M, 10/14/19

Fall'19 CSCE 629

Analysis of Algorithms

Fang Song
Texas A&M U

Lecture 18

- An excursion to data structures
- Amortized analysis

Credit: based on slides by K. Wayne

Recall: priority queue for Dijkstra's algorithm

PriorityQueue Q: set of n elements w. associated key values (alarm)

- Change-key(x). change key value of an element
- Delete-min. Return the element with smallest key, and remove it.
- Can be done in $O(\log n)$ time (by a heap)

Dijkstra(G, s) // initialize $d(s) = 0$, others $d(u) = \infty$
Make Q from V using $d(\cdot)$ as key value
While Q not empty
    $u \leftarrow$ Delete–min(Q)
    // pick node with shortest distance to s
    For all edges $(u, v) \in E$
        If $d(v) > d(u) + l(u, v)$
            $d(v) \leftarrow d(u) + l(u, v)$ and Change-key(v)

$O(n\log n)$

$O(m\log n)$

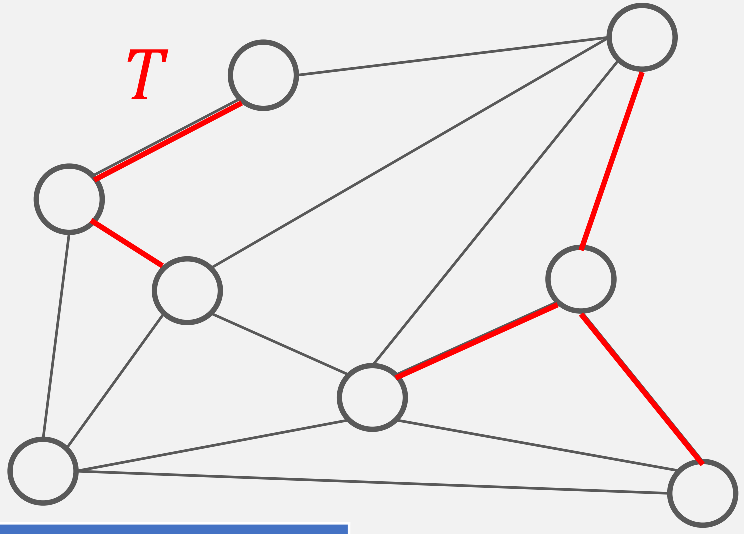
Dijkstra
$O((m + n)\log n)$

Further improvement possible by Fibonacci heap [More to come]

N.B. BFS uses ordinary Queue. Dijkstra = BFS+Priority Queue

Recall: disjoint-set for Kruskal's algorithm

- Disjoint-set (aka Union-Find) data structure
  - Make−Set(x): create a singleton set containing x
  - Find−Set(x): return the "name" of the unique set containing x
  - Union(x, y): merge the sets containing x and y respectively



	Linked list	Balanced tree
Find (worst-case)	$\Theta(1)$	$\Theta(\log n)$
Union (worst-case)	$\Theta(n)$	$\Theta(\log n)$
Amortized analysis: k unions and k finds, starting from singleton	$\Theta(k \log k)$	$\Theta(k \log k)$

Today

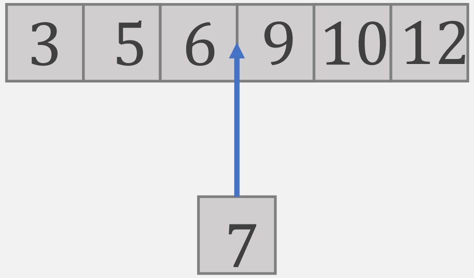
A taste of data structures & amortized analysis

Implementing Priority Queue

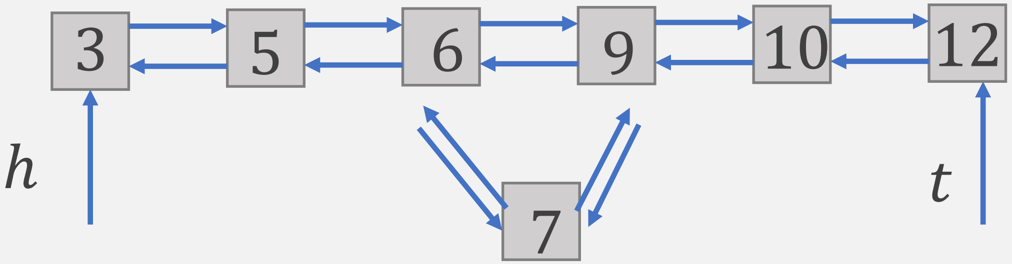
PriorityQueue: set of n elements w. associated key values

- Change-key. change key value of an element
- Delete-min. Return the element with smallest key, and remove it.
- Insert/Delete
- Goal: $O(\log n)$ time worst-case

- (Sorted) Array?
  - ☺ Change-key: $O(1)$?
  - ☹ Insert: $\Omega(n)$

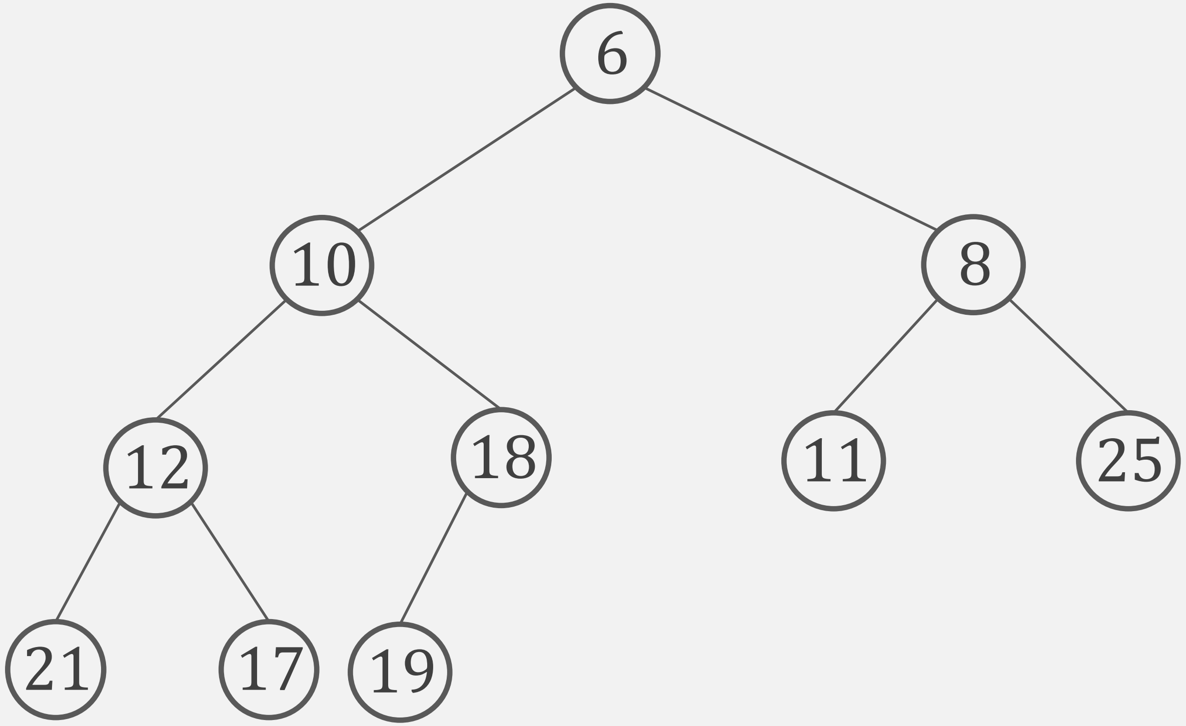


- (Sorted) Linked list?
  - ☺ Delete-min: $O(1)$
  - ☹ Insert: $\Omega(n)$



Binary heaps

- Binary complete tree. Perfectly balanced, except for bottom level
- Heap-ordered tree. For every node, $key(child) \geq key(parent)$
- Binary heap. Heap-ordered complete binary tree

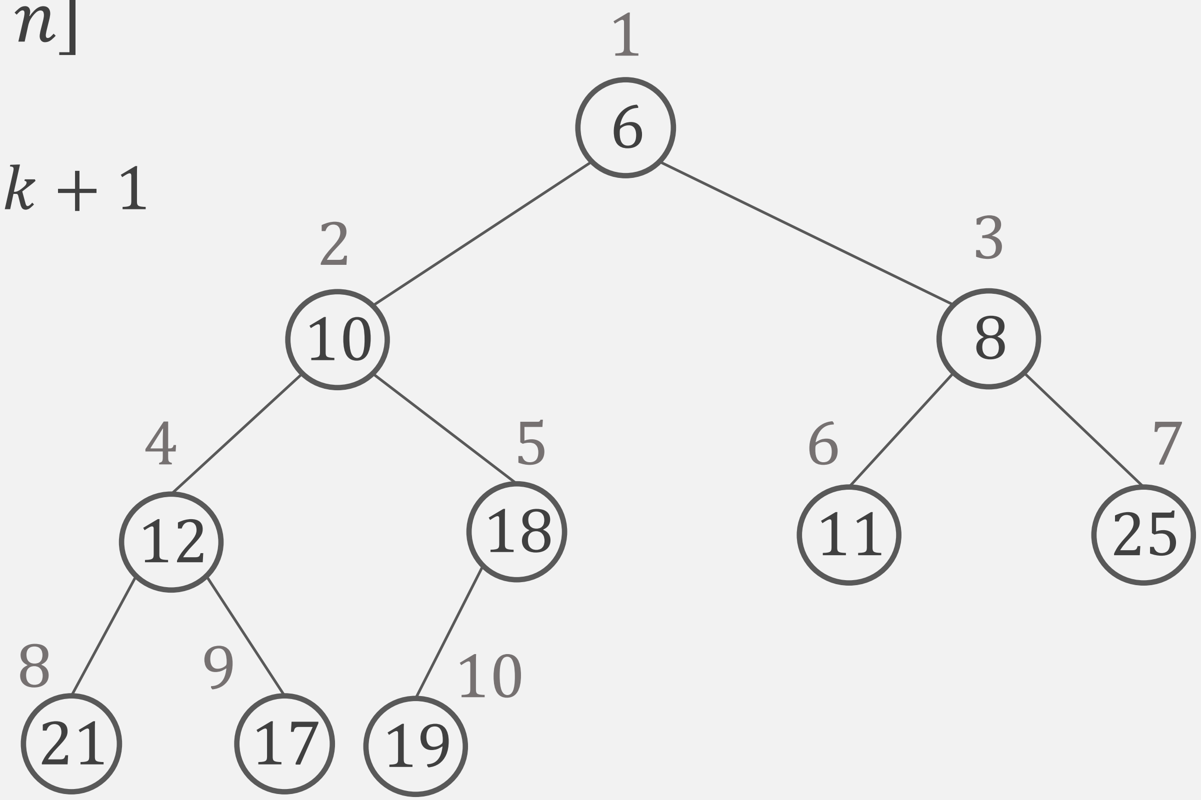


https://photos.com/featured/doum-palm-hyphaene-coriacea-and-james-warwick.html?product=poster

Representing a binary heap

- Array representation. $H[1,2,\dots,n]$
  - Parent of node at k is at $\lfloor k/2 \rfloor$
  - Children of node at k is at $2k$ and $2k+1$

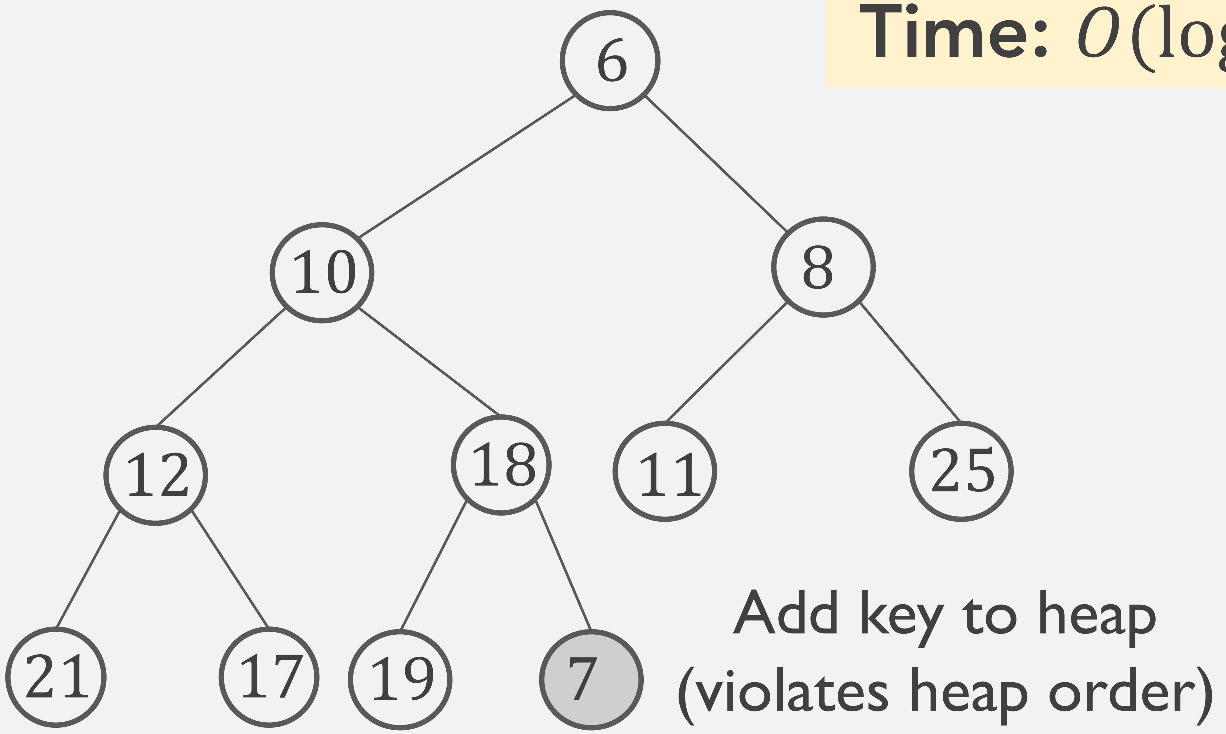
	1	2	3	4	5	6	7	8	9	10	11
H	6	10	8	12	18	11	25	21	17	19	

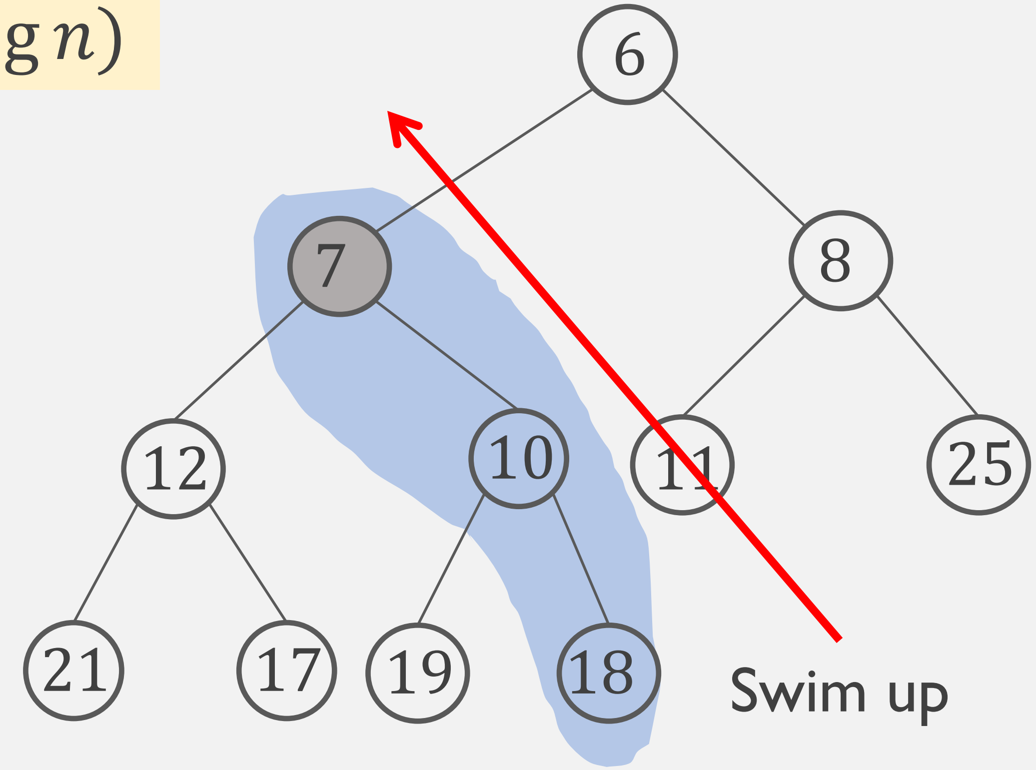


Binary heap: Insert

- Insert. Add new node at end; repeatedly exchange new node with its parent until heap order is restored.

Time: $O(\log n)$

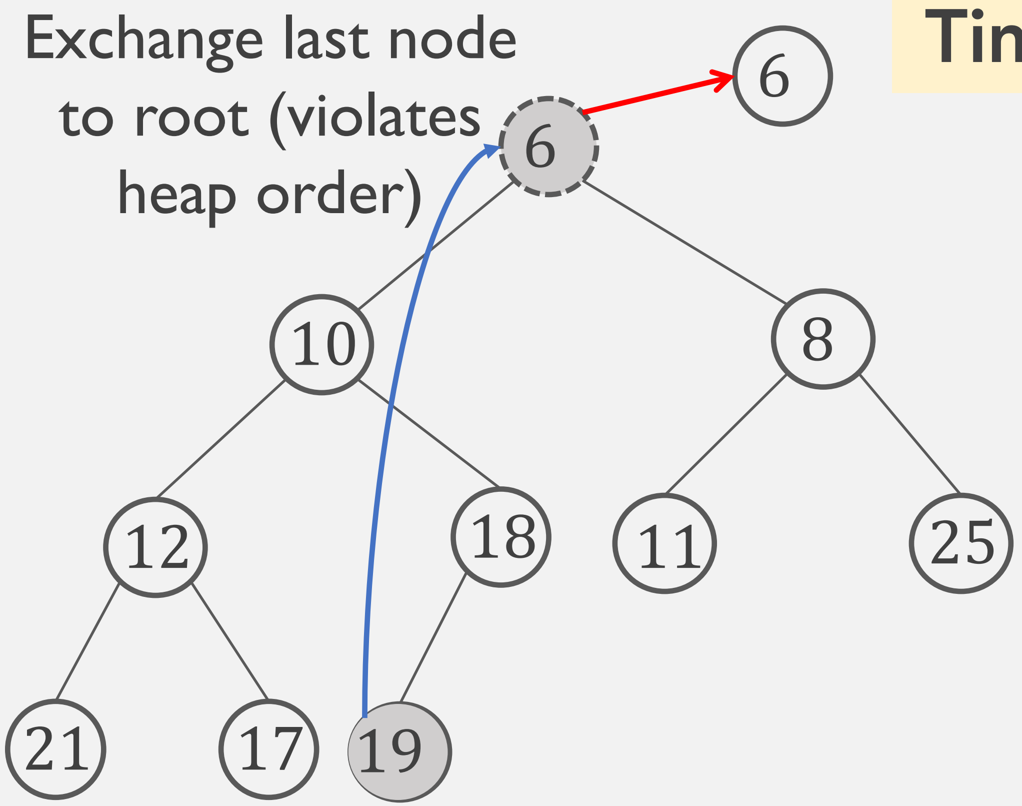


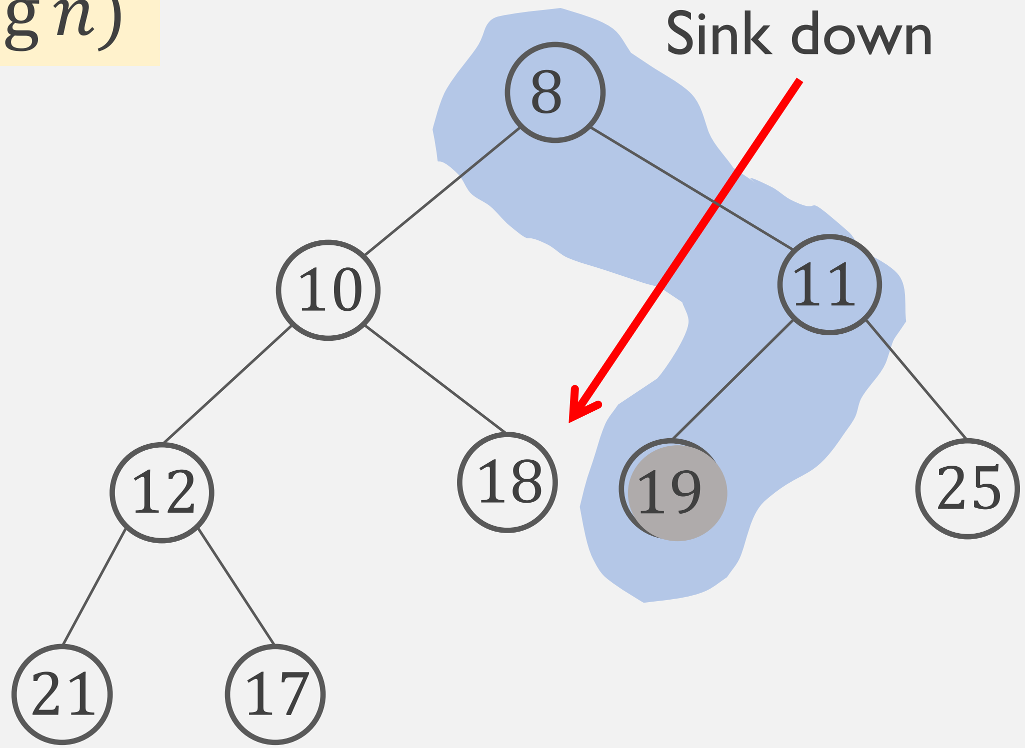


Binary heap: Delete-min

- Extract Min at root; upgrade last node to root and "heapify" it!

Time: $O(\log n)$





Implementing priority queue

Operation	Linked list	Binary heap	Fibonacci Heap*
Insert	$O(n)$	$O(\log n)$	$O(1)$
Delete-min	$O(1)$	$O(\log n)$	$O(\log n)$
Change-key	$O(n)$	$O(\log n)$	$O(1)$

Disjoint-set data structure

- Goal. Three operations on a collection of disjoint sets.
  - $\text{Make}-\text{Set}(x)$: create a singleton set containing x
  - $\text{Find} - \text{Set}(x)$: return "name" of the unique set containing x
  - $\text{Union}(x, y)$: merge the sets containing x and y respectively

- Performance parameters
  - k=number of calls to the three op's
  - n=number of elements

Simple implementation by an array

- Array $Component[x]$: name of the set containing x
  - FIND(x): $O(1)$
  - UNION(x, y): $\Theta(n)$ update all nodes in sets containing x and y

- Some improvement
  - Maintain the list of elements in each set.
  - Choose the name for the union to be the name of the larger set [so changes are fewer]

☹ UNION(x, y): still $\Theta(n)$ in the worst-case

But this rarely happens…
can we refine the analysis?

Amortized analysis

- Amortized analysis. Determine worst-case running time of a sequence of k data structure operations.
  - Standard (worst-case) analysis can be too pessimistic if the only way to encounter an expensive operation is when there were lots of previous cheap operations

---

Theorem. A sequence of k Union costs $O(k\log k)$. [contrast w. $O(k^2)$]

- Pf. [Aggregate method]
  - Start from singletons. After k unions, at most $2k$ nodes involved.
  - Any $Component[x]$ changes only when merged with a larger set;
  - i.e., change of name implies doubling of the set size;

  ➔ For any x, # changes at most $\log_2(2k)$

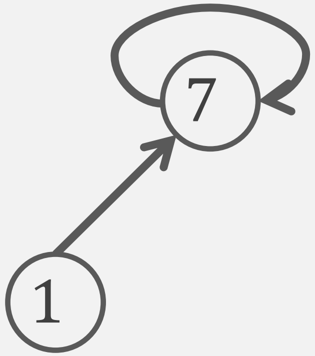
  ➔ $O(k \log k)$ for a sequence of k Unions [i.e., each has amortized cost $O(\log k)$].

Parent-link representation

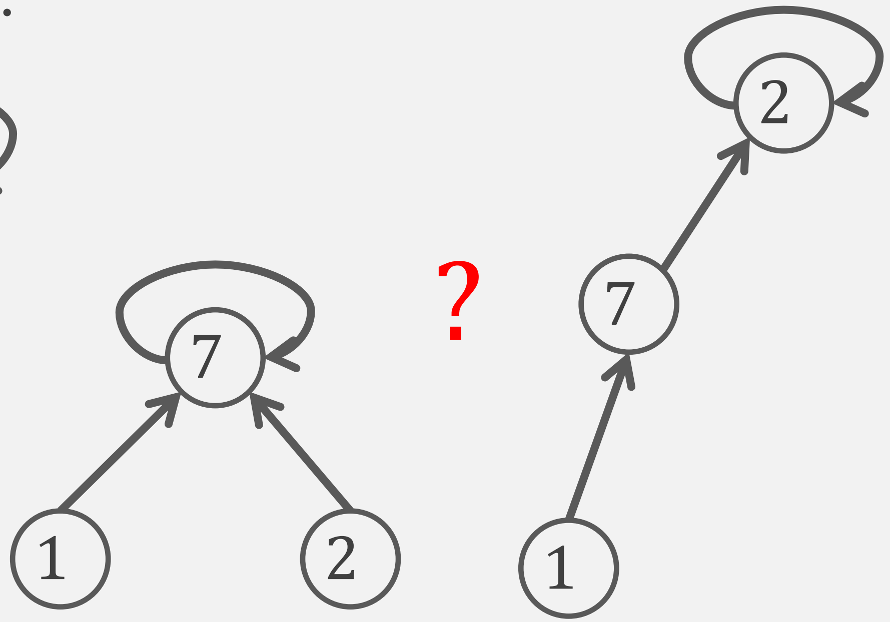
- Represent each set as a tree
  - Each element has an explicit parent pointer in the tree
  - The root (points to itself) serves as the "name"
  - FIND(x): find the root of the tree containing x
  - UNION(x, y): merge trees containing x and y.

Make-set

Union(1,7)

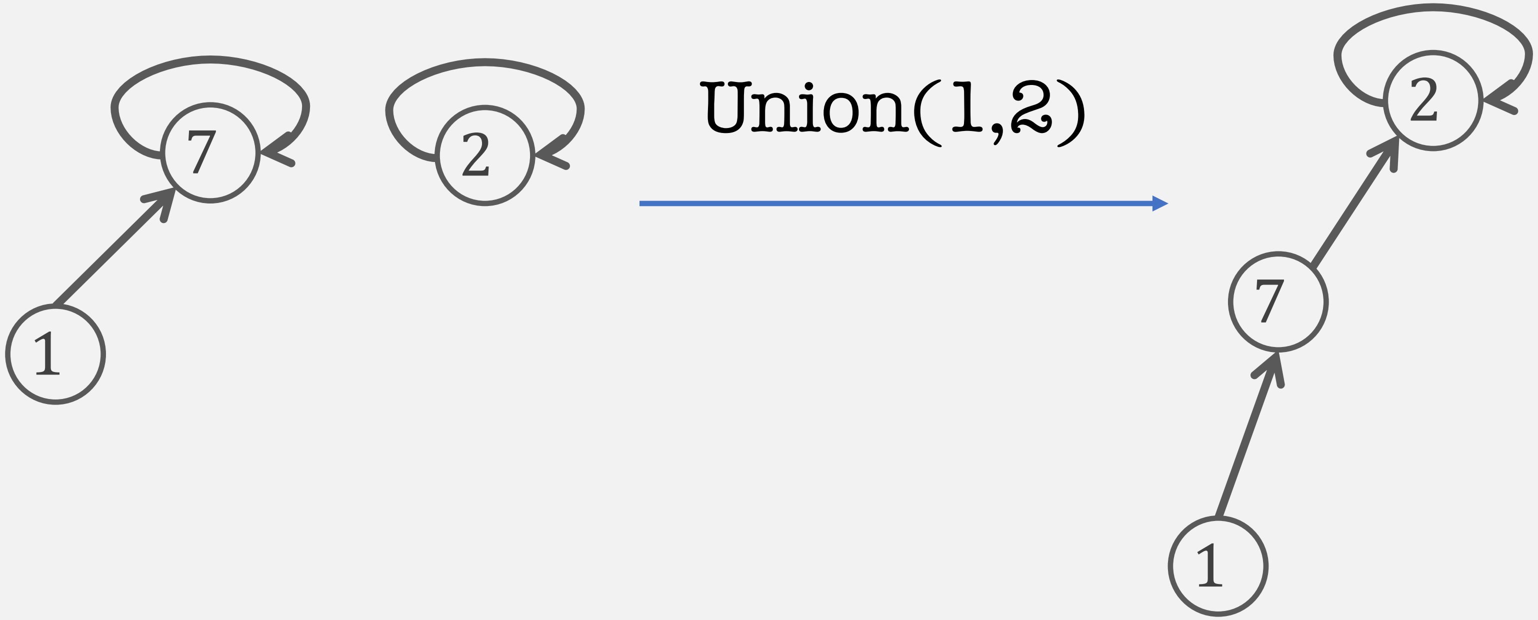


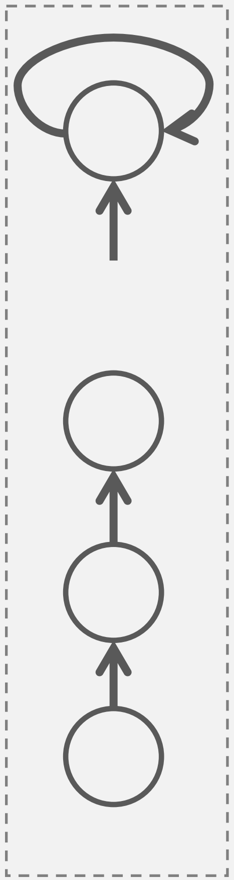
Union(1,2)



Naïve linking

- **Naïve linking:** link root of first tree to root of second tree

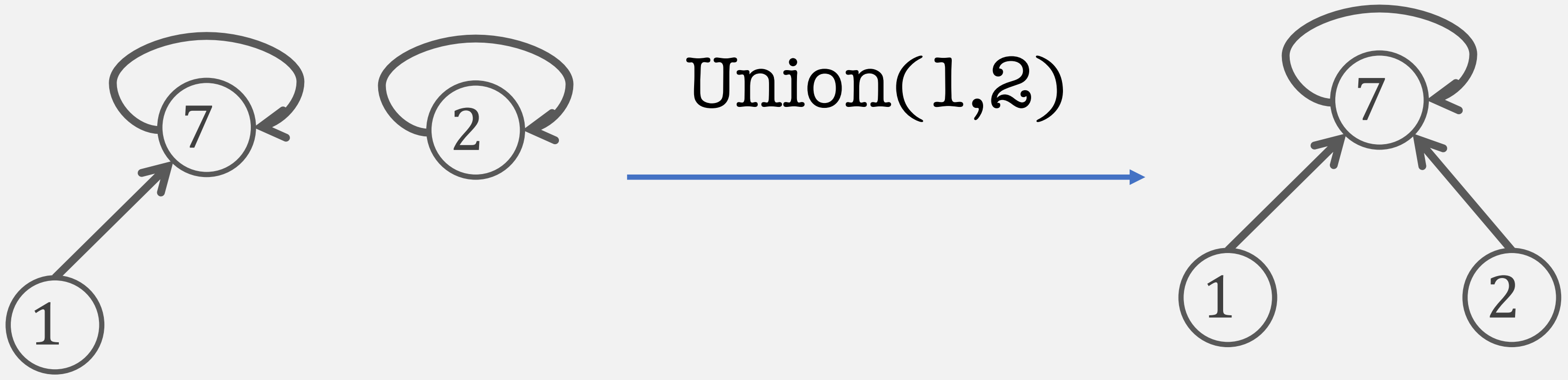




- **Observation.** A Union can take $\Theta(n)$ in the worst case
  - Find root of this tree: determined by the height of the tree

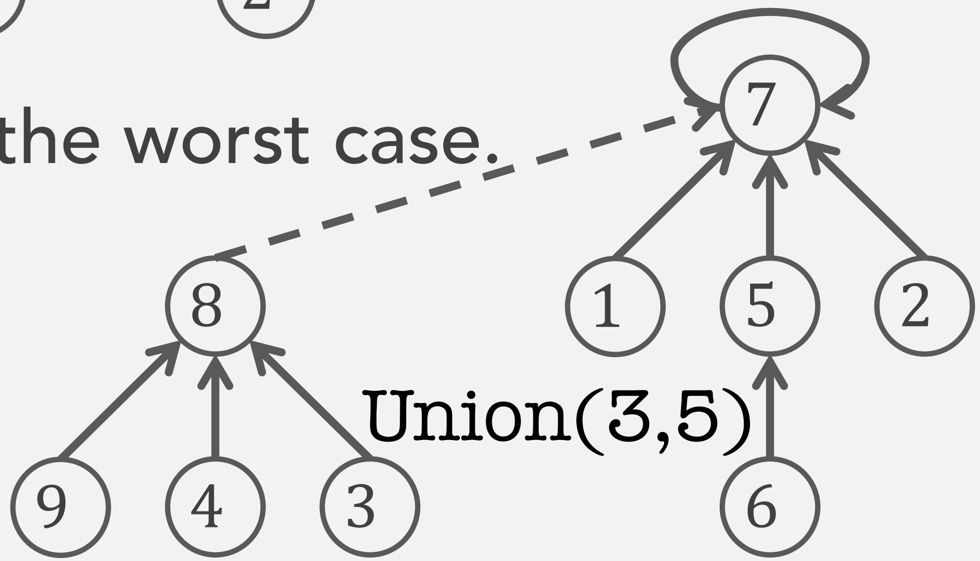
Link-by-size

- Link-by-size: maintain a tree size (# of nodes in the set) for each root node; link smaller tree to larger



- Observation. Union takes $O(\log n)$ in the worst case.
- Pf. [NB. time ∝ height]
  - (By Induction) For every root node r:
    $size[r] \geq 2^{height(r)}$

  ➔(worst-case) height $\leq \log n$



Disjoint-set summary

	Array / Naïve linking	Link-by-Size (Balanced tree)	Link-by-Size w. path-compressing
Find (worst-case)	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Union (worst-case)	$\Theta(n)$	$\Theta(\log n)$	$\Theta(\log n)$
Amortized cost: k unions and k finds, starting from singleton	$\Theta(k \log k)$	$\Theta(k \log k)$	$\Theta(k\alpha(k))$

$\alpha(n)$: inverse Ackermann function;
≤ 4 for any practical cases