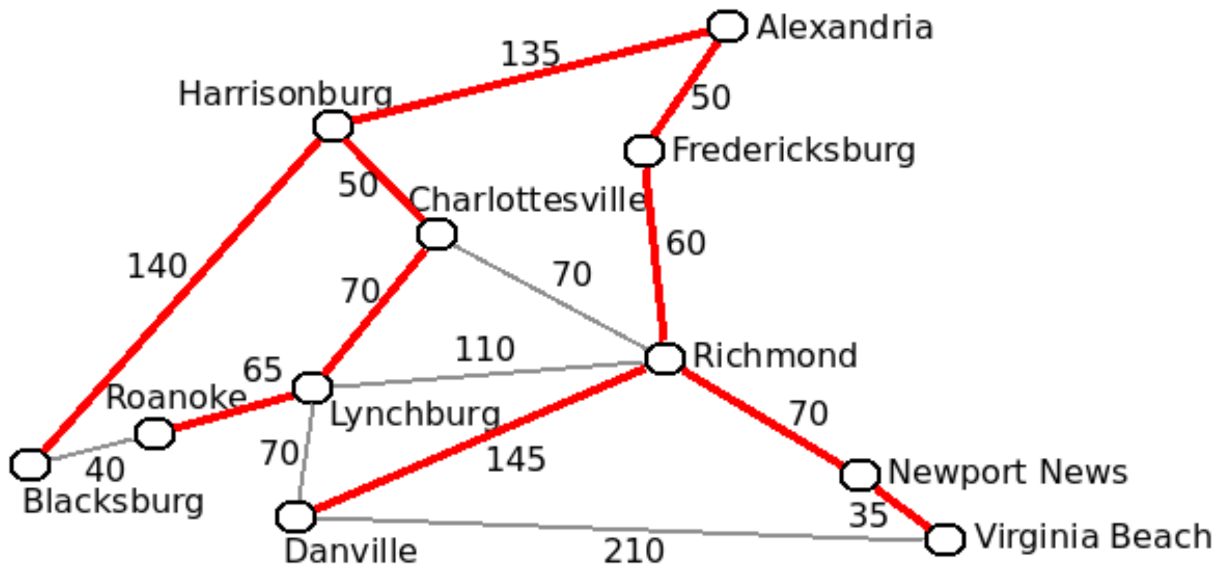
Minimum spanning trees (very quickly) and UNION-FIND data structure

Minimum spanning trees

Computation of a minimal-cost tree
covering a connected undirected graph



Various applications in network design (telephone, electric, roads,…)

Algorithms

Prim's algorithm

suitable for adjacency matrices

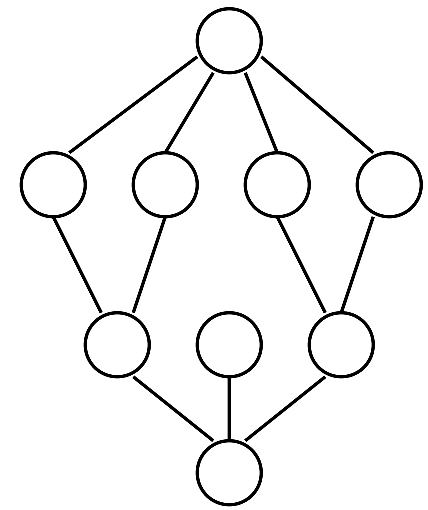
$O(n^2)$, $O(m \log n)$, can be made $O(m + n \cdot \log n)$

Kruskal's algorithm

suitable for adjacency lists and sparse graphs ($|A| \ll n^2$)

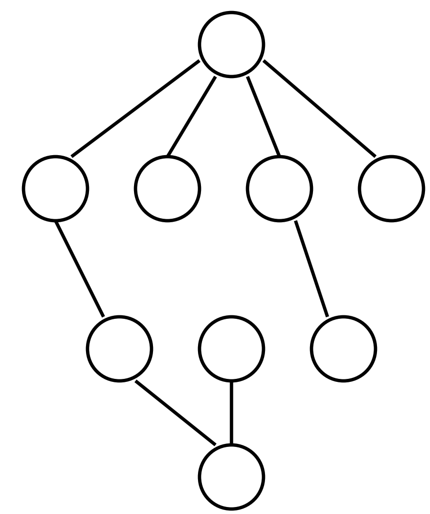
$O(m \log n)$

Spanning tree



$G = (V, E), |V| = n$, undirected connected graph

Spanning tree
$G = (V, E'), E' \subseteq E : E'$ connects all vertices of V and is acyclic (i.e. forms a tree)



Any spanning tree has $n - 1$ edges

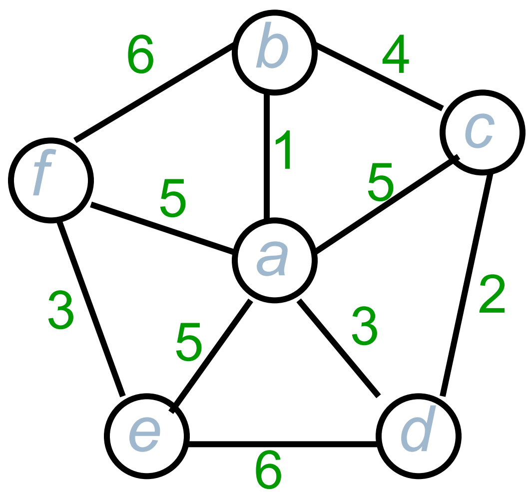
A spanning tree can be found by DFS or BFS in time $O(m + n)$

Minimum(-weight) spanning tree

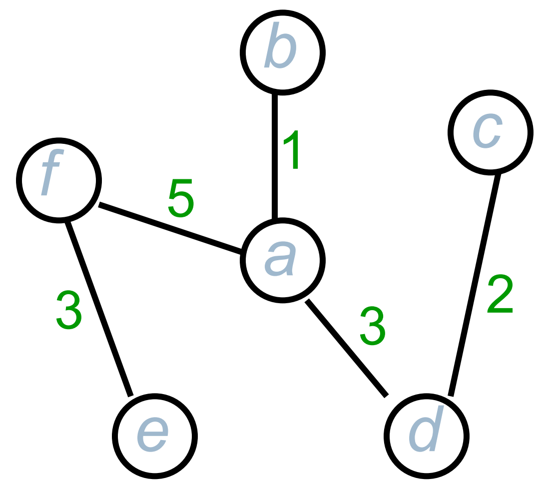
Weighted graph $G = (V, E, w)$ with weights $w : E \to \mathbb{R}$
undirected and connected

Cost of a subgraph $G' = (V', E')$: $\sum_{(p,q)\in E'} w(p,q)$

Problem: compute a spanning tree of G with minimal cost



cost = 14



Exercises about properties of minimum spanning trees

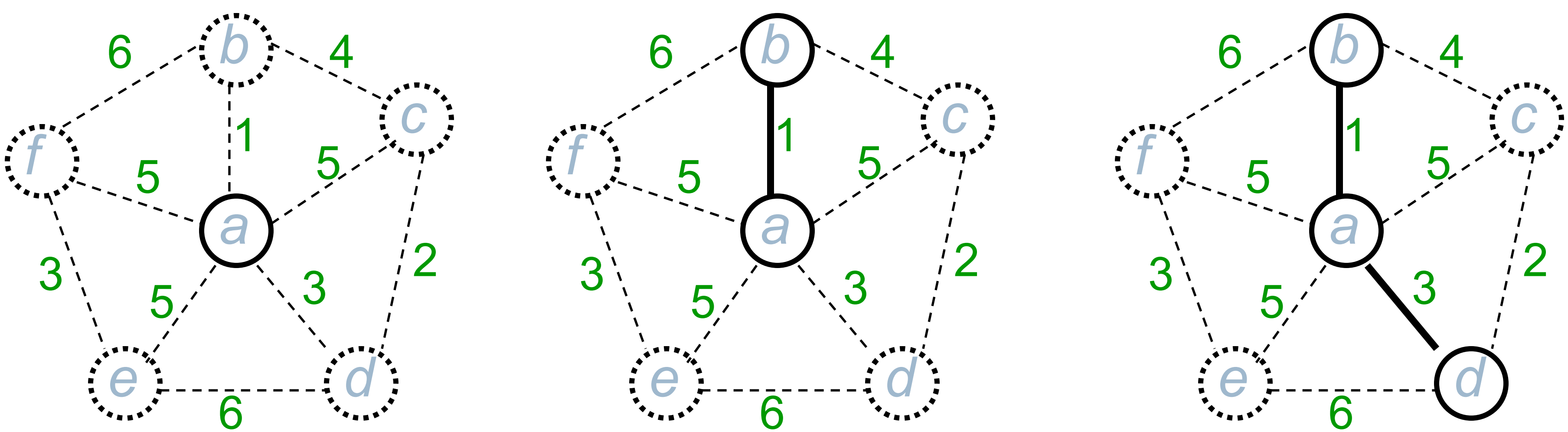
Consider a weighted undirected graph $G = (V, E, w)$.

- Prove that an edge $e = (p, q)$ does not belong to *any* spanning tree if there exists a path between p and q consisting entirely of edges whose cost is smaller than the cost of e.
- *A bit more difficult*: prove that the inverse is true as well, i.e. if e does not belong to *any* spanning tree, then such a path always exists.
- Using the above facts, propose an $O(m + n)$ algorithm that checks if a given edge e belongs to at least one spanning tree

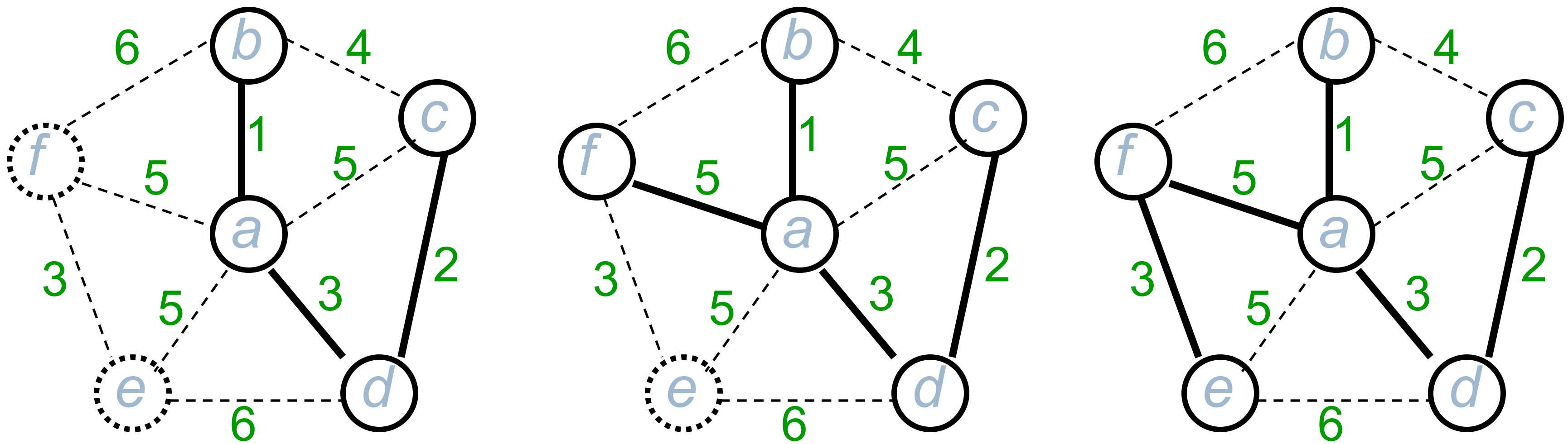
Prim's algorithm

Compute a minimal spanning tree:
"grow" a tree until all graph nodes are covered
(very similar to $stMinCut$ of Stoer-Wagner)

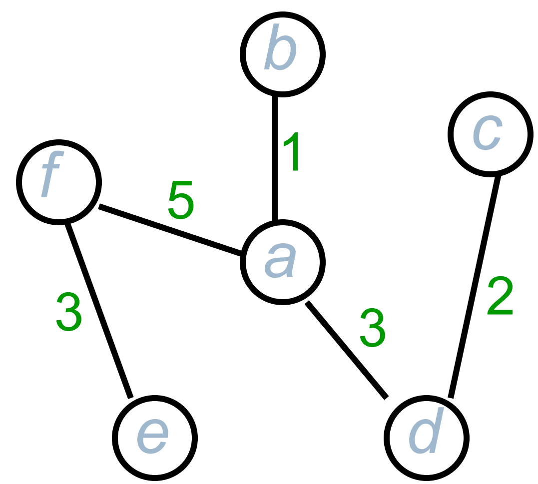
Example:



Example (cont)



minimal ST
cost = 14



Prim's algorithm

```
MST-PRIM( {1, 2, ..., n}, E, w ) {
        T = {1} ;
        B = ∅ ;
        while |T| < n do {
                {p, q} = minimum cost edge
                              such that p ∈ T and q ∉ T ;
                T = T + {q} ;
                B = B + {p, q} ;
        }
        return (B) ;
}
```

Implementation: min-priority queue

With a binary heap:

extracting the minimal cost edge: $O(\log n)$

updating the costs: $O(m)$ updates overall,

each in time $O(\log n)$

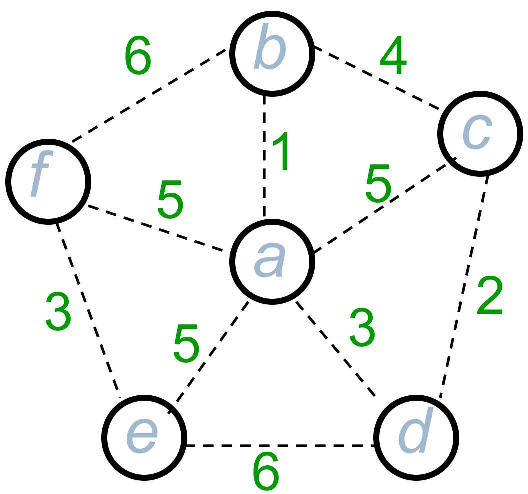
Altogether: $O(n \cdot \log n + m \cdot \log n) = O(m \cdot \log n)$

This can be improved to $O(m + n \cdot \log n)$ with Fibonacci heaps

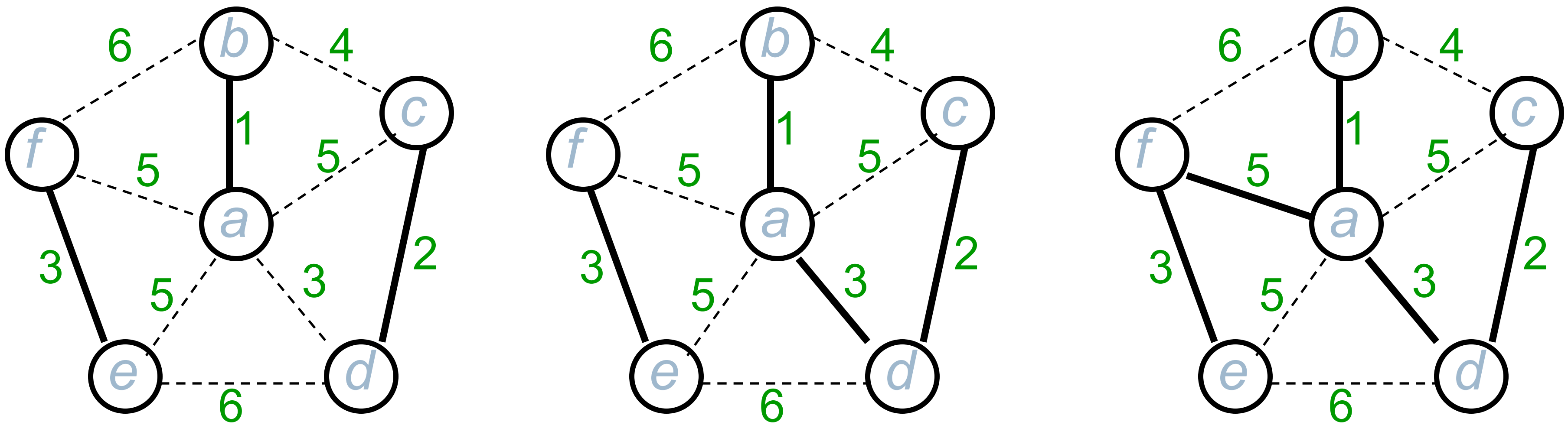
Kruskal's algorithm (1956)

Computation step:
connect two disjoint subtrees by a minimal cost edge

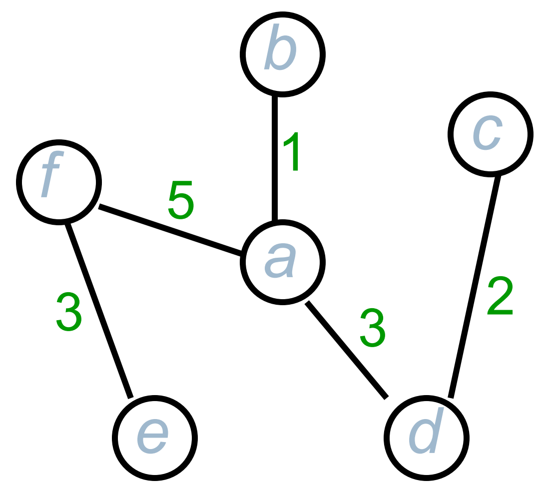
Example:



Example (cont)



minimal ST
cost = 14



Kruskal's algorithm

```
MST-KRUSKAL({1, 2, ..., n}, E, w ) {
        Partition ← { {1}, {2}, ..., {n} } ;
        B = ∅ ;
        while |Partition| > 1 do {
                {p, q} = minimum cost edge such that
                             FIND(p) ≠ FIND(q) ;
                B = B + {p, q} ;
                UNION(p,q);
                // merge the sets containing p and q in Partition;
        }
        return ( B ) ;
}
```

where **FIND**(p) returns an “identifier” of the set containing p

Implementation of Kruskal's algorithm

- Pre-sort all the edges E by increasing cost, in time $O(m \cdot \log m) = O(m \cdot \log n)$
- Process edges in increasing order and repeatedly find the minimum-cost one that has its endpoints in distinct trees

- $2 \cdot m$ operations **`CLASS`** and m operations **`UNION`**
- if **`CLASS`** and **`UNION`** can be implemented in $O(\log n)$, the resulting complexity would be $O(m \cdot \log n)$
- on the next slides, we will see that **`CLASS`** and **`UNION`** can be implemented even in a smaller time than $\log n$

UNION/FIND (maintaining disjoint sets)

Maintaining partitions of {1, 2, …, n} under **operations**

FIND(*p*) : compute a representative (identifier) of the class of *p*

UNION(*p,q*) : union of disjoint classes of *p* and of *q*

Example: n = 7

UNION(1, 2);	{1, 2} {3} {4} {5} {6} {7}
UNION(5, 6);	{1, 2} {3} {4} {5, 6} {7}
UNION(3, 4);	{1, 2} {3, 4} {5, 6} {7}
UNION(1, 4);	{1, 2, 3, 4} {5, 6} {7}
FIND(2)=**FIND**(3)?	YES

Many applications: Kruskal's spanning tree algorithm, computing connected components of an undirected graph, checking equivalence of deterministic finite automata, …

How **UNION** and **FIND** are implemented?

Array implementation

	1	2	3	4	5	6	7
CLASS	1	1	3	3	5	5	7

represents {1,2} {3,4} {5,6} {7}

```
UNION(p,q)
{     x ← FIND(p) ;   y ← FIND(q) ;
      for k ← 1 to n do
            if CLASS [k] = y then
                  CLASS [k] ← x ;
}
```

Time
FIND : O(1)
UNION : O(n)

UNION(1, 4)

	1	2	3	4	5	6	7
CLASS	1	1	1	1	5	5	7

represents {1,2,3,4} {5,6} {7}

Linked list implementation

FIND(*p*): return the head (or tail) of the list of *p*
UNION(*p*,*q*): concatenate the list of *p* with the list of *q* (or vice versa)

1. **Simple linked lists**:
   FIND(*p*): $O(n)$ (*returns tail*)
   UNION(*p*,*q*): $O(n)$ (*requires* FIND)

2. **Linked lists with elements pointing to the head**
   FIND(*p*): $O(1)$
   UNION(*p*,*q*): $O(n)$

3. **Linked lists with elements pointing to the head and length counter** (weighted-union heuristic)
   a sequence of m operations **UNION/FIND** on a set of n elements takes time $O(m + n \cdot \log(n))$

Tree implementation

Main idea: For each set, store its elements in nodes of a tree

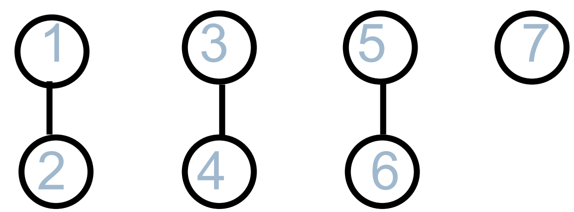
```
FIND(p) {
        k ← p ;
        while parent(k) is defined do k ← parent(k) ;
        return (k) ;
}

UNION(p,q) {
        attach the root of the tree of p as a new child
                of the root of tree of q;
}
```

Time
FIND : $O(n)$
UNION : $O(n)$ (requires finding roots)

UNION: example

partition {1,2} {3,4} {5,6} {7}

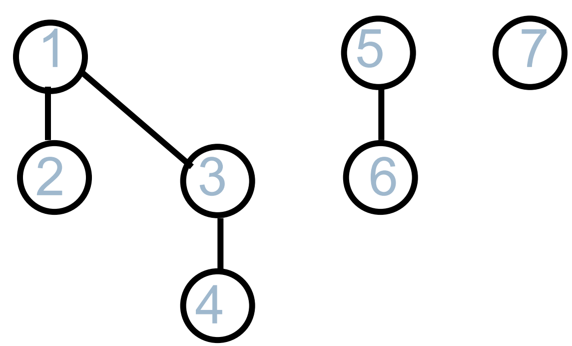
forest



UNION(4,2)

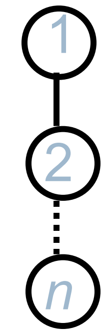
partition {1,2,3,4} {5,6} {7}

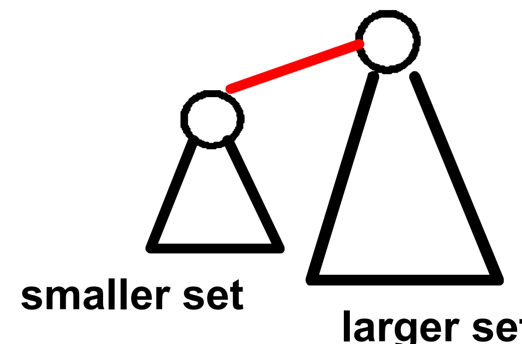
forest



Optimizations

- *Goal*: "balance" trees to reduce the computation time of **`FIND`**(p)
- strategy **union-by-rank**:
  - with each element, store a *rank* that will upper-bound the height of this element in the tree
  - for a single-element tree, the rank $= 0$
  - when merging two tree, compare the ranks of the roots; attach the tree with smaller rank to the one with larger rank
  - in case the two ranks are equal, attach arbitrarily and increment the rank of the root by 1





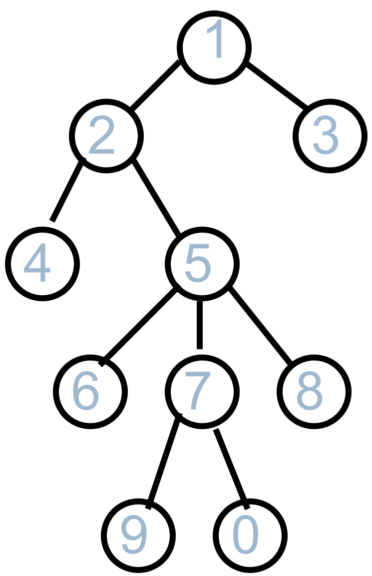
Complexity of UNION with union-by-rank strategy

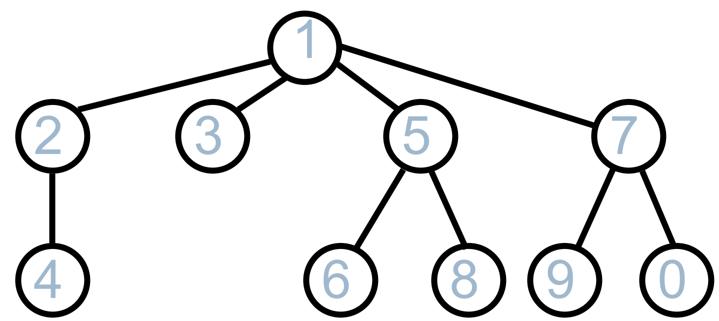
Time
`FIND` : O(log *n*)
`UNION` : O(log *n*) (requires finding the roots)

Proof (sketch**)**
with union-by-rank strategy alone, the rank equals the actual height of the node. It can be shown that every node has rank at most $\lfloor \log_2 n \rfloor$.

Path compression

Idea : flatten the tree by attaching the nodes traversed by **`FIND`**(p) directly to the root





after **`FIND(`**7**`)`**

Combining both strategies union-by-rank and path-compression, time of m calls to **`UNION and FIND`** is $O(m \cdot \alpha(n))$,

where $\alpha(n)$ is the inverse of the Ackermann function.
$\alpha(n) \leq 4$ for all practical purposes