Portland State University

W'21 CS 584/684

Algorithm Design & Analysis

Fang Song

Lecture 12

- Minimum spanning tree
- Amortized analysis

Greedy algorithms for MST

- Kruskal's. Start with $T = \emptyset$. Insert edges in ascending order of weights, unless it creates a cycle.
- Reverse-Delete. Start with $T = E$. Remove edges in descending order of weights, unless it disconnects T.

Edge-driven

- Prim's. Start with some node s. Grow a tree T from s outward. Add v to T such that $w(u, v)$ cheapest and $u \in T$.

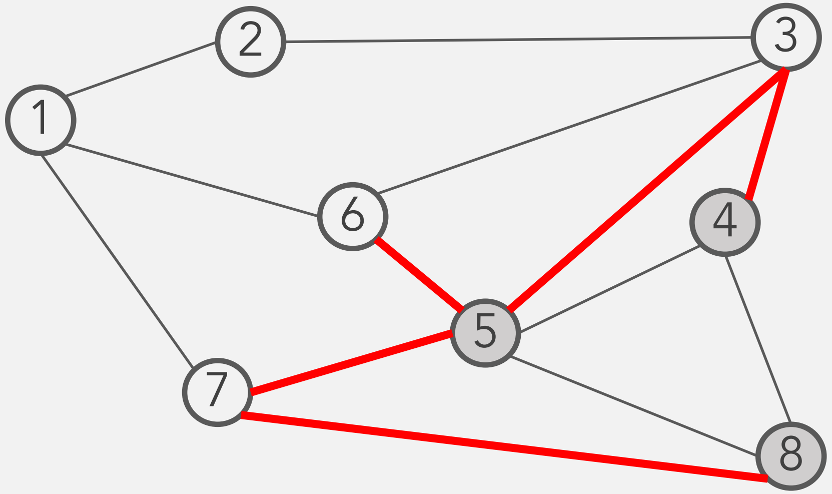
Node-driven

Sounds familiar? Dijkstra's?

☺ In this extremely lucky case, all of them work! But correctness proofs are non-trivial. We need the following tools to prove them.

Cycles and cuts

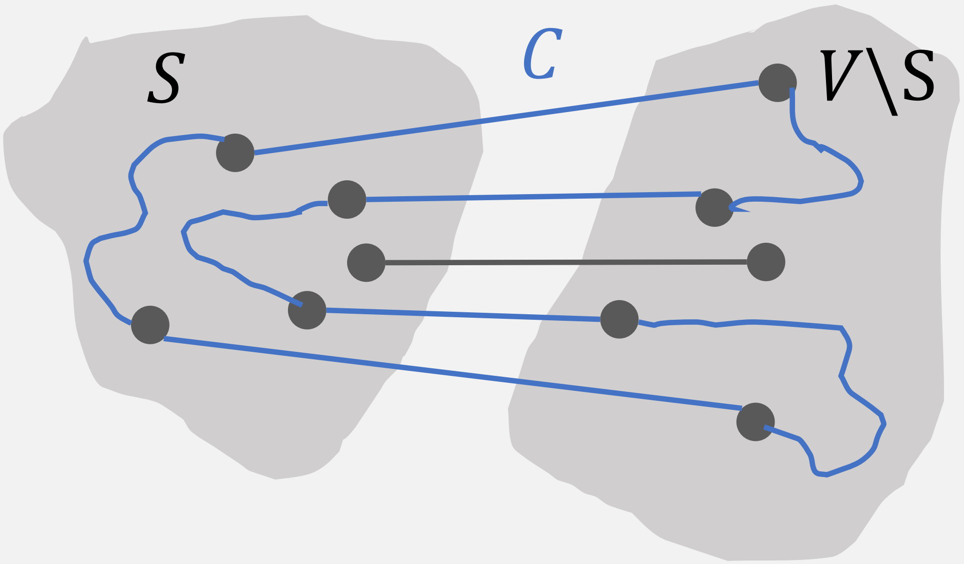
- Cycle: set of edges of form $(a, b), (b, c), \ldots, (z, a)$.
- Cut: a subset of nodes $S \subseteq V$.
- Cutset $D(S)$: subset of edges with exactly one endpoint in S.



Ex. Cut $S = \{4,5,8\}$
Cutset $D(S) = \{(4,3), (5,7), (5,6), (7,8)\}$

Observation: cycle-cut intersection

Claim*. A *cycle* & a *cutset* intersect in an even number of edges.



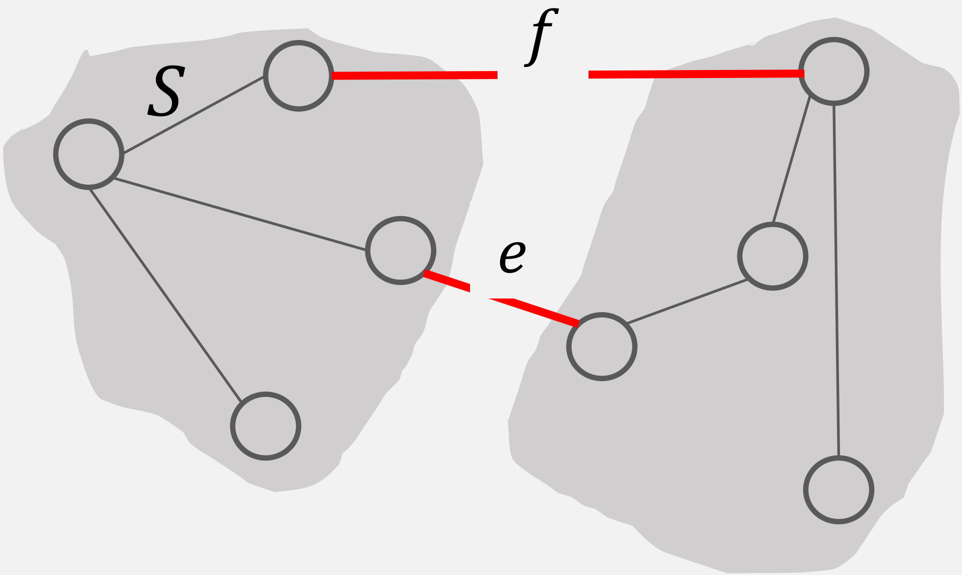
- Proof. A cycle has to leave & enter the cut the same number of times.

Cut Property

Cut property. Let S be a subset of nodes. Let e be the min weight edge with exactly one endpoint in S. Then any MST T contains e.

- Proof. (exchange argument)
  - Suppose e does not belong to T
  - Adding e to T creates a cycle C
  - Edge e is both in C and in the cutset $D(S)$

  ➔ there exists another edge, say f, that is in both C and D. [Claim*]
  - $T' \coloneqq T \cup \{e\} - \{f\}$ is also a spanning tree
  - $w_e < w_f$ ➔ $w(T') < w(T)$. Contradiction!

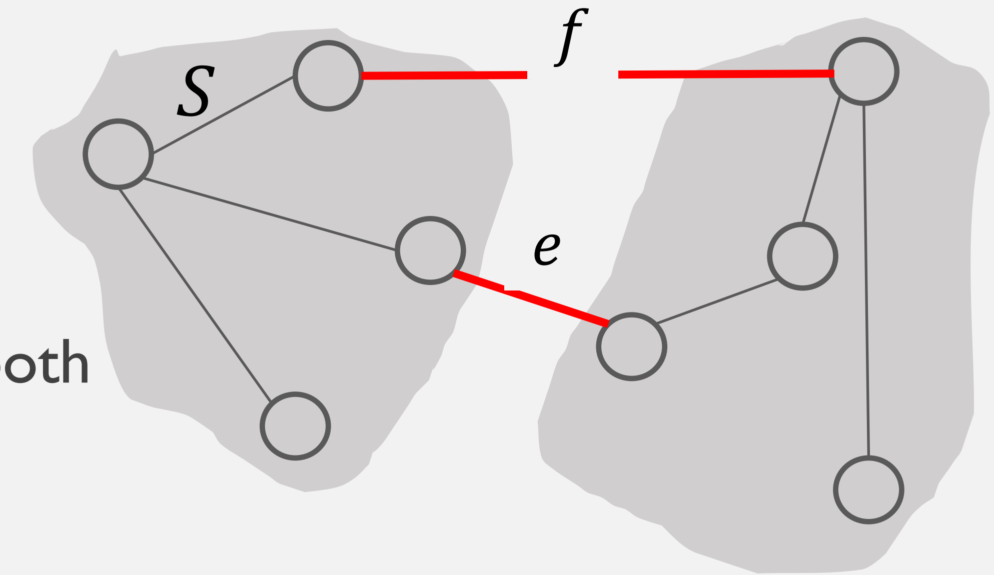


Cycle property

Cycle property. Let C be a cycle, and let f be the max weight edge in C. Then any MST T does not contain f.

- Proof. (exchange argument)
  - Suppose f belongs to T
  - Deleting f creates a cut S
  - Edge f is both in C and in the cutset $D(S)$

  ➔ there exists another edge, say e, that is in both C and D.
  - $T' \coloneqq T \cup \{e\} - \{f\}$ is also a spanning tree
  - $w_e < w_f$ ➔ $w(T') < w(T)$. Contradiction!



Pop quiz 2

Let G be a connected undirected graph w. distinct edge weights.

- Let e be the cheapest edge in G. Some MST of G contains e?

  True. By cut property

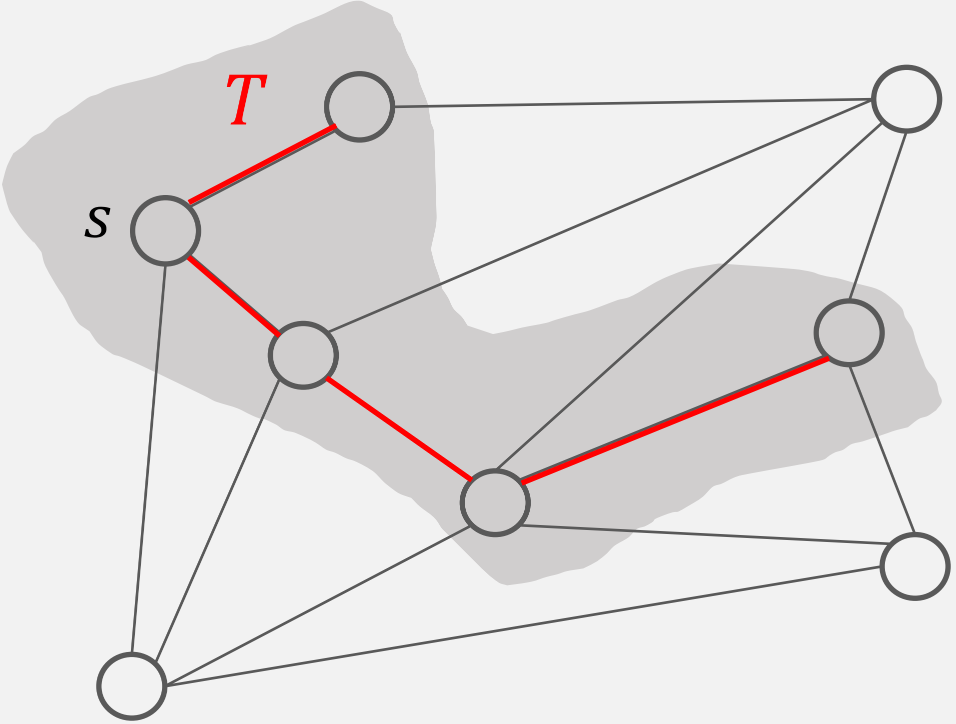
- Let e be the most expensive edge in G. No MST of G contains e?

  False. Counterexample: if G is a tree, all its edges are in the MST

Prim's algorithm: correctness

Prim's algorithm [Janik 1930, Prim 1959]

Start with some node s. Grow a tree T from s outward. Add v to T such that $w(u, v)$ cheapest and $u \in T$.



- Correctness
  - Apply cut property to T
  - When edge weights are distinct, every edge that is added must be in the MST

  ➔ Prim's algorithm outputs the MST

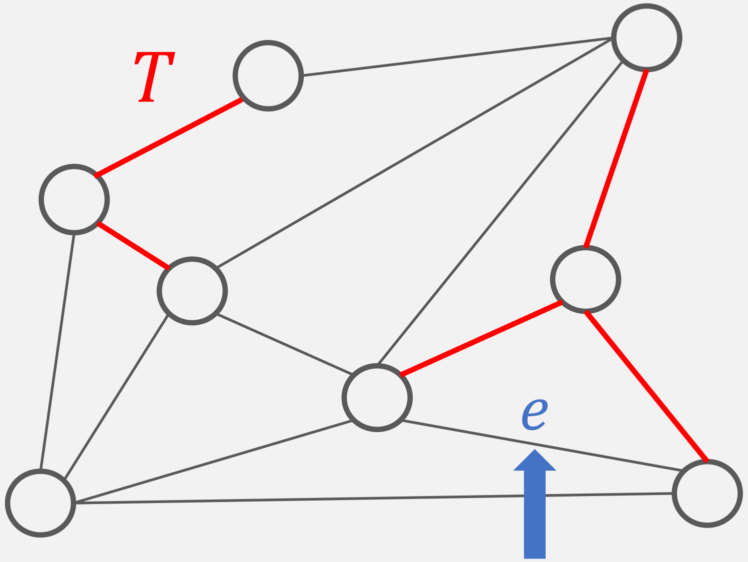
Kruskal's algorithm: correctness

Kruskal's algorithm [Kruskal 1956]

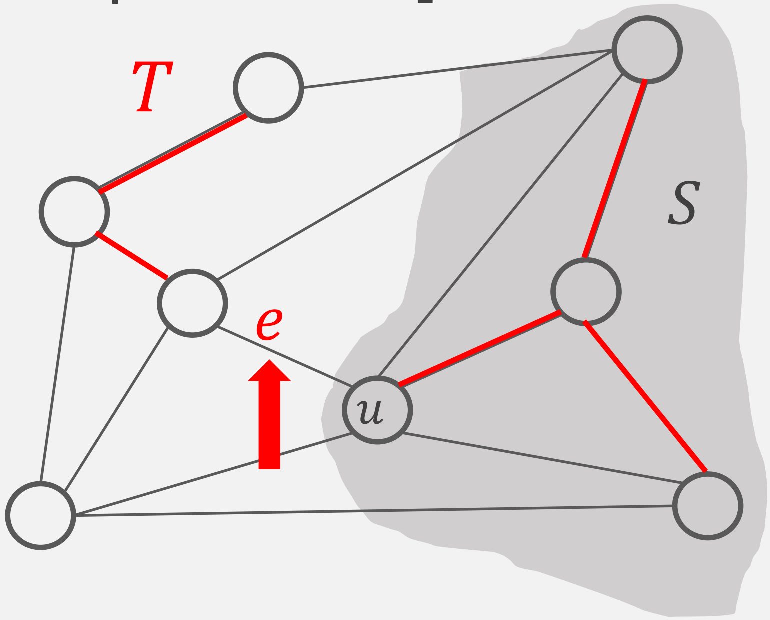
Start with $T = \emptyset$. Insert edges in ascending order of weights, unless it creates a cycle.

- Correctness

Case 1. If adding e to T creates a cycle, discard e according to cycle property.

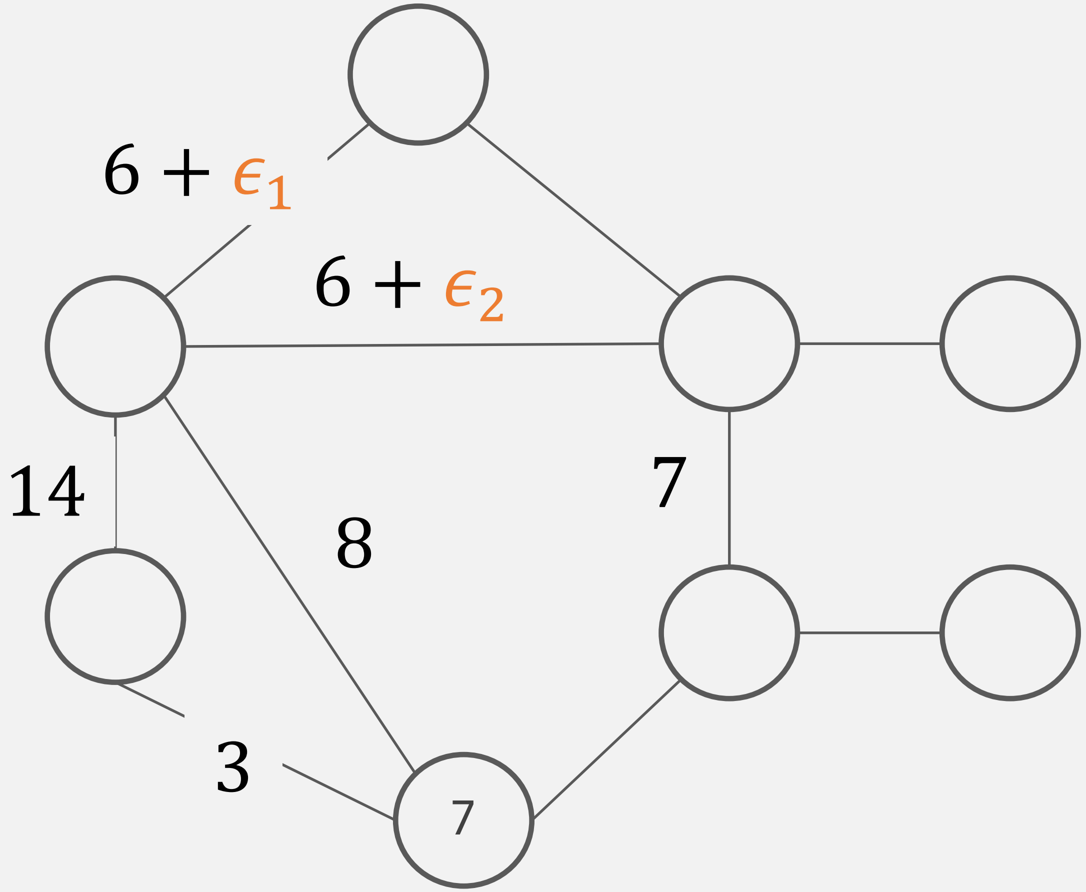


Case 2. Adding $e = (u, v)$ to T according to cut property. [S = connected component of u]



Removing distinct weight assumption

- Perturbation argument



$\sum \epsilon_i \ll |w(T') - w(T)|$

Implementing Prim's

- Maintain $V - T$ as a priority queue. [as in Dijkstra's]
- $Key(v)$: weight of the least-weight edge connecting it to a vertex in T

$\boldsymbol{Prim}(G, \{w_e\})$

1. $Q \leftarrow MakeQueue(V)$
2. $key[s] \leftarrow 0$ for an $s \in V$; $key[v] \leftarrow \infty$ otherwise
3. While Q not empty
   - $u \leftarrow$ Delete−min(Q) // add u to T
   - For $v \in Adj[u]$ // consider neighbors of u
     - If $v \in Q$ and $w(u, v) < key[v]$
       - $key[v] \leftarrow w(u, v)$
       - Change−key(v)
       - $parent(v) \leftarrow u$
4. Return $T \leftarrow \{(v, parent(v))\}$

$O(n)$

n Delete-min

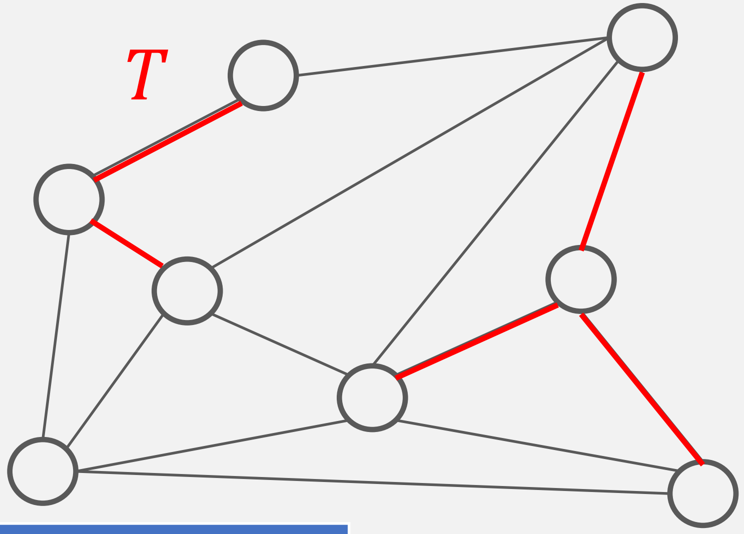
m Change-key

Time: $O((m+n)\log n)$

Same as Dijkstra's

Implementing Kruskal's

- Disjoint-set (aka Union-Find) data structure
  - $\mathrm{Make-Set}(x)$: create a singleton set containing x.
  - $\mathrm{Find-Set}(x)$: return the "name" of the unique set containing x.
  - $\mathrm{Union}(x, y)$: 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	$\Theta(k \log k)$	$\Theta(k \log k)$

Implementing Kruskal's

Kruskal$(G, \{w_e\})$

// $T \leftarrow \emptyset$; sort m edges so that $w(e_1) \leq w(e_2) \leq \cdots$ $\Big\}$ $O(m \log m)$

1. For $v \in V$, MakeSet(v)
2. For $i = 1, \ldots, m$

    $(u, v) \leftarrow e_i$ // ith cheapest edge

    If Find−Set$(u) \neq$ Find−Set(v) // same component?

    $T \leftarrow T \cup \{e_i\}$

    Union−Set(u, v)

3. Return T

$2m$ Find-Set
n Union-Set

Running time: $O(m \log m + n \log n) = O(m \log n)$

Warning on Greedy algorithms

Correctness

Greedy algorithms are tempting but rarely work!
Only with care (as sanity check or last resort)

"You will not receive any credit for any greedy algorithm, on any homework or exam, even if the algorithm is correct, without a formal proof of correctness." –Erickson

I second, and we adopt this policy in this class too!

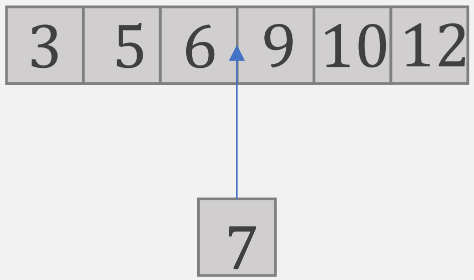
A taste of data structures & amortized analysis

Implementing Priority Queue

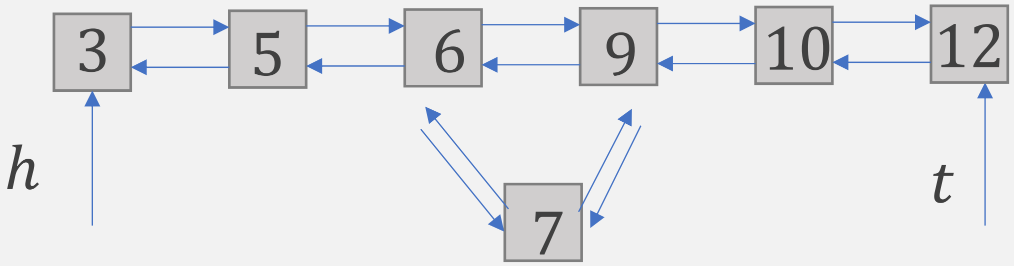
PriorityQueue: set of n elements w. associated key values

- 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: $\Omega(n)$

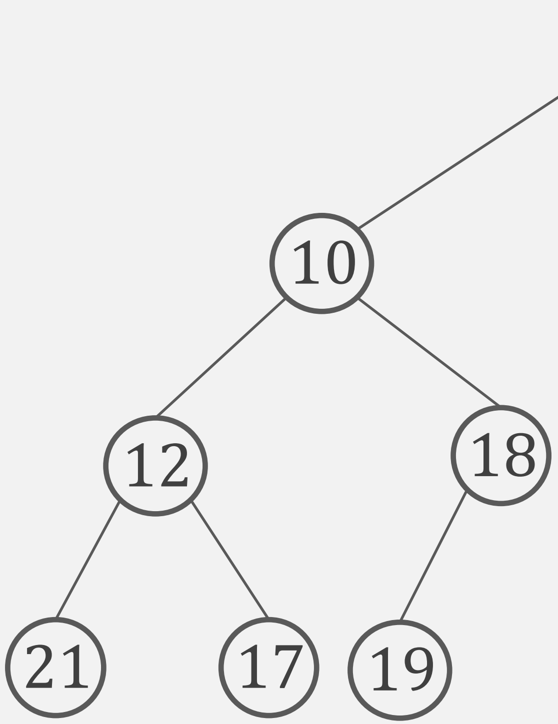


- (Sorted) Linked list?
  - ☺ Delete-min: $O(1)$
  - ☹ Insert: $\Omega(n)$



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

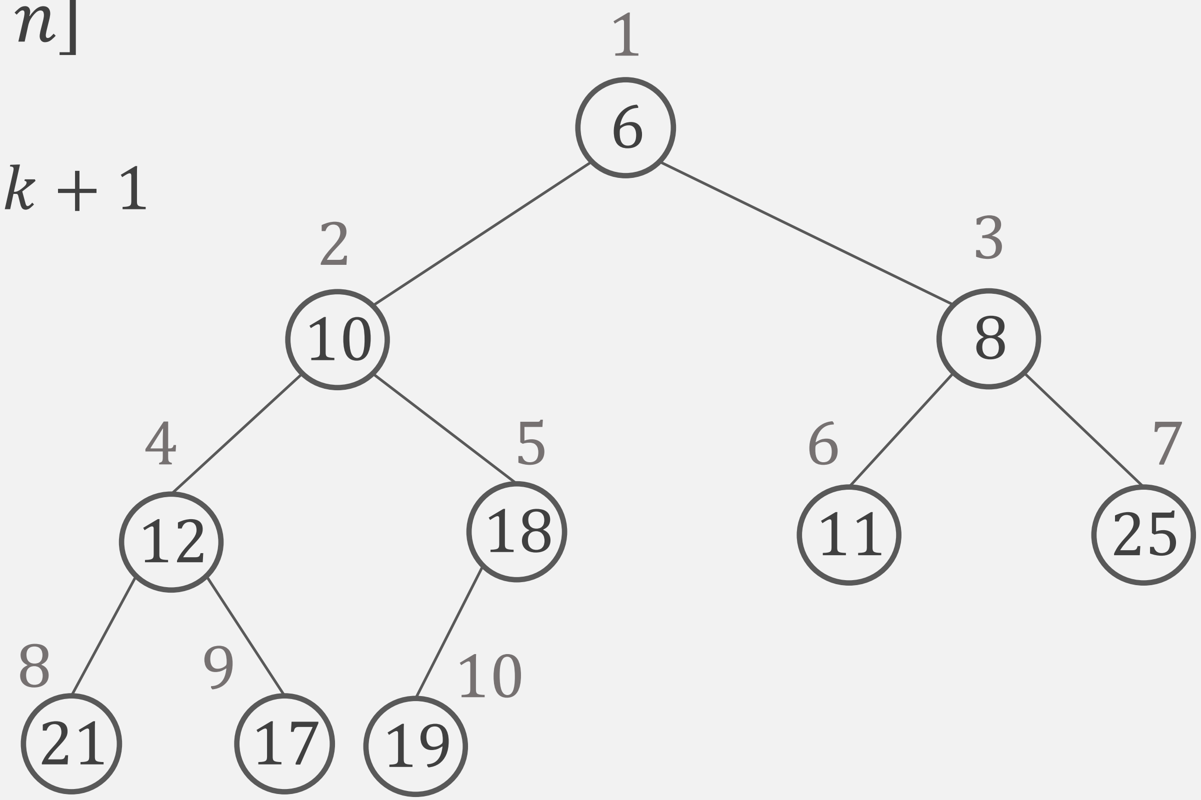


https://photos.com/featured/doum-palm-hyphaene-coriacea-and-james-warwick.html?product=poster

Representing a binary heap

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

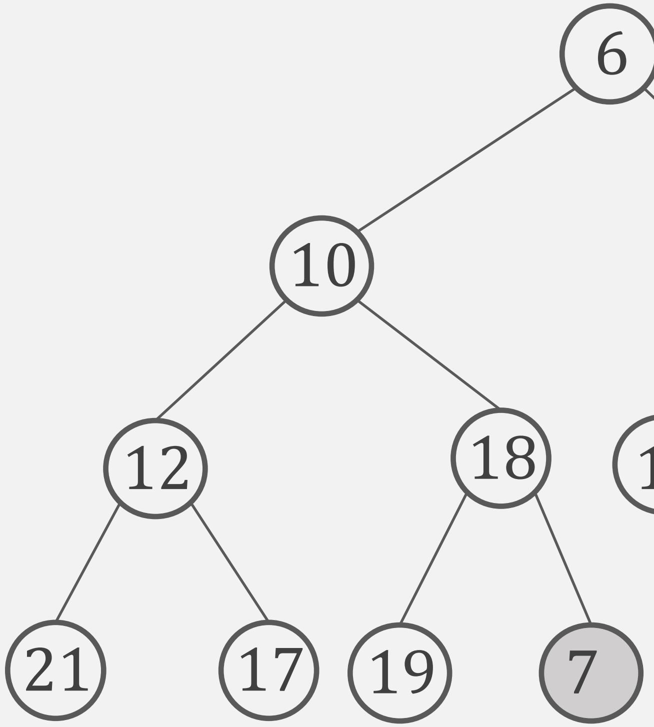
	1	2	3	4	5	6	7	8	9	10	11
H	6	10	8	12	18	11	25	21	17	19	



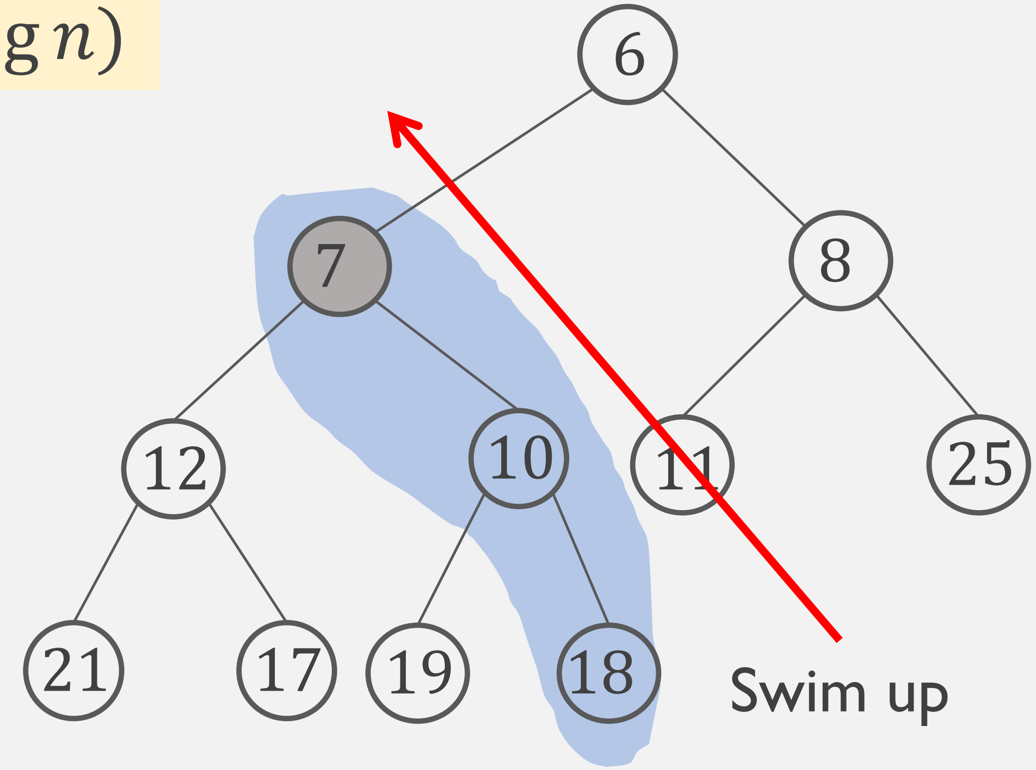
Binary heap: Insert

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

Time: $O(\log n)$



Add key to heap
(violates heap order)

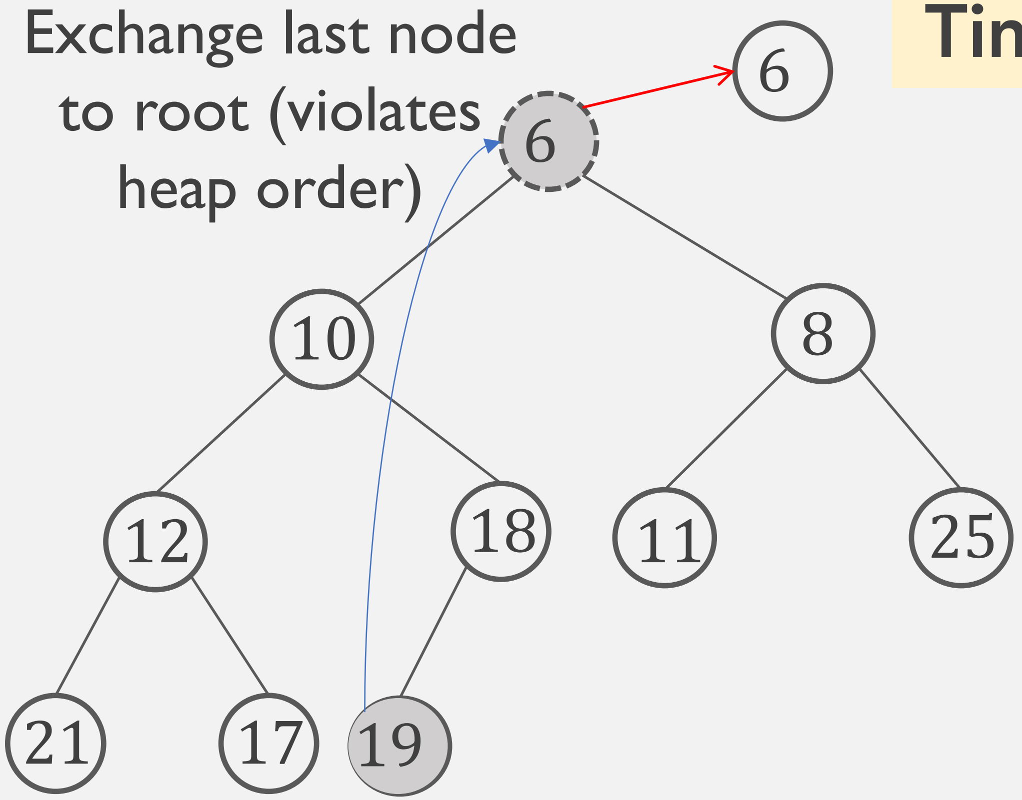


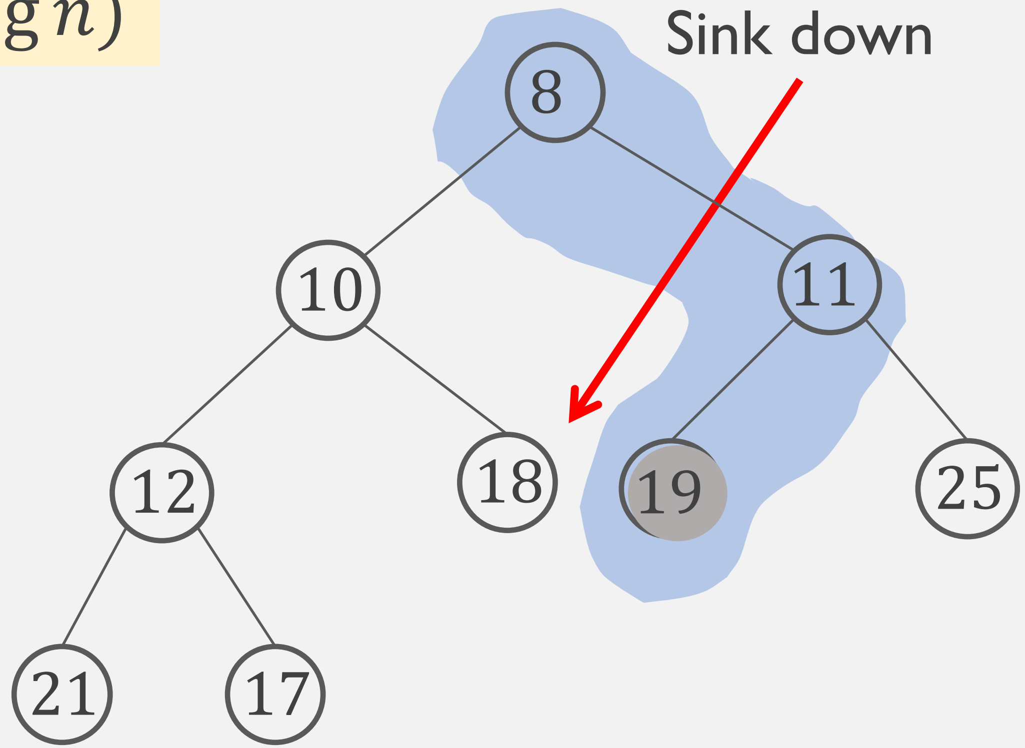
Swim up

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. $O\left(k^2\right)$]

- 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 $\log_2(2k)$

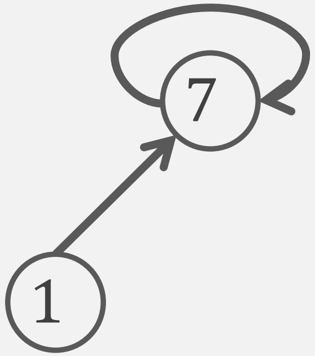
➔ $O(k \log k)$ for a sequence of k Unions [i.e., each has amortized cost $O(\log k)$].

Parent-link representation

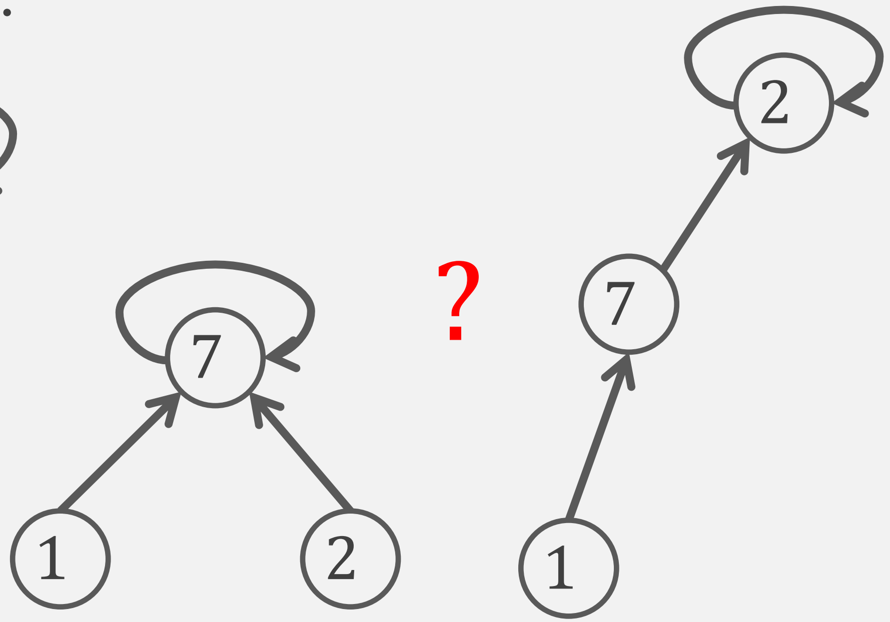
- Represent each set as a tree
  - Each element has an explicit parent pointer in the tree
  - The root (points to itself) serves as the "name"
  - FIND(x): find the root of the tree containing x
  - UNION(x, y): merge trees containing x and y.

Make-set

Union(1,7)

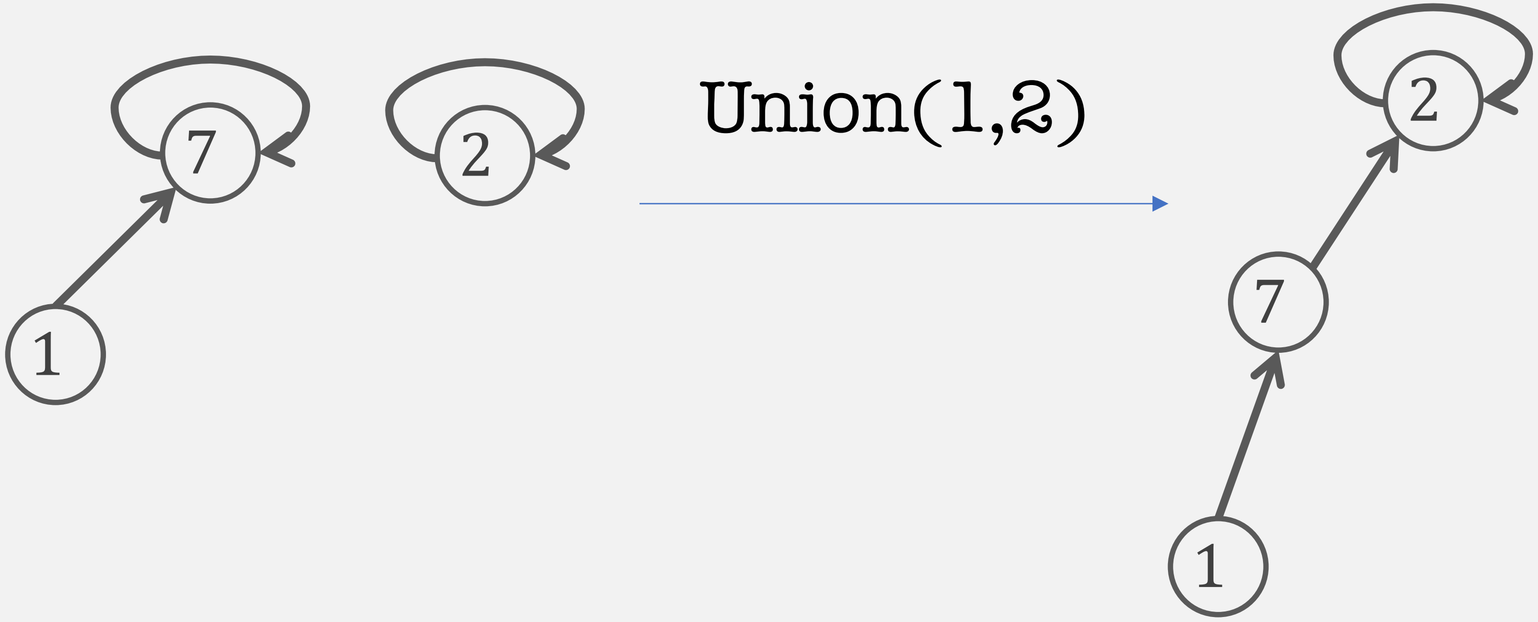


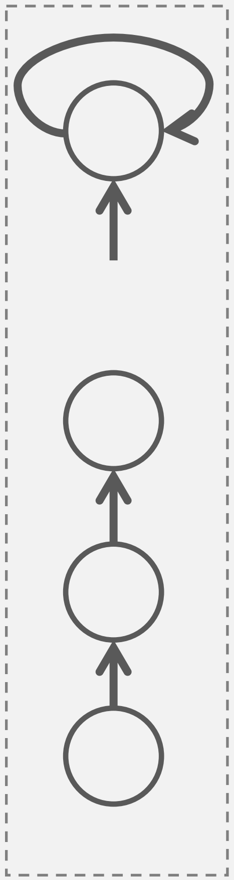
Union(1,2)



Naïve linking

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

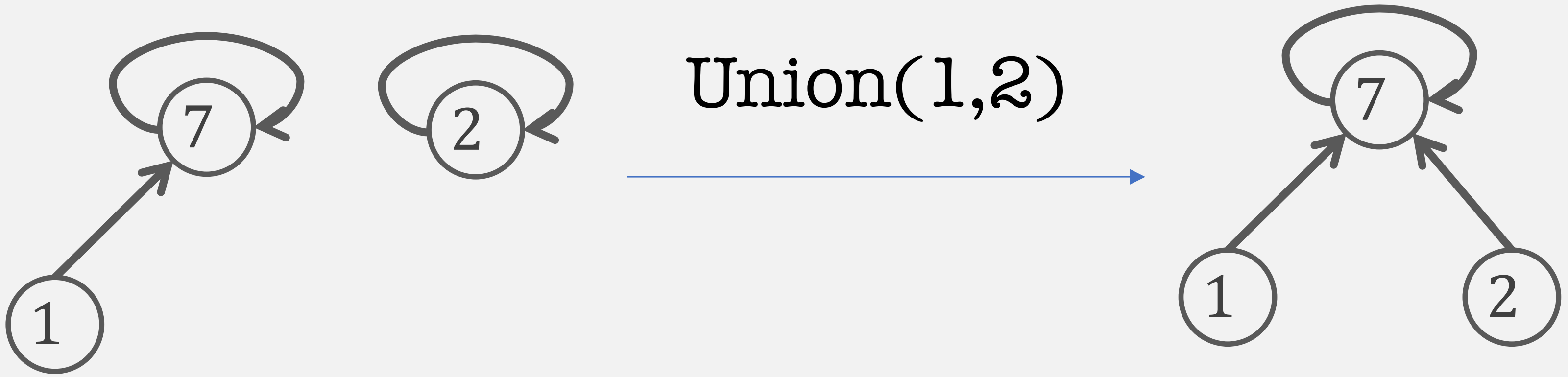




- **Observation.** A Union can take $\Theta(n)$ in the worst case
  - Find root of this tree: determined by the height of the tree

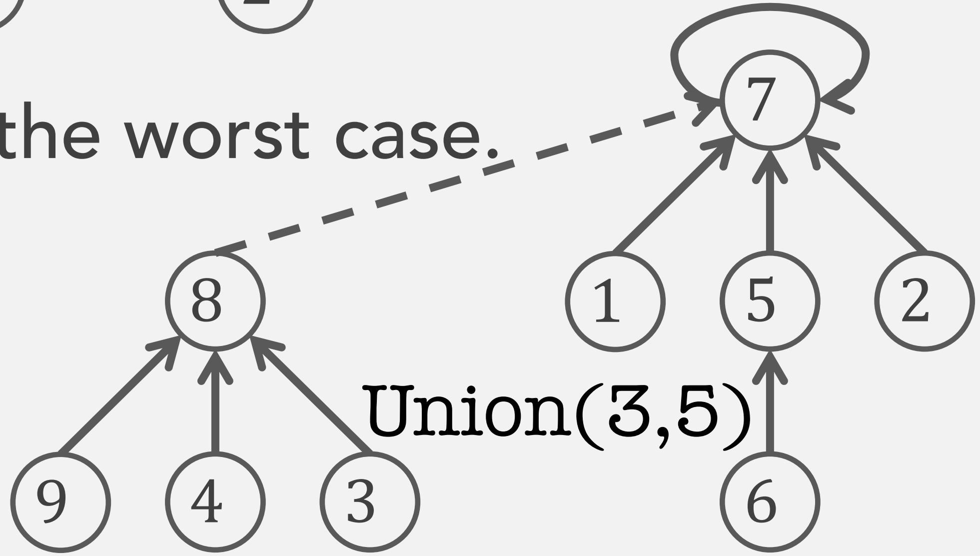
Link-by-size

- 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