

CS5800 – ALGORITHMS

MODULE 5. DATA STRUCTURES & GRAPHS

Lesson 4: Union-Find

Ravi Sundaram

Topics

- Union-Find or Disjoint Set data structure – what & why
- Implementation as forest of up-trees
 - Union
 - Find
- Improvements
 - Union by rank or size
 - Path compression during Find
- Summary

Union-Find Data Structure

- Keeps track of a set of elements partitioned into disjoint subsets
- Initially, each element e is a set in itself: e.g., $\{\{e_1\}, \{e_2\}, \{e_3\}\}$
- Union(x, y) – Combine two sets x and y into a single set
 - Before: $\{\{e_3, e_5, e_7\}, \{e_4, e_2, e_8\}, \{e_9\}, \{e_1, e_6\}\}$
 - After Union(e_5, e_1): $\{\{e_3, e_5, e_7, e_1, e_6\}, \{e_4, e_2, e_8\}, \{e_9\}\}$
- Find(x) Determine which set a particular element is in
 - Each set has a unique name; name is arbitrary; what matters is: $\text{find}(a) == \text{find}(b)$ iff a and b are in the same set
 - Usually, one of the members of the set is the "representative" (i.e. name) of the set: e.g., $\{\{e_3, e_5, e_7, e_1, e_6\}, \{e_4, e_2, e_8\}, \{e_9\}\}$; Find(e_1) = e_5 ; Find(e_4) = e_8

Union-Find Data Structure: Applications

- Keeping track of connected components in undirected graph
- Maze construction
- Forming equivalence classes
- Unification – solving symbolic equations in logic
- Computational Geometry – analysis involving Davenport-Schinzel sequences
- Kruskal's algorithm for Minimum Spanning Tree
- ...

Key idea: use up-trees

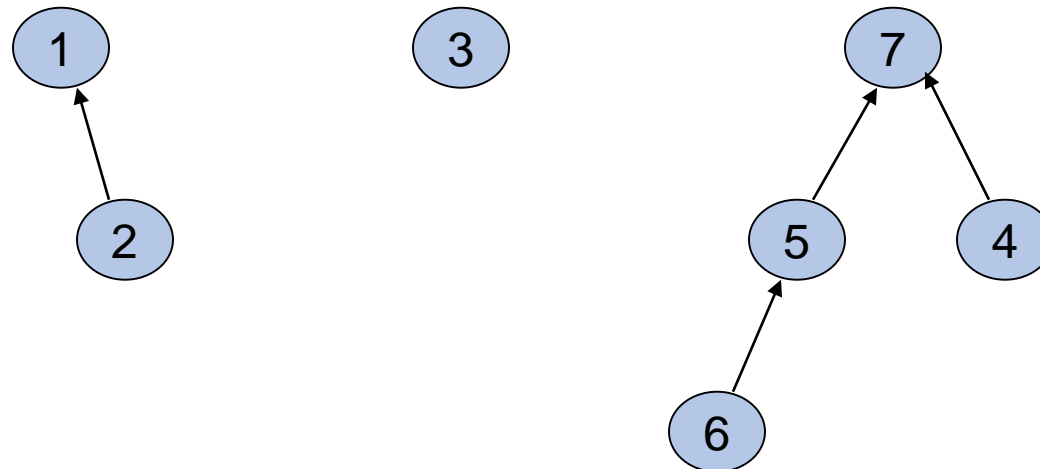
- Why trees?
- *Trees* let us find many elements given one root (i.e. representative)
- Why up-trees?
- If we *reverse* the pointers (make them point up from child to parent), we can find a single root from any element in the tree

Initial state



Diagram showing seven separate nodes labeled 1 through 7, representing the initial state where each element is its own root.

Intermediate state



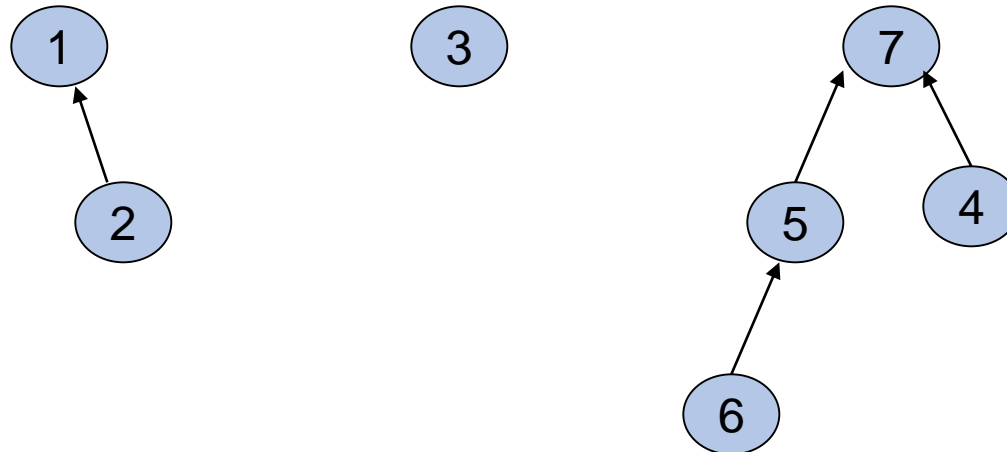
Roots are the names
of each set.

Array Implementation

- Array of indices

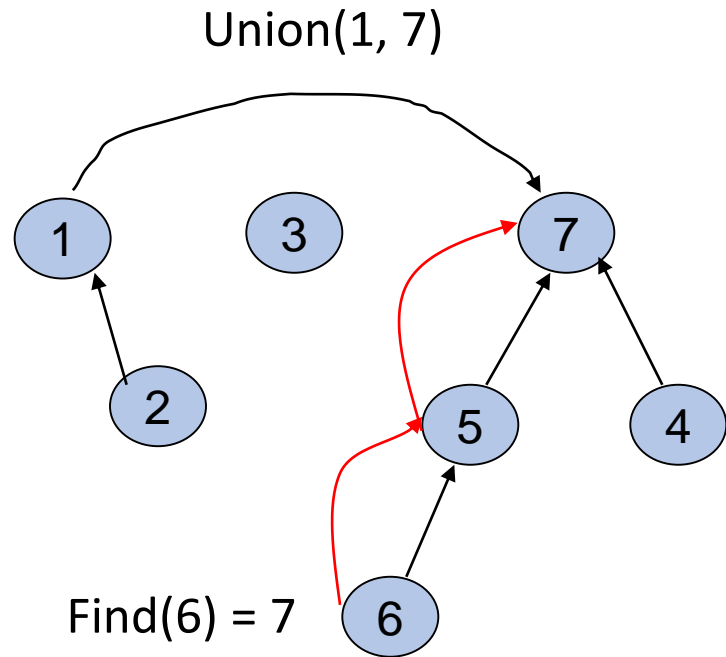
	1	2	3	4	5	6	7
up	0	1	0	7	7	5	0

Up[x] = 0 means
x is a root.

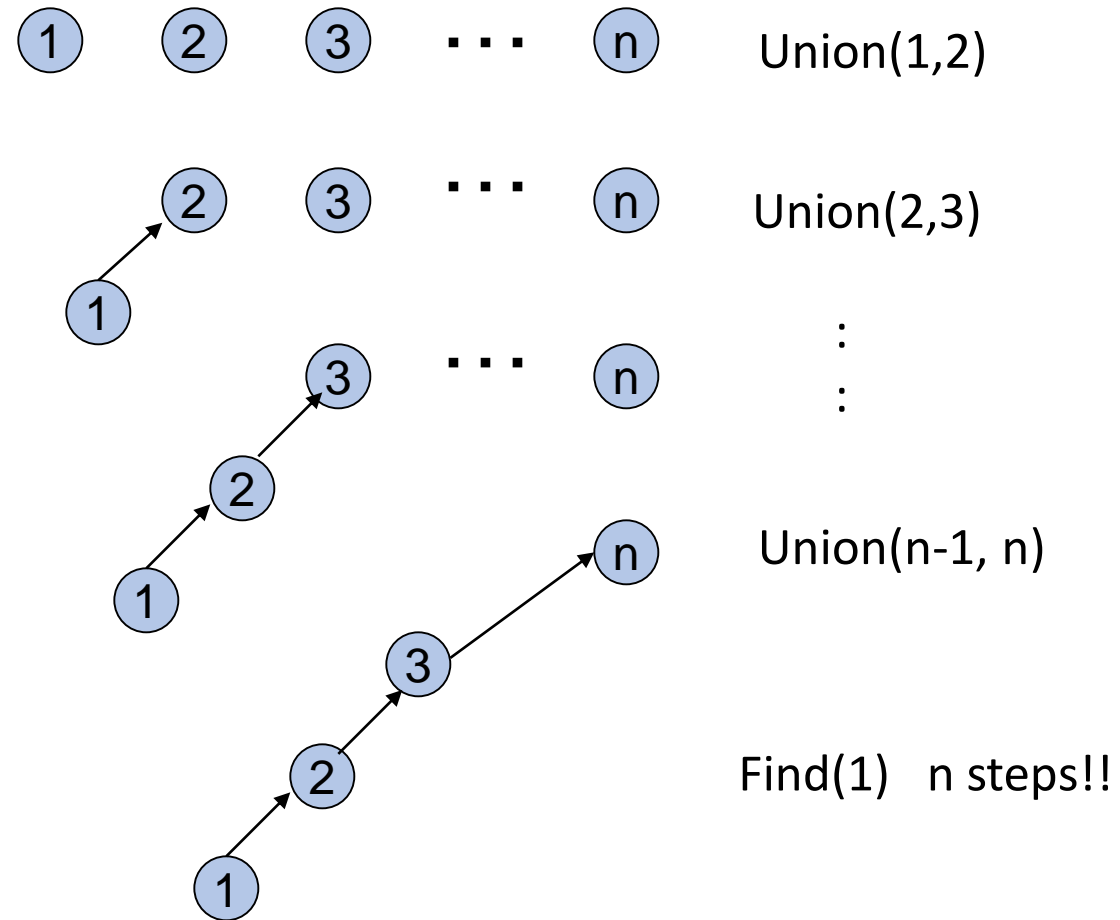


Implementation

- $\text{Union}(x, y)$ – assuming x and y roots, points x to y
- $\text{Find}(x)$: follows x to root and returns root

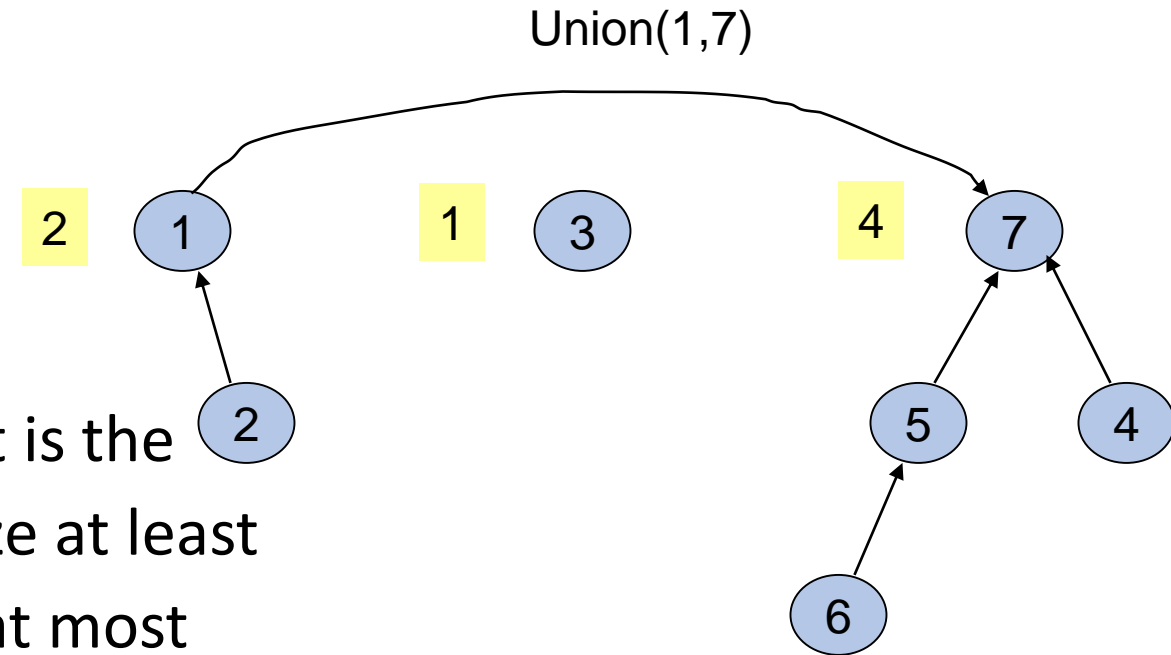


Bad Example

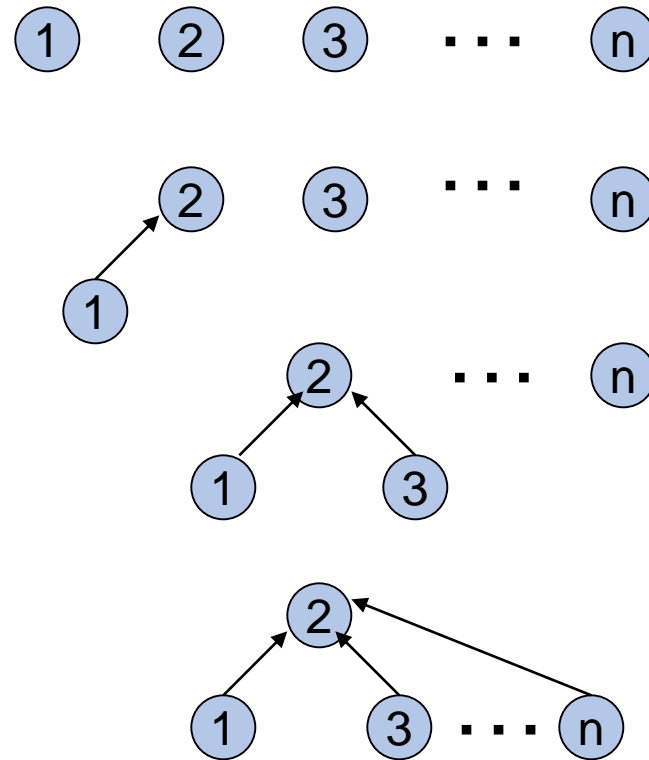


Improvements: Union by Size

- How to reduce the time *find* takes?
- Observation: the time *find* takes depends on the depth.
- How do we keep the trees shallow?
- Union by size (aka union by rank)
 - Keep track of size of each component
 - Point the smaller tree to the larger tree
- Theorem: *find* takes $O(\log n)$ time.
- Pf: The depth of a tree increases only when it is the smaller of the two in a Union; and then its size at least doubles. Given n nodes the size can double at most $O(\log n)$ times. ■



Example Again



$\text{Union}(1, 2)$

$\text{Union}(2, 3)$

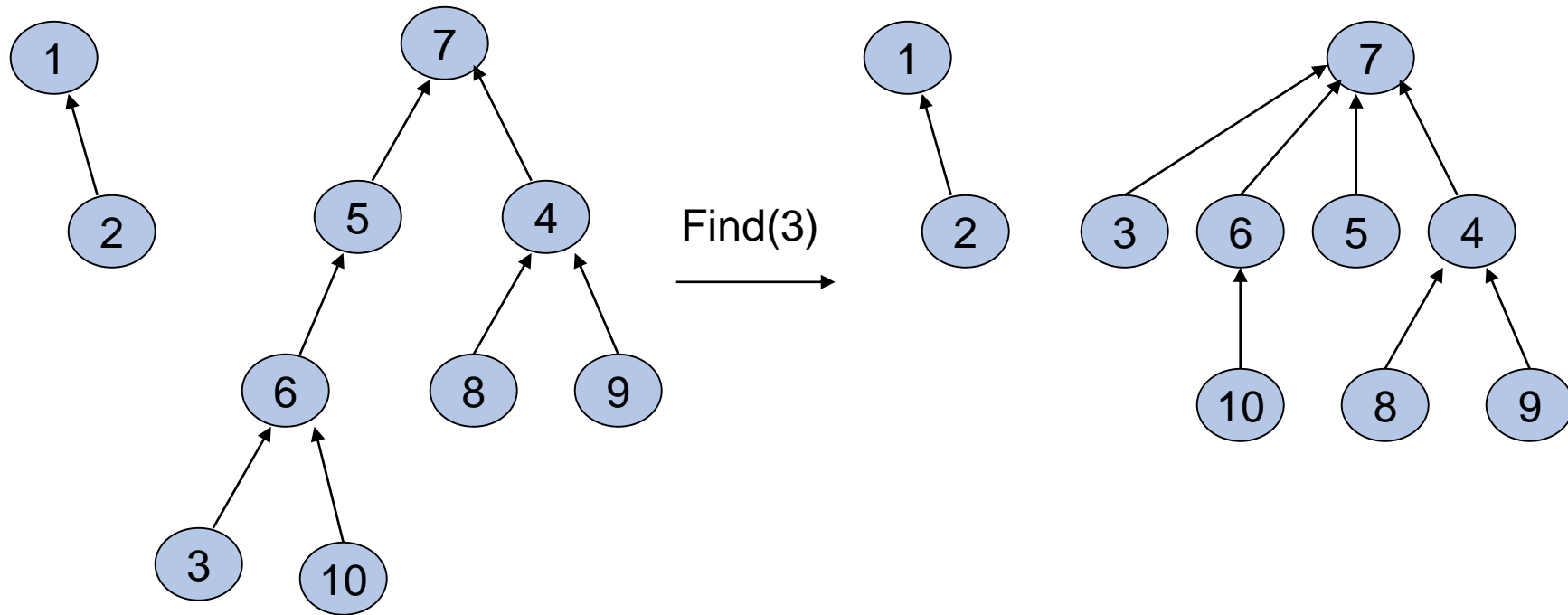
\vdots

$\text{Union}(n-1, n)$

$\text{Find}(1)$ constant time

Improvements: Path Compression

- On a Find operation point all the nodes on the search path directly to the root.



Improvements: Amortized Complexity

- With the two improvements – union by rank and path compression –
Time complexity for $m \geq n$ Union/Find operations on n elements is $\Theta(m \alpha(m, n))$
 - $\alpha(m, n)$ is the inverse Ackerman function and essentially a constant.
 - The proof is beyond the scope of this course
- For disjoint union / find with union by rank and path compression
 - average time per operation, or amortized complexity is essentially a constant
 - worst case time for a Union is $\Theta(1)$ and Find is $\Theta(\log n)$
- As we will see later this means the bottleneck of Kruskal's actually becomes the sorting of the edges

Summary

- A collection of disjoint sets can be maintained using a forest of up-trees
- The implementation can be improved through two optimizations
 - Union by size
 - Path compression
- The amortized complexity of m Union/Find operations is essentially a constant.
- Main Takeaway: Union/Find or Disjoint Set data structure can be implemented to achieve (almost) constant amortized complexity.