

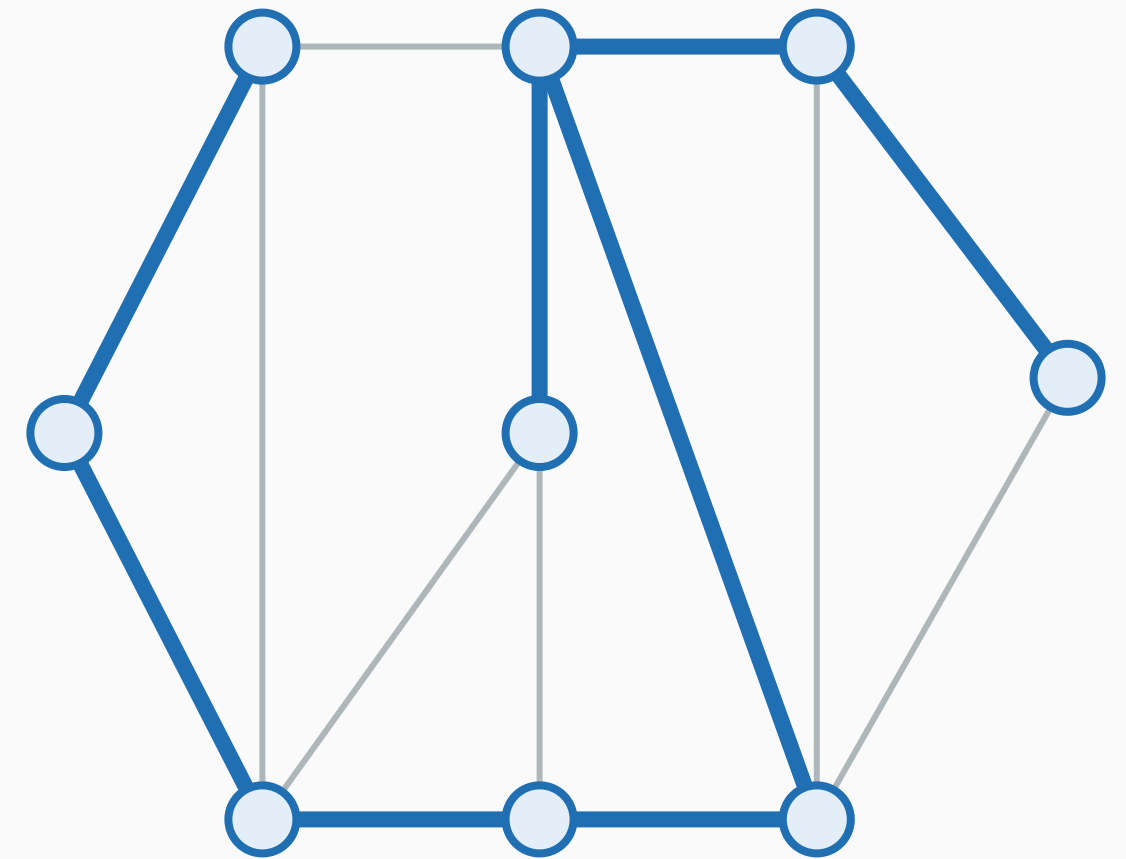
Minimum spanning trees: running time

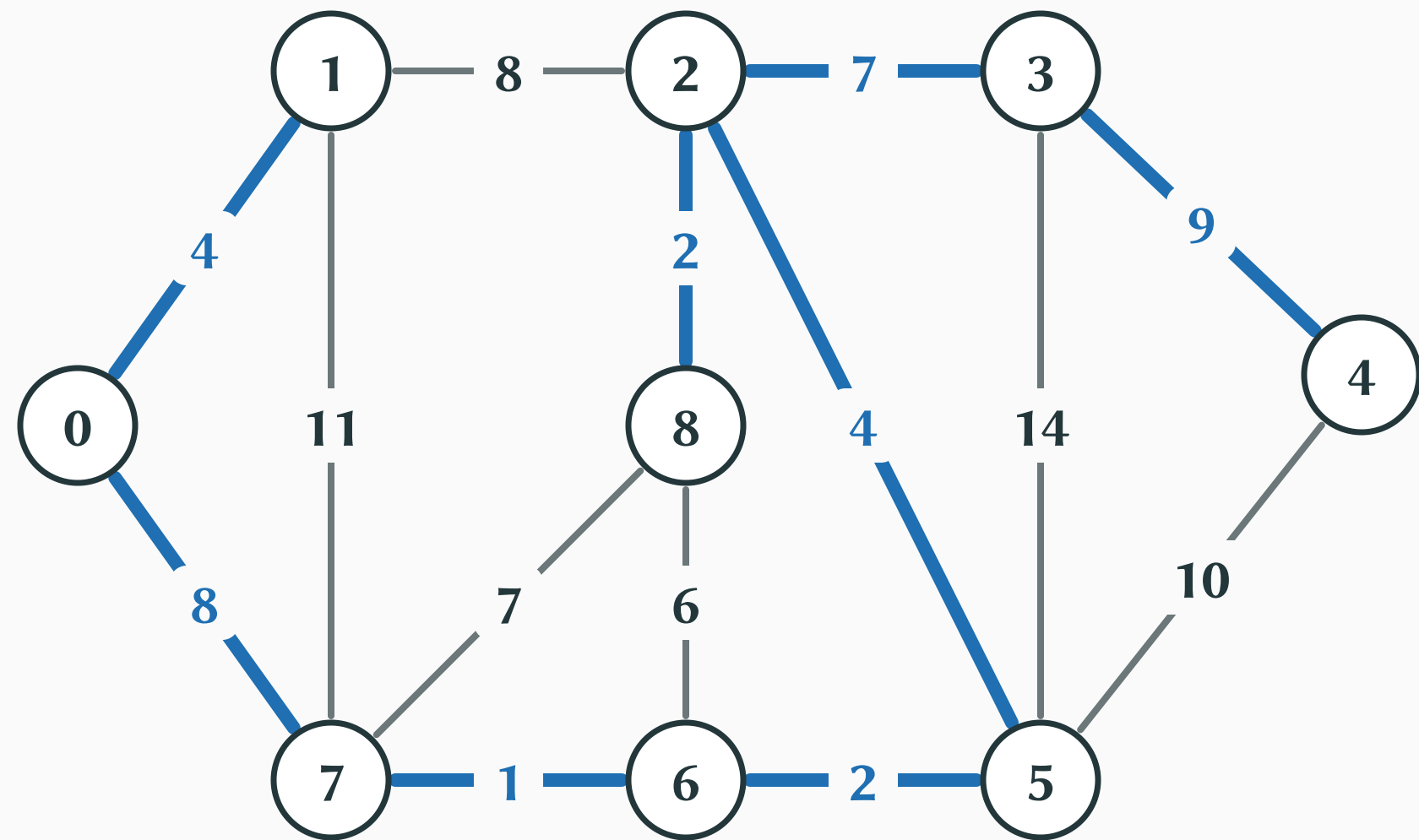
Priority queues and Union-Find

Heng Guo

School of Informatics, University of Edinburgh

Thursday 15 October 2026 · 16:10–17:00





Both give this **tree**, of cost 37.

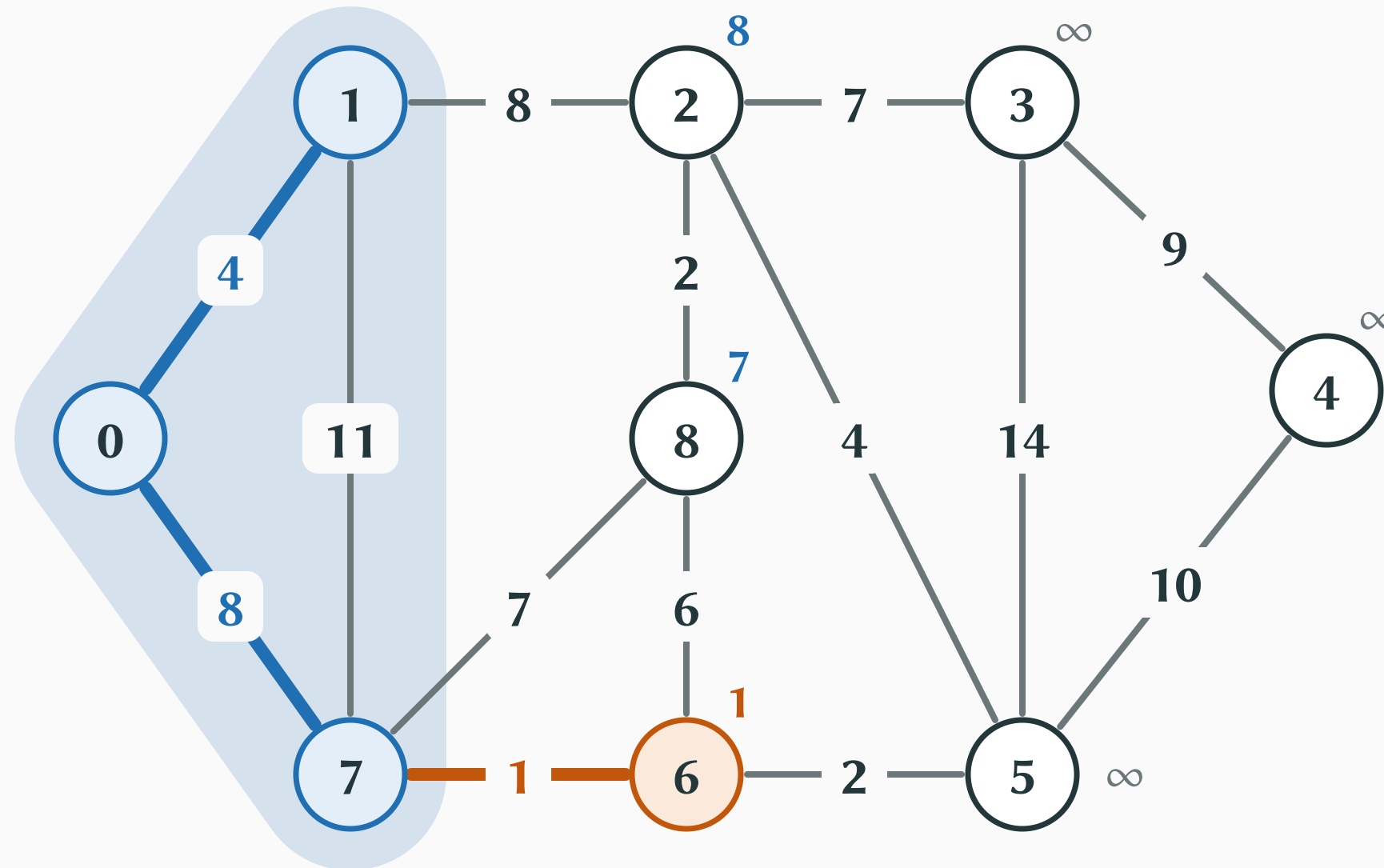
Find a spanning tree of minimum cost in a connected graph with n nodes and m edges.

Kruskal: sort the edges;
add an edge unless it closes a cycle.

Prim: start at s ;
add the cheapest edge leaving the tree.

? *Both are correct. How fast are they?*

Today: **priority queues** for Prim, **Union-Find** for Kruskal.



Numbers: $a(v)$; ∞ : no edge into S .

After three steps, Prim's tree has the nodes $S = \{0, 1, 7\}$.

Node 8 can join by the edge $\{7, 8\}$: its **attachment cost** is $a(8) = 7$.

In general, the cheapest edge into S :

$$a(v) = \min_{e=\{u,v\}, u \in S} c_e \quad \text{for } v \notin S.$$

Prim adds the node with the **smallest** $a(v)$. Scan all candidates: $\Theta(n)$ per step, $n - 1$ steps, so $\Theta(n^2)$ in total.

? *Can we find the minimum faster?*

Yes: keep the $a(v)$ in a **priority queue**.

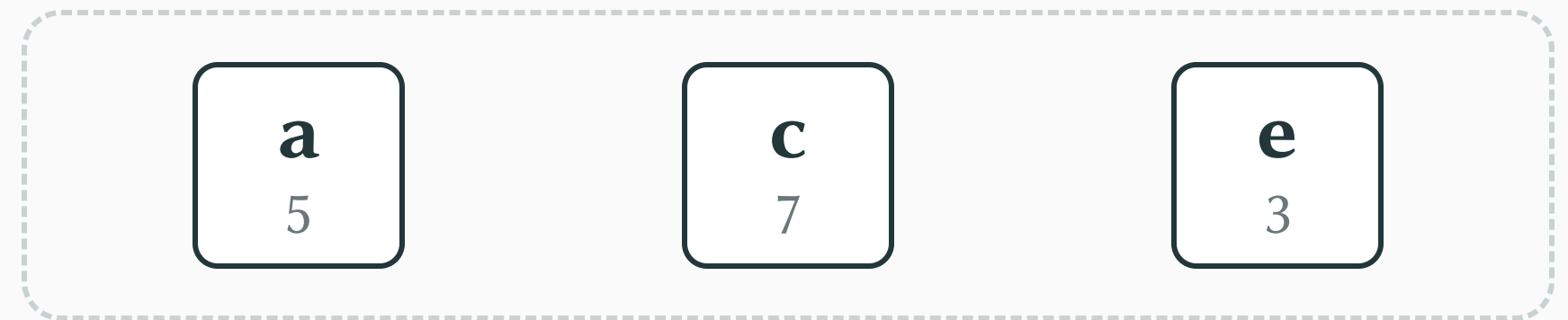
A **priority queue** Q : items with **keys**; the smallest key has the highest priority.

INSERT(Q, v, α): add item v with key α

FINDMIN(Q): the item with the smallest key

EXTRACTMIN(Q): return it and remove it

CHANGEKEY(Q, v, α): set $\text{key}(v) = \alpha$



FindMin(Q) $\rightarrow$ d (key 4)

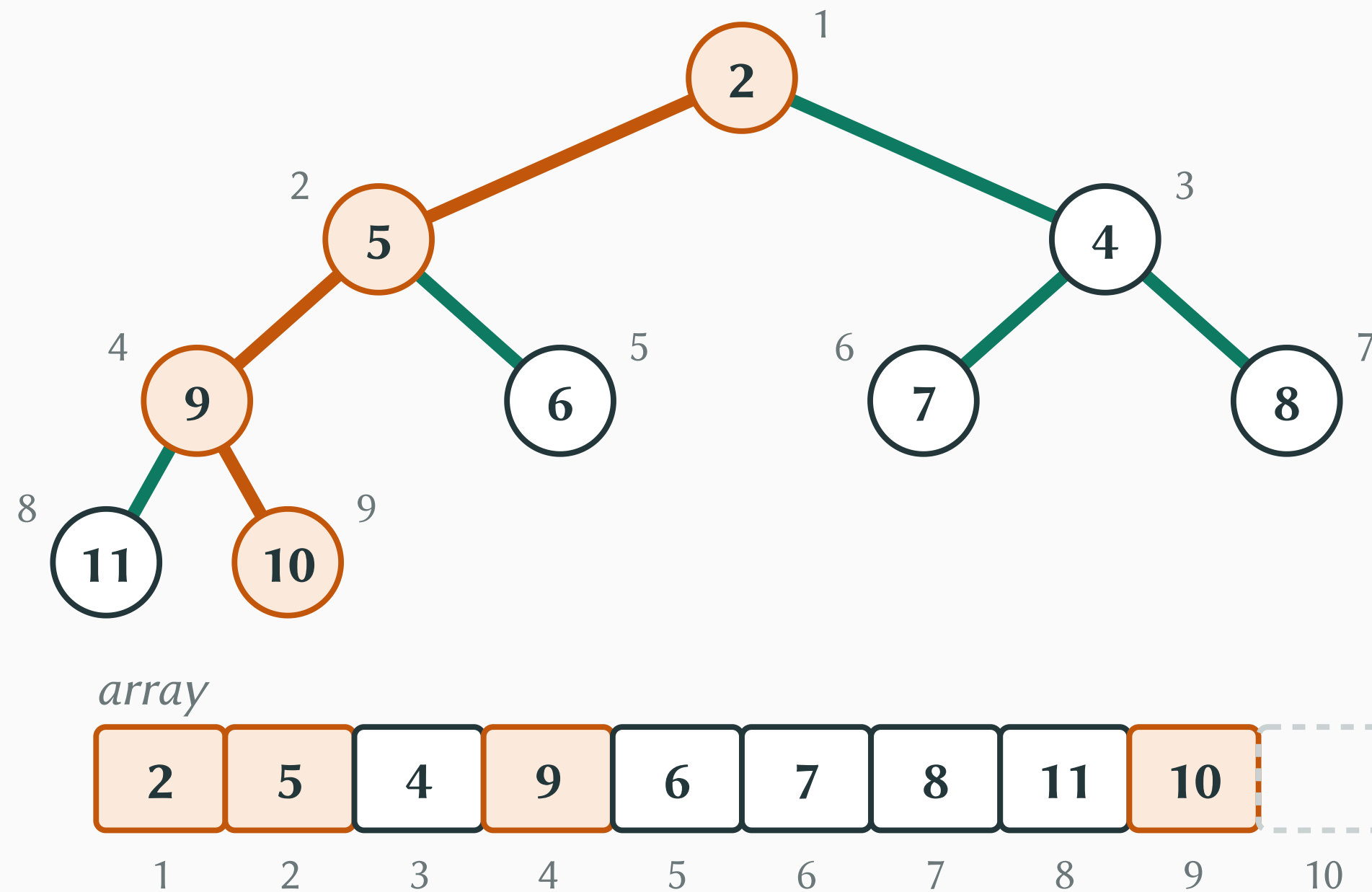
ExtractMin(Q) $\rightarrow$ d (key 4)

Insert($Q, e, 3$)

ChangeKey($Q, b, 1$)

ExtractMin(Q) $\rightarrow$ b (key 1)

Goal: every operation in $O(\log n)$ time, for n items.



A **nearly complete binary tree**: levels are filled from the top, each one from the left.

Stored in an **array**: slot i has parent $\lfloor i/2 \rfloor$ and children $2i$ and $2i + 1$.

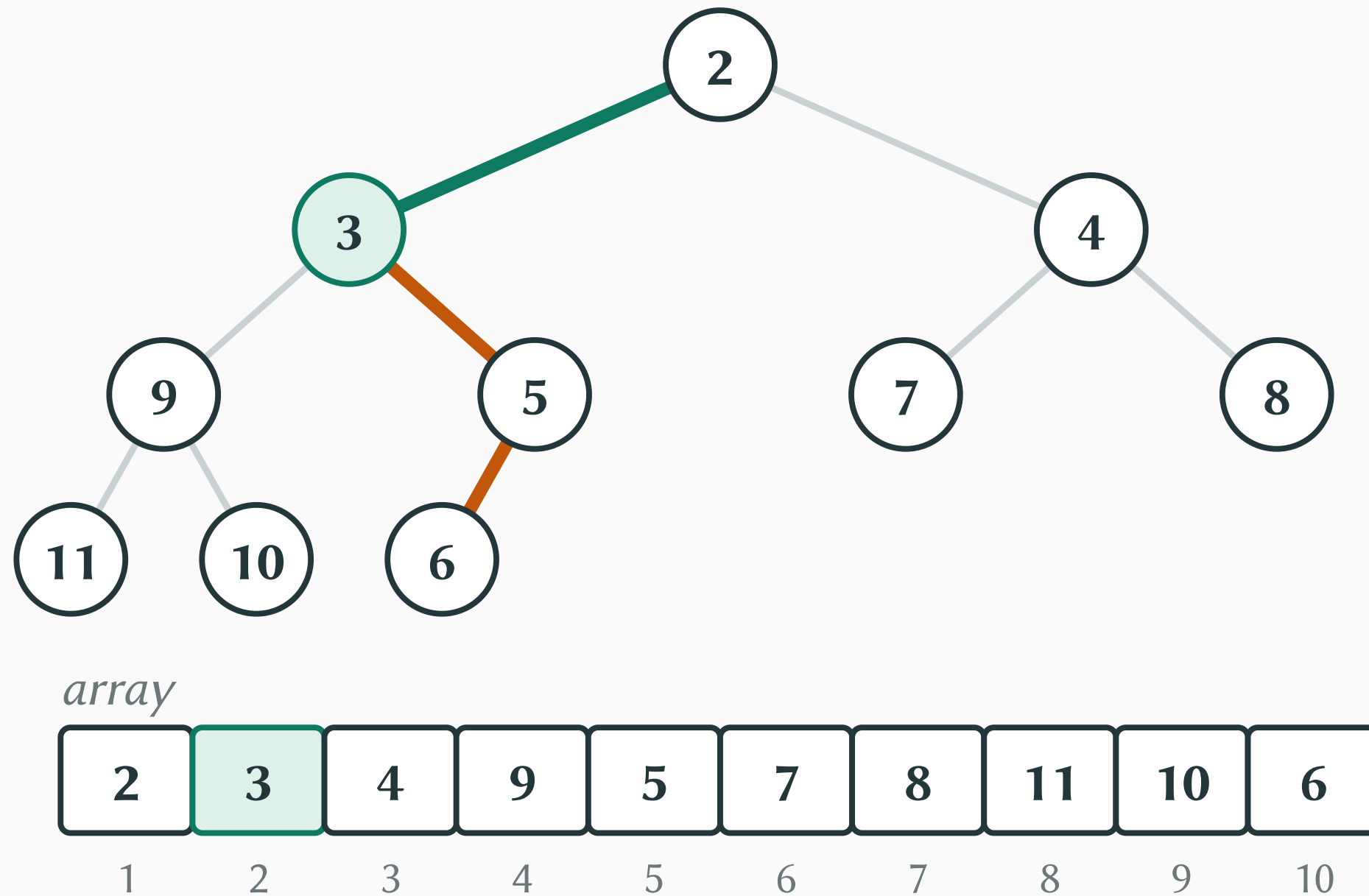
Slot 4: parent 2, children 8 and 9.

Heap property: on every edge, $\text{key}(\text{parent}) \leq \text{key}(\text{child})$.

The minimum is in slot 1:

FindMin takes $O(1)$.

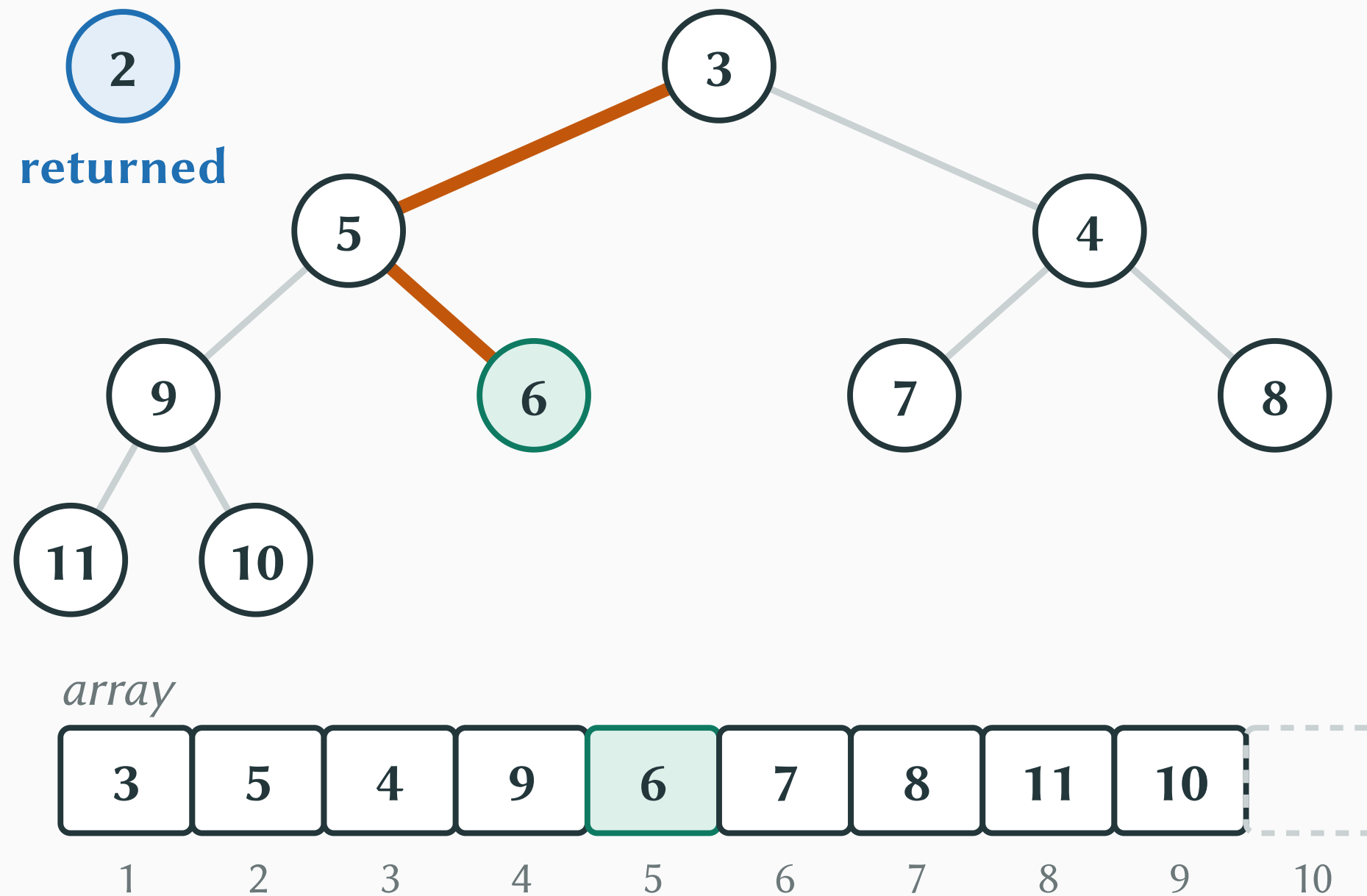
Height $\lfloor \log_2 n \rfloor$: a **path** from the root down has at most $\log_2 n$ edges.



- 1 put the new key in the first free slot
- 2 **while** its parent is larger:
swap with the parent
- 3 stop: the heap property holds

$3 \geq 2$: stop. 2 swaps in total.

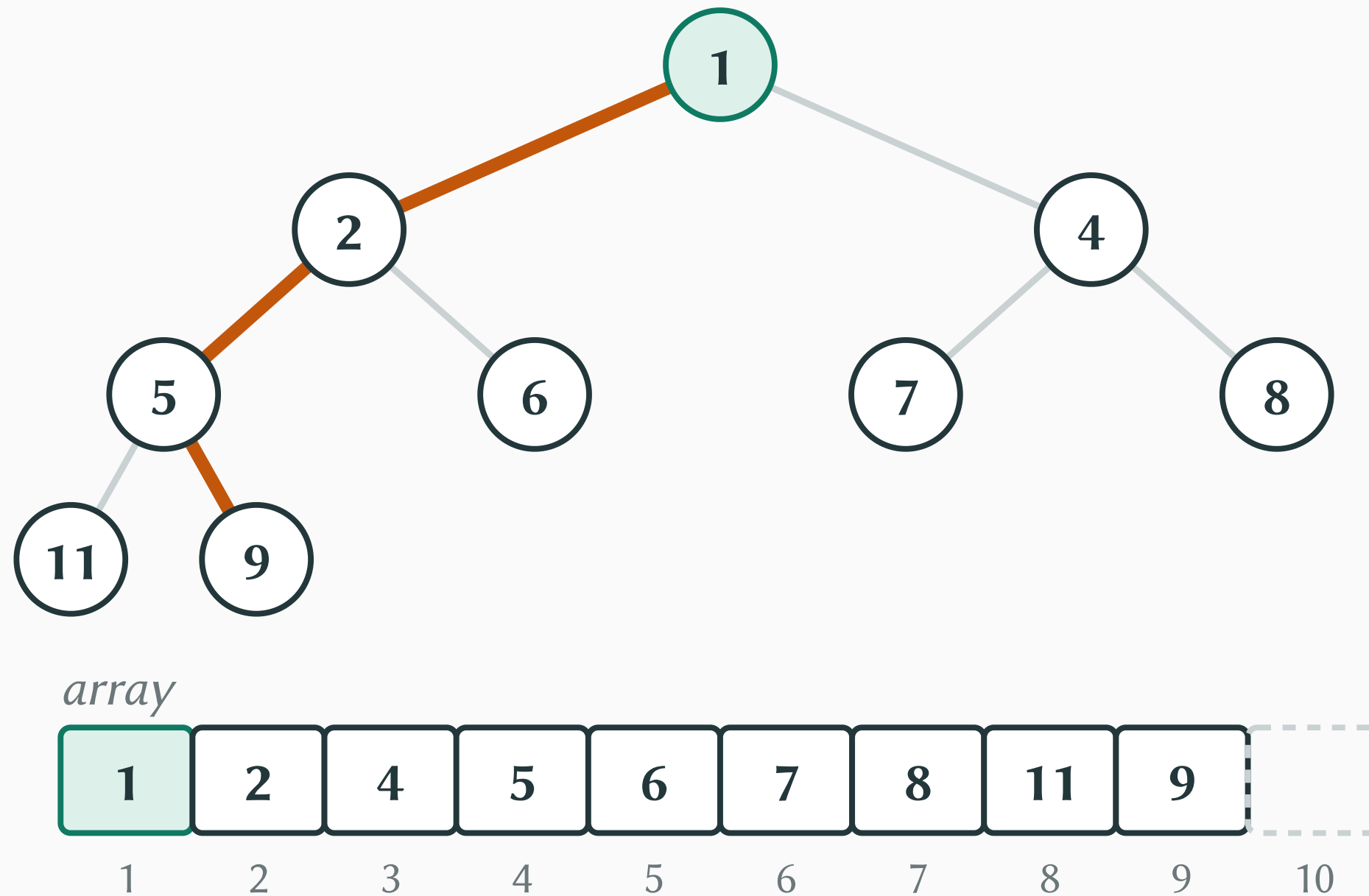
One swap per level: $O(\log n)$ time.



- 1 return the key in the root
- 2 move the last key to the root
- 3 **while** a child is smaller:
swap with the smaller child
- 4 stop: the heap property holds

No children: stop.

One swap per level: $O(\log n)$ time.



- 1 change the key in its slot
 - 2 smaller: sift up, as for Insert
 - 3 larger: sift down, as for ExtractMin
- an array $pos[v]$ gives the slot of v*

Reached the root: stop. 3 swaps in total.

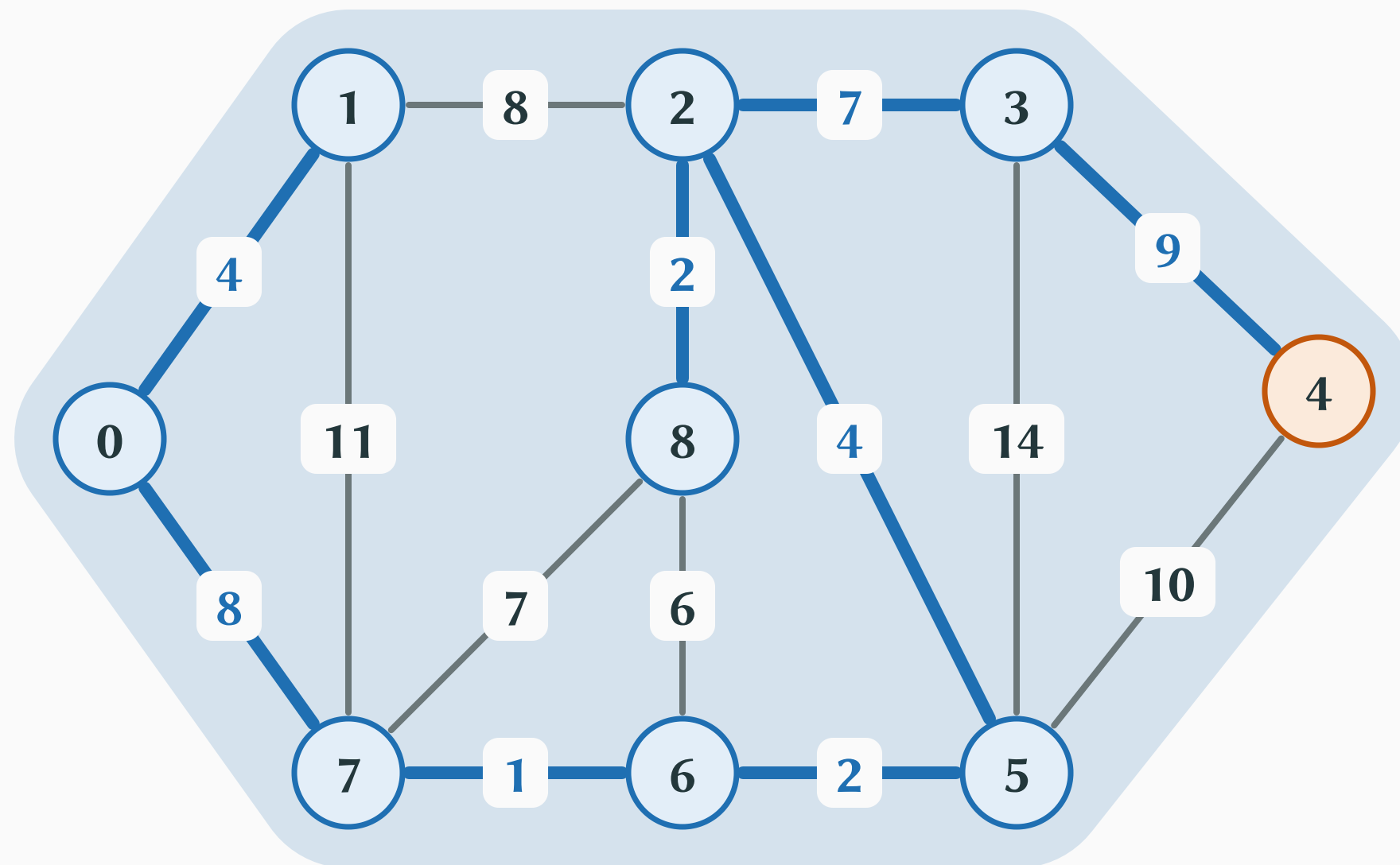
Insert	$O(\log n)$
--------	-------------

FindMin	$O(1)$
---------	--------

ExtractMin	$O(\log n)$
------------	-------------

ChangeKey	$O(\log n)$
-----------	-------------

Binary heaps: KT 2.5; CLRS 6.1–6.3, 6.5.



ExtractMin $\rightarrow$ 4 (key 9)
no ChangeKey. Tree cost 37.

```

1 key(v)  $\leftarrow \infty$  for all v, key(s)  $\leftarrow 0$ 
2 Q  $\leftarrow$  all nodes (n Inserts)
3 while Q  $\neq \emptyset$ 
4   u  $\leftarrow$  ExtractMin(Q); add u to the tree
5   for each neighbour v  $\in$  Q of u
6     if  $c_{uv} < \text{key}(v)$ : ChangeKey(Q, v,  $c_{uv}$ )
    key(v) is the attachment cost a(v)
    
```

Q is empty

9

Insert

9

ExtractMin

13

ChangeKey

Nodes left Q in the order 0, 1, 7, 6, 5, 2, 8, 3, 4.

The heap operations of Prim, for n nodes and m edges:

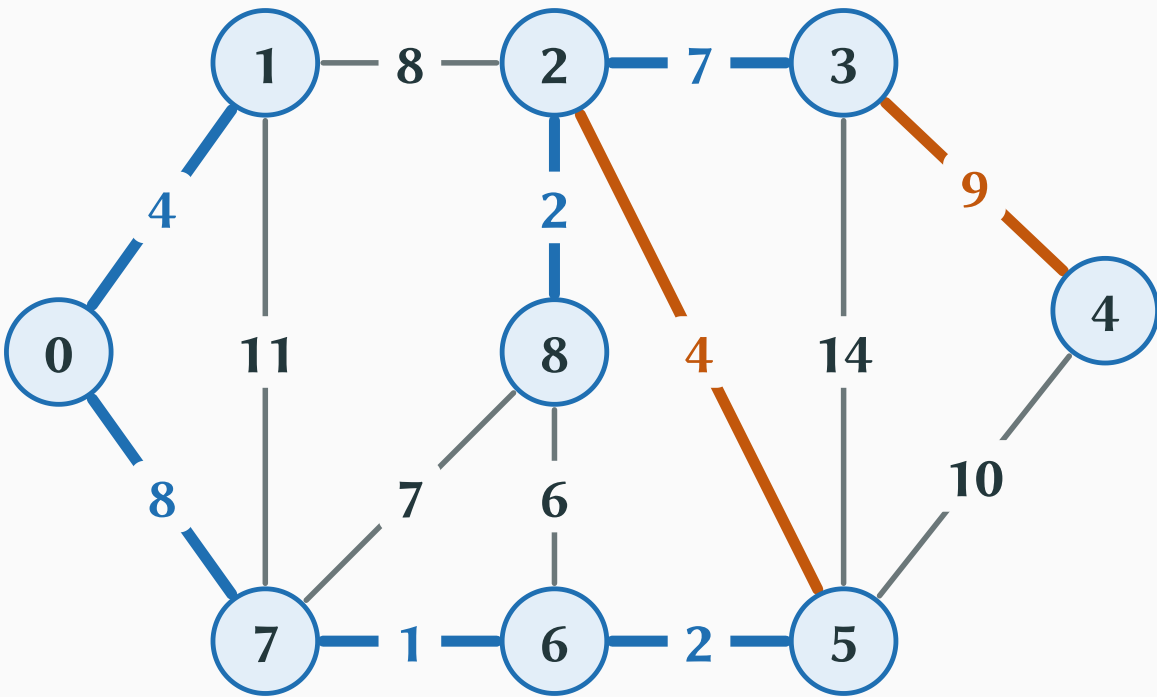
- ① n Inserts: every node enters Q once. $n \cdot O(\log n)$
- ② n ExtractMins: every node leaves Q once. $n \cdot O(\log n)$
- ③ $\leq m$ ChangeKeys: at most one per edge $\{u, v\}$,
when the first of u, v leaves Q . $m \cdot O(\log n)$

THEOREM. Prim with a binary heap runs in $O((n + m) \log n) = O(m \log n)$ time, because $m \geq n - 1$.

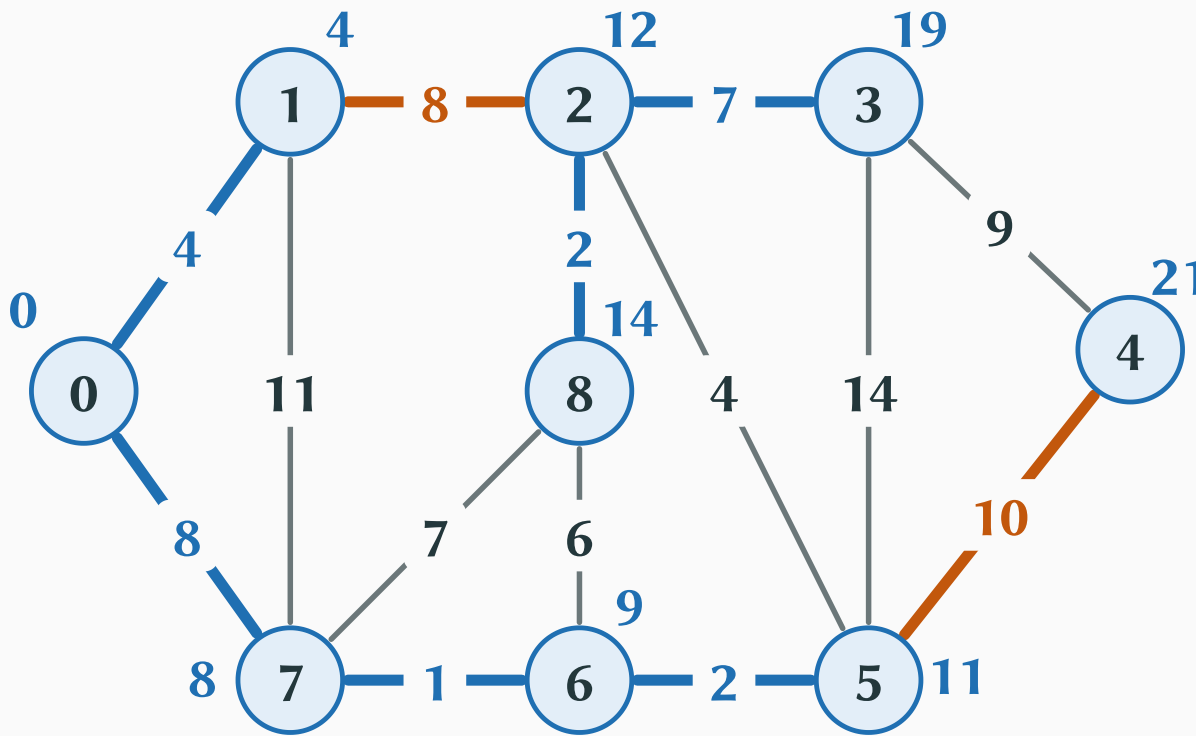
Scanning all candidates takes $\Theta(n^2)$:
the heap is better when $m \log n < n^2$, for sparse graphs.

With a Fibonacci heap: $O(m + n \log n)$. Fredman and Tarjan, 1987; CLRS 21.2.

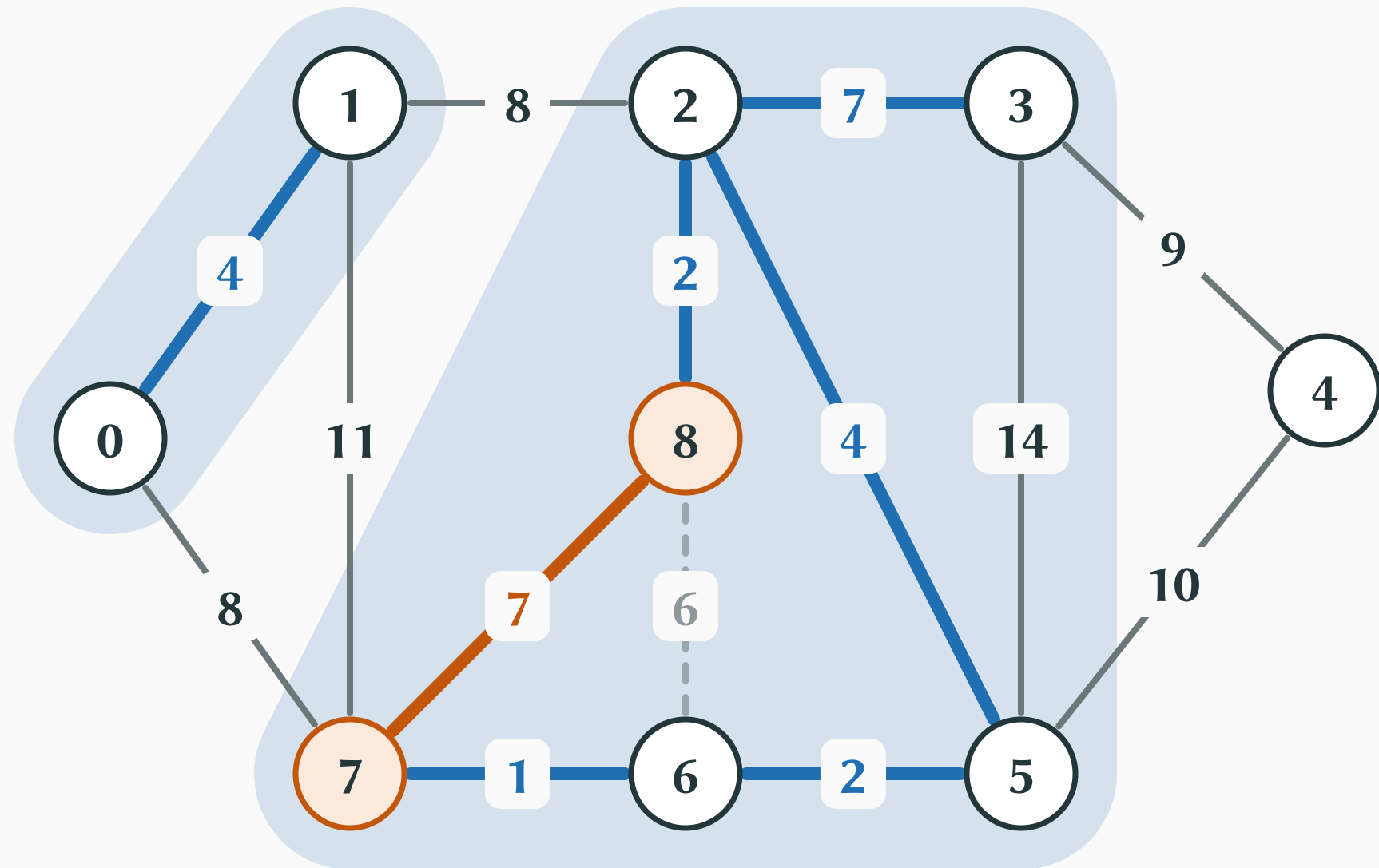
	Prim's algorithm (slide 9)	Dijkstra's algorithm (IADS)
finds	a minimum spanning tree	shortest paths from s
key(v)	cheapest edge from the tree to v	shortest path from s to v so far
new key via {u, v}	c_{uv}	$\text{key}(u) + c_{uv}$
binary heap	$O(m \log n)$ for both: the same operations as on slide 10	



Prim's algorithm from 0: an MST, cost 37



Dijkstra's algorithm from 0: distances, cost 42



Edge $\{7, 8\}$, cost 7: are 7 and 8 in one component of (V, T) ?

? *Components only ever merge.
Can we use that?*

Sort the edges by cost.

$O(m \log n)$

Check whether $\{u, v\}$ should join T .

Check whether u, v are in one component.

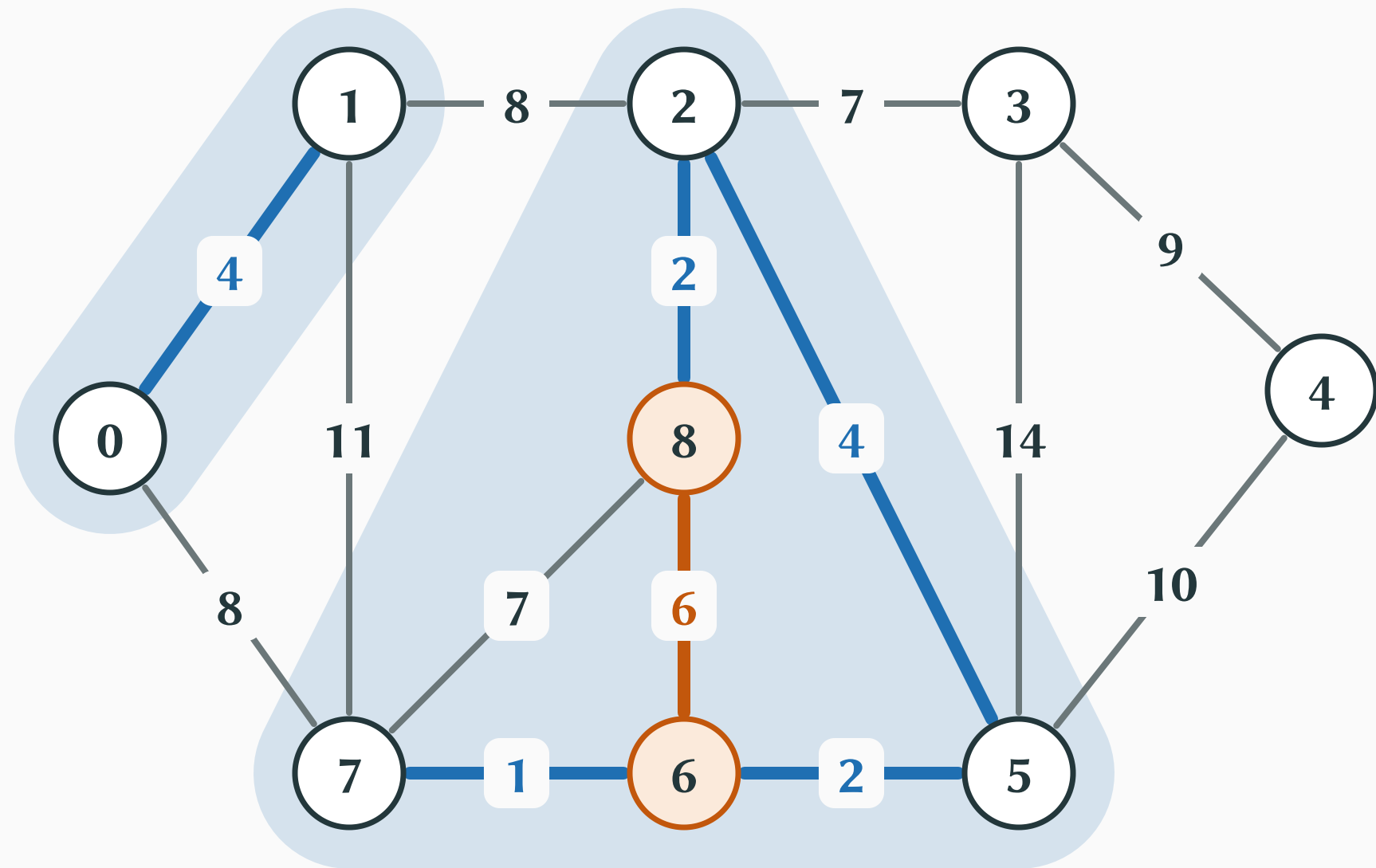
Use BFS/DFS from u in (V, T) .

$O(|V| + |T|)$

Repeat for all m edges;
each search costs $O(n)$, as $|T| < n$.

$O(mn)$

Yes: **Union-Find**,
a data structure for sets that merge.



MAKEUNIONFIND(S):

each element of S alone in a set

FIND(u): the name of the set containing u

UNION(A, B): merge the sets A and B

MakeUnionFind(V): 9 sets

$\{6, 7\}$: Find(6) $\neq$ Find(7): Union

$\{5, 6\}$: Find(5) $\neq$ Find(6): Union

$\{2, 8\}$: Find(2) $\neq$ Find(8): Union

$\{0, 1\}$: Find(0) $\neq$ Find(1): Union

$\{2, 5\}$: Find(2) $\neq$ Find(5): Union

$\{6, 8\}$: Find(6) = Find(8): skip

sets:

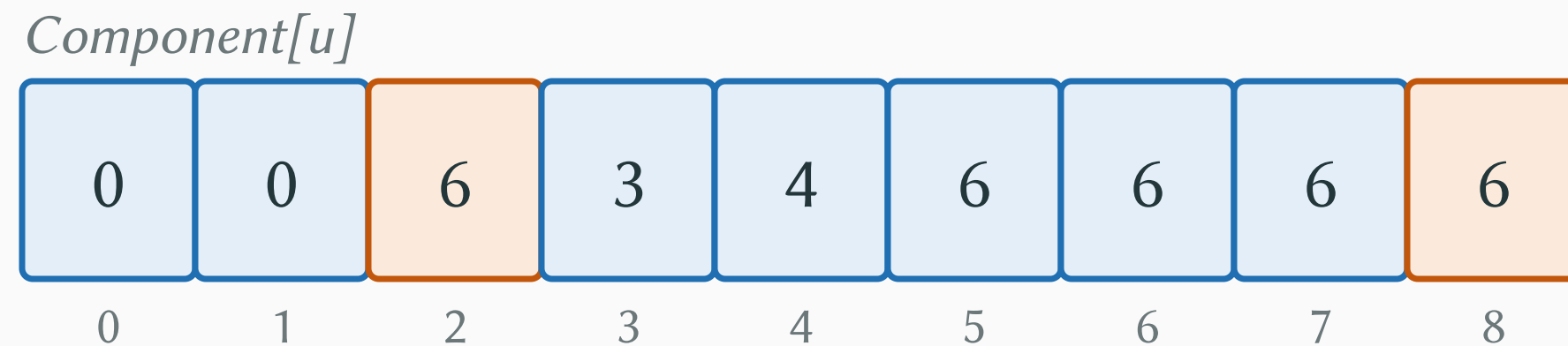
$\{0, 1\}$ $\{2, 5, 6, 7, 8\}$ $\{3\}$ $\{4\}$

- 1 sort the edges by cost $O(m \log n)$
- 2 $\text{MakeUnionFind}(V); T \leftarrow \emptyset$ once
- 3 **for** each edge $\{u, v\}$ in this order
- 4 $A \leftarrow \text{Find}(u); B \leftarrow \text{Find}(v)$ $2m$ Finds
- 5 **if** $A \neq B$: add $\{u, v\}$ to T ; $\text{Union}(A, B)$ at most $n - 1$ Unions
- 6 **return** T

TIME. $O(m \log n) + T_{\text{make}} + 2m \cdot T_{\text{find}} + (n - 1) \cdot T_{\text{union}}$

$T_{\text{make}}, T_{\text{find}}, T_{\text{union}}$: the time of one MakeUnionFind , Find , Union .

Goal: all Union-Find operations together in $O(m \log n)$.



Union(2, 6): all 9 cells visited, 2 relabelled to 6.

0: {0, 1}

6: {2, 5, 6, 7, 8}

3: {3}

4: {4}

Component[u]: the name of u's set.

MakeUnionFind: $\text{Component}[u] = u$, time $O(n)$.

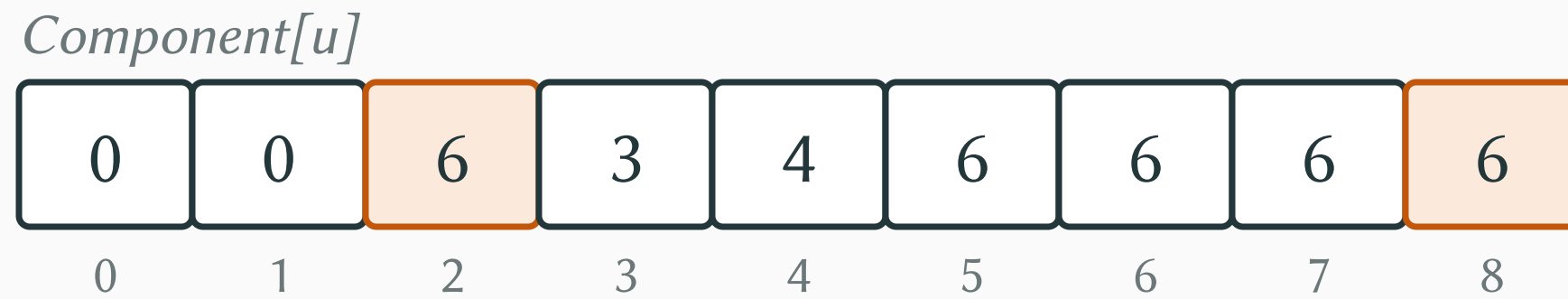
Find(u) returns $\text{Component}[u]$: $O(1)$.

Union(A, B): visit **every** cell, relabel those named A to B: $O(n)$.

? $n - 1$ Unions: $O(n^2)$.

Can we avoid scanning everything?

Yes: keep a **list of members** for every set.



member lists of the sets

0: 0 1

3: 3

4: 4

6: 6 7 5 2 8

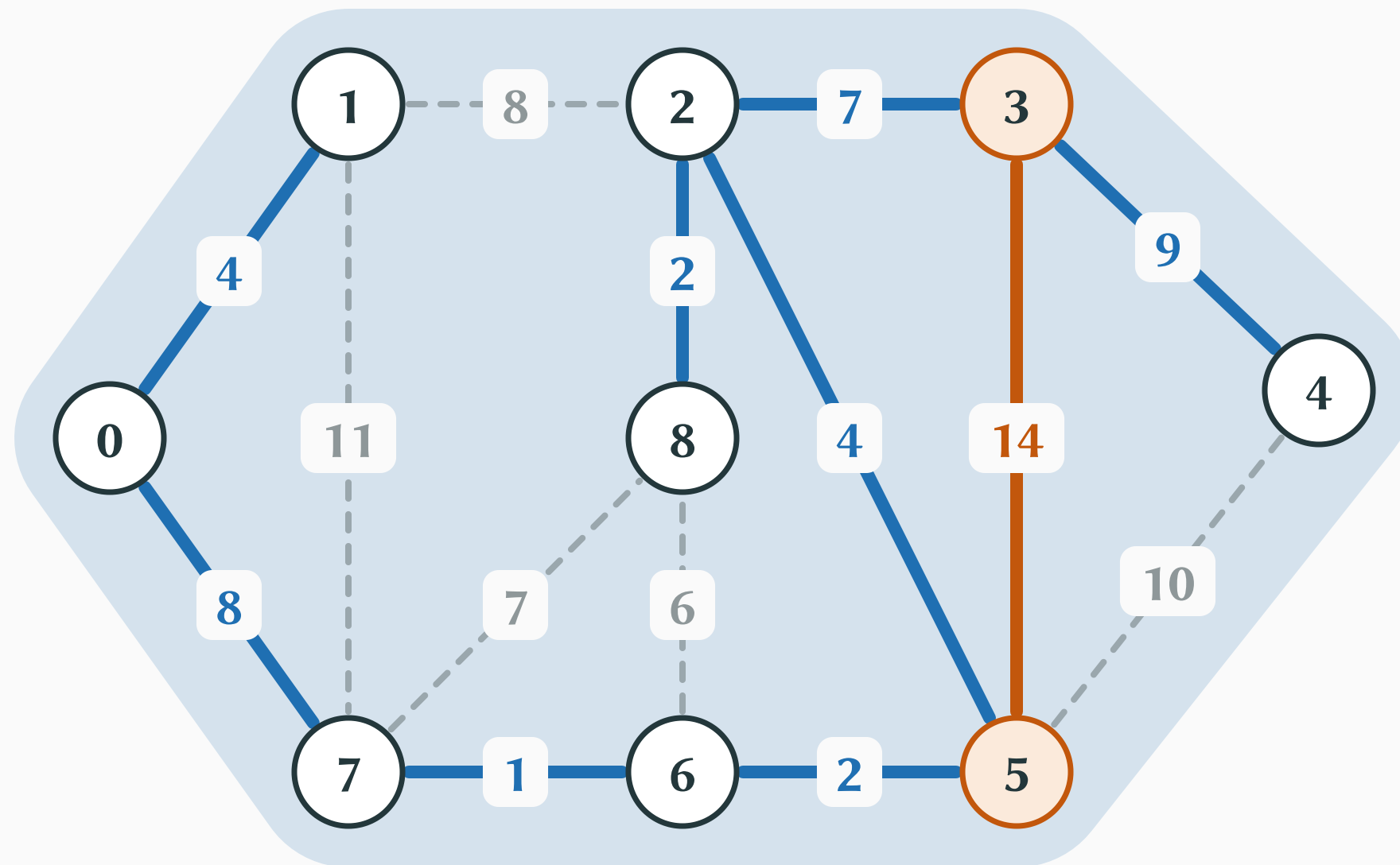
Keep a **list of members** and its size for every set: no scan of the array.

Edge {2, 5}:

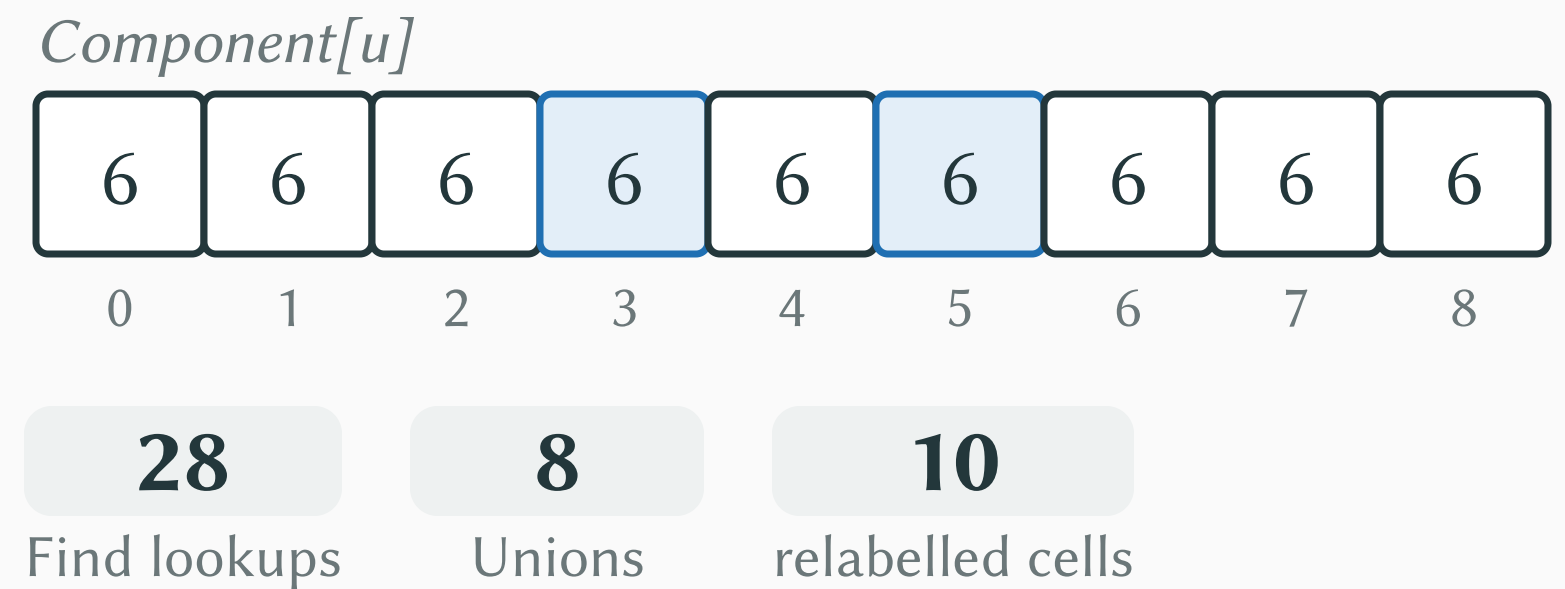
$\text{Component}[2] = 2 \neq 6 = \text{Component}[5]$

Union: the larger set keeps its name; relabel only **the smaller set**: $O(\min(|A|, |B|))$.

One Union can still cost $\Theta(n)$:
two sets of size $n/2$.



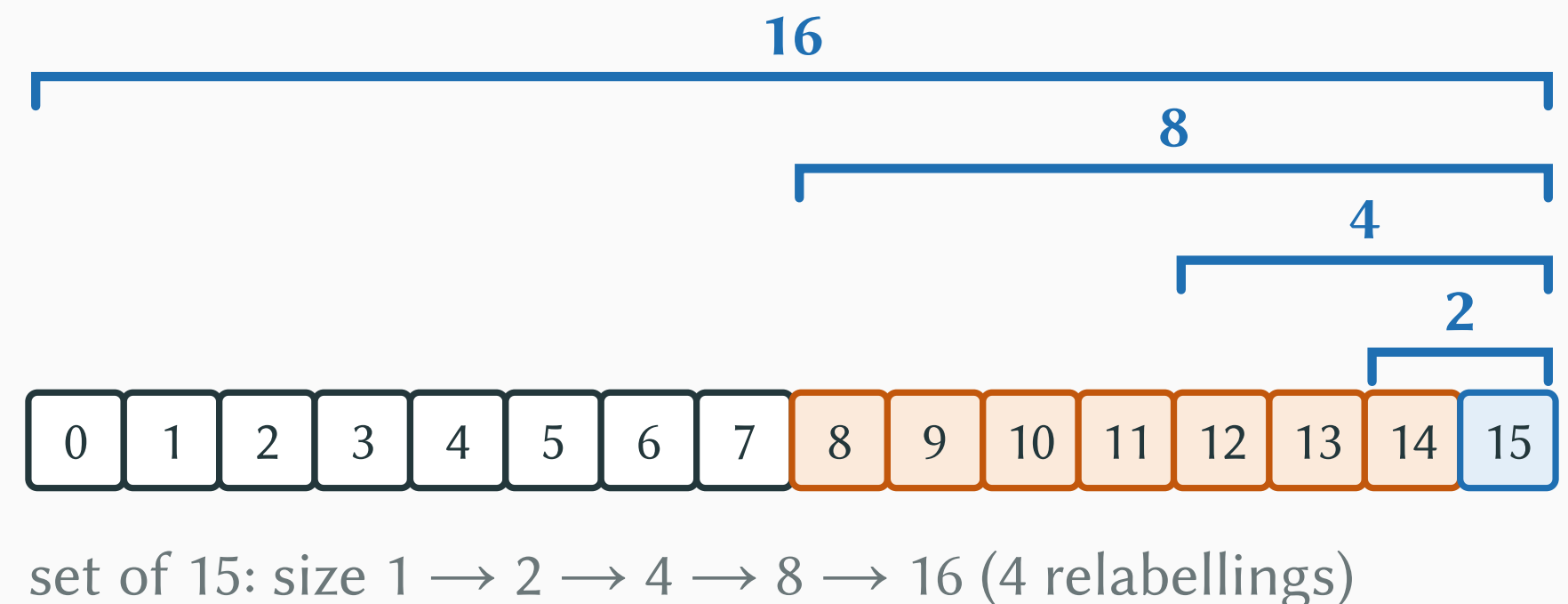
$\{3, 5\}$: $\text{Component}[3] = 6 = \text{Component}[5]$.
Same set: skip the edge.



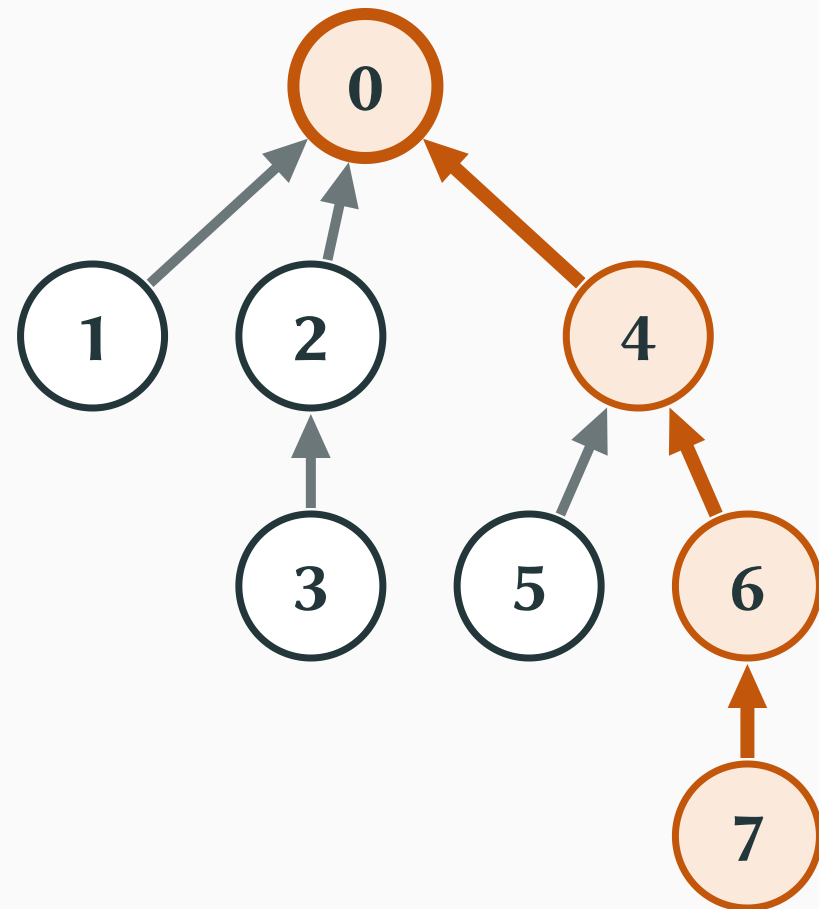
8 Unions relabelled 10 cells in total. Relabelling everything each time would have been $8 \times 9 = 72$.

LEMMA. Starting from MakeUnionFind (all sets singletons), any sequence of k Unions takes $O(k \log k)$ time.

- 1 Cost = relabelled cells.
A singleton is never relabelled; each Union makes at most two elements non-singleton.
So at most $2k$ elements are ever relabelled.
- 2 When v (here 15) is relabelled, its set was the smaller: it at least **doubles**.
- 3 Sizes go from 1 up to at most $2k$:
 v is relabelled at most $\log_2 2k$ times.
- 4 Total: $\leq 2k \cdot \log_2 2k = O(k \log k)$:
on average (**amortised**) $O(\log k)$ per Union.



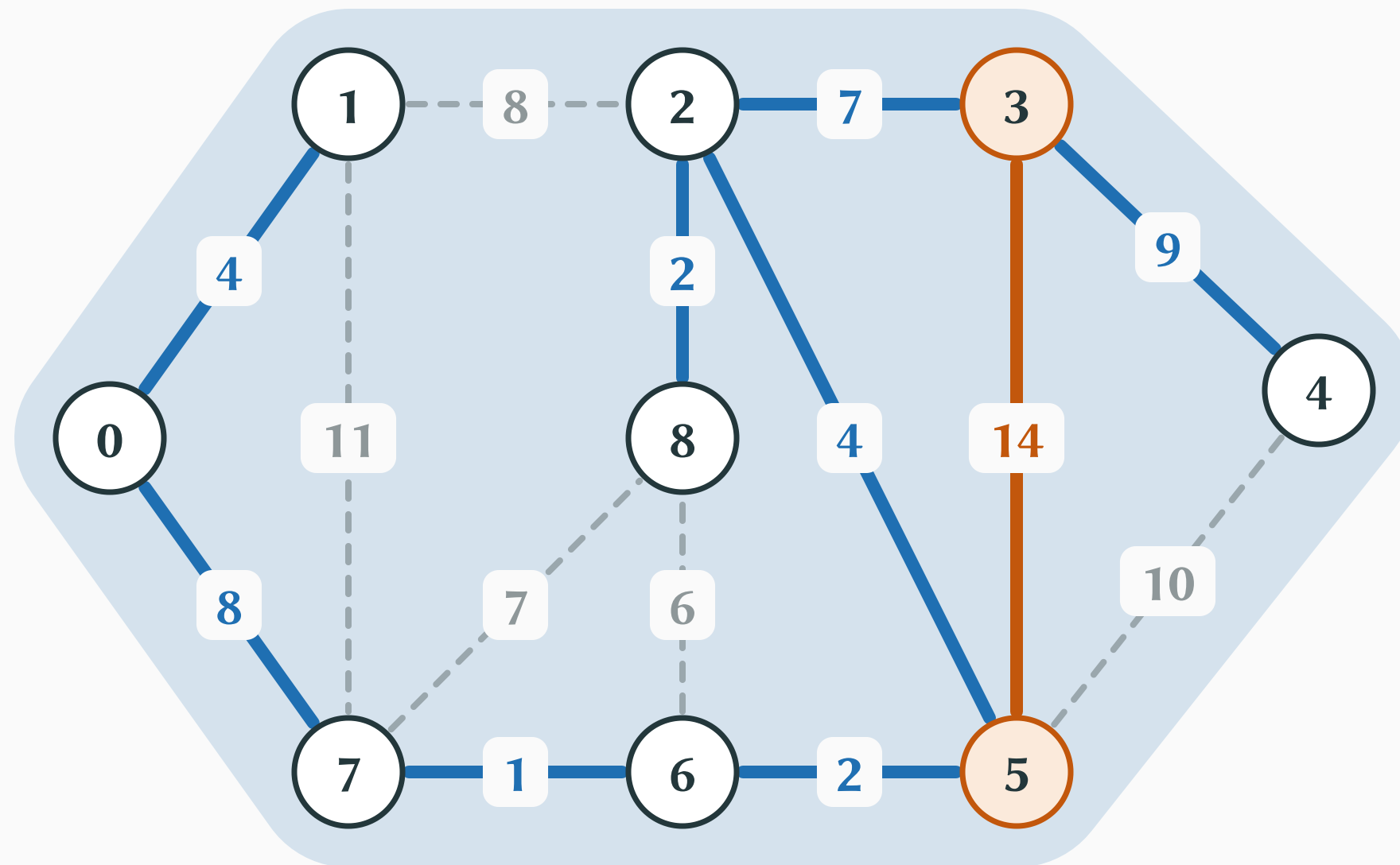
Element 7: depth $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$
tree size $1 \rightarrow 2 \rightarrow 4 \rightarrow 8$



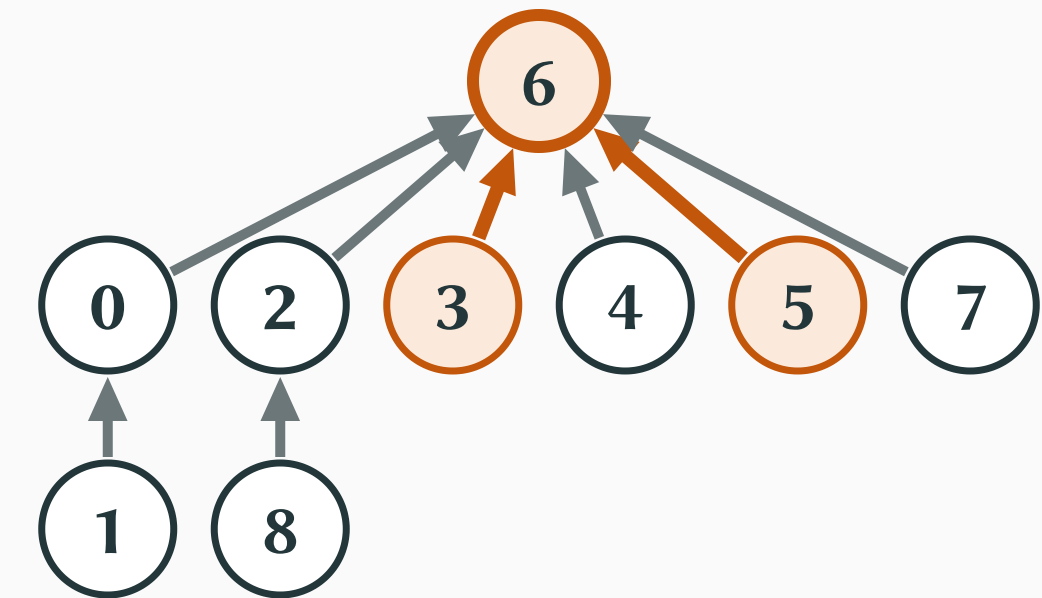
- 1 The depth of u (here 7) grows only when its root goes below another root: by one.
- 2 Union by size: the other tree is at least as large, so the tree of u at least **doubles**.
- 3 Sizes run from 1 to at most n :
depth $\leq \log_2 n$.

Find takes $O(\log n)$ time.

Tight: here $n = 8$, and 7 has depth 3.



Find(3): 1 step $\rightarrow$ 6. Find(5): 1 step $\rightarrow$ 6.
Same root: skip the edge.



28

Finds

19

pointers followed

8

Unions

28 Finds followed 19 pointers, 8 Unions changed 8.
Deepest element: depth 2.

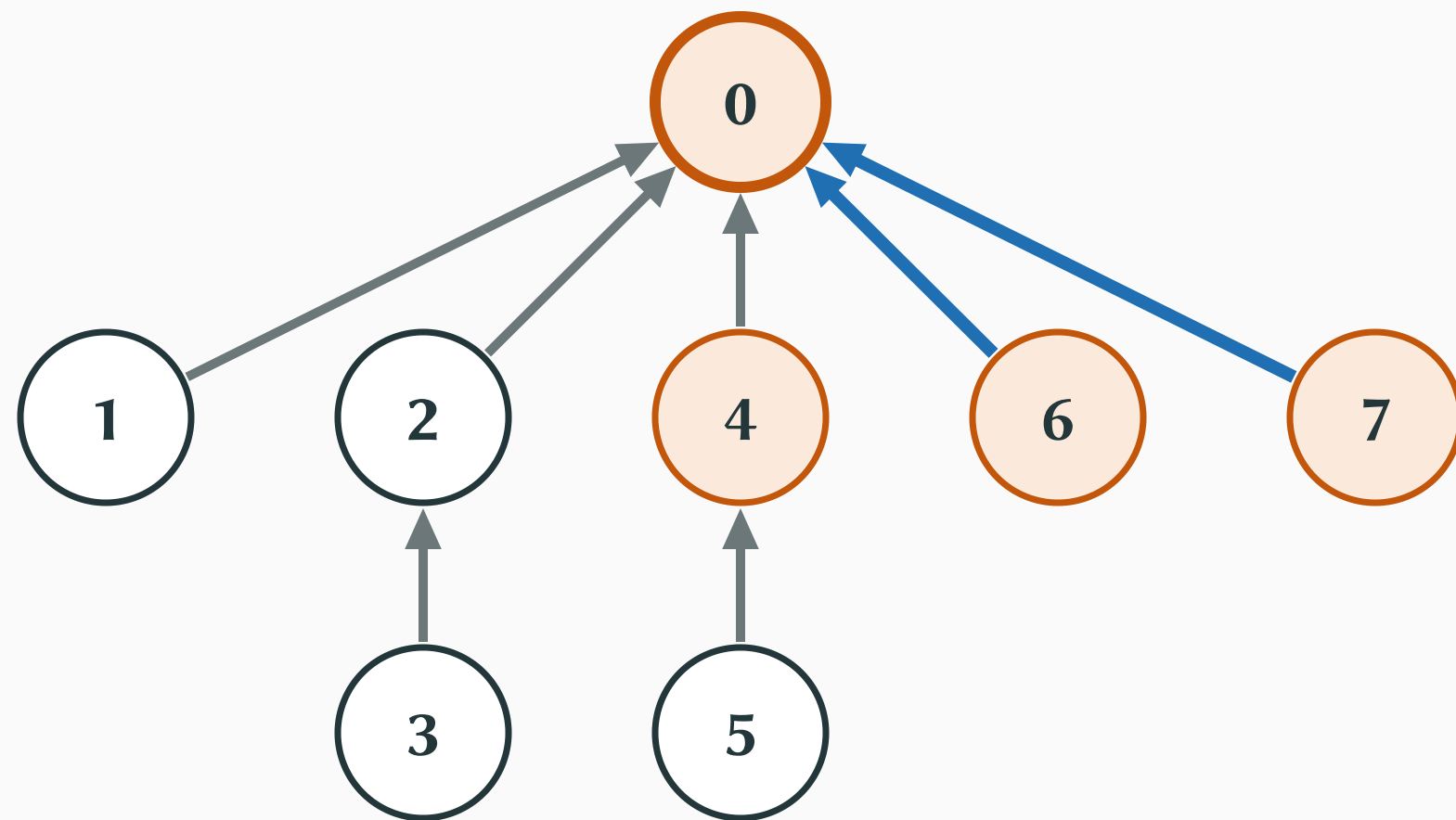
	array of names	forest, union by size
MakeUnionFind	$O(n)$	$O(n)$
$2m$ Finds	$2m \cdot O(1)$	$2m \cdot O(\log n)$
$n - 1$ Unions	$O(n \log n)$ together	$(n - 1) \cdot O(1)$
plus sorting: $O(m \log n)$	$O(m \log n)$	$O(m \log n)$

Kruskal runs in $O(m \log n)$ time, like Prim.

Sorting alone costs $O(m \log n)$. The Union-Find part alone:
array $O(m + n \log n)$, forest $O(m \log n)$.

Reverse-delete, with a BFS/DFS for every edge: $O(m^2)$.

After Find(7) with path compression.



Path compression: after Find(u), point every node on the **path** **directly at the root**.

Same sets, shorter paths:
the next Find(7) takes **1** step.

With union by size **and** path compression, Kruskal's $2m$ Finds and $n - 1$ Unions take $O(m \log^* n)$ time: almost constant each.

$\log^* n$: how often $\log_2$ must be applied to n to reach 1;
at most 5 for $n \leq 2^{65536}$.

Details: KT pp. 197–199.

1 Priority queue as a binary heap.

Insert, ExtractMin, ChangeKey: $O(\log n)$, one walk along a path.

Prim: n ExtractMins, $\leq m$ ChangeKeys.

2 Kruskal needs Union-Find: sorting, $2m$ Finds, $n - 1$ Unions.

- **Doubling** bounds both versions:
relabelling in the array (k Unions: $O(k \log k)$),
depth in the forest (Find: $O(\log n)$).

Prim and Kruskal both run in $O(m \log n)$ time.

The right data structure is part of the algorithm.

Reference: CLRS, Chapter 21.2