|V|^2)$, *runtime* $\in O(|E|\log|V|)$

Implementing the Disjoint Sets ADT (1 of 2)

- If we have *n* elements:
  - What is the total cost of m find()s + ≤ n-1 union()s?

Can we have >n union()s?

- Goal: O(m+n)
  - i.e. O(1) amortized
  - Can get O(1) worst-case union()
  - Would be nice if we could get O(1) worst-case find(), but…
  - *Known result*: both find() and union() can't have worst-case O(1)

Implementing the Disjoint Sets ADT (2 of 2)

- *Observation*:
  - Trees let us find many elements given a single root
- *Idea*:
  - If we reverse the pointers (ie, point up from child to parent), we can find a single root from many elements
- *Decision*:
  - One up-tree for each set
  - The ID of the set is (hash of) the tree root

Lecture Outline

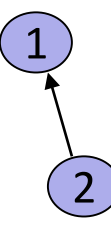
- Review: Disjoint Sets ADT
- Up-Tree Data Structure
  - **Representation**
  - Optimization: Weighted Union
  - Optimization: Path Compression

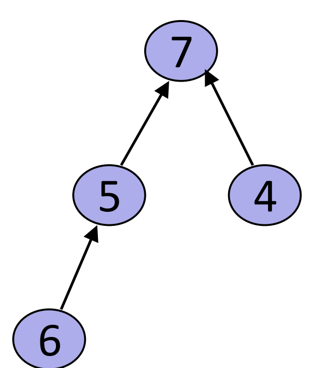
Lecture questions: pollev.com/cse332

Up-Tree Data Structure for Disjoint Sets ADT

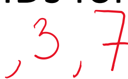
- Initial State:

- After several union()s:



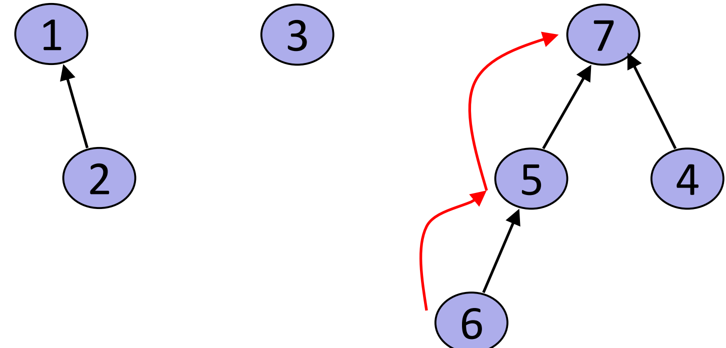


- Roots are the IDs for each set



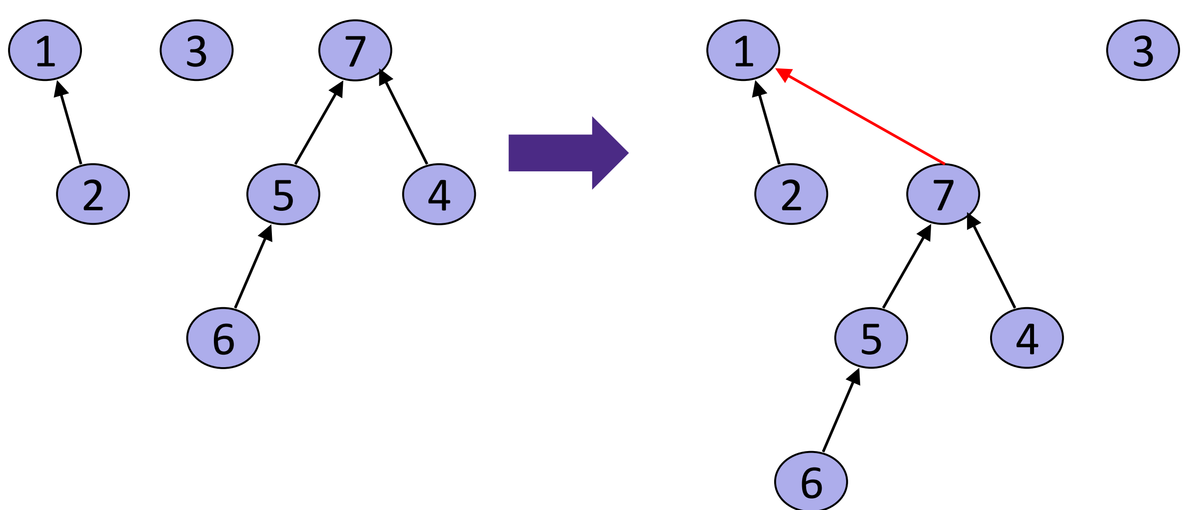
Up-Tree Find

- find(x): follow x to the root and return the root
  - Eg: find(6) = 7



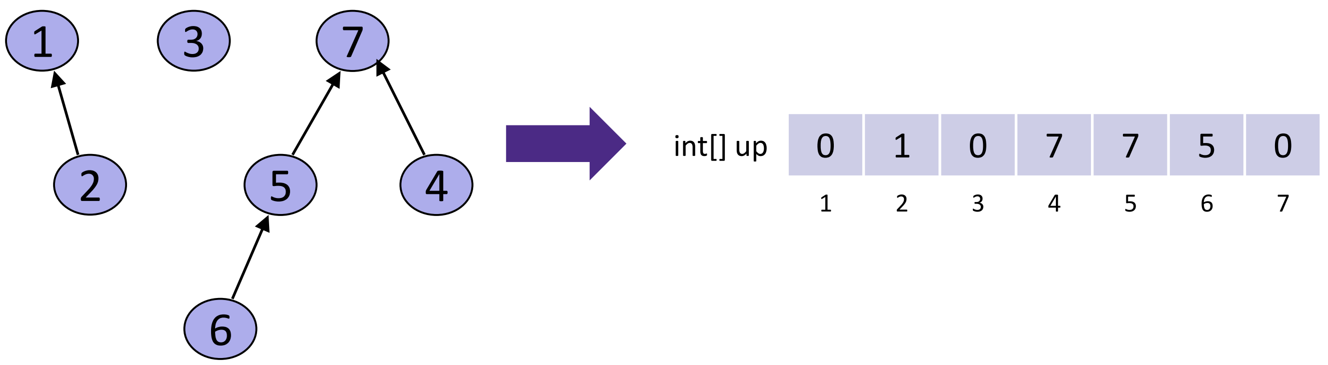
Up-Tree Union

- union(x): assuming x and y are roots, point y to x
  - If x or y are not roots, can require caller to call find() first
  - Eg: union(1, 7)



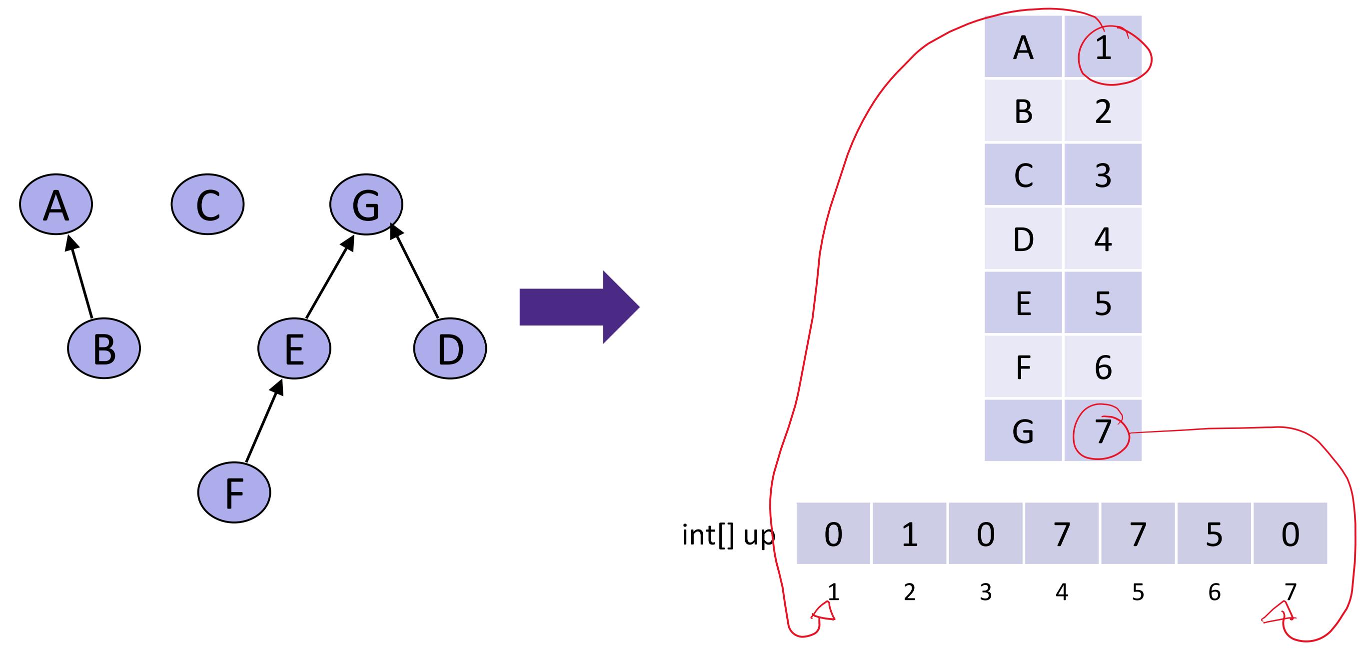
Up-Tree Representation (1 of 2)

- Up-Trees can be represented as an array of indices
  - up[x] = 0 means x is a root
  - Note: array is 1-indexed



Up-Tree Representation (2 of 2)

- Up-Trees can be represented as an array of indices
  - Can contain non-integer values using a hash table to map to indices



Up-Tree Implementation

```
void union(int x, int y) {
  up[y] = x;
}
```

```
int find(int x) {
  while (up[x] != 0) {
    x = up[x];
  }
  return x;
}
```

- Worst-case runtime for union():
- Worst-case runtime for find():
- Total runtime for n-1 union()s and m find()s:

pollev.com/cse332

- What is the runtime for …
  - union(), worst-case
  - find(), worst-case
  - n-1 union()s + m find()s

```
void union(int x, int y) {
  up[y] = x;
}
```

```
int find(int x) {
  while (up[x] != 0) {
    x = up[x];
  }
  return x;
}
```

A. $\Theta(1)$ / $O(1)$ / $O(n + m)$

B. $\Theta(1)$ / $O(h)$ / $O(n + mh)$

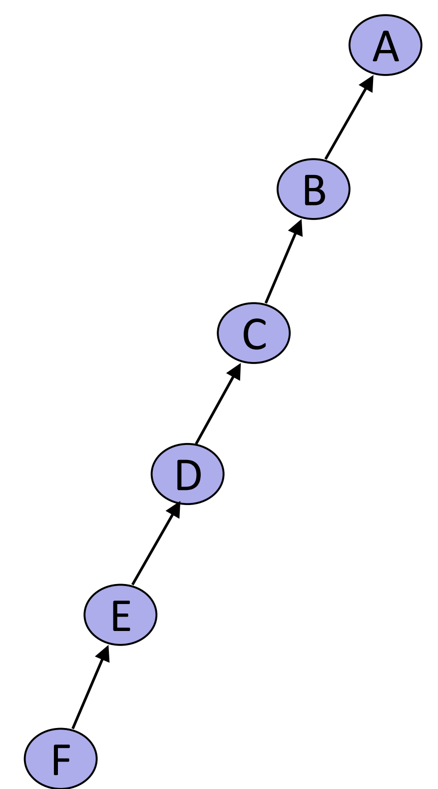
C. $\Theta(1)$ / $O(n)$ / $O(n + m^2)$

D. I'm not sure …

Tight bound uses h, but since we haven't bound h (yet), $h \in O(n)$

Worst-case Union

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



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

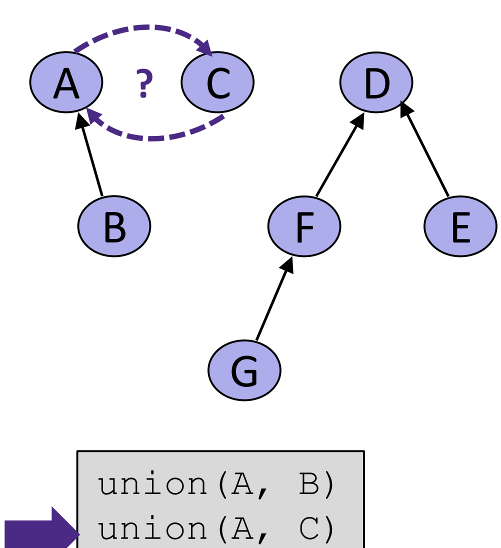
Lecture Outline

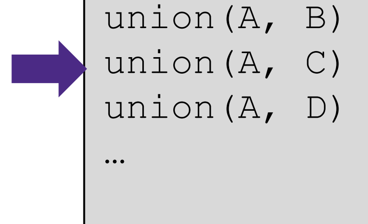
- Review: Disjoint Sets ADT
- Up-Tree Data Structure
  - Representation
  - **Optimization: Weighted Union**
  - Optimization: Path Compression

Lecture questions: pollev.com/cse332

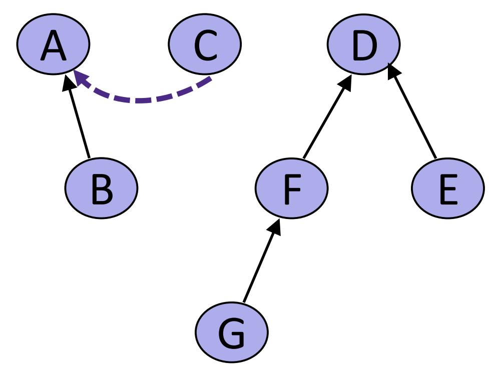
Weighted Union (1 of 2)

- Our naïve union() always picked the same argument (the second argument) to become the child in the unioned structure



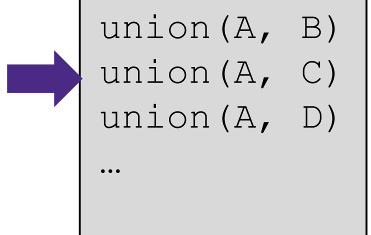


Weighted Union (2 of 2)

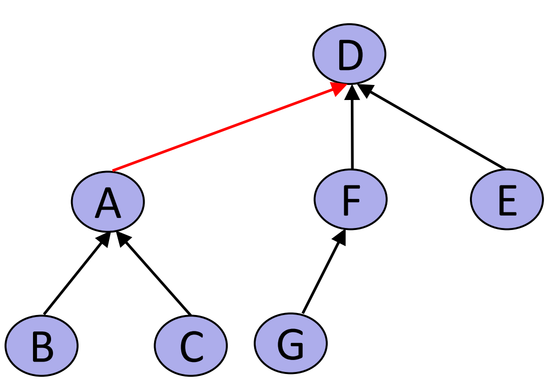


- Our naïve union() always picked the same argument (the second argument) to become the child in the unioned structure

- Instead:
  - Pick the smaller tree (ie, tree with fewer nodes) to be the new child
    - Let "weight" = "num nodes"
  - Add the new child to the root

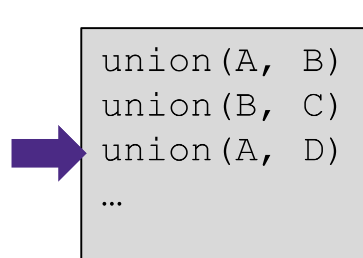


Weighted Union (2 of 2)



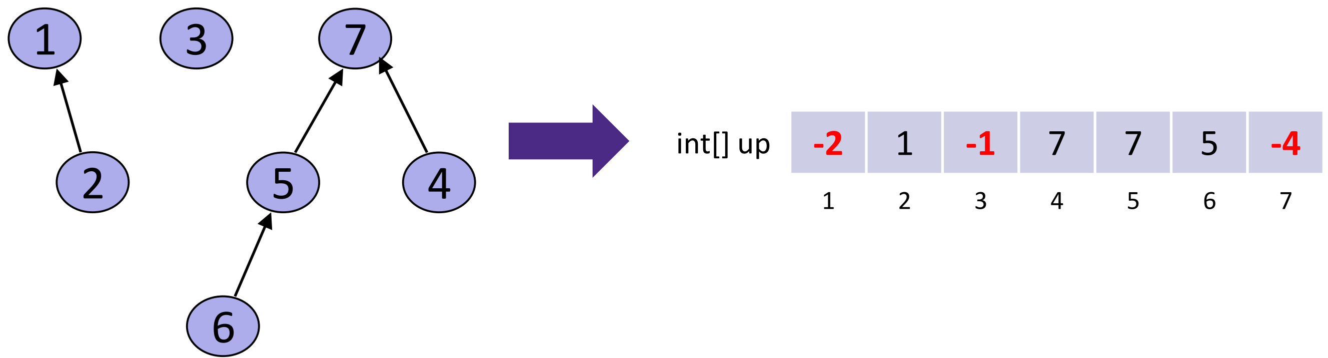
- Our naïve union() always picked the same argument (the second argument) to become the child in the unioned structure

- Instead:
  - Pick the smaller tree (ie, tree with fewer nodes) to be the new child
    - Let "weight" = "num nodes"
  - Add the new child to the root



Weighted Union: Representation

- Need to store *number of nodes* (or "weight") of each tree
- Instead of '0', we can store the root's weight instead!
  - Use negative values to indicate they're not indices
  - See Weiss, 8.4



Weighted Union: Implementation

```
void union(int x, int y) {
  up[y] = x;
}
```

```
weightedUnion(int x, int y) {
  wx = weight[x];
  wy = weight[y];
  if (wx < wy) {
    up[x] = y;
    weight[y] = wx + wy;
  } else {
    up[y] = x;
    weight[x] = wx +wy;
  }
}
```

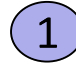
union()'s runtime is still O(1)!

Does this (slightly) added complexity help us balance the up-trees and improve find()?

Weighted Union: Performance

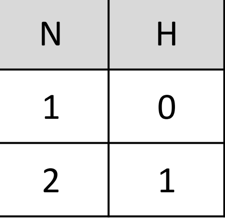
- Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are "spindly"

N	H
1	0

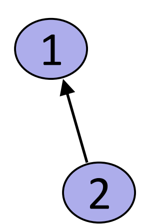


Weighted Union: Performance

- Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are “spindly”

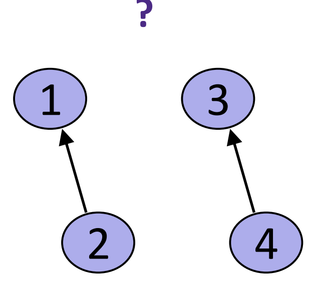


N	H
1	0
2	1



Weighted Union: Performance

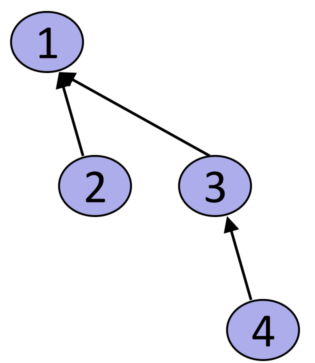
- Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are "spindly"



N	H
1	0
2	1
4	?

Weighted Union: Performance

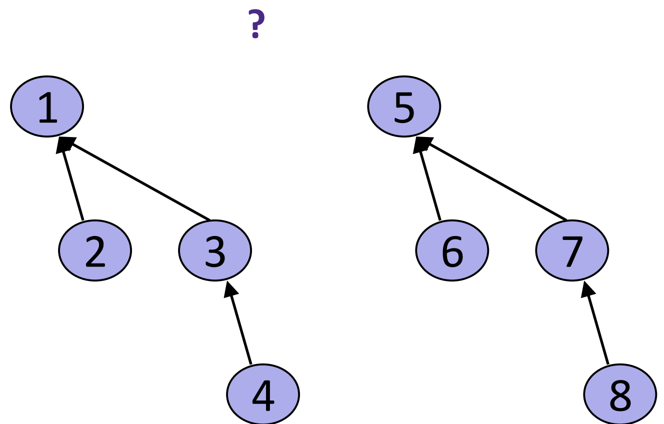
- Consider the worst case: tree height grows as fast as possible



N	H
1	0
2	1
4	2

Weighted Union: Performance

- Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are "spindly"



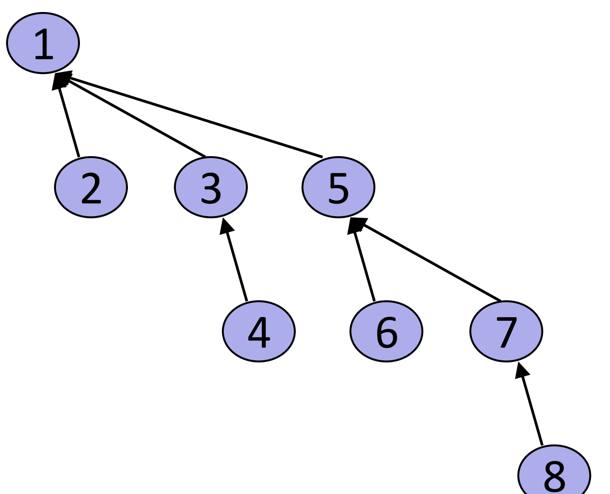
N	H
1	0
2	1
4	2
8	?

Weighted Union: Performance

- Consider the worst case: tree height grows as fast as possible
  - ie, up-tree and up-subtrees are "spindly"

N	H
1	0
2	1
4	2
8	3
2^n	n

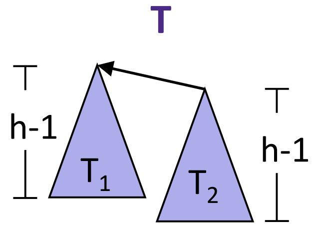
- Worst-case height and worst-case find() is Θ(log N)



Weighted Union Performance: Proof

- An up-tree with height h using weighted union has weight at least 2^h

- Proof by induction
  - *Base-case*: h = 0. The up-tree has one node and $2^0 = 1$
  - *Inductive step*: Assume true for all h' < h



Minimum weight up-tree of height h formed by weighted unions

We know:

$W(T_1) \geq 2^{h-1}$
$W(T_2) \geq 2^{h-1}$ } *Induction hypothesis*

$W(T_1) \geq W(T_2)$ } *Definition of weighted union*

Since $W(\mathbf{T}) = W(T_1) + W(T_2)$, we know that

$W(\mathbf{T}) \geq W(T_1) + W(T_2)$
$= 2^{h-1} + 2^{h-1}$
$= 2^h$

Therefore $W(\mathbf{T}) \geq 2^h$

Poll Everywhere

pollev.com/cse332

- What is the runtime for …
  - weighted union(), worst-case
  - find(), worst-case
  - n-1 union()s + m find()s

A. $\Theta(1)$ / $\Theta(1)$ / $O(n + m)$

B. $\Theta(1)$ / $\Theta(n)$ / $O(n + m^2)$

C. $\Theta(1)$ / $\Theta(\log n)$ / $O(n + m^2)$

D. $\Theta(1)$ / $\Theta(\log n)$ / $O(n + m \log n)$

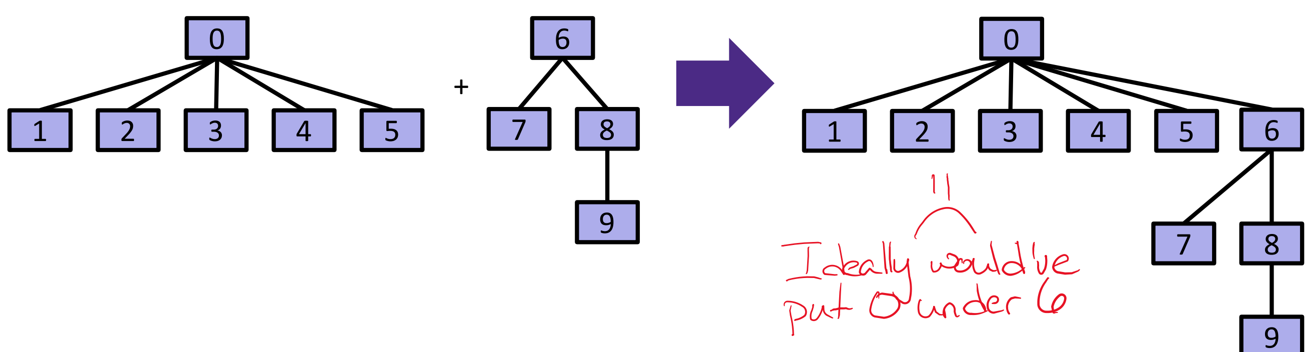
E. I'm not sure …

```
weightedUnion(int x, int y) {
  wx = weight[x];
  wy = weight[y];
  if (wx < wy) {
    up[x] = y;
    weight[y] = wx + wy;
  } else {
    up[y] = x;
    weight[x] = wx +wy;
  }
}
```

```
int find(int x) {
  while (up[x] > 0) {
    x = up[x];
  }
  return x;
}
```

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?
  - Heighted Union's runtime is asymptotically the same: Θ(log(N))
    - Proof is left as an exercise to the reader ;)



  - Easier to track weights than heights, and heighted union doesn't combine very well with the next optimization technique for find()

Lecture Outline

- Review: Disjoint Sets ADT
- Up-Tree Data Structure
  - Representation
  - Optimization: Weighted Union
  - **Optimization: Path Compression**

Lecture questions: pollev.com/cse332

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 AVL tree invariants
  - **Promoting keys / splitting leaves**: preserve B-tree invariants (eg, L+1 keys stored in a leaf node)

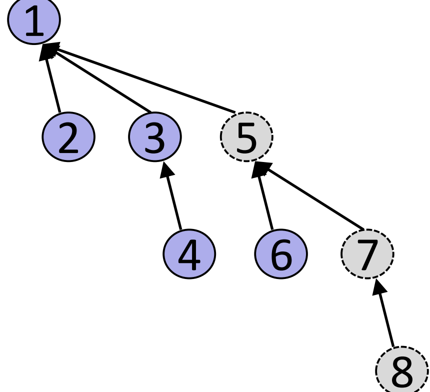
- Notably, the modifications don't improve runtime *between identical method calls*
  - If avl.find(x) takes 2 µs, we expect future calls to take ~2 µs
  - If we call avl.find(x) m times, the total runtime should be ~2m µs

Modifying Data Structures for Future Gains

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

Path Compression: Idea

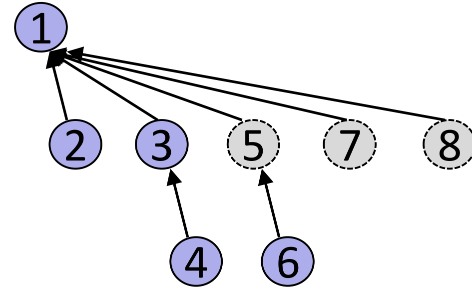
- Recall the worst-case structure if we use weighted union:



- *Idea*: When we find(8), move all visited nodes under the root
  - Additional cost is insignificant (same order of growth) , so run path compression on every find()

Path Compression: Example

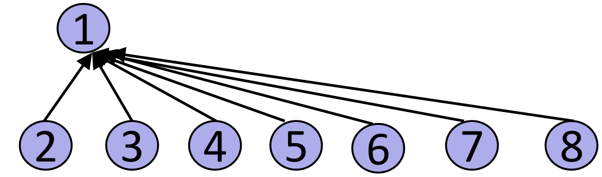
- Recall the worst-case structure if we use weighted union



- *Idea*: When we find(8), move all visited nodes under the root
  - Additional cost is insignificant (same order of growth), so run path compression on every find()
  - Doesn't meaningfully change runtime for *this* invocation of find(8), but *subsequent* find(8)s (and subsequent find(7)s and find(5)s and …) will be faster!

Path Compression: Details and Runtime

- With "enough" path compression aka "enough" find()s …



  - How much is "enough"? Probably m>n (ish. No guarantee the find()s are on unique elements)

Path Compression: Implementation

```
int find(int x) {
  while (up[x] != 0) {
    x = up[x];
  }
  return x;
}
```

```
int pathCompressionFind(int x) {
  while (up[x] > 0) {
    x = up[x];
  }
  int root = x;

  // Change the parent for all
  // nodes along this path
  while (up[x] > 0) {
    x = up[x];
    up[x] = root;
  }
  return root;
}
```

find()'s worst-case runtime is still O(log n)!

Does this (slightly) added complexity help us make the up-trees shallower and improve sequences of find()?

Path Compression: Runtime

- In total, m>1 find()s will take O(m log*n) time
  - log*n is the "iterated log": the number of times you need to apply log to n before the result is <=1
  - See Weiss for proof

- find()s and union()s are now amortized constant time
  - log*n < 5 for any realistic n
  - If log*n ≈ 5, then m 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

Interlude: A Really Slow Function

- Ackermann's function is a really big function A(x, y) with inverse α(x, y) which is really small

- α shows up in:
  - Computation Geometry (surface complexity)
  - Combinatorics of sequences

- How fast does α(x, y) grow?
  - Even slower than iterated log!
  - For all practical purposes, $\alpha(x, y) < 4$

Path Compression: Tighter Runtime

- A sequence of p union()s + find()s on a set of n elements has worst-case time of $O(p \cdot \alpha(p, n))$
  - Assumes weighted union and path compression
  - Proved by Robert Tarjan in 1984 (see also Fibonacci heaps and splay trees)

- So find()'s amortized runtime is O(1)
  - Since $O(p \cdot 4)$ for p operations!

- Complex analysis, but inverse-Ackermann's is a tighter bound than iterated-log

Summary

- Worst-case runtime for up-trees with weighted union and path compression:
  - union is O(1)
  - find is O(log n)
- Total time for $m \geq n$ operations on n elements is $O(m \cdot \alpha(m, n))$
  - An individual operation can be costly, but over time the "average" cost per operation is not
  - Logic is similar to array-resizing: an individual insert() can be costly *(due to resizing)*, but over time the "average" insert() is not
- Using "ranked union" gives an even better bound theoretically