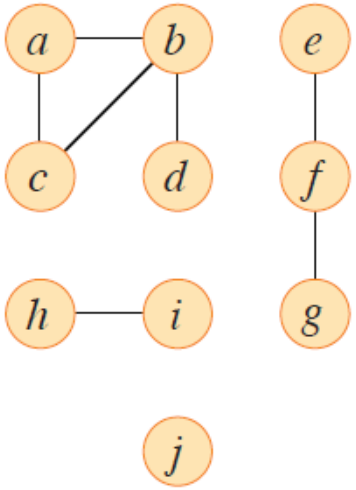
Data Structures for Disjoint-Set



Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b, d)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e, f)	$\{a\}$	$\{b, d\}$	$\{c\}$		$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(a, c)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(h, i)	$\{a, c\}$	$\{b, d\}$			$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(a, b)	$\{a, b, c, d\}$				$\{e, f\}$		$\{g\}$	$\{h, i\}$		$\{j\}$
(f, g)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$
(b, c)	$\{a, b, c, d\}$				$\{e, f, g\}$			$\{h, i\}$		$\{j\}$

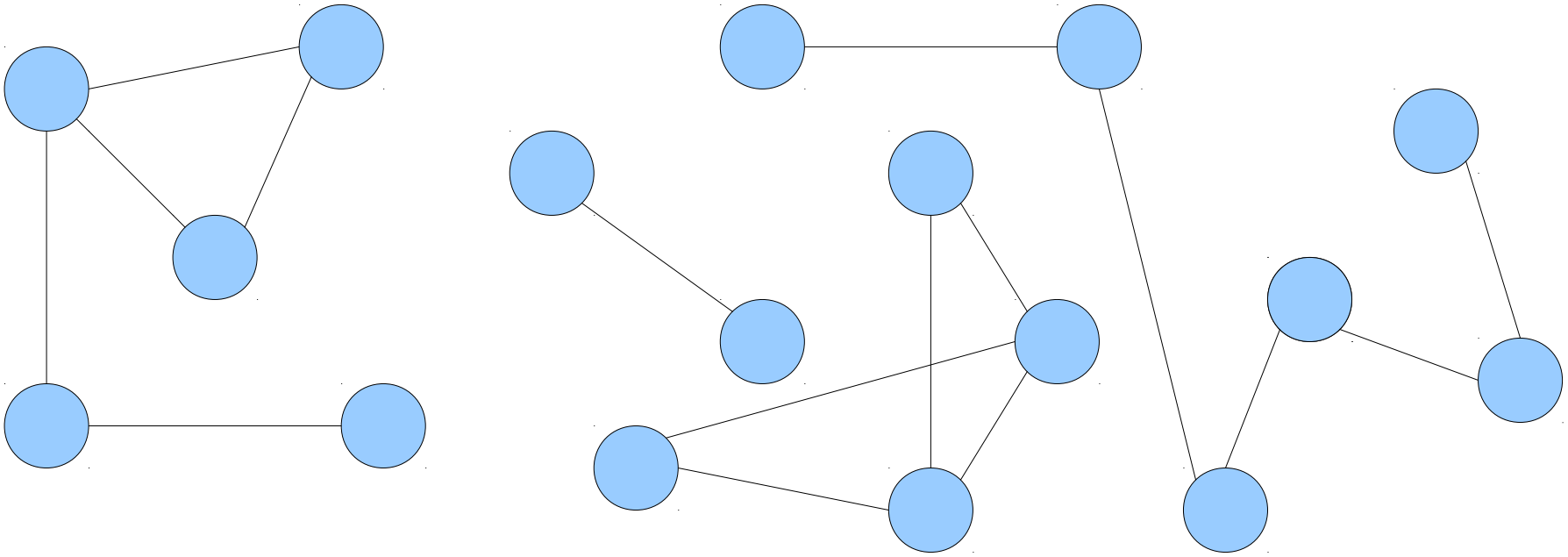
(b)

Disjoint-Set Forests

The Connectivity Problem

- The ***graph connectivity problem*** is the following:

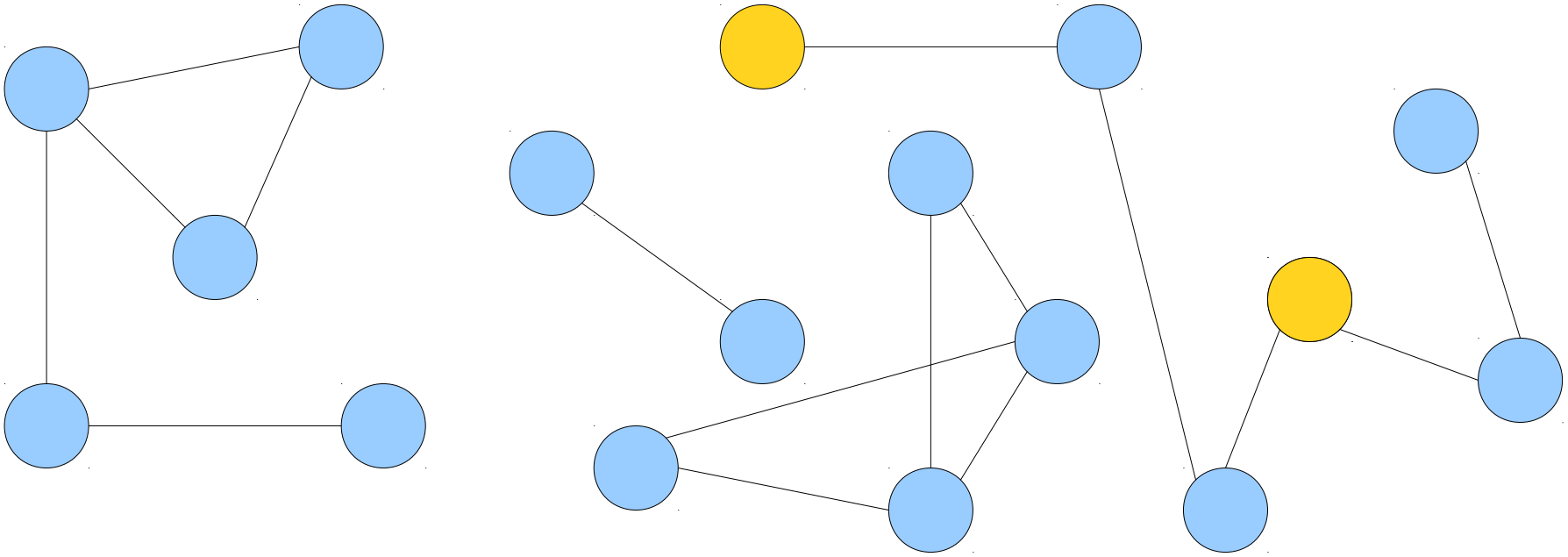
  Given an undirected graph *G*, preprocess the graph so that queries of the form "are nodes *u* and *v* connected?"



The Connectivity Problem

- The ***graph connectivity problem*** is the following:

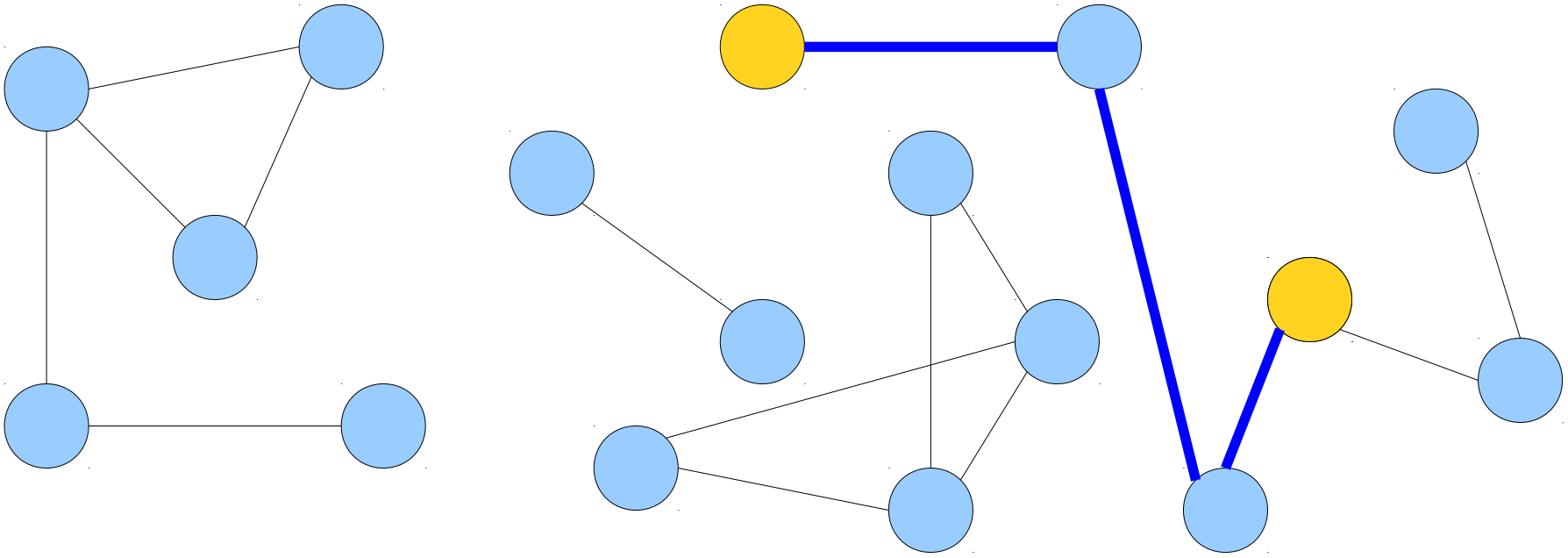
  Given an undirected graph *G*, preprocess the graph so that queries of the form "are nodes *u* and *v* connected?"



The Connectivity Problem

- The ***graph connectivity problem*** is the following:

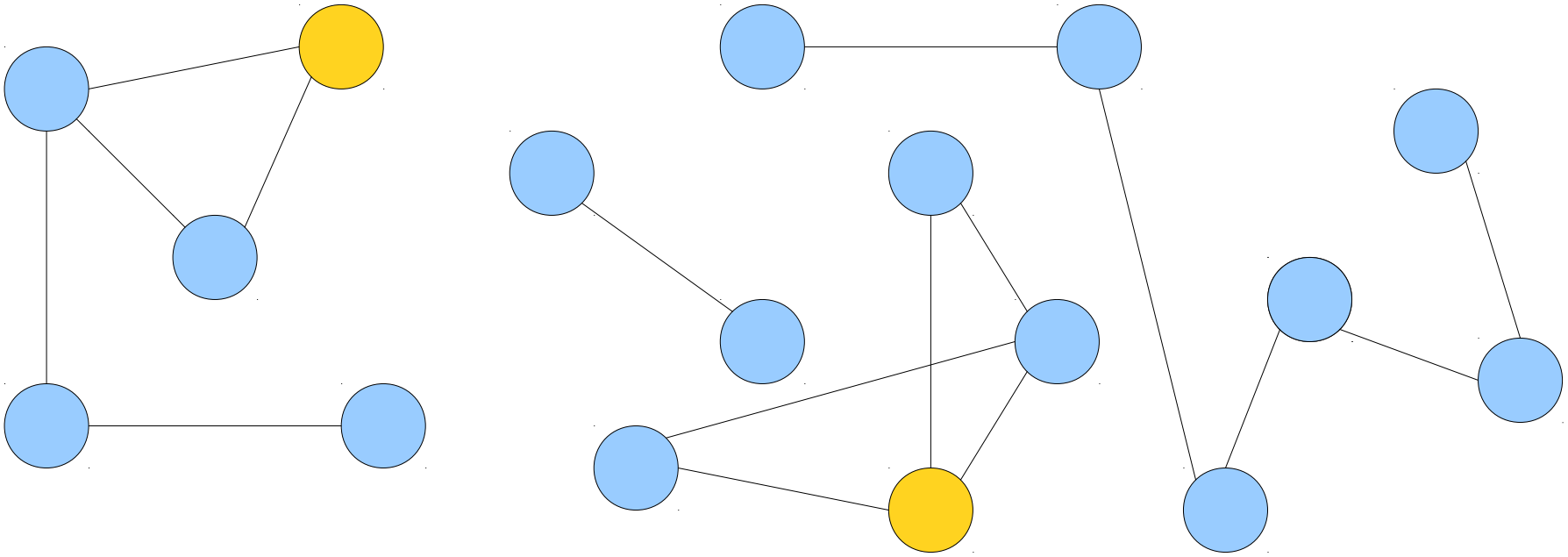
  Given an undirected graph *G*, preprocess the graph so that queries of the form "are nodes *u* and *v* connected?"



The Connectivity Problem

- The ***graph connectivity problem*** is the following:

  Given an undirected graph *G*, preprocess the graph so that queries of the form "are nodes *u* and *v* connected?"

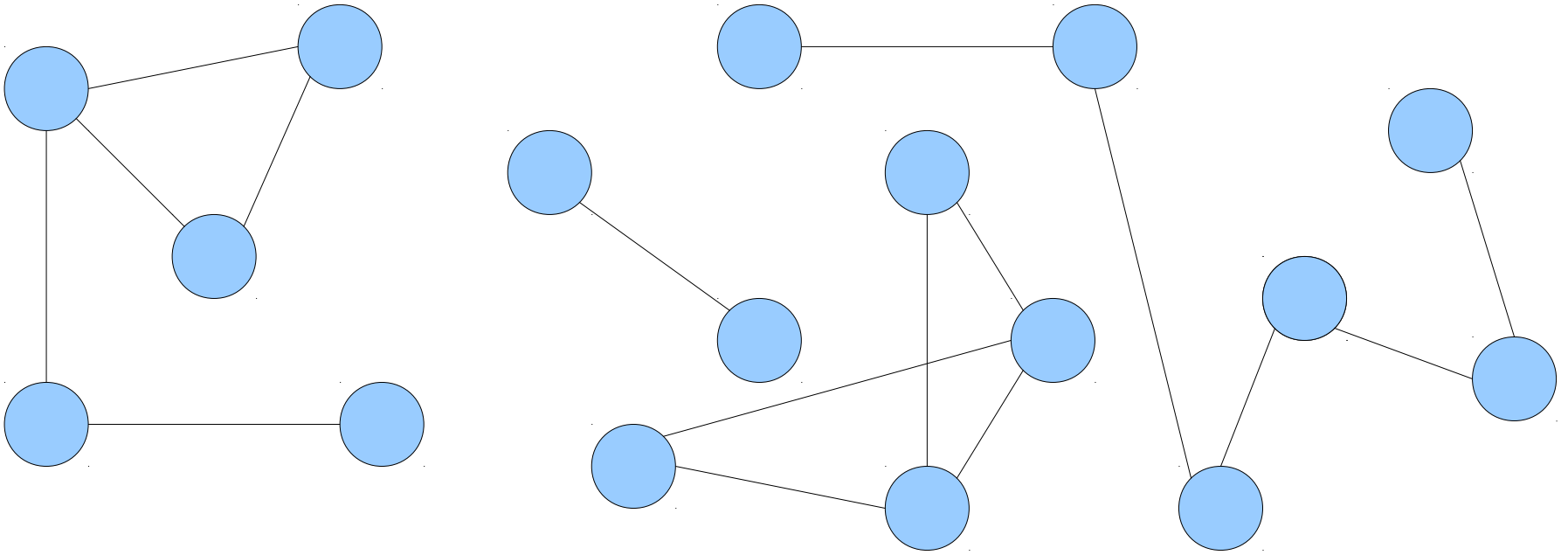


The Connectivity Problem

- The ***graph connectivity problem*** is the following:

  Given an undirected graph G, preprocess the graph so that queries of the form "are nodes u and v connected?"

- Using $\Theta(m + n)$ preprocessing, can preprocess the graph to answer queries in time O(1).

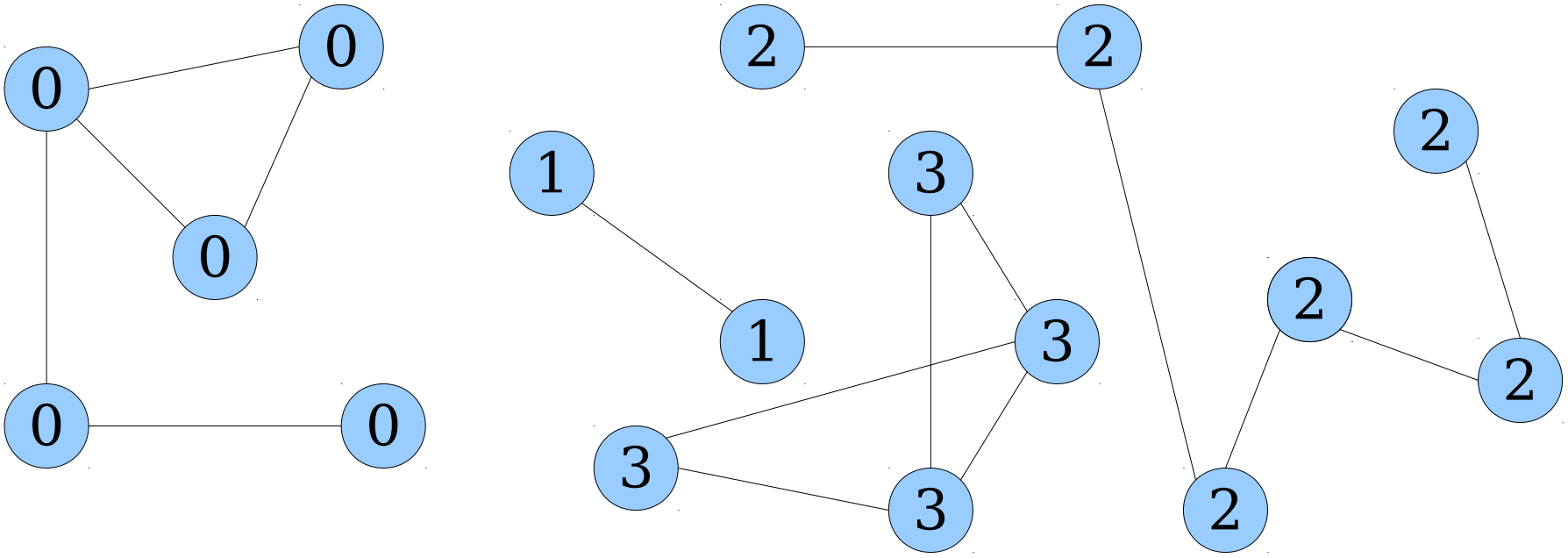


The Connectivity Problem

- The ***graph connectivity problem*** is the following:

  Given an undirected graph G, preprocess the graph so that queries of the form "are nodes u and v connected?"

- Using $\Theta(m + n)$ preprocessing, can preprocess the graph to answer queries in time O(1).

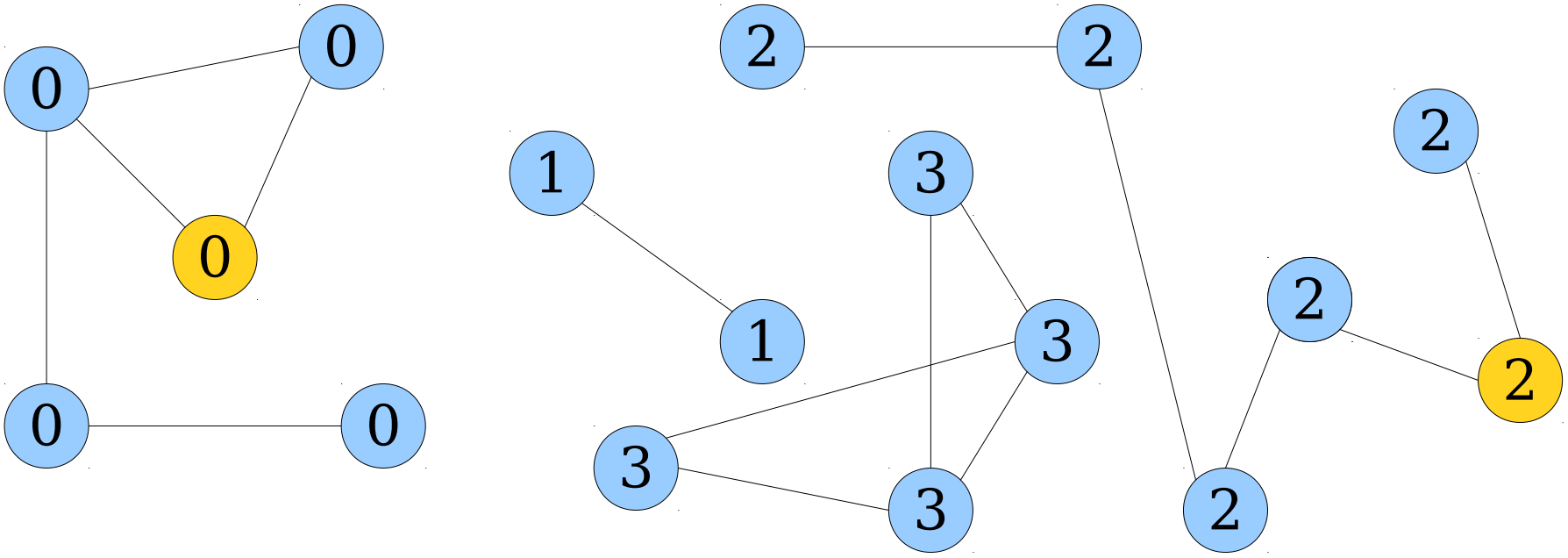


The Connectivity Problem

- The ***graph connectivity problem*** is the following:

  Given an undirected graph G, preprocess the graph so that queries of the form "are nodes u and v connected?"

- Using $\Theta(m + n)$ preprocessing, can preprocess the graph to answer queries in time O(1).

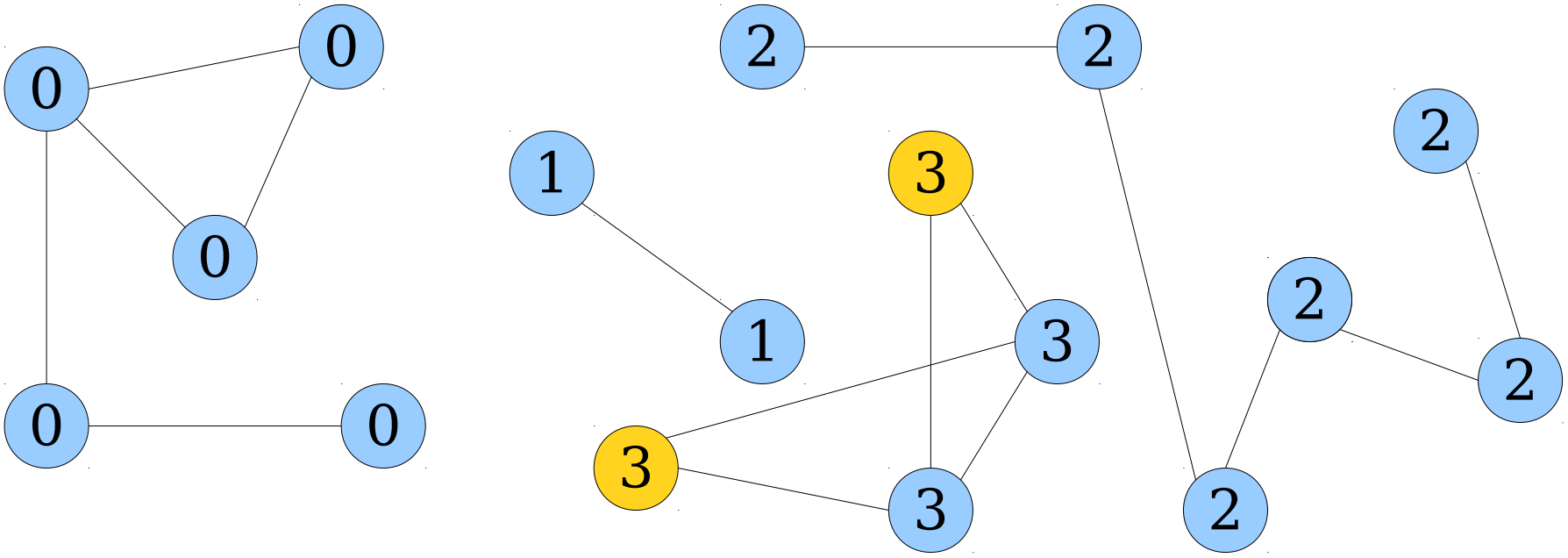


The Connectivity Problem

- The ***graph connectivity problem*** is the following:

  Given an undirected graph G, preprocess the graph so that queries of the form "are nodes u and v connected?"

- Using $\Theta(m + n)$ preprocessing, can preprocess the graph to answer queries in time O(1).

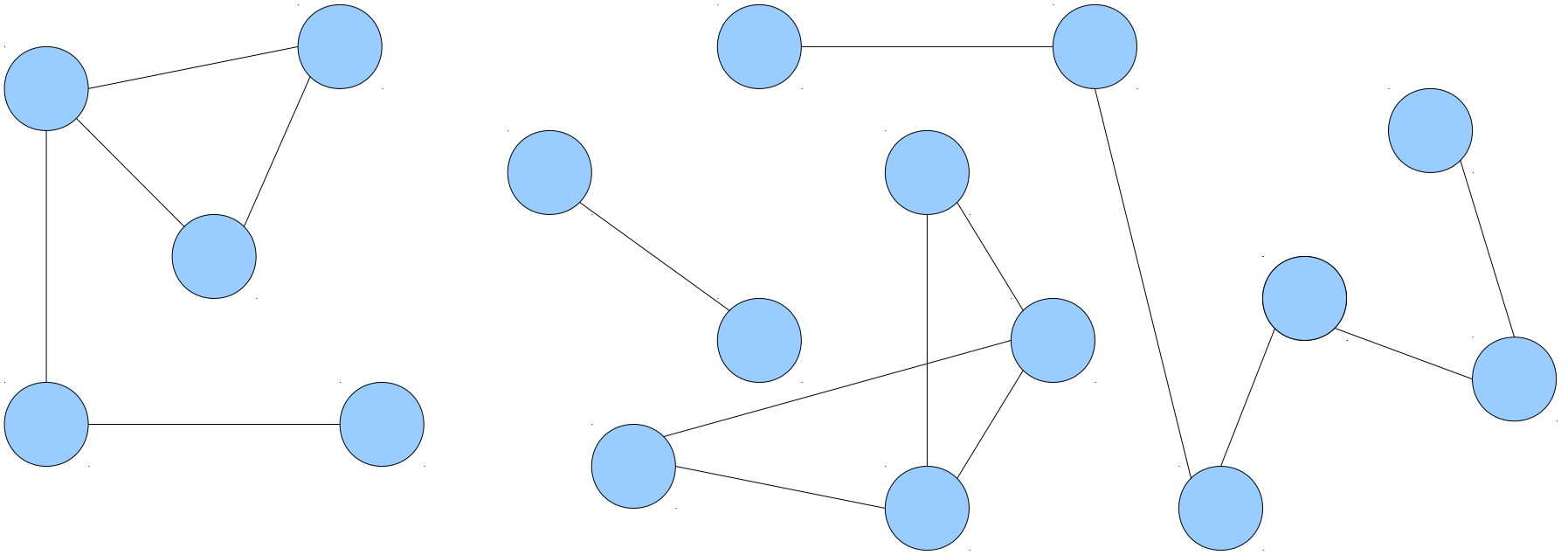


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

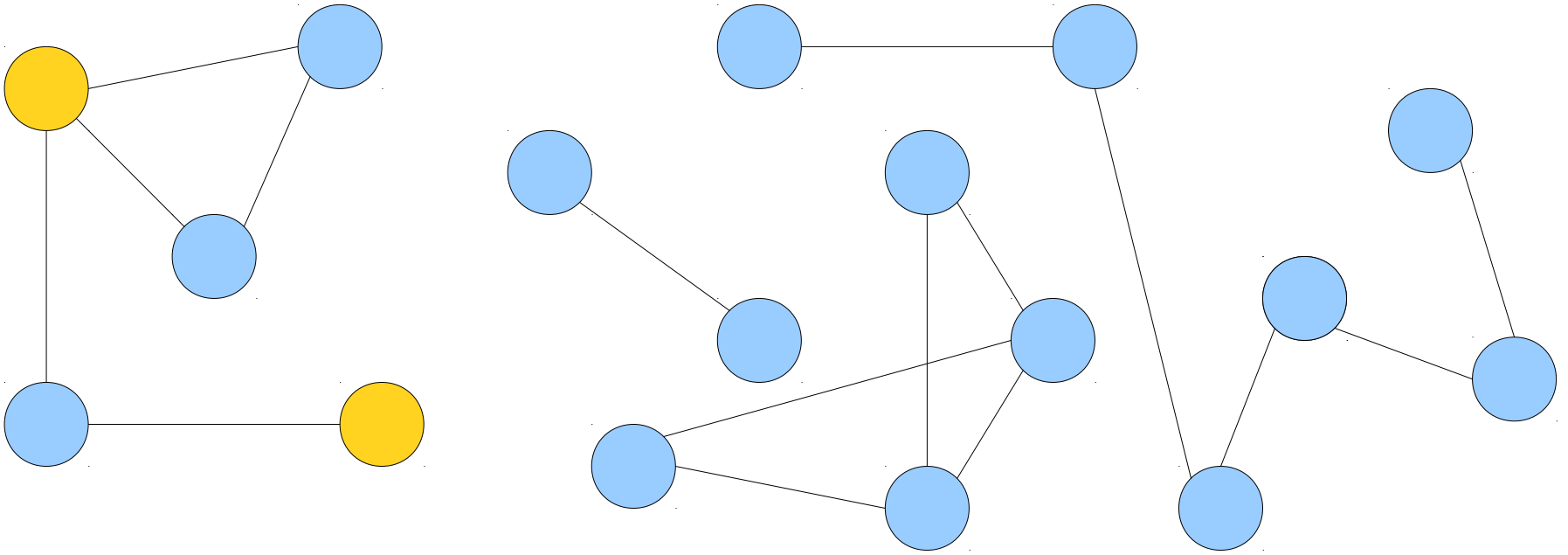


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

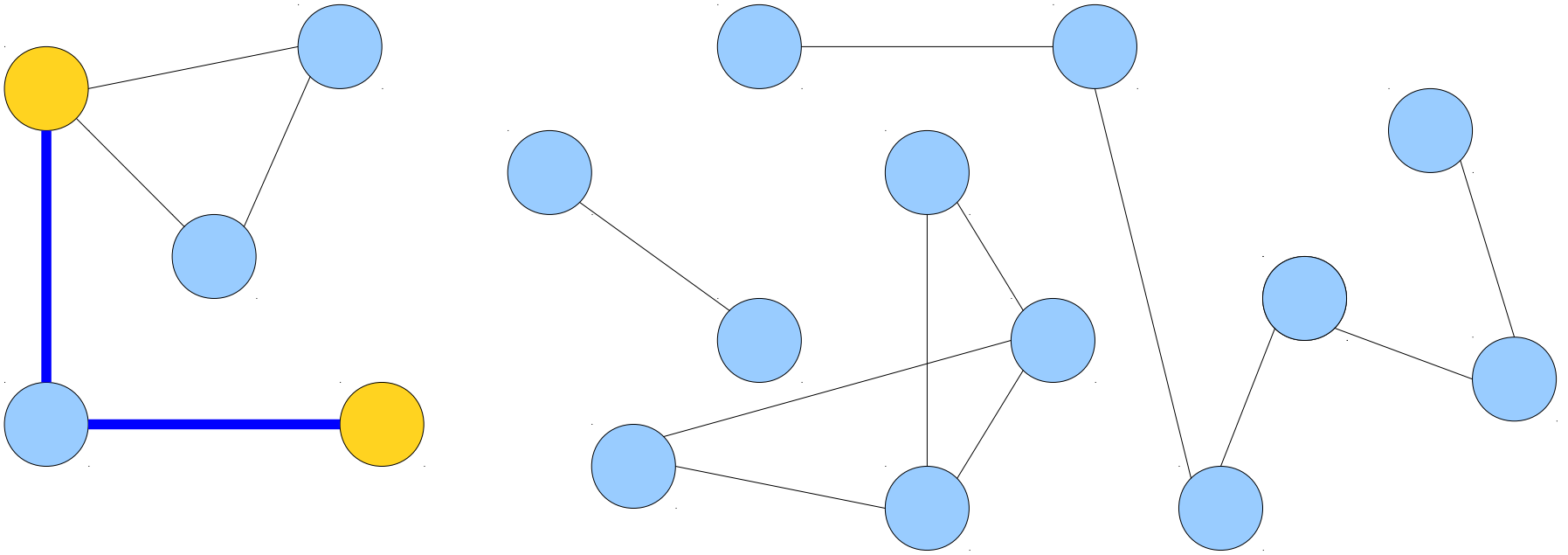


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

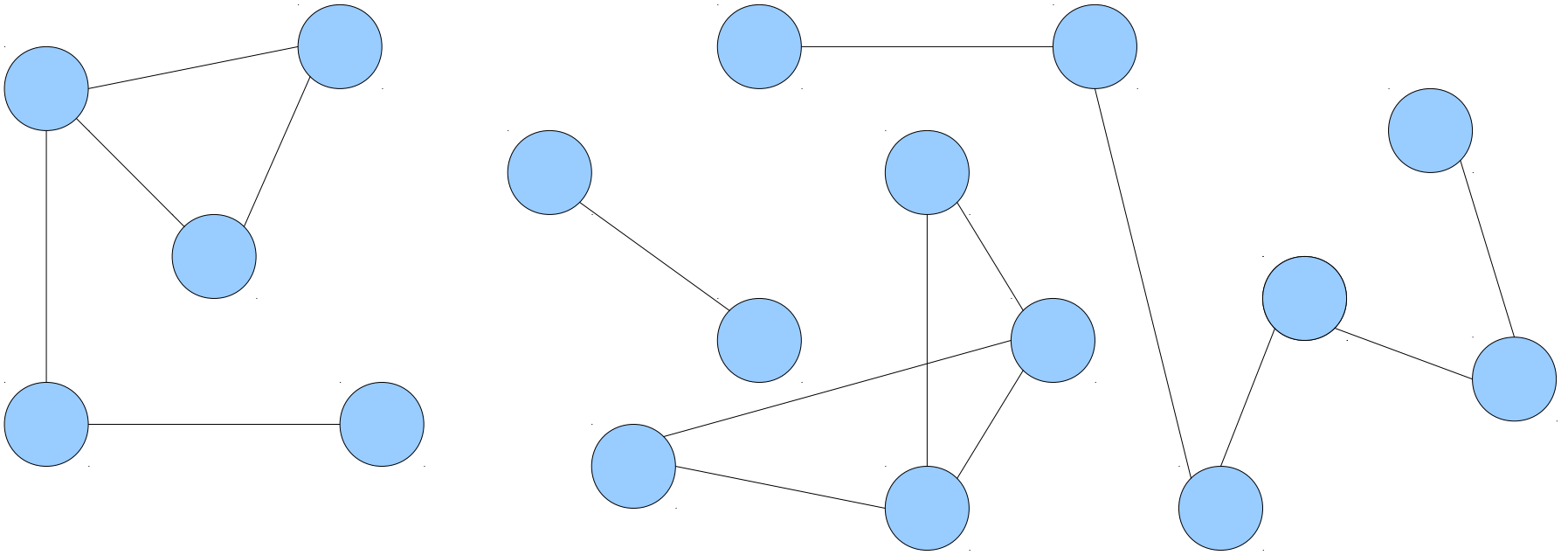


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

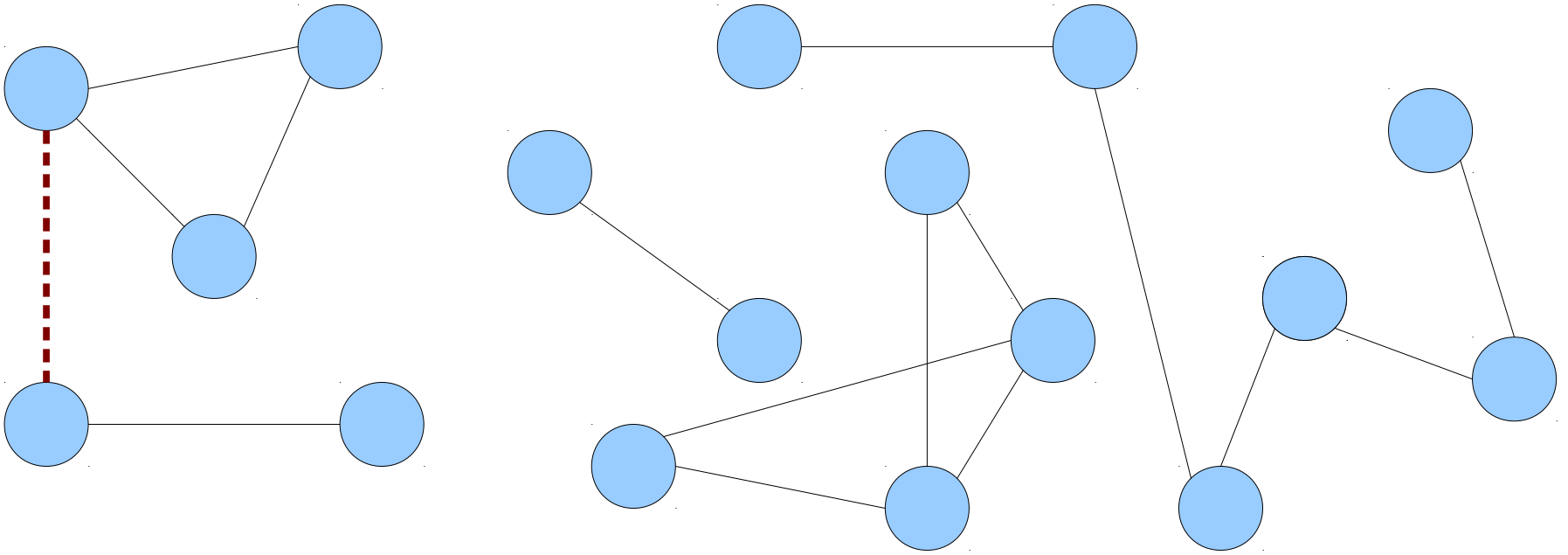


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

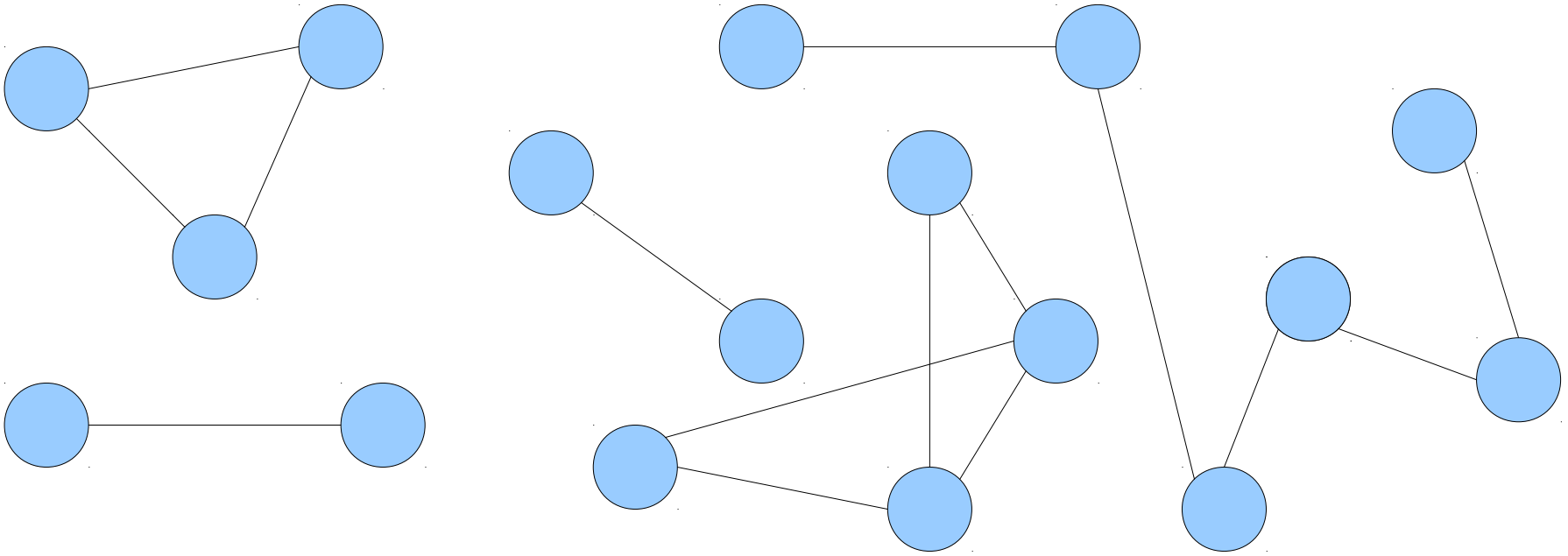


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

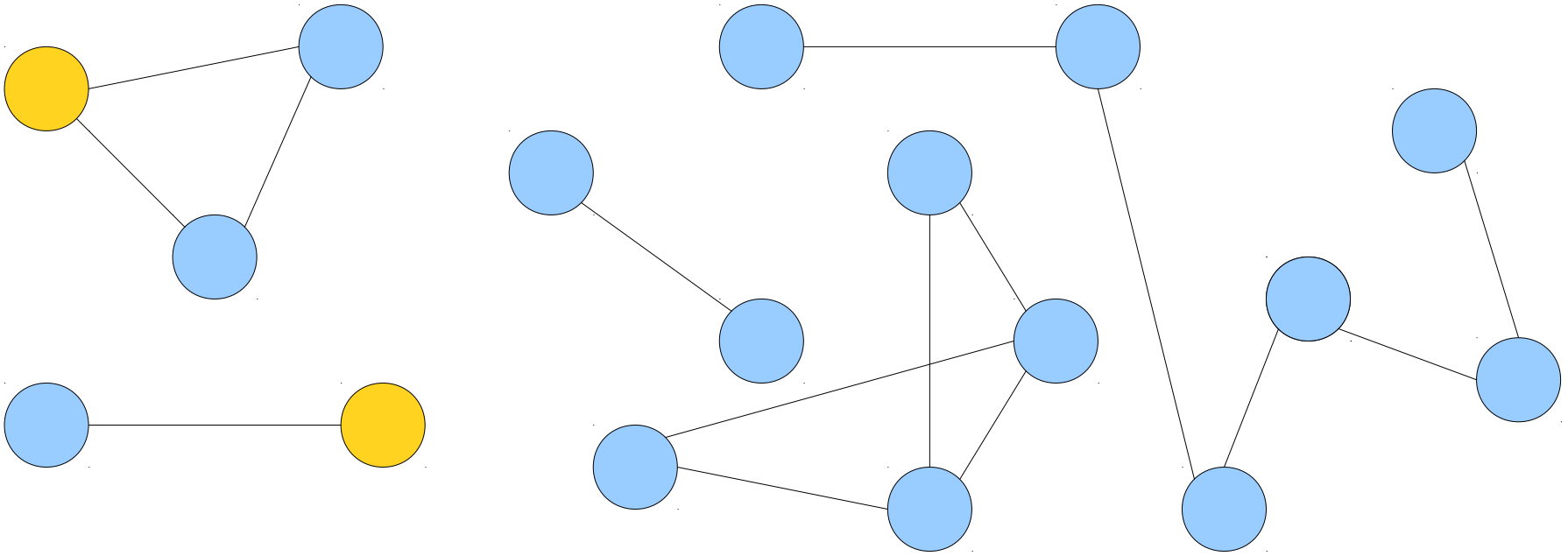


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

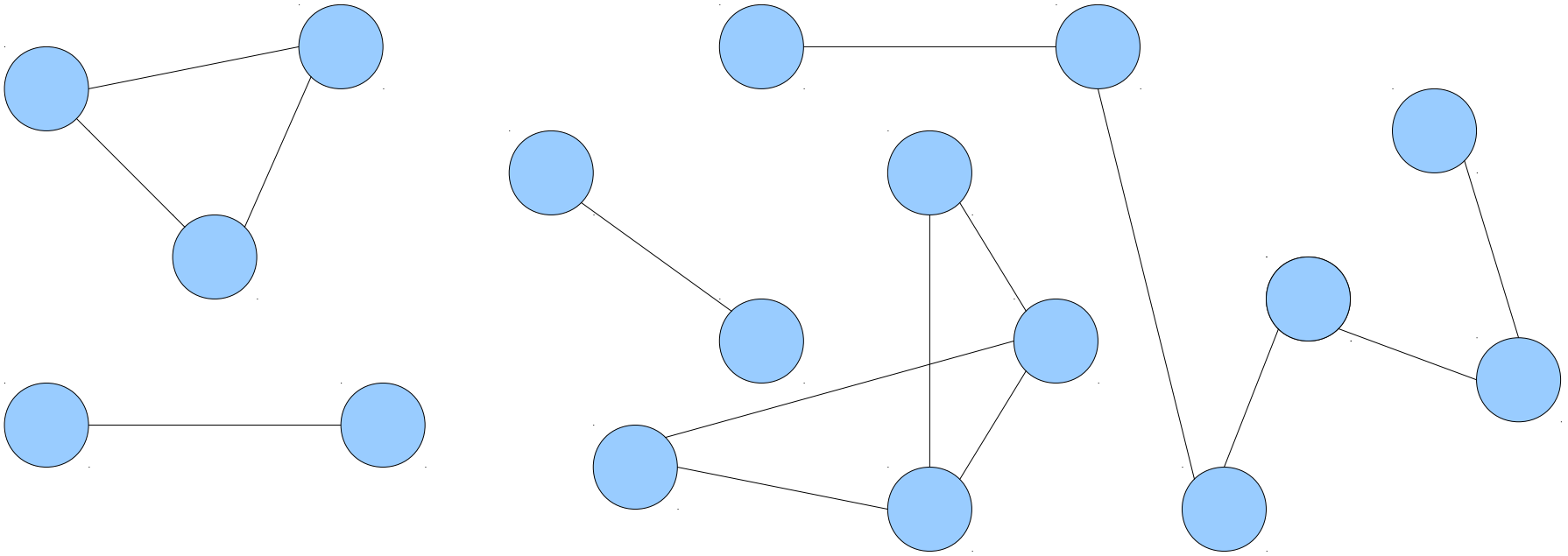


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

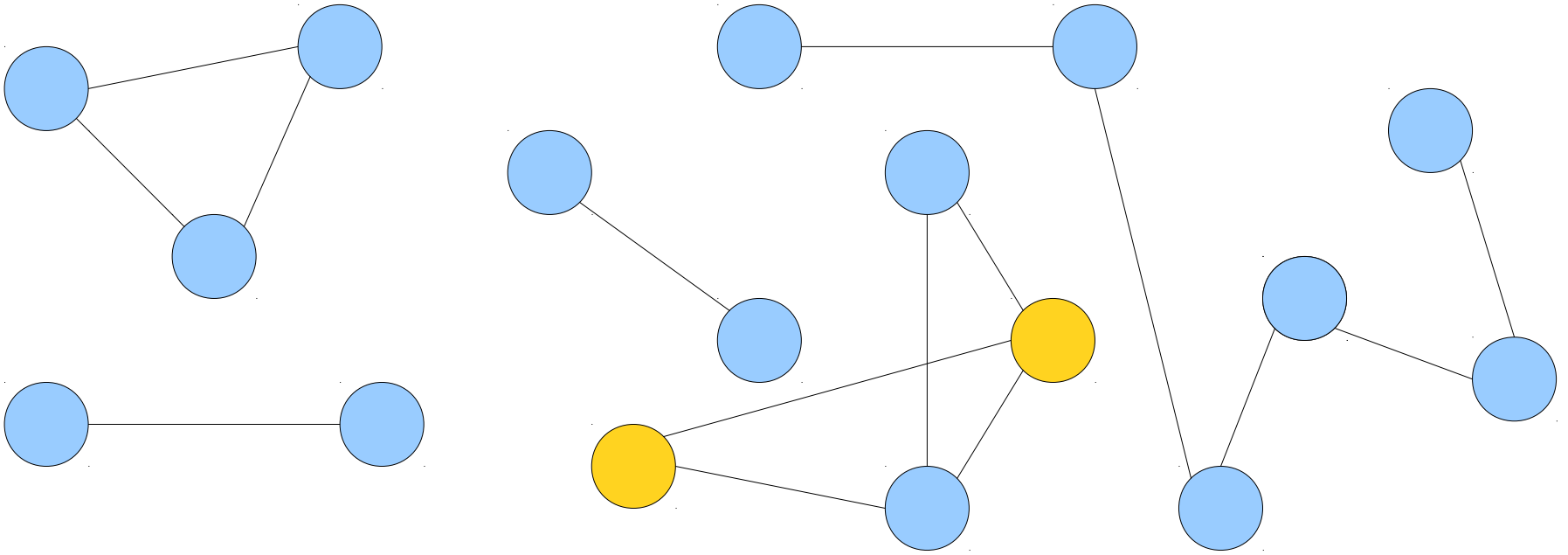


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

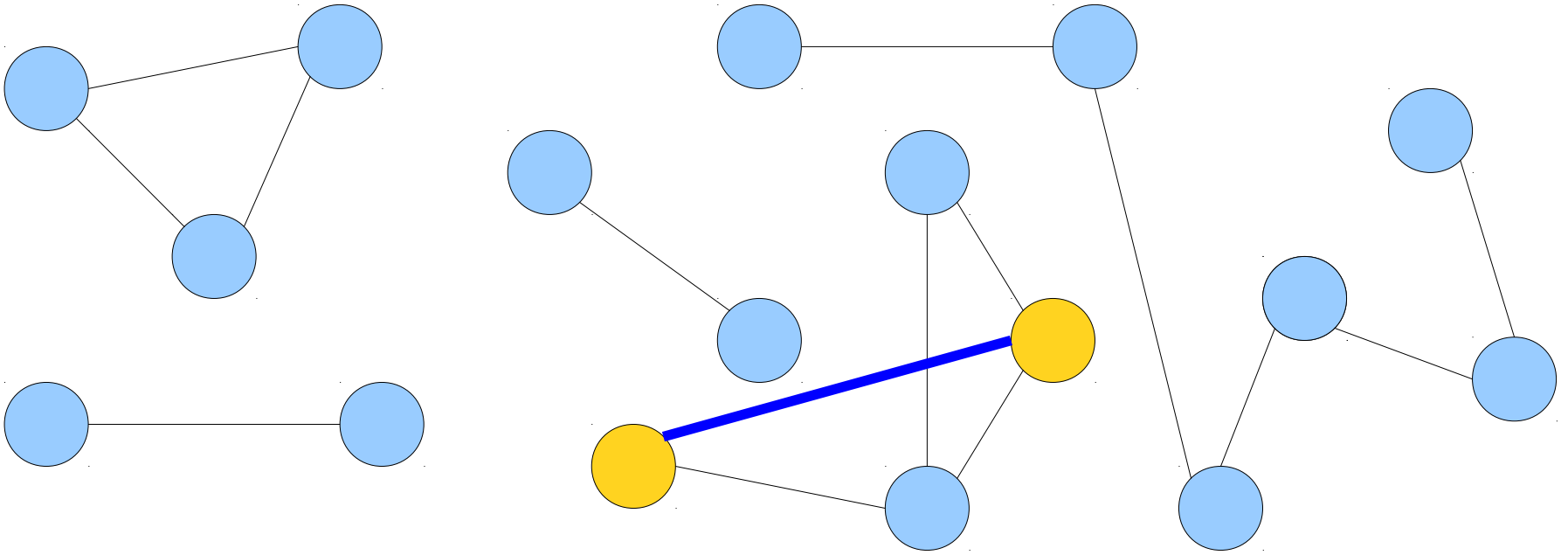


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

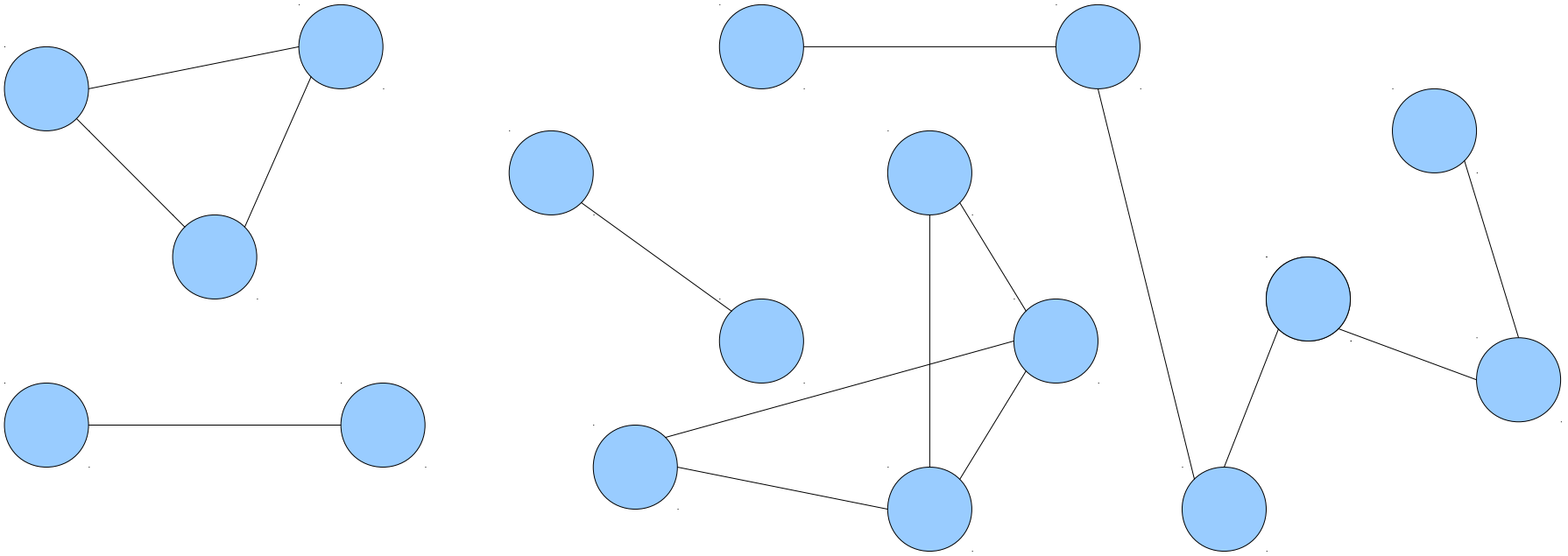


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

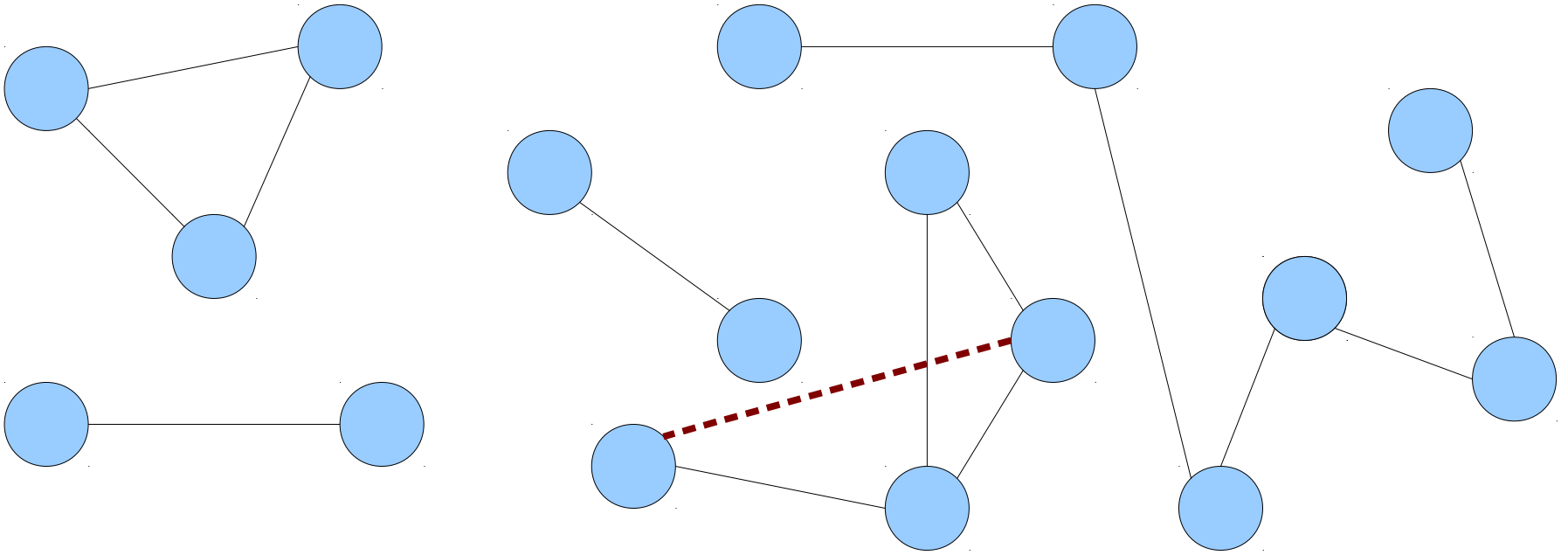


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

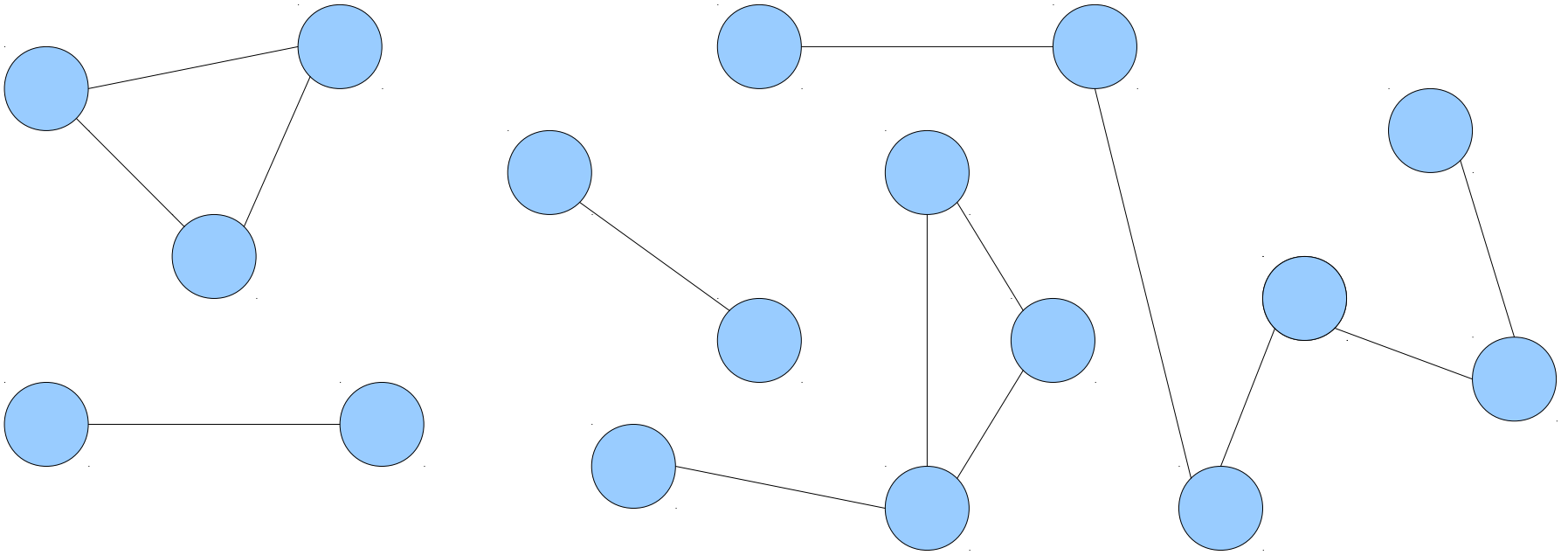


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

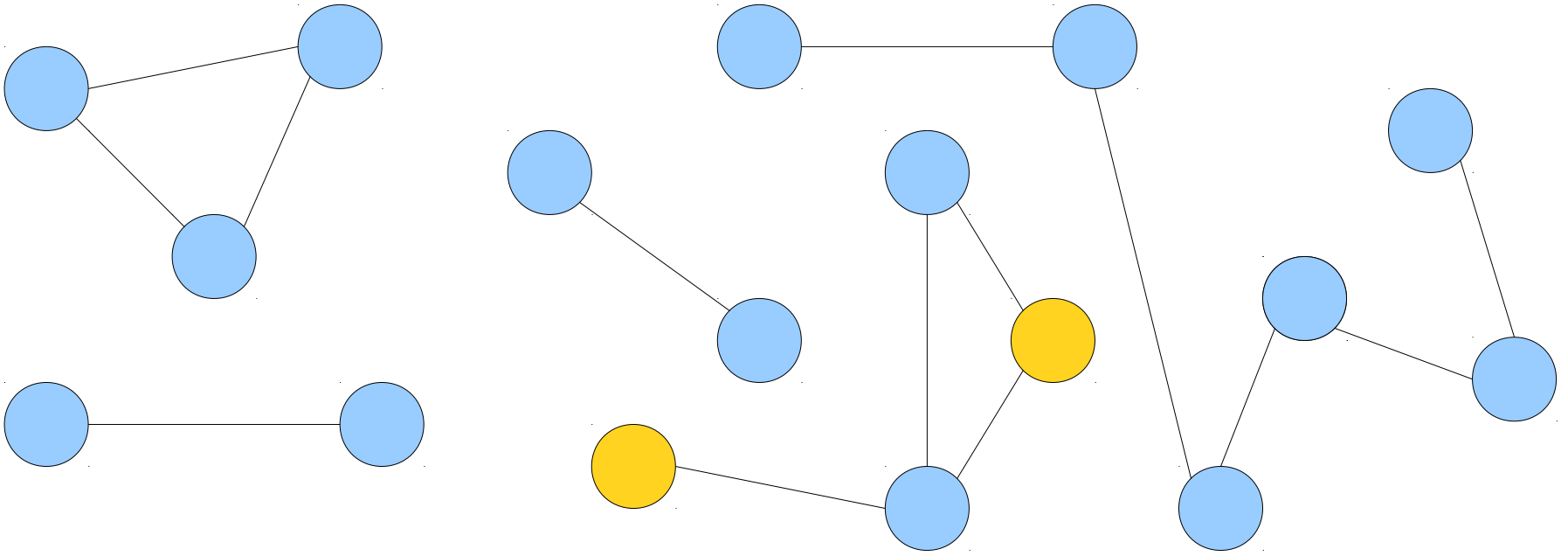


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

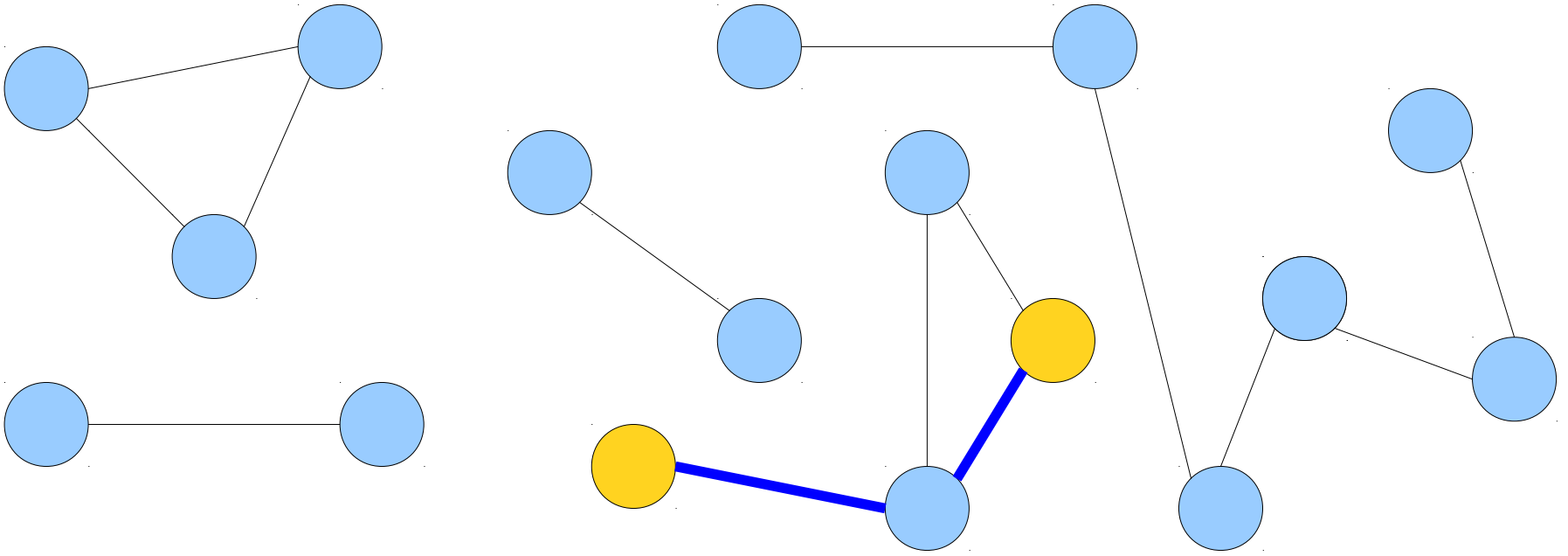


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

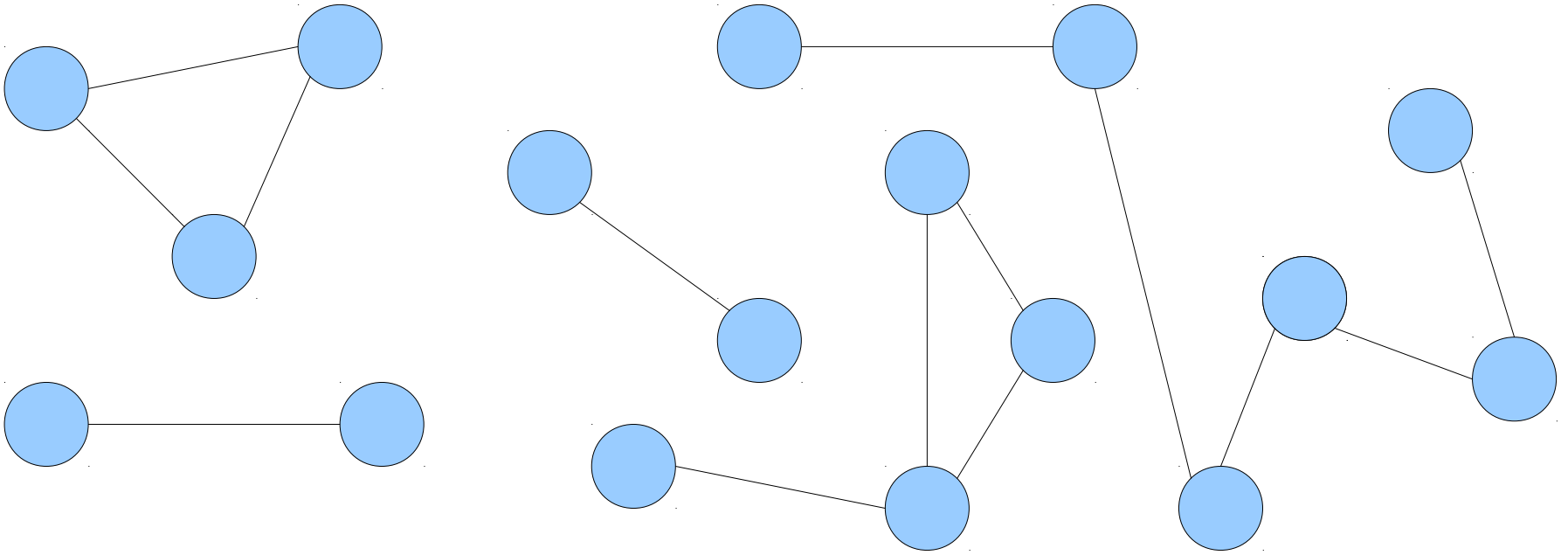


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

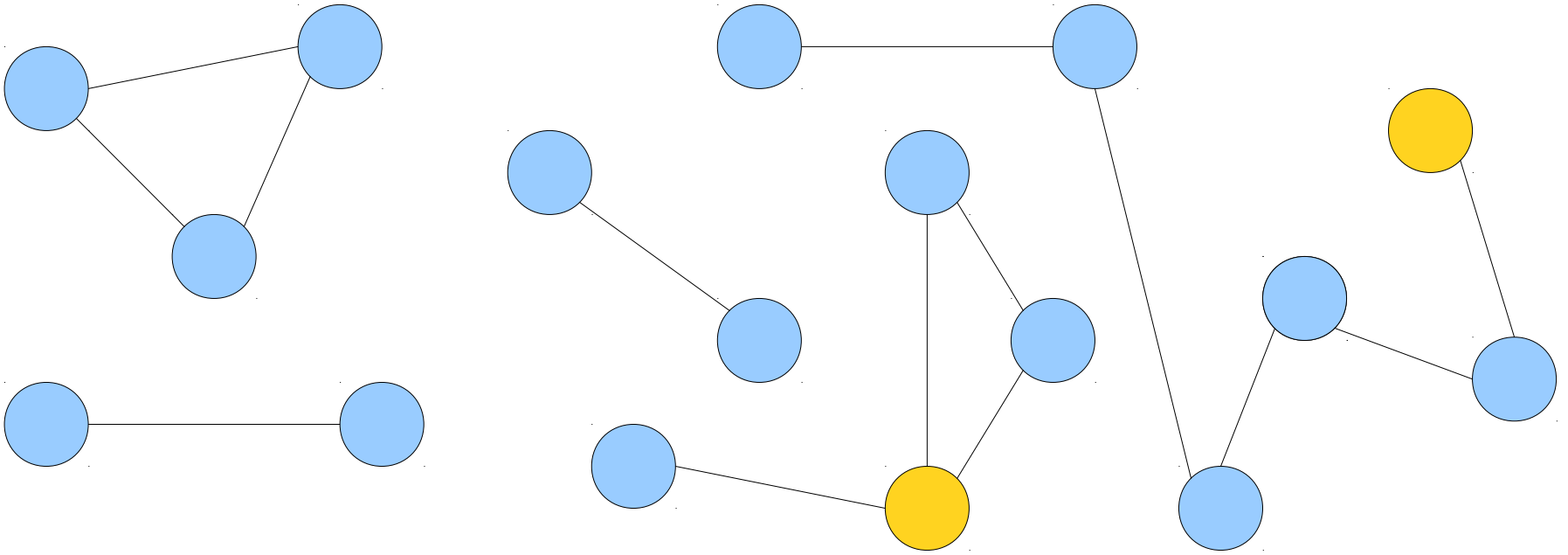


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

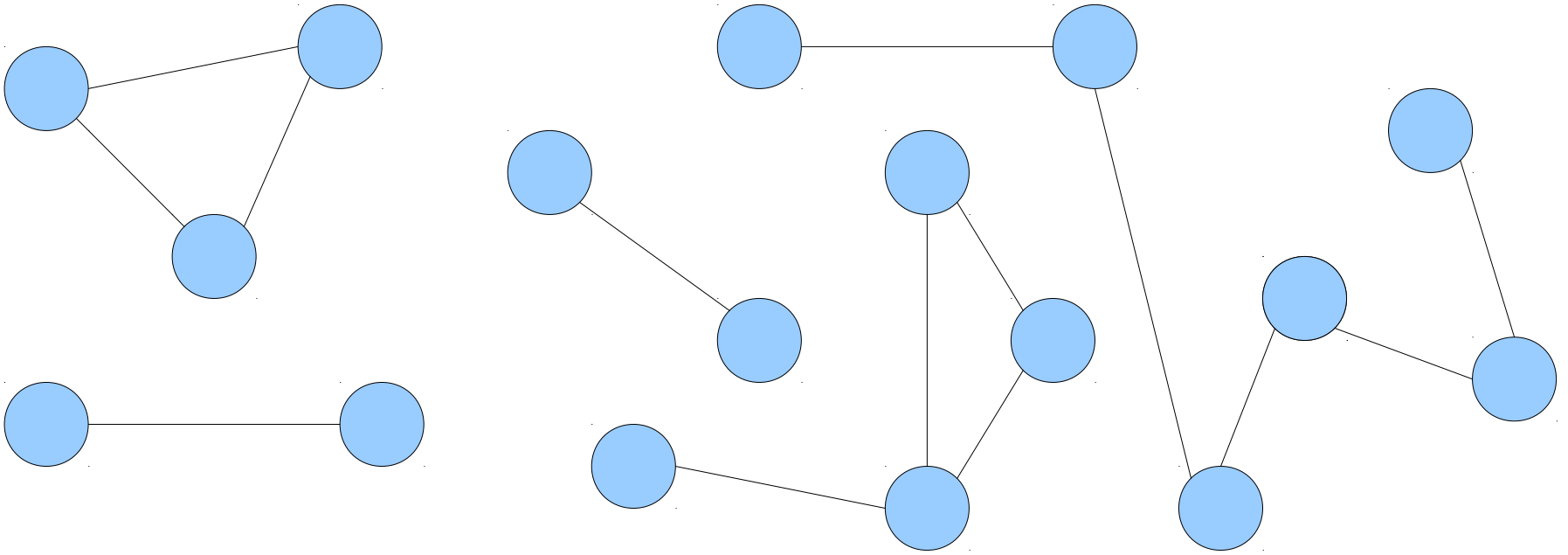


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

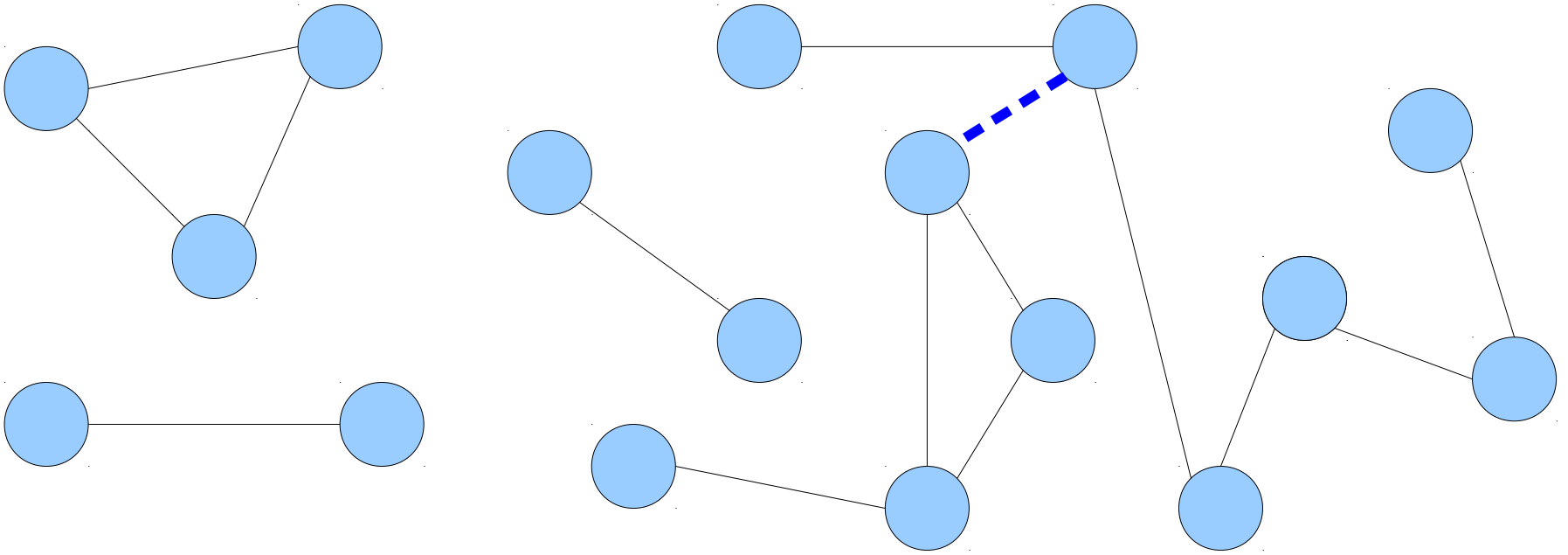


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

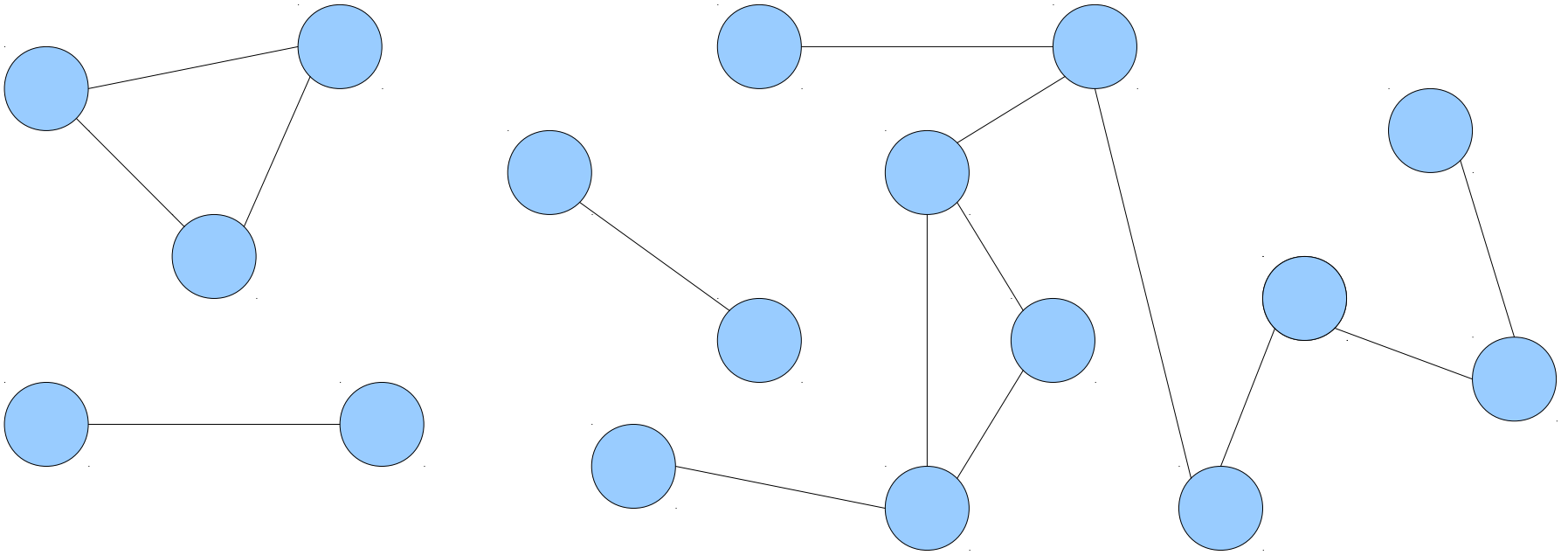


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

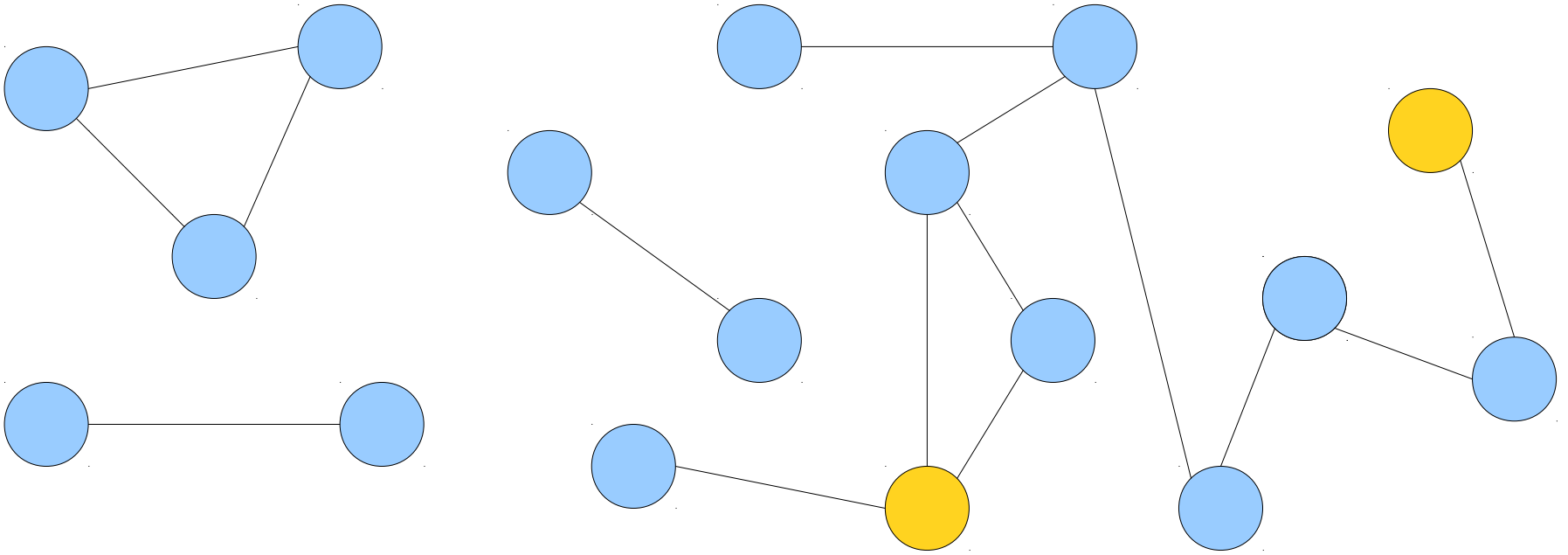


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!

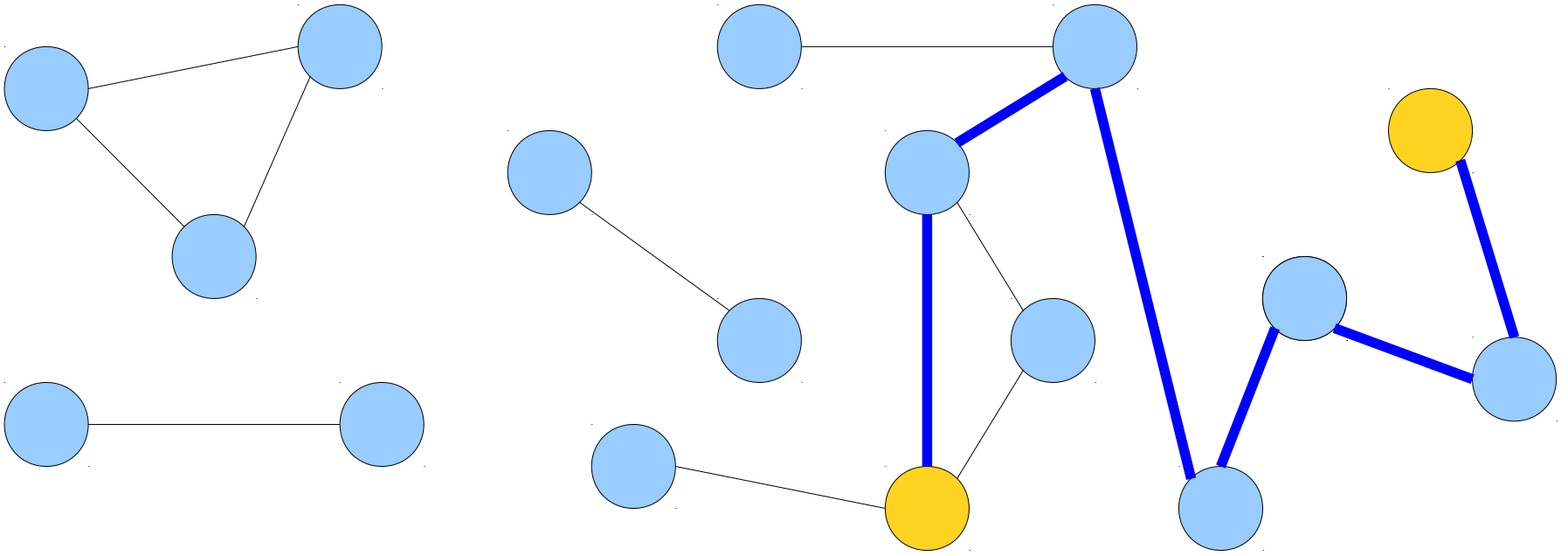


Dynamic Connectivity

- The ***dynamic connectivity problem*** is the following:

  Maintain an undirected graph G so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- This is a *much* harder problem!



Dynamic Connectivity

- Today, we'll focus on the ***incremental dynamic connectivity problem:*** maintaining connectivity when edges can only be added, not deleted.
- Has applications to Kruskal's MST algorithm and to many other online connectivity settings.
  - Look up ***percolation theory*** for an example.
- These data structures are also used as building blocks in other algorithms:
  - Speeding up Edmond's blossom algorithm for finding maximum matchings.
  - As a subroutine in Tarjan's offline lowest common ancestors algorithm.
  - Building meldable priority queues out of non-meldable queues.

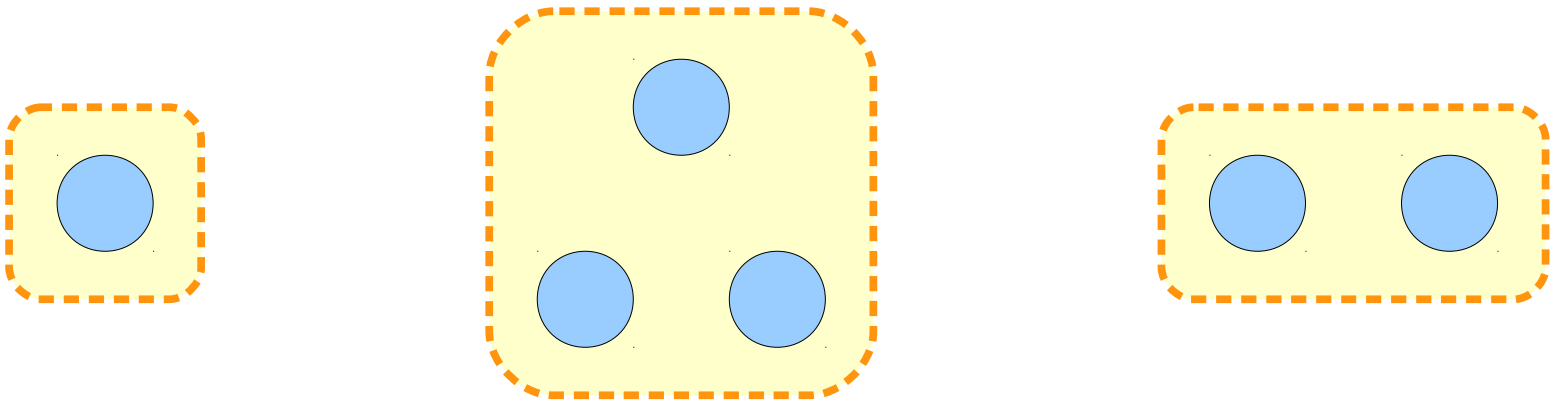
Incremental Connectivity and Partitions

Set Partitions

- The incremental connectivity problem is equivalent to maintaining a partition of a set.
- Initially, each node belongs to its own set.
- As edges are added, the sets at the endpoints become connected and are merged together.
- Querying for connectivity is equivalent to querying for whether two elements belong to the same set.

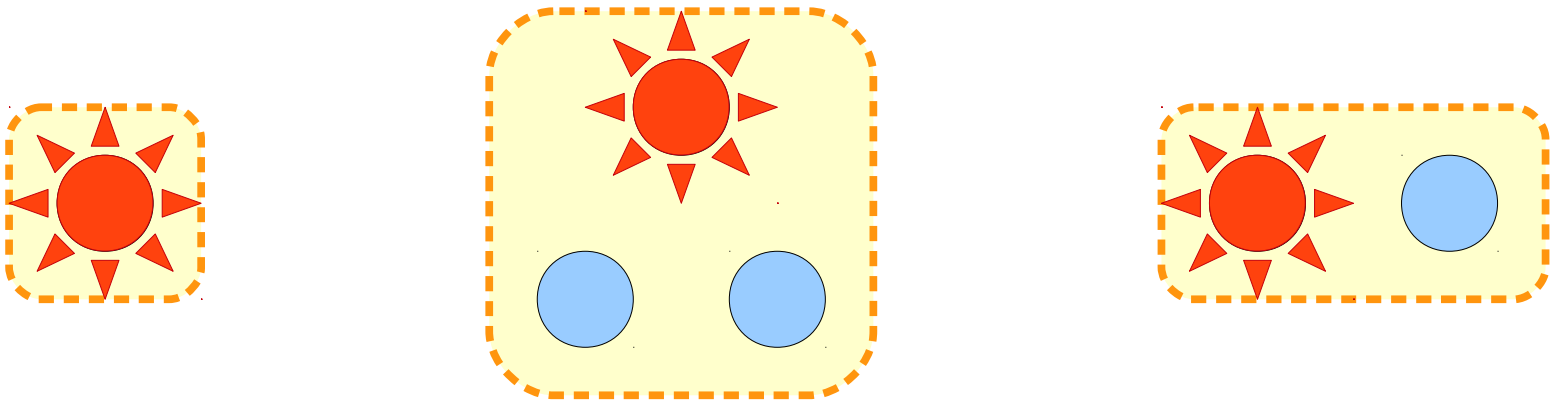
Representatives

- Given a partition of a set S, we can choose one ***representative*** from each of the sets in the partition.
- Representatives give a simple proxy for which set an element belongs to: two elements are in the same set in the partition iff their set has the same representative.



Representatives

- Given a partition of a set S, we can choose one ***representative*** from each of the sets in the partition.
- Representatives give a simple proxy for which set an element belongs to: two elements are in the same set in the partition iff their set has the same representative.

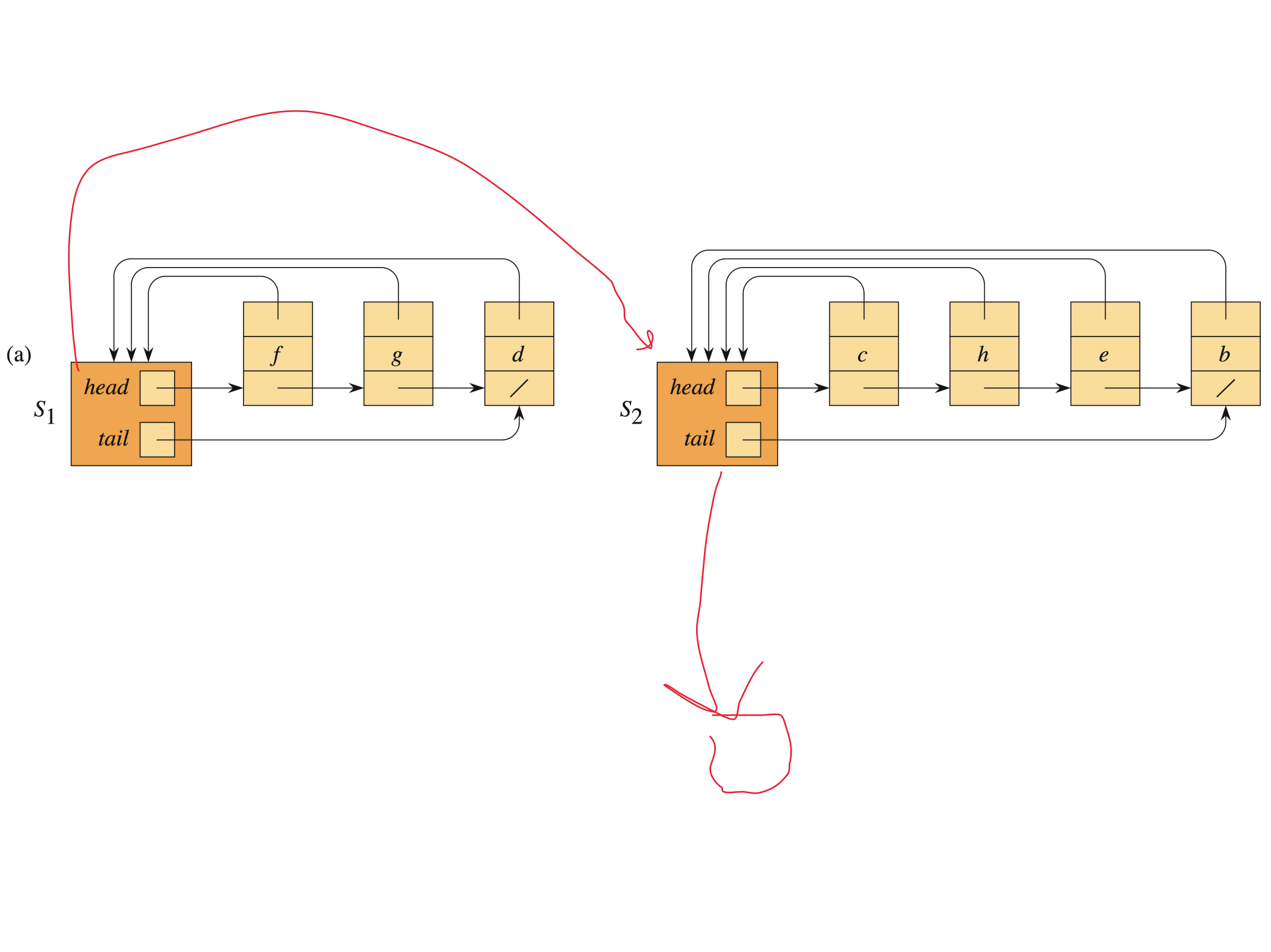


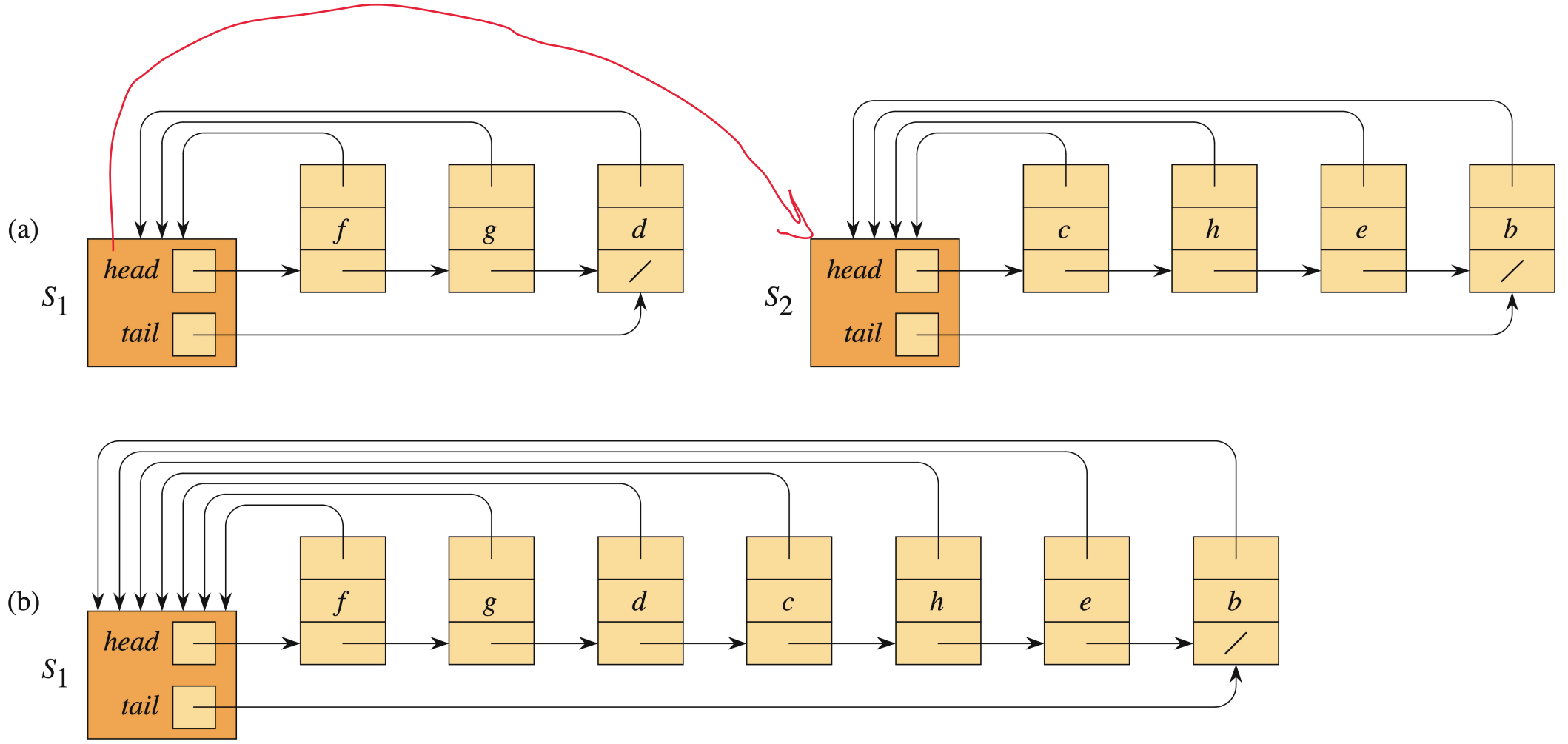
Union-Find Structures

- A ***union-find structure*** is a data structure supporting the following operations:
  - ***find***(x), which returns the representative of the set containing node x, and
  - ***union***(x, y), which merges the sets containing x and y into a single set.
- We'll focus on these sorts of structures as a solution to incremental connectivity.

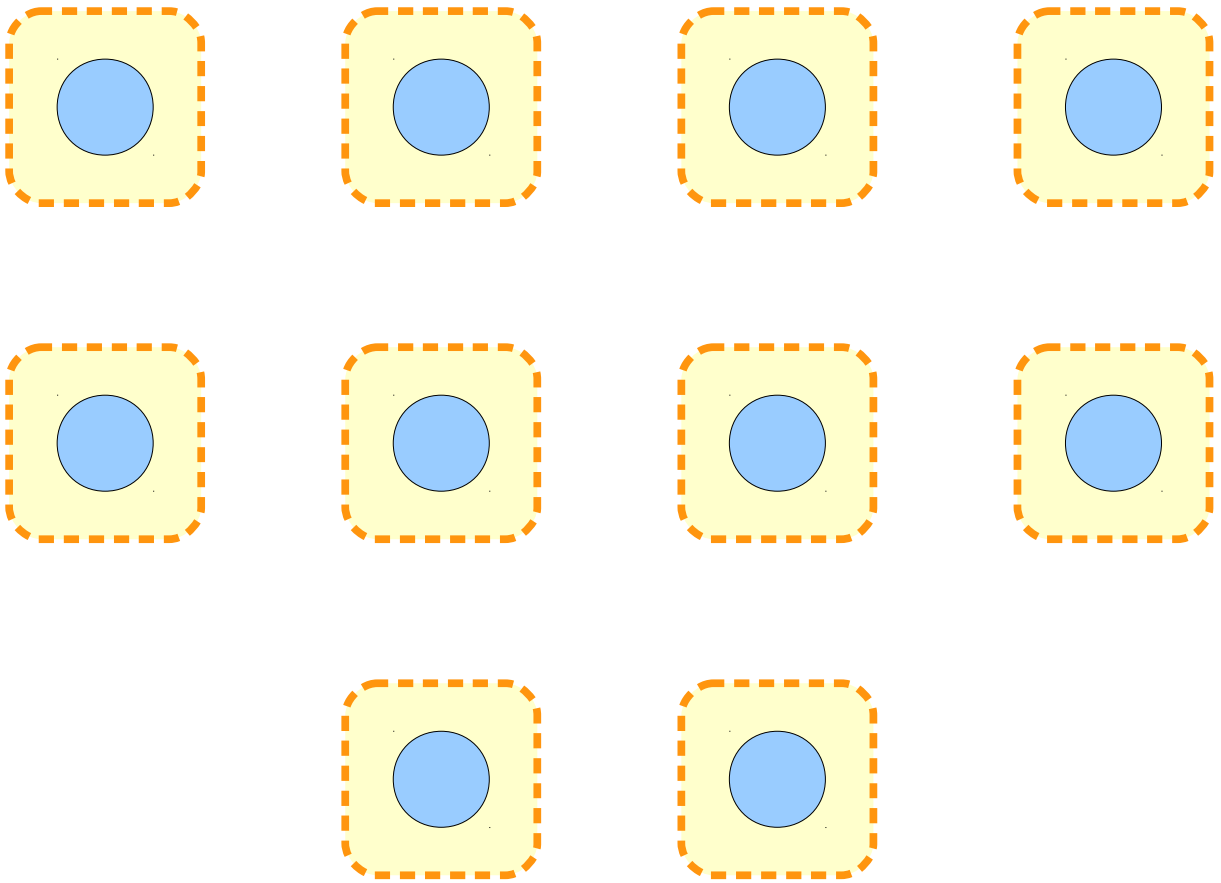
Data Structure Idea

- ***Idea:*** Have each element store a pointer directly to its representative.
- To determine if two nodes are in the same set, check if they have the same representative.
- To link two sets together, change all elements of the two sets so they reference a single representative.

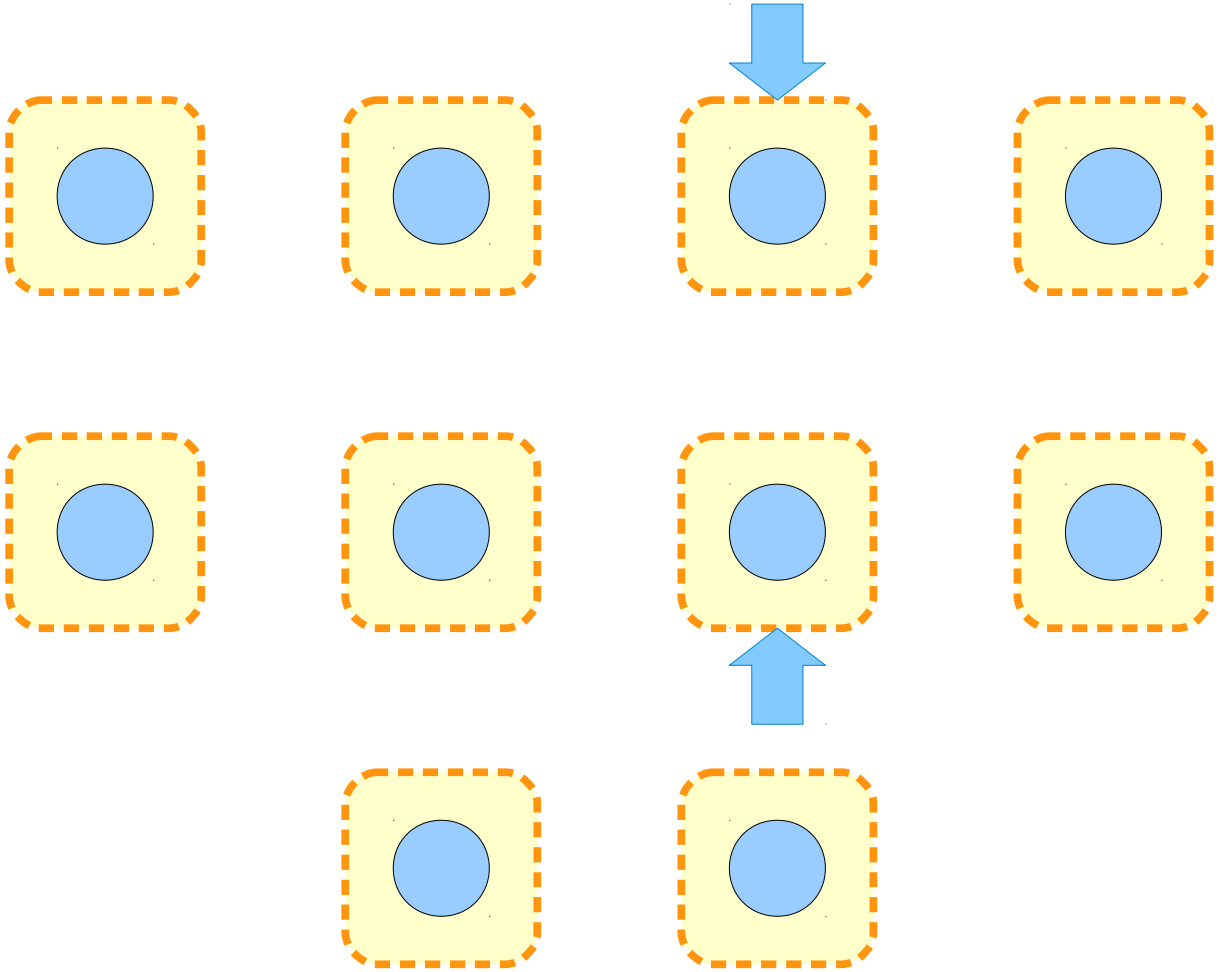

(a)
S_1
head
tail
f
g
d
S_2
head
tail
c
h
e
b


(a)
S_1
head
tail
f
g
d
S_2
head
tail
c
h
e
b
(b)
S_1
head
tail
f
g
d
c
h
e
b

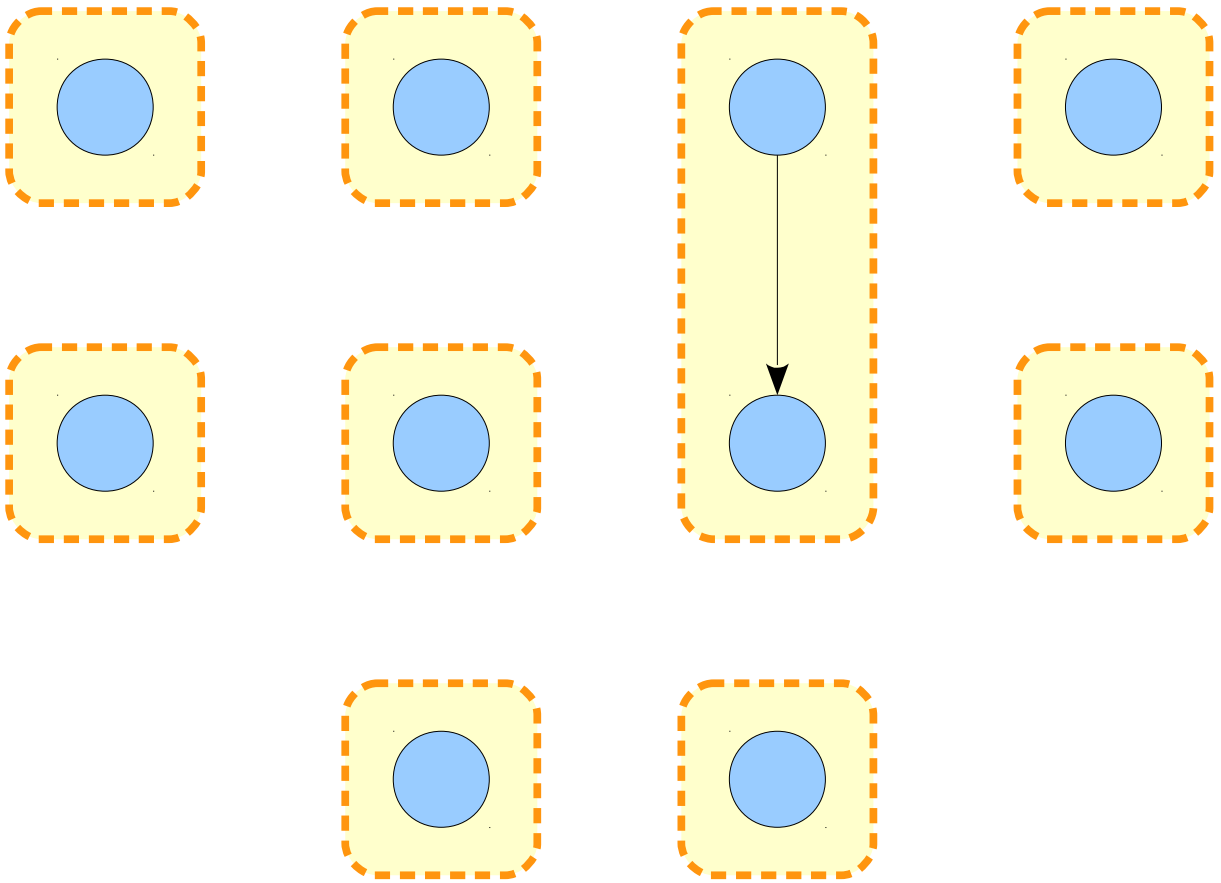
Using Representatives



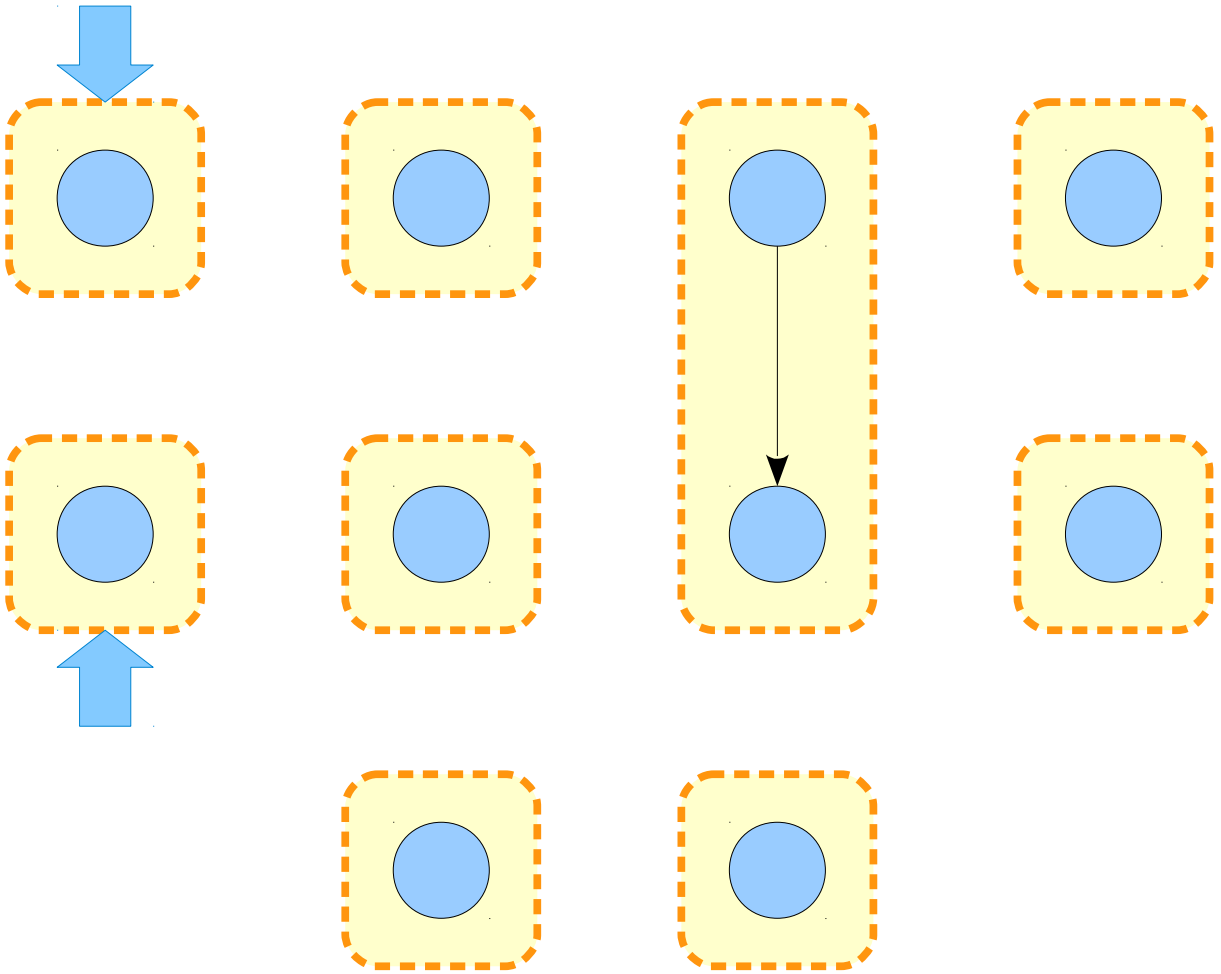
Using Representatives



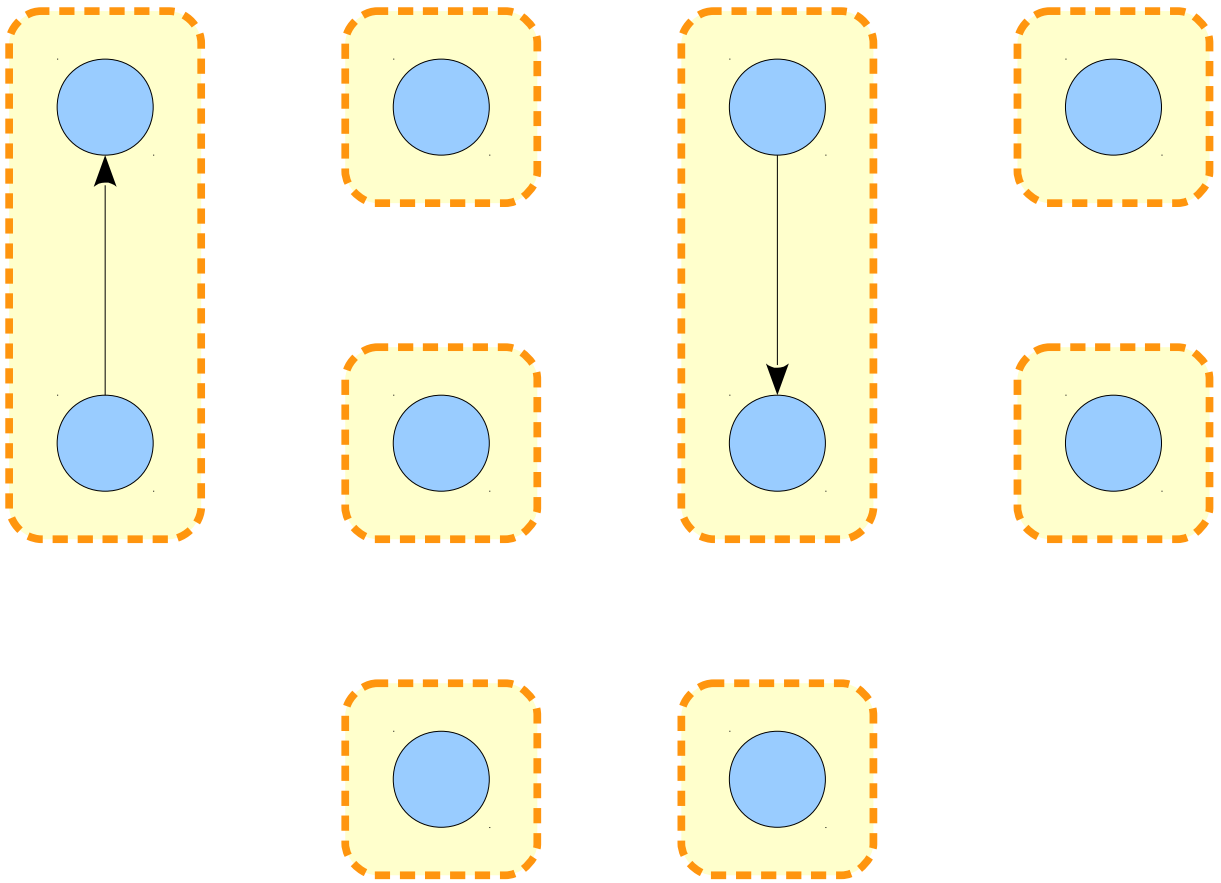
Using Representatives



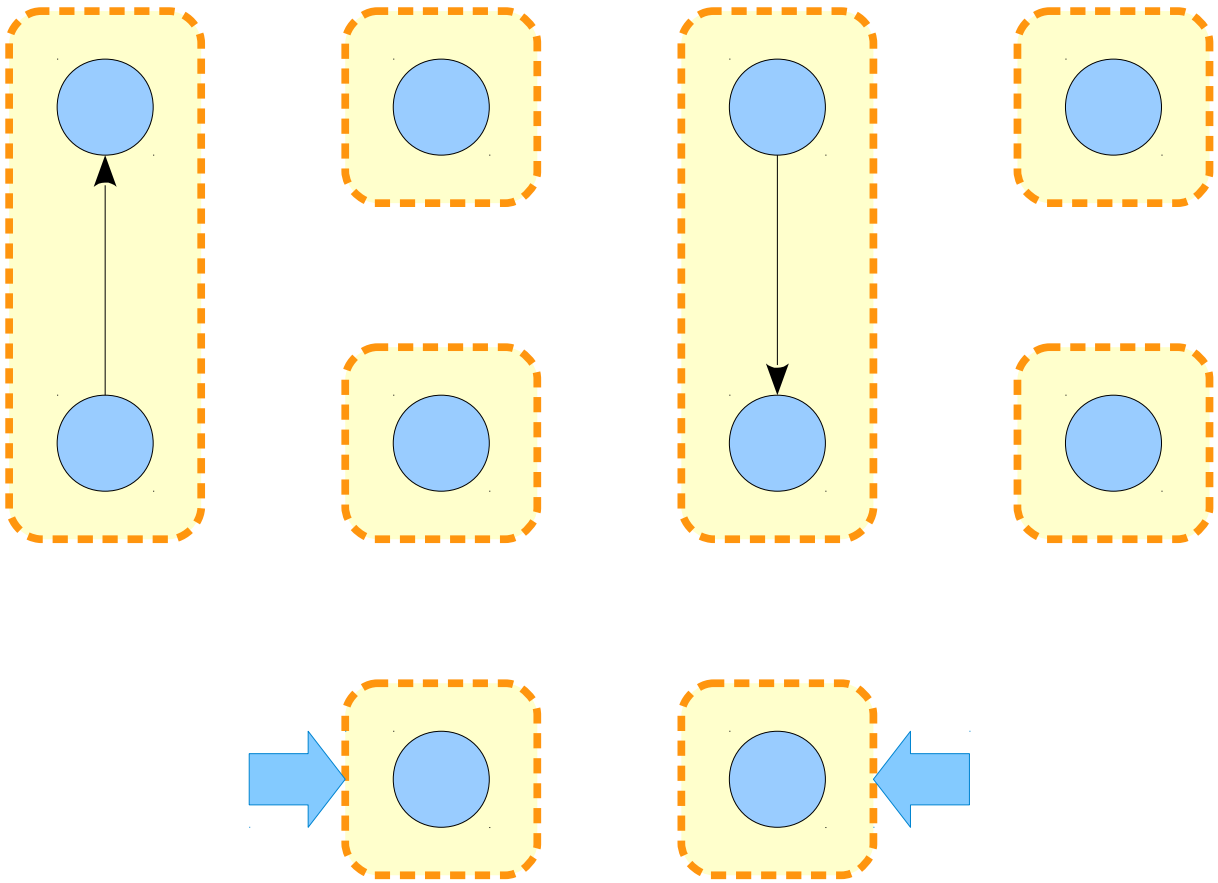
Using Representatives



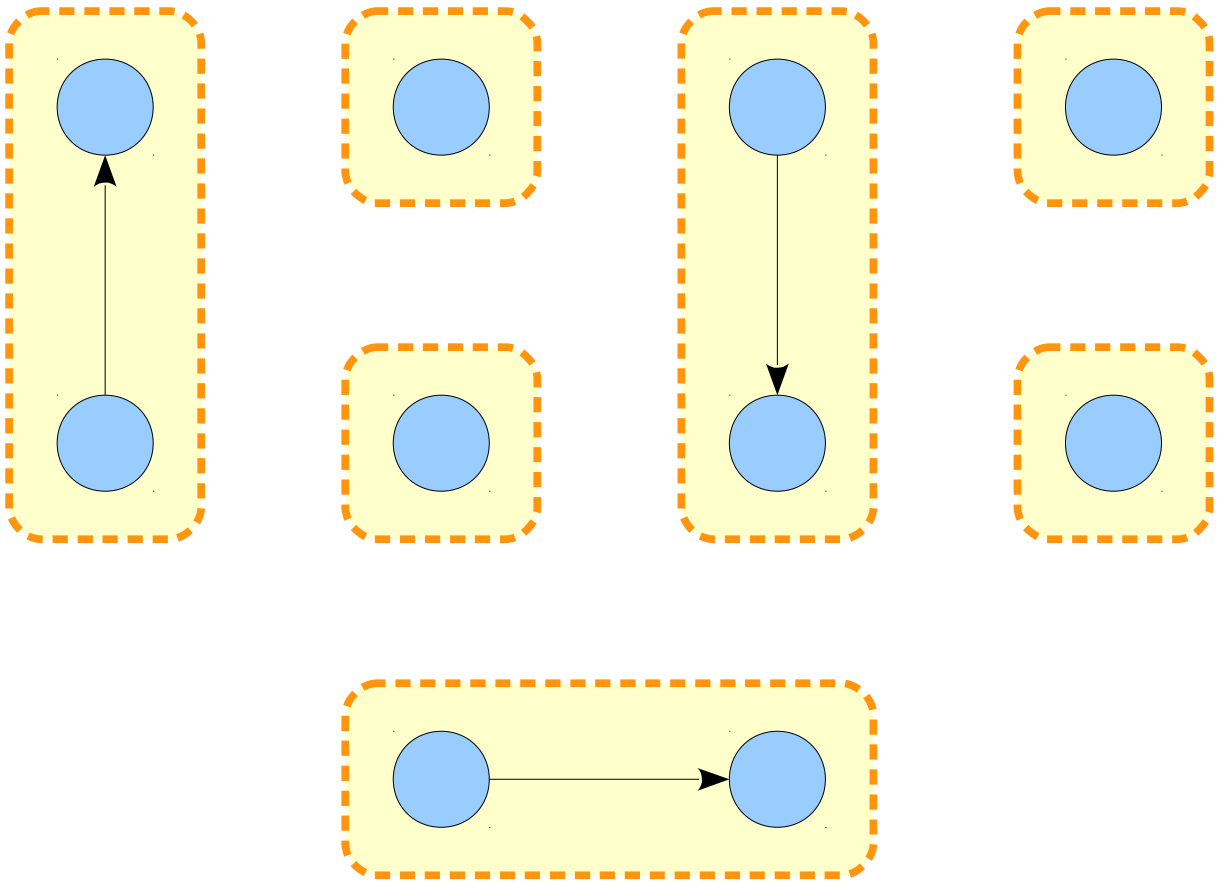
Using Representatives



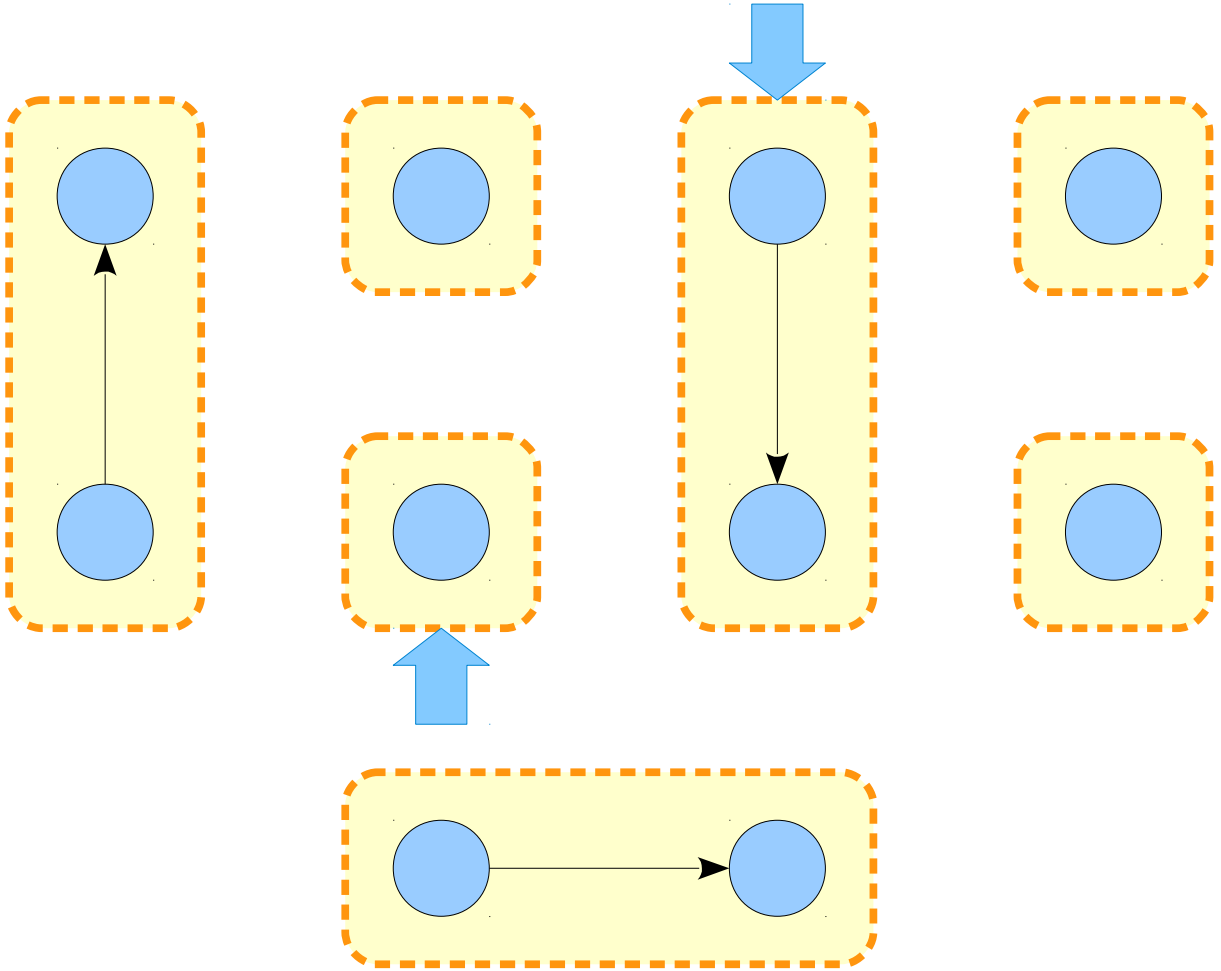
Using Representatives



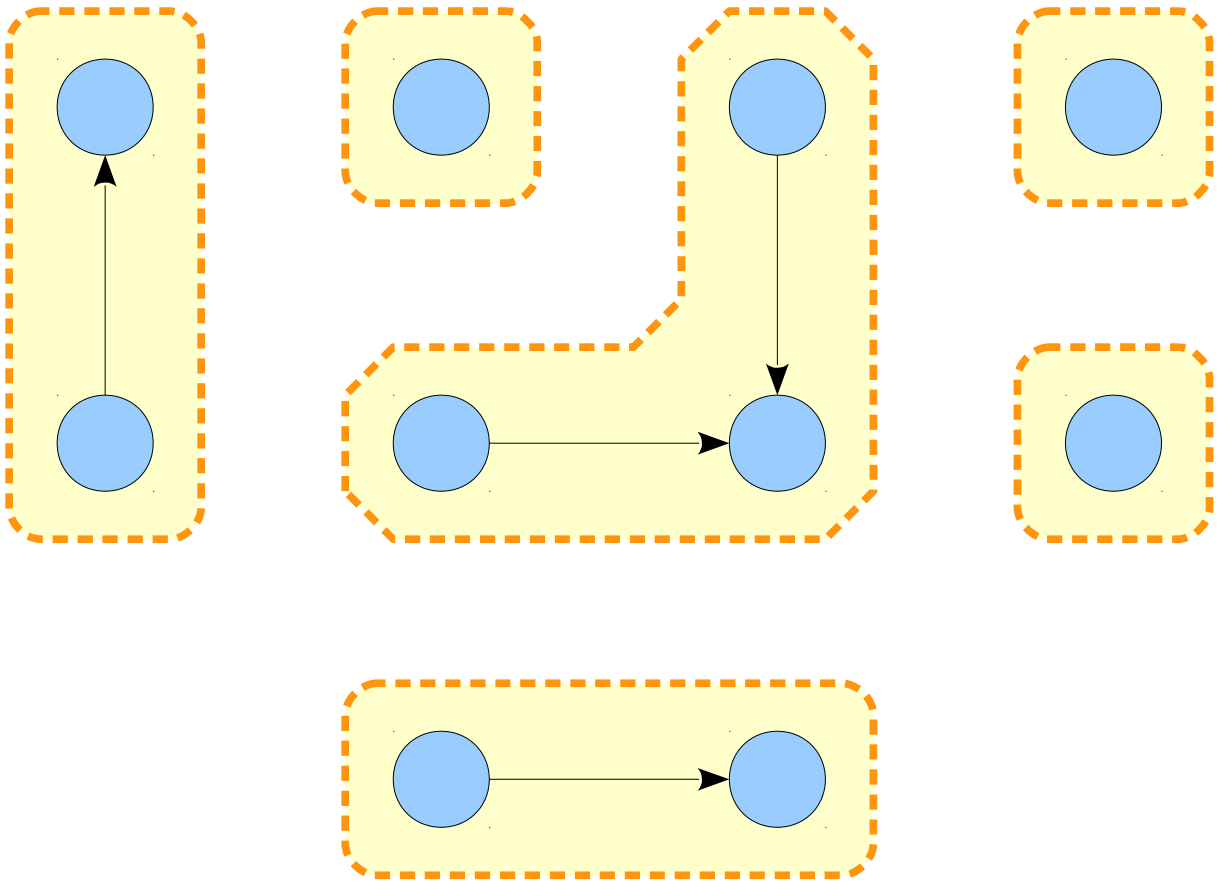
Using Representatives



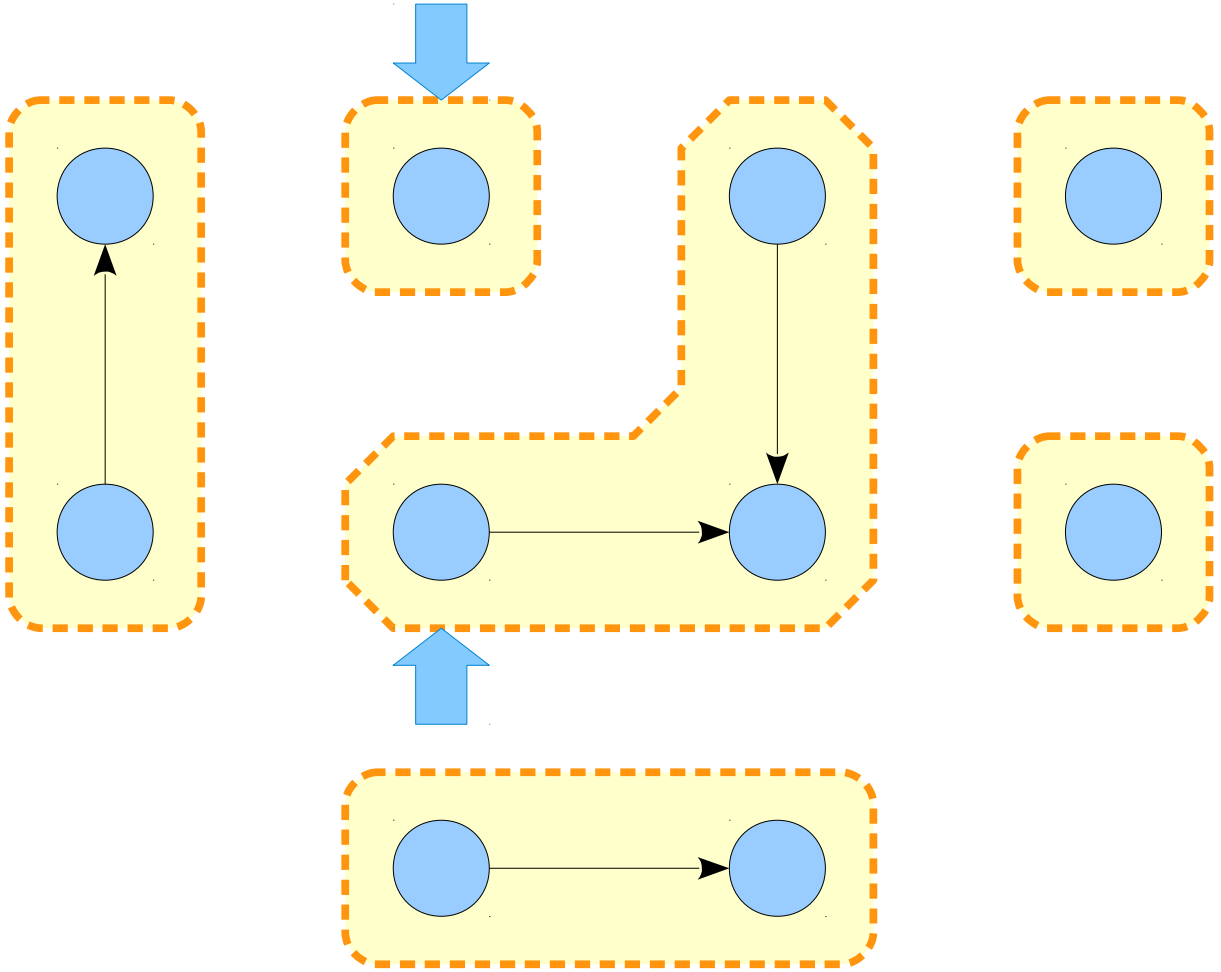
Using Representatives



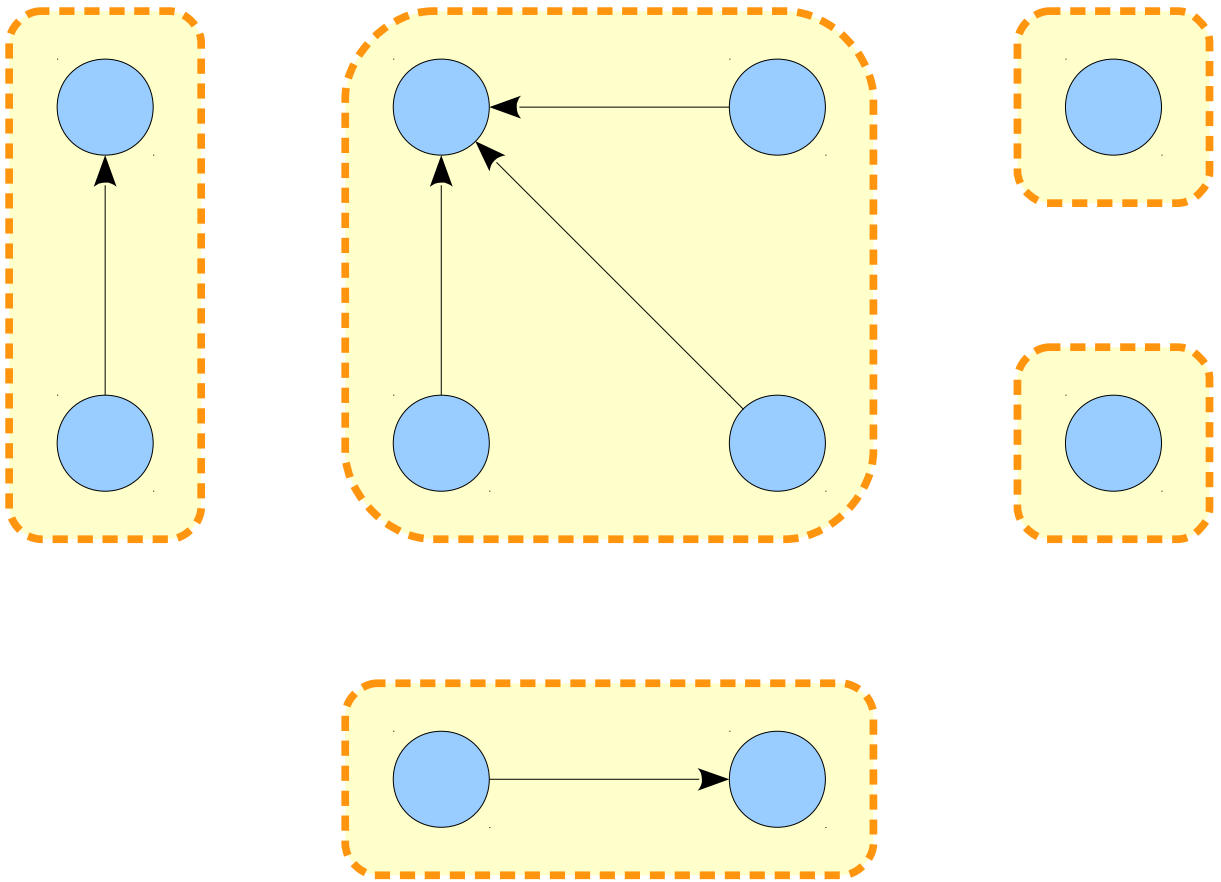
Using Representatives



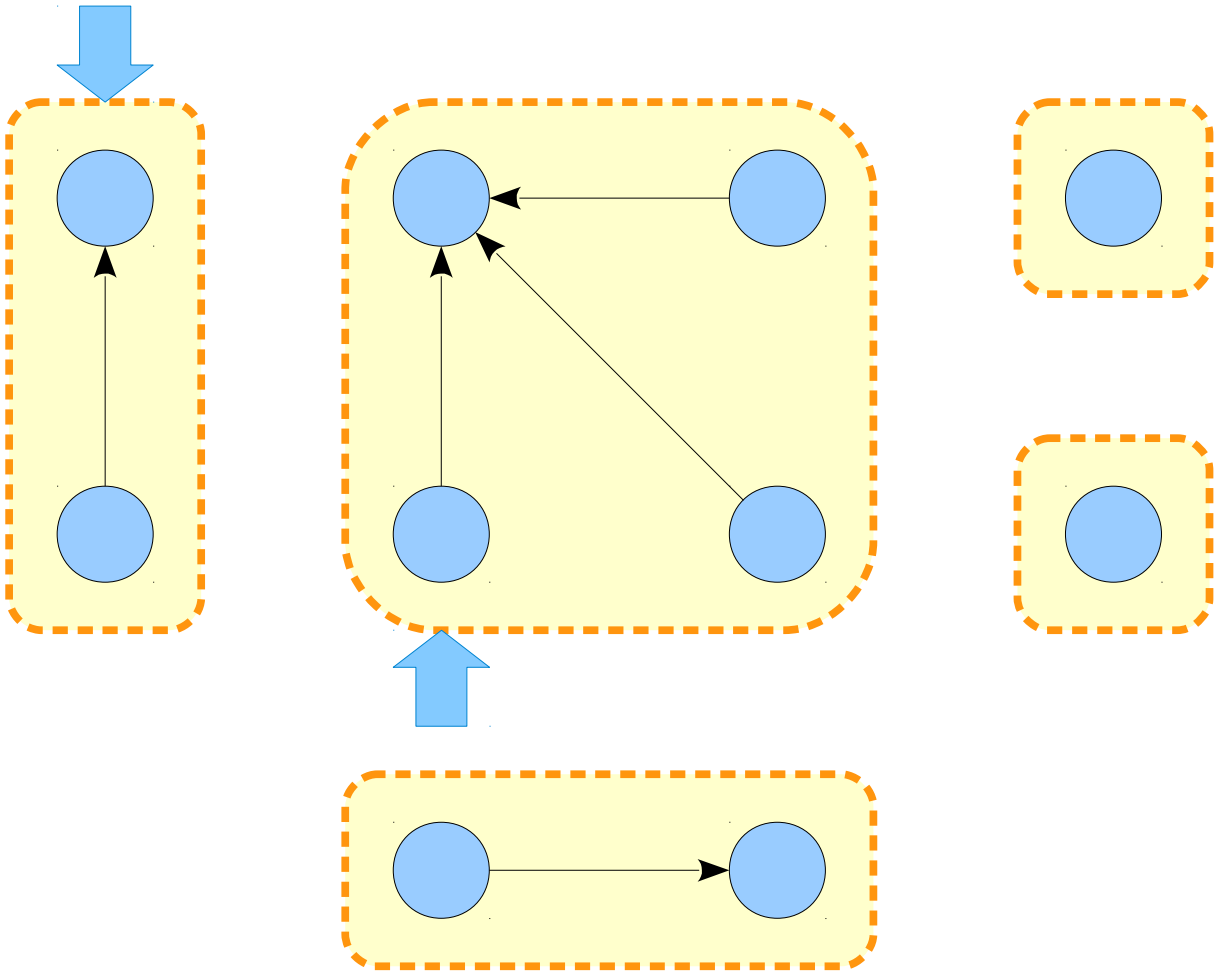
Using Representatives



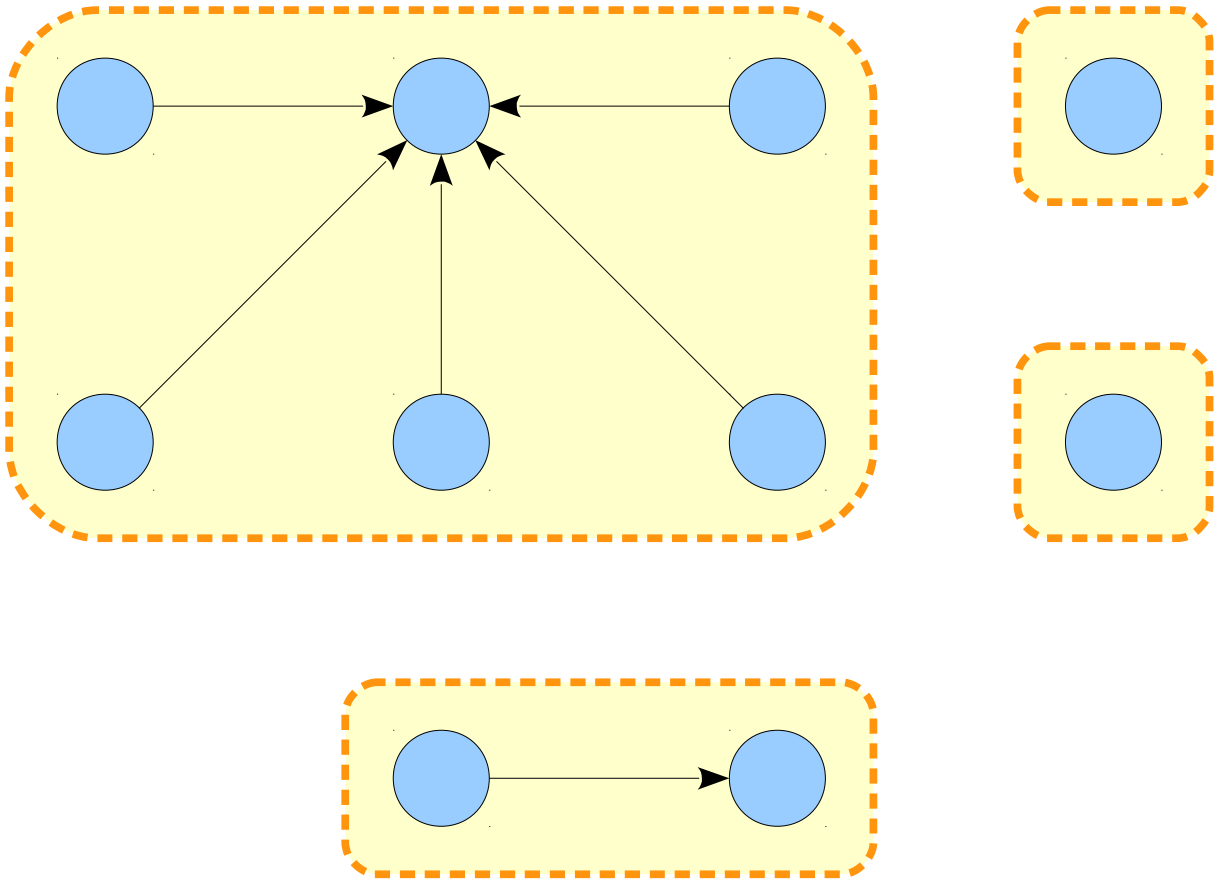
Using Representatives



Using Representatives



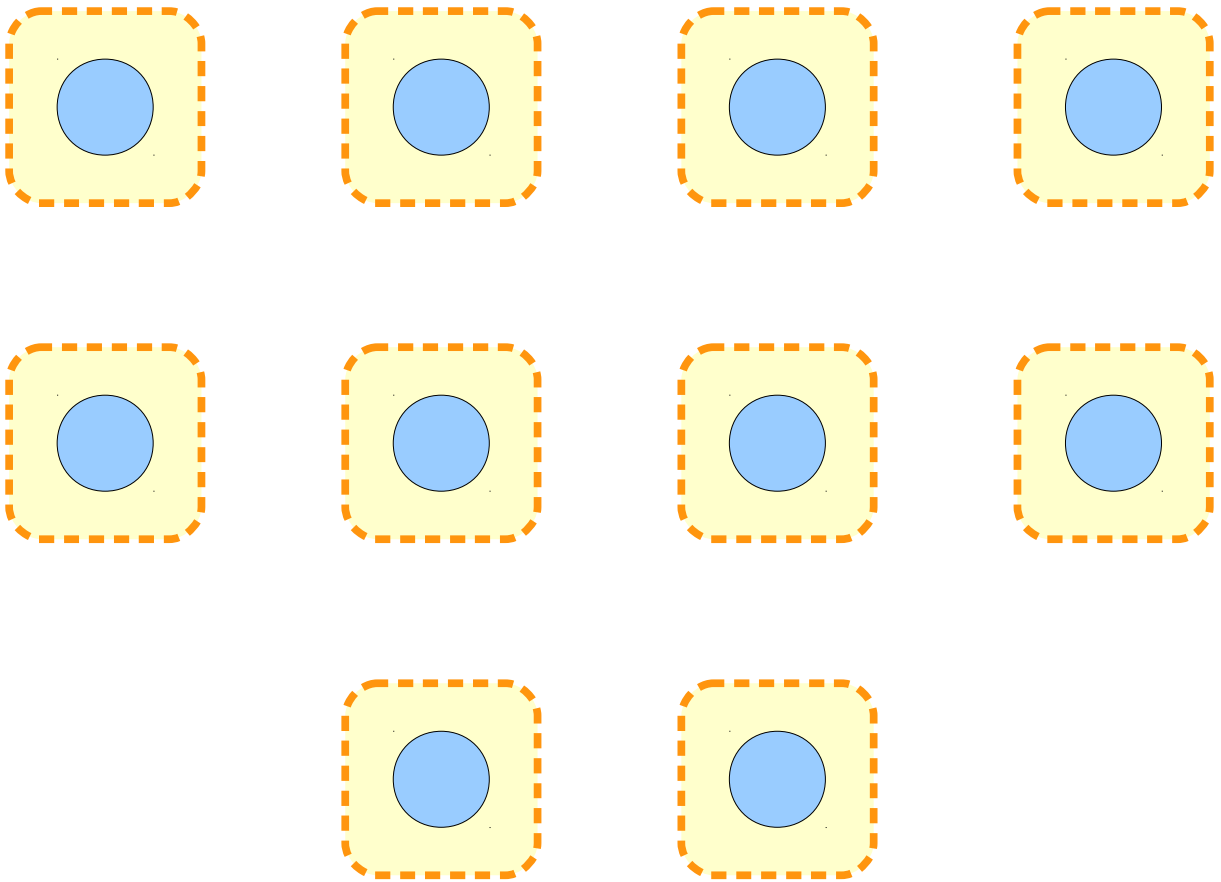
Using Representatives



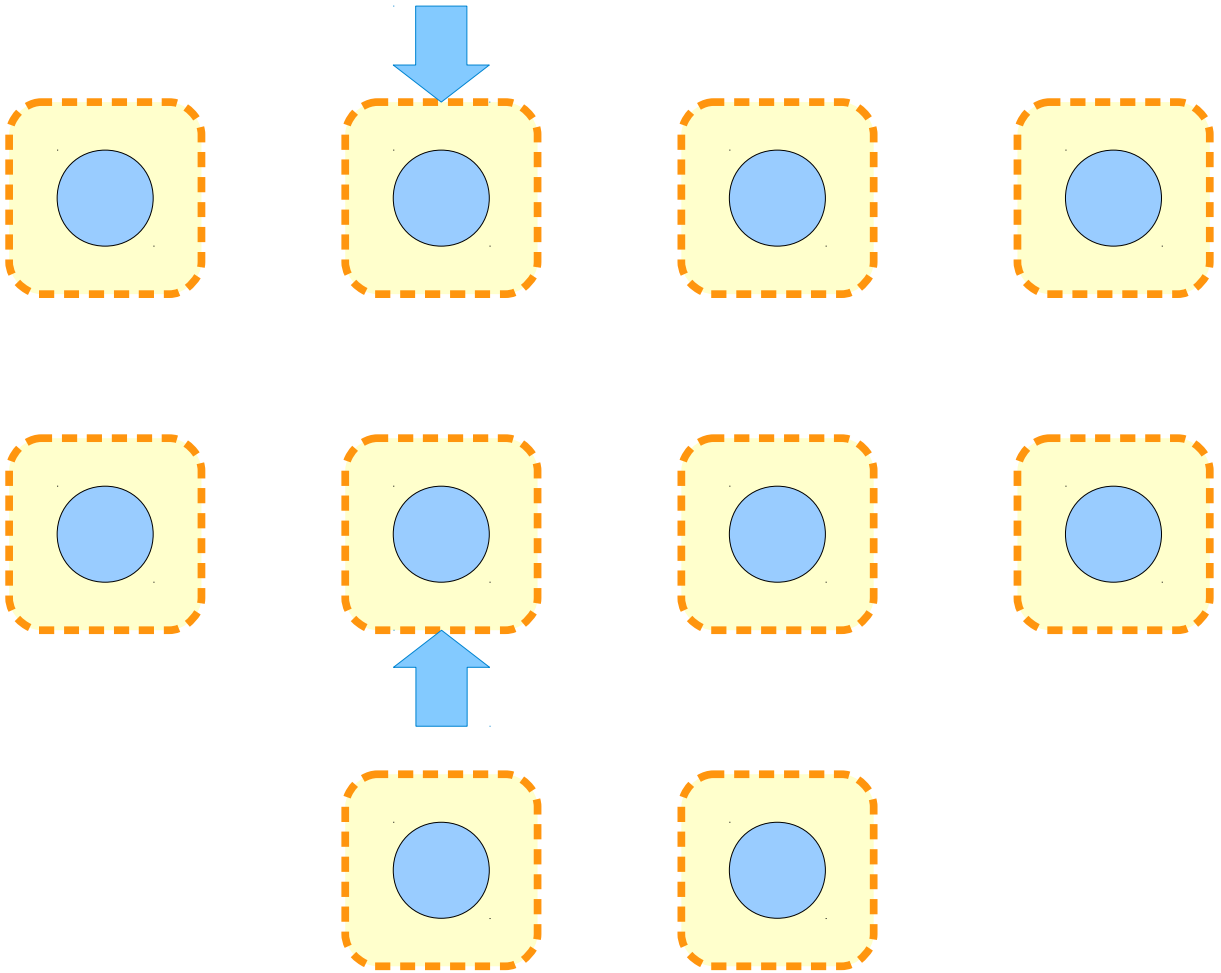
Using Representatives

- If we update all the representative pointers in a set when doing a ***union***, we may spend time O(n) per ***union*** operation.
  - If you're clever with how you change the pointers, you can make it amortized O(log n) per operation. Do you see how?
- Can we avoid paying this cost?

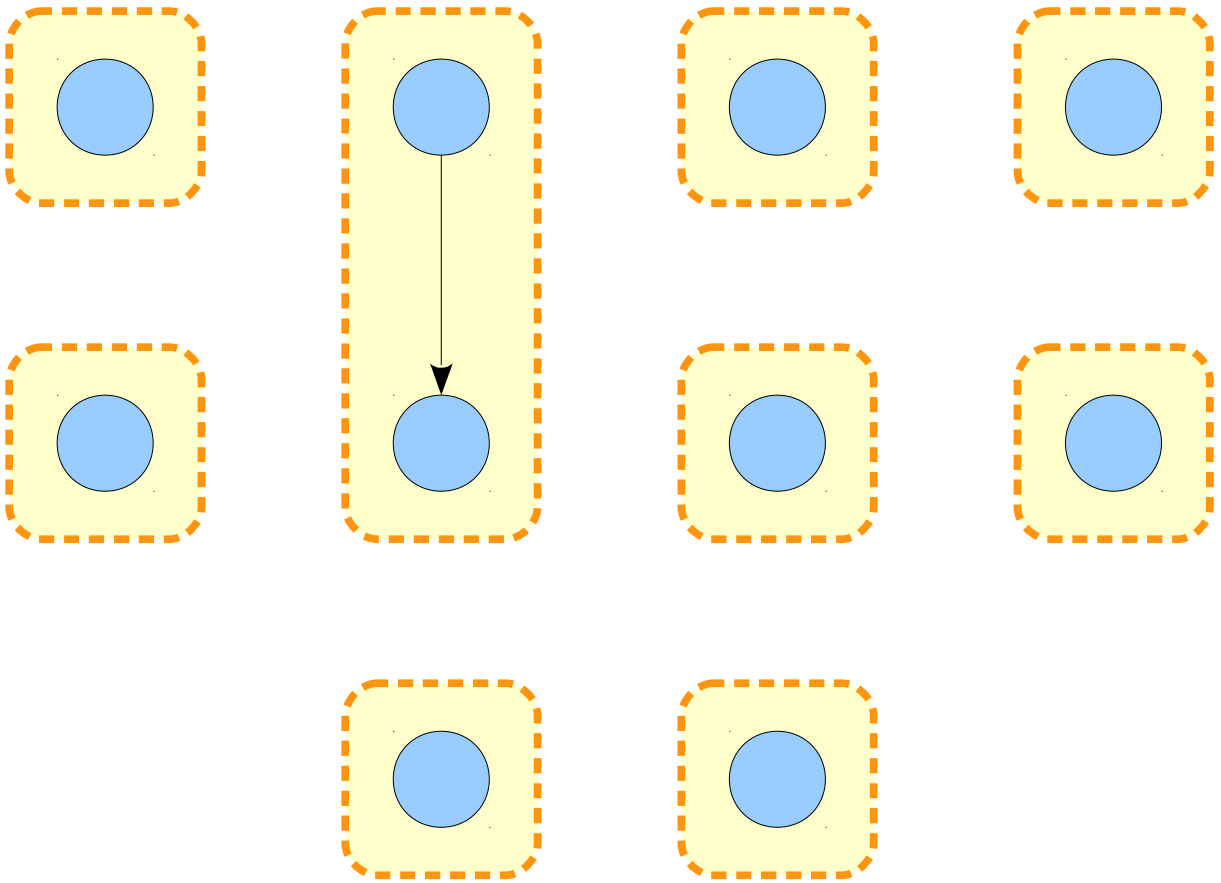
Hierarchical Representatives



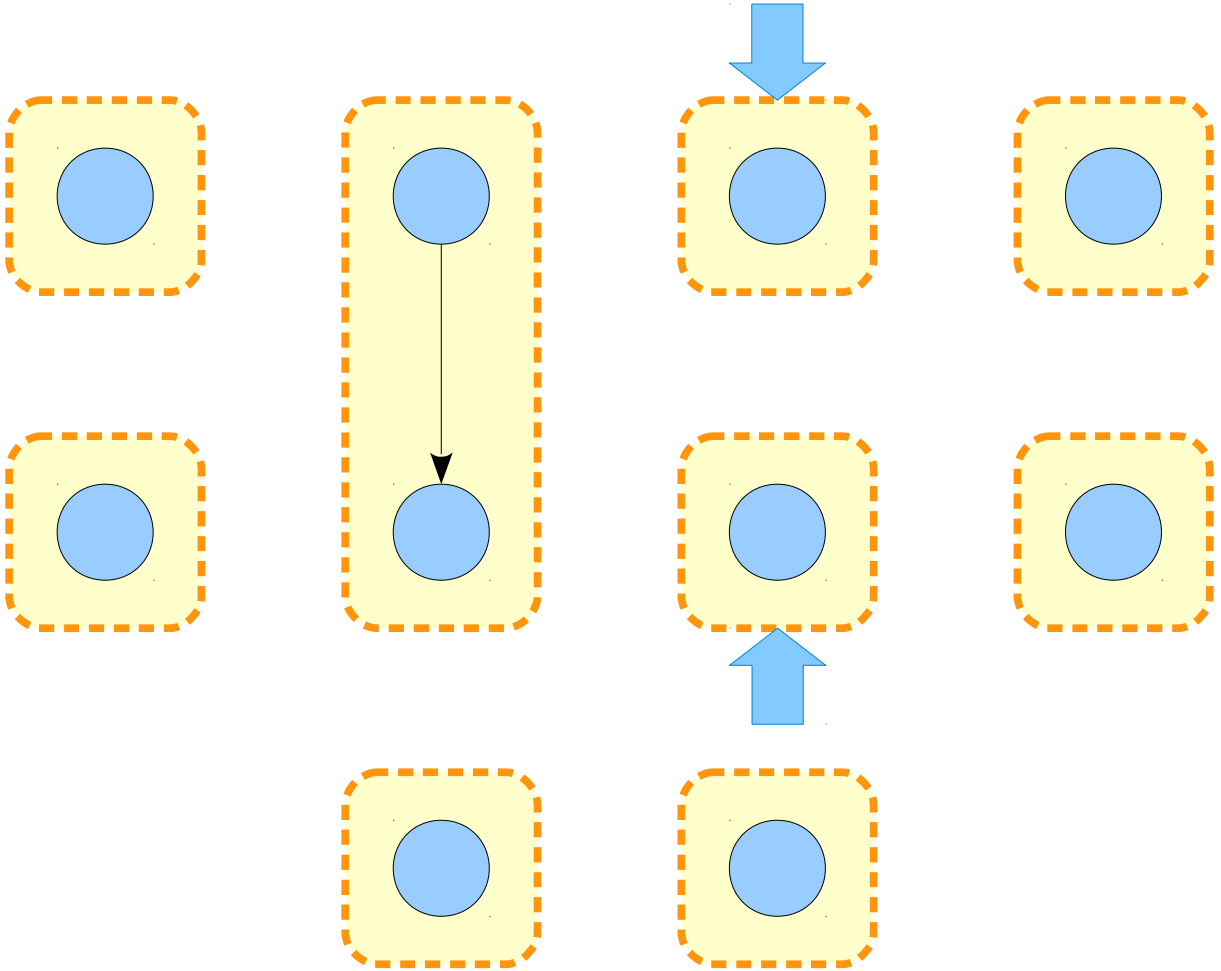
Hierarchical Representatives



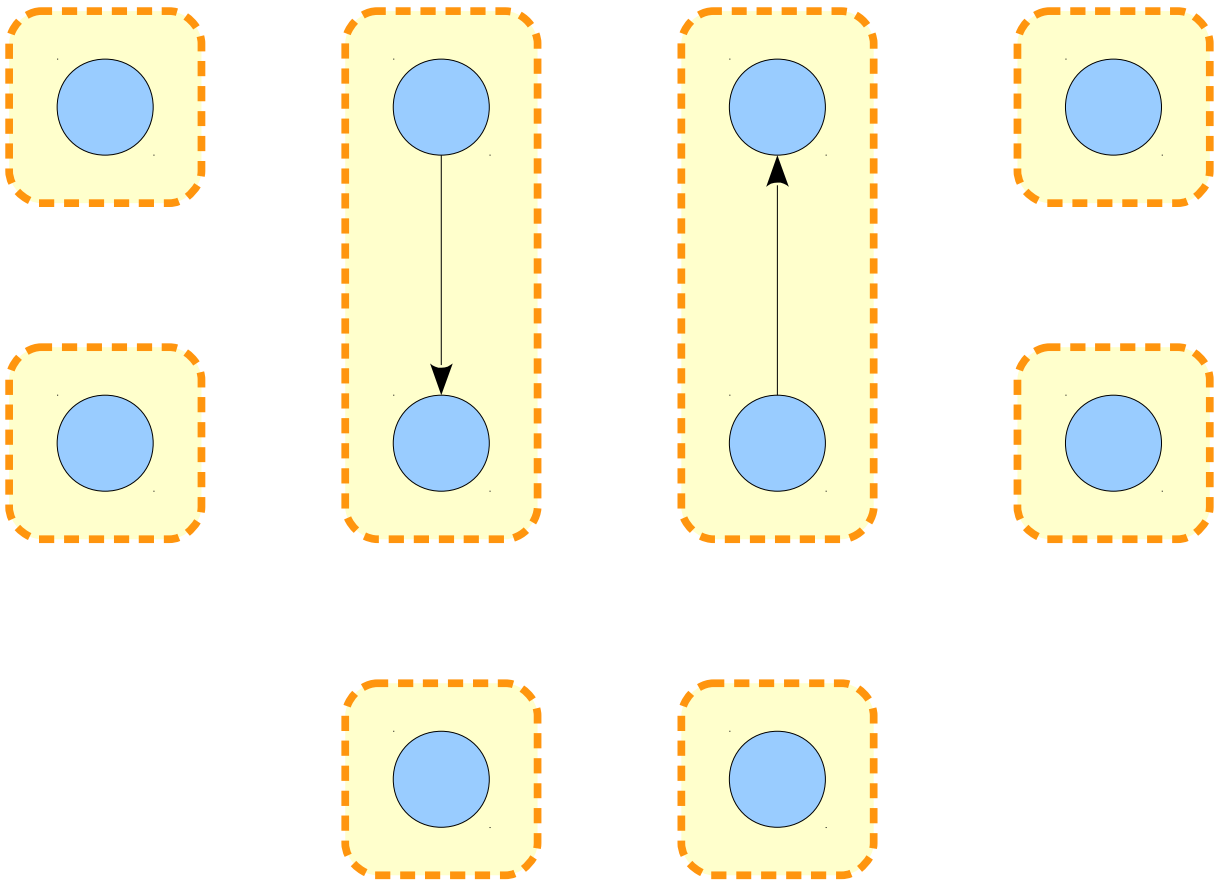
Hierarchical Representatives



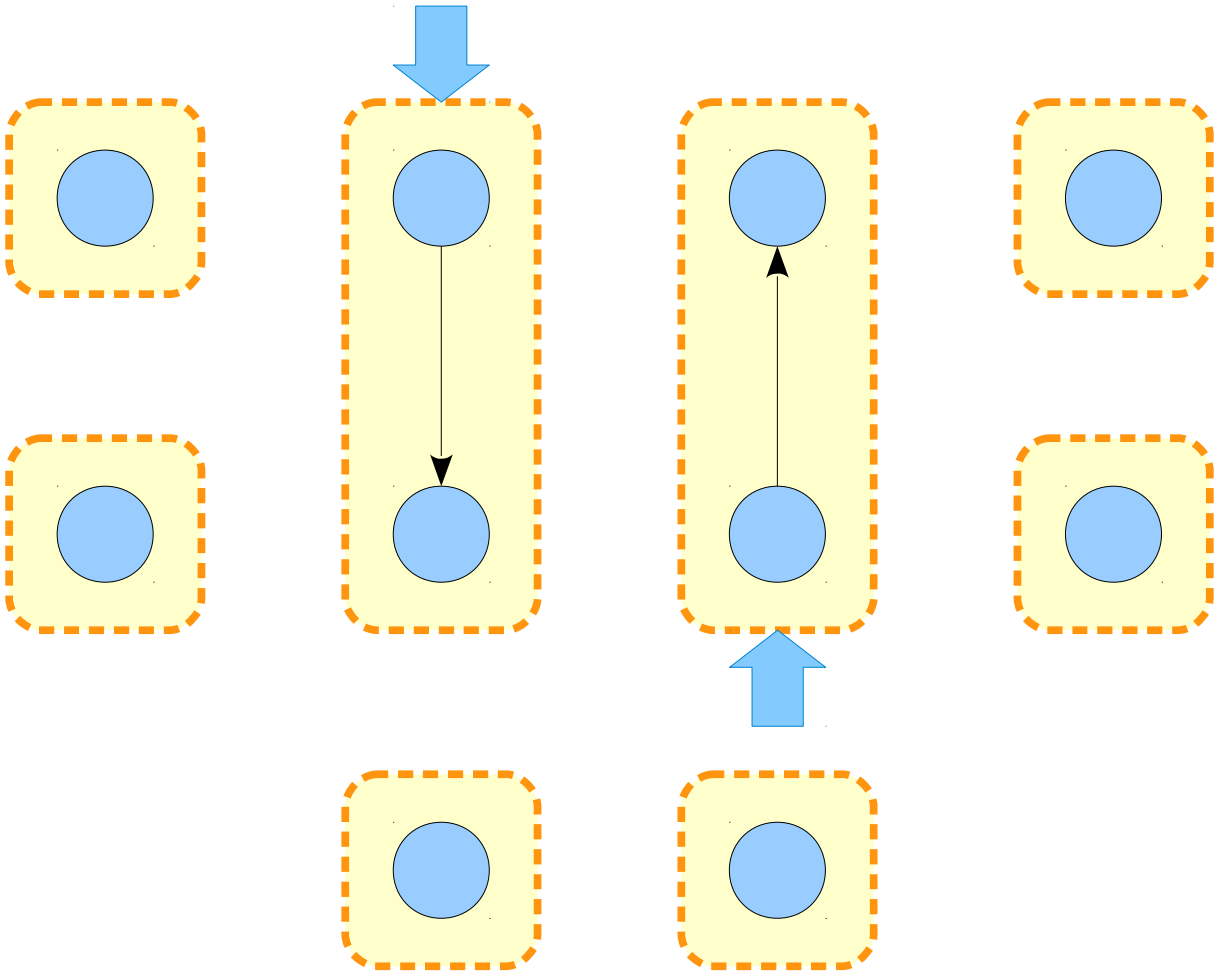
Hierarchical Representatives



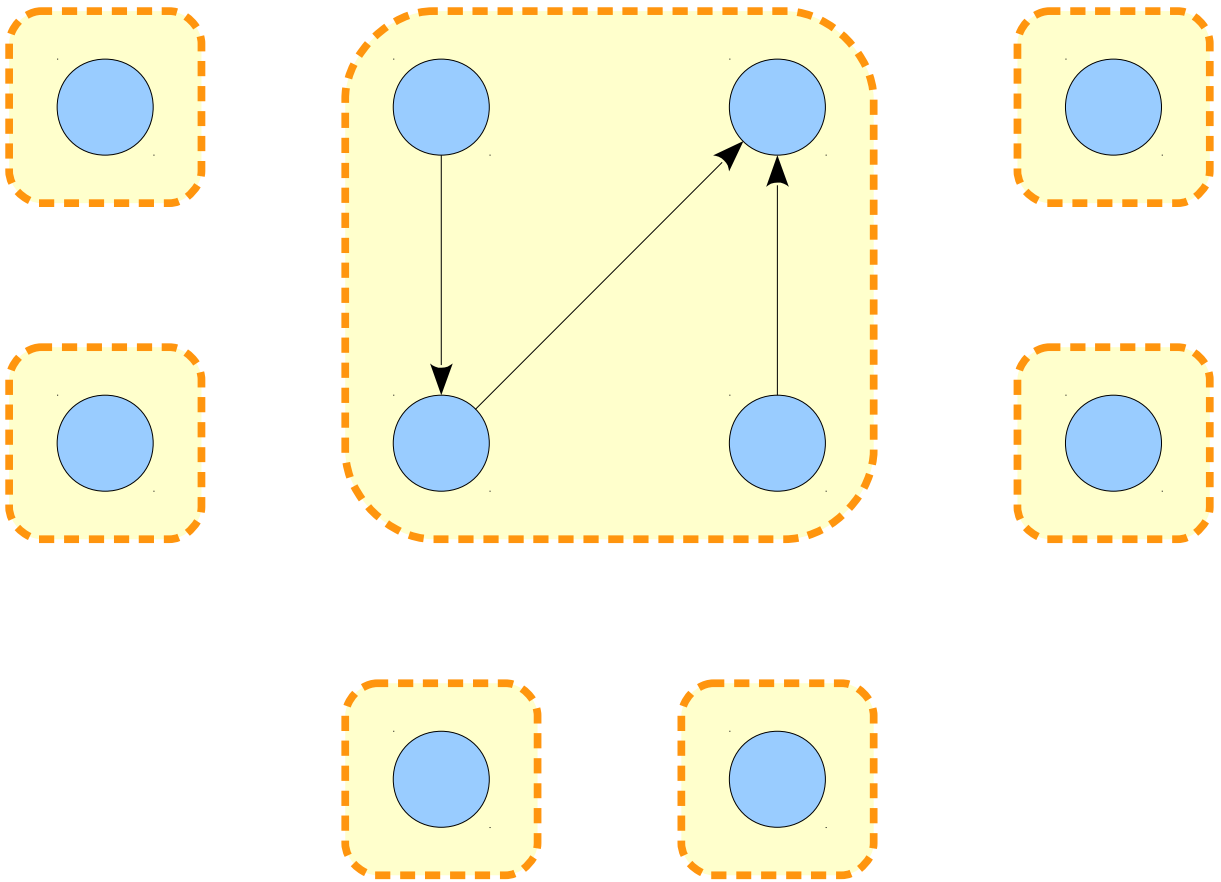
Hierarchical Representatives



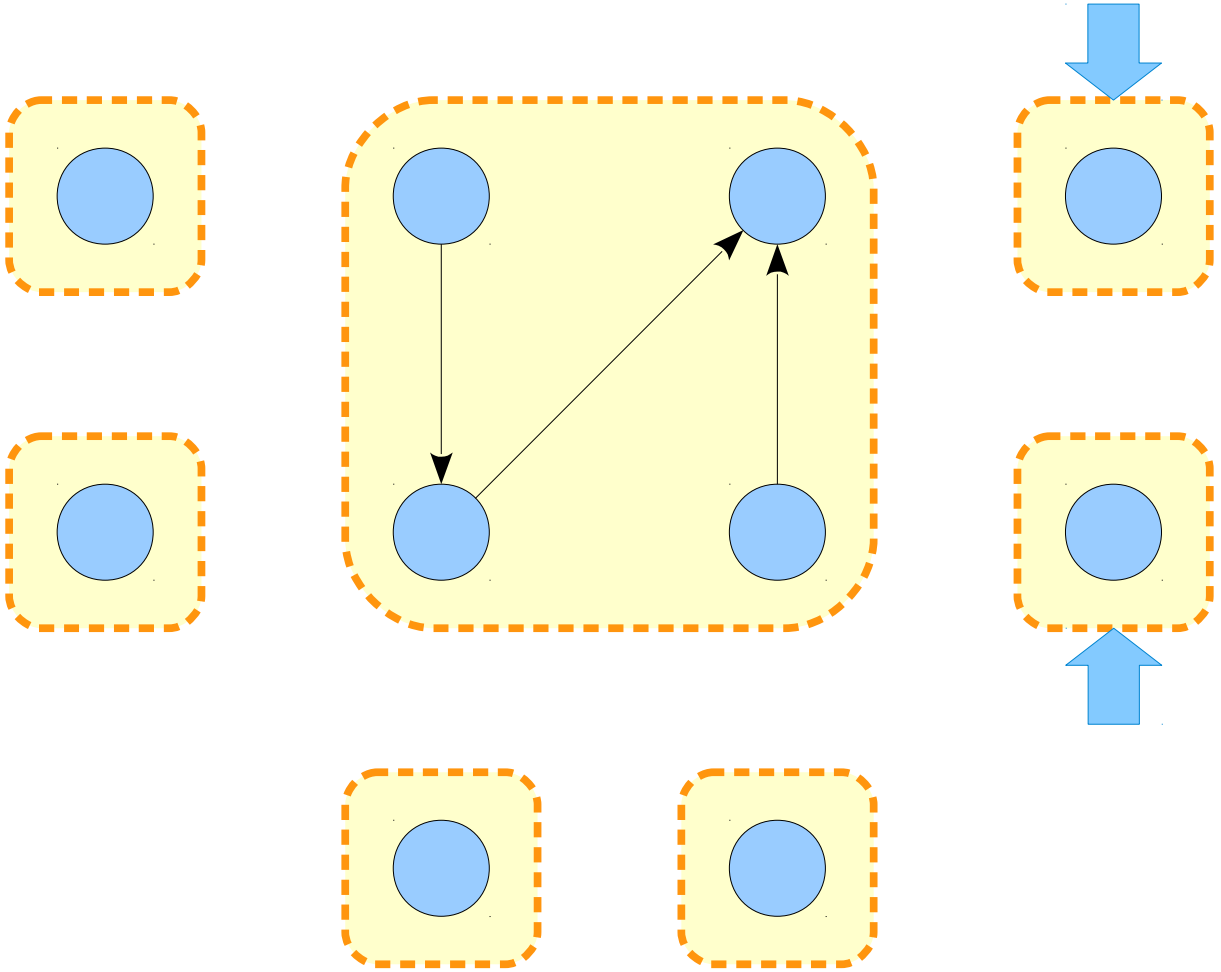
Hierarchical Representatives



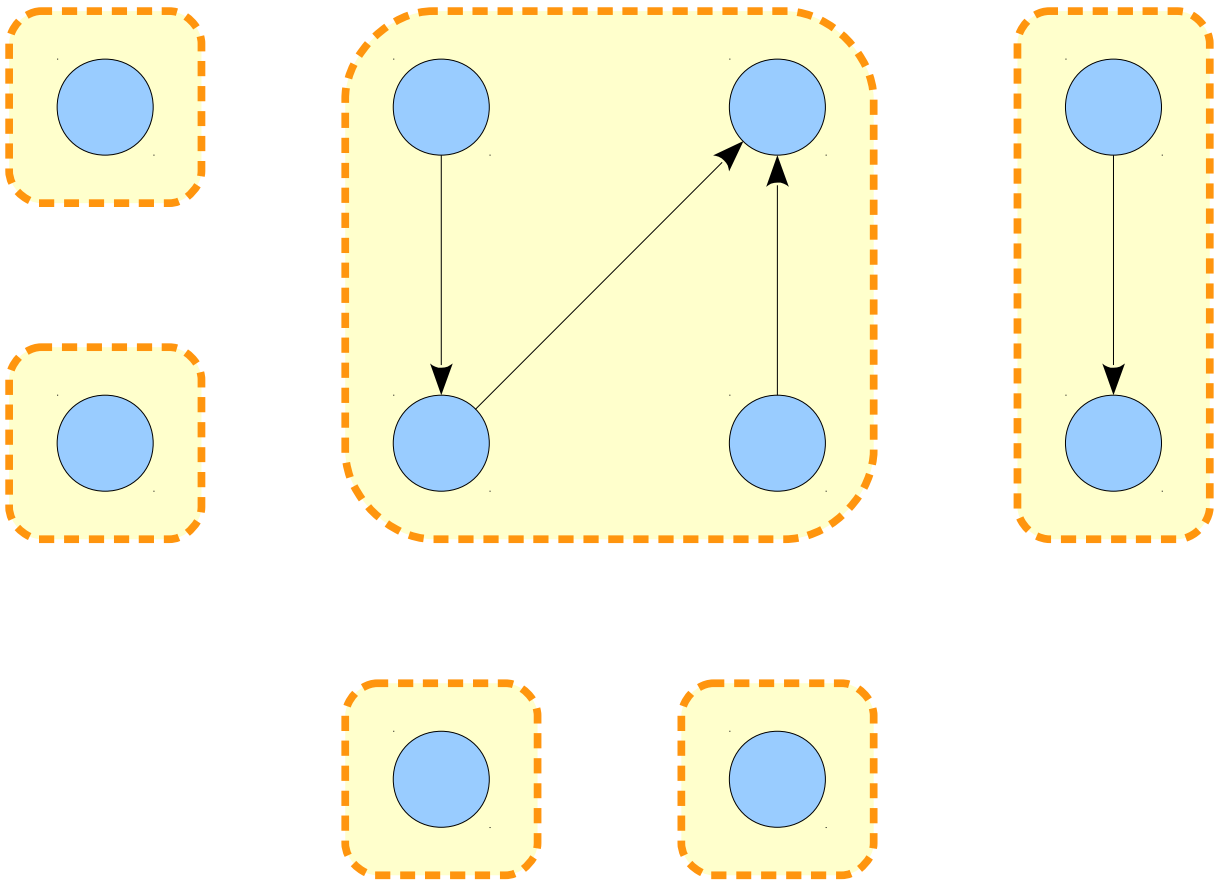
Hierarchical Representatives



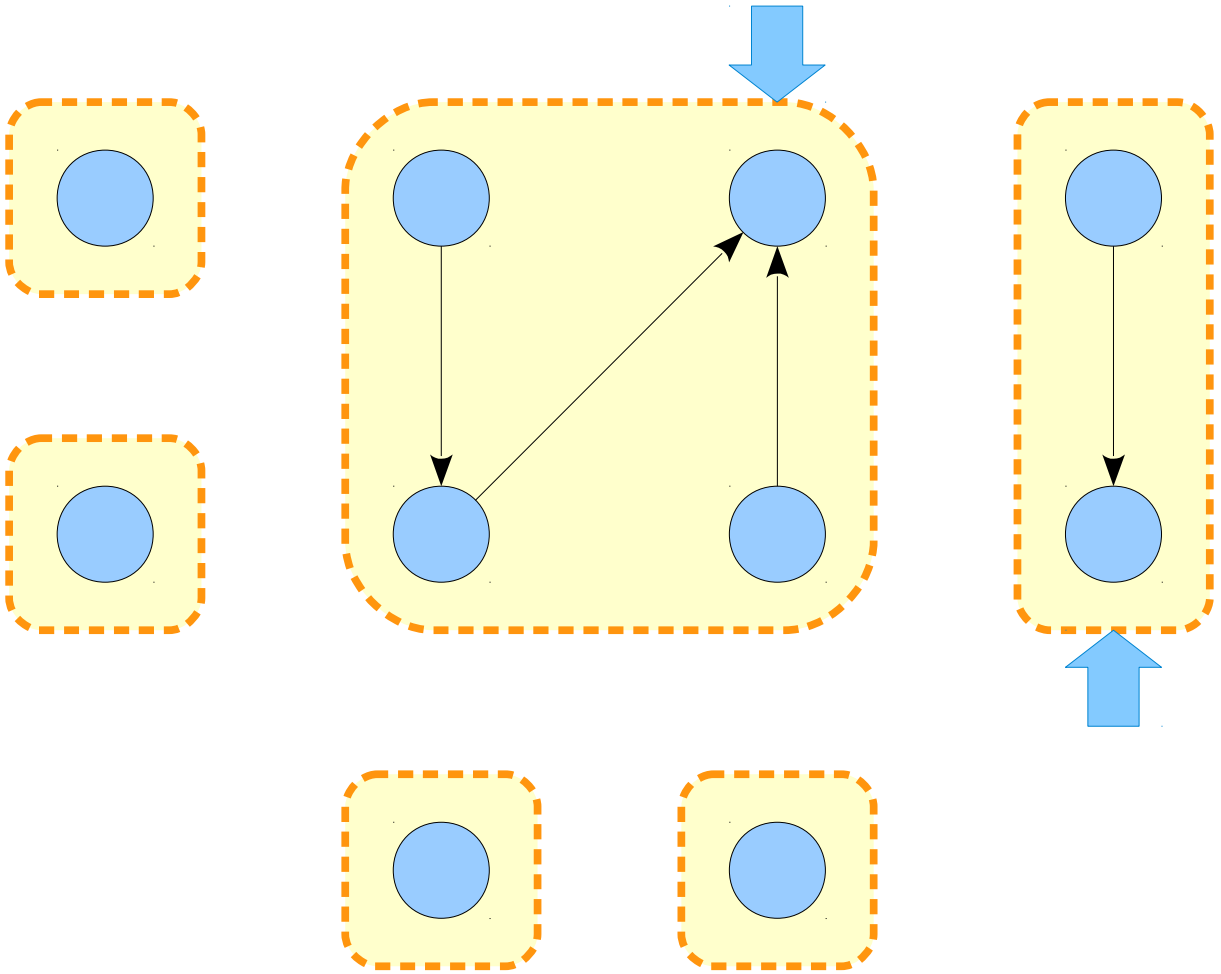
Hierarchical Representatives



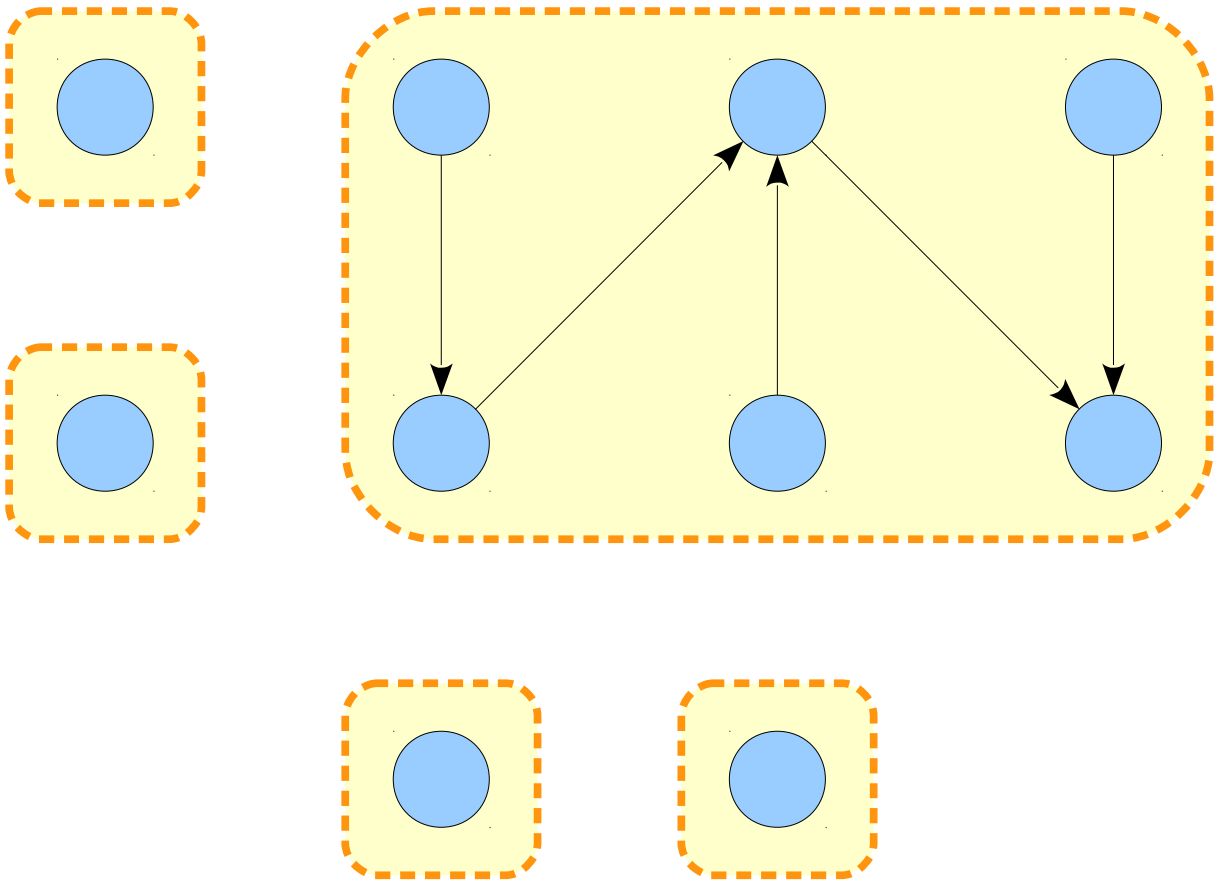
Hierarchical Representatives



Hierarchical Representatives



Hierarchical Representatives



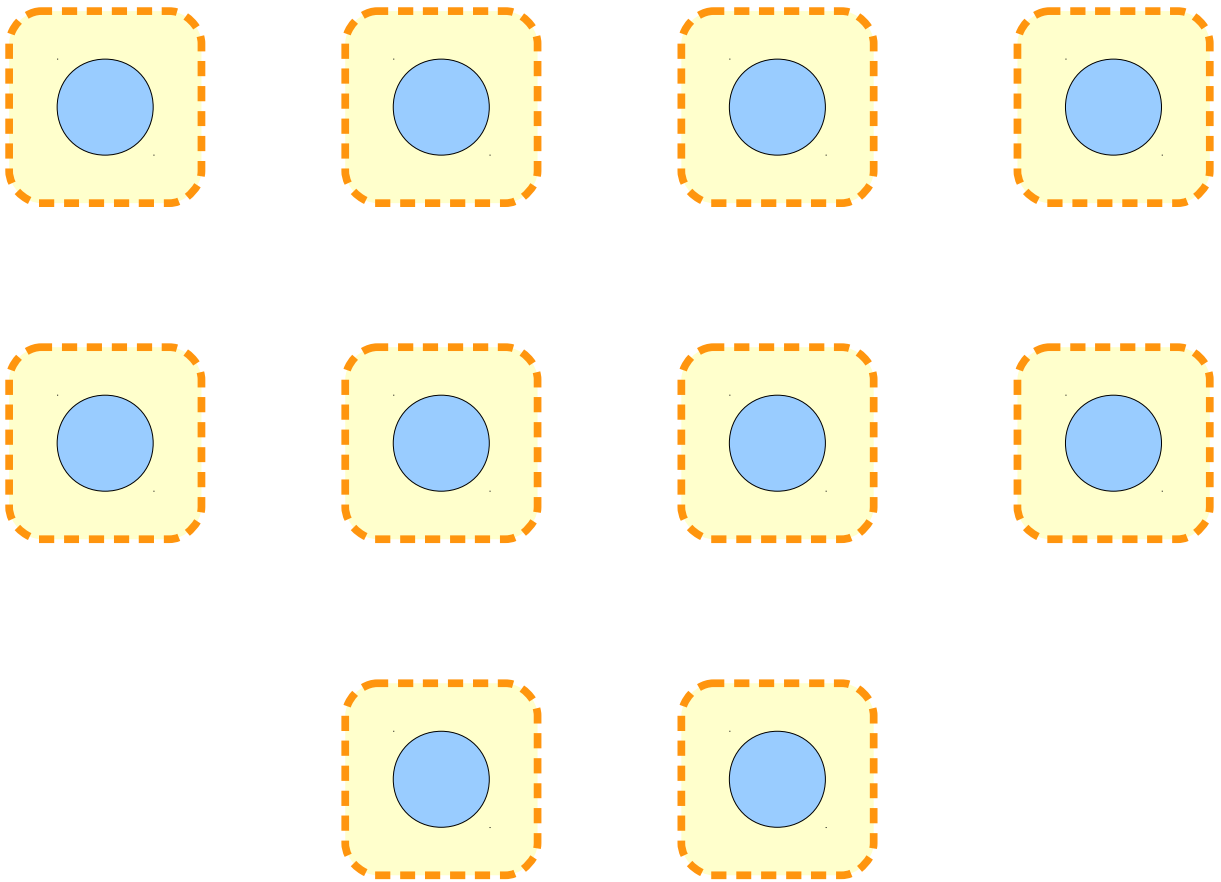
Hierarchical Representatives

- In a degenerate case, a hierarchical representative approach will require time $\Theta(n)$ for some ***find*** operations.
- Therefore, some ***union*** operations will take time $\Theta(n)$ as well.
- Can we avoid these degenerate cases?

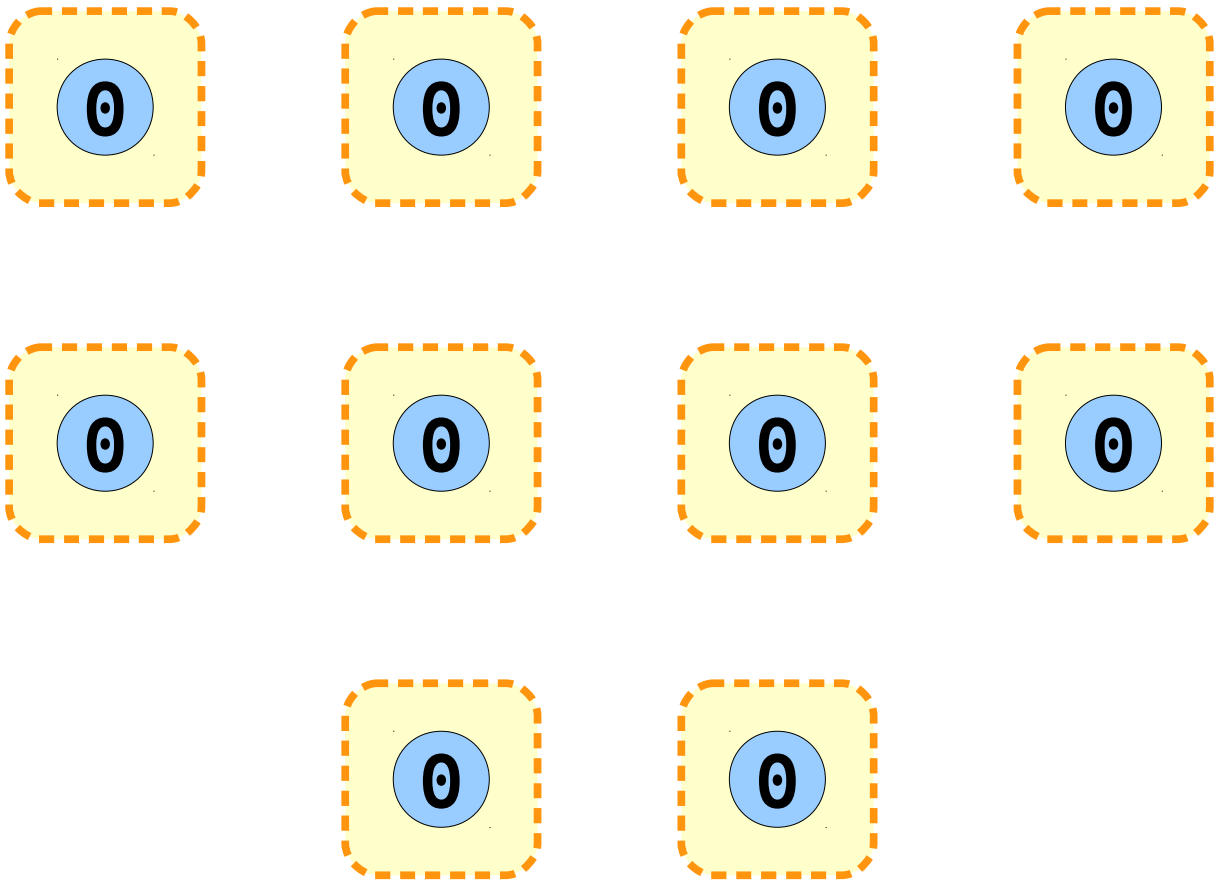
Union by Rank

- Assign to each node a *rank* that is initially zero.
- To link two trees, link the tree of the smaller rank to the tree of the larger rank.
- If both trees have the same rank, link one to the other and increase the rank of the other tree by one.

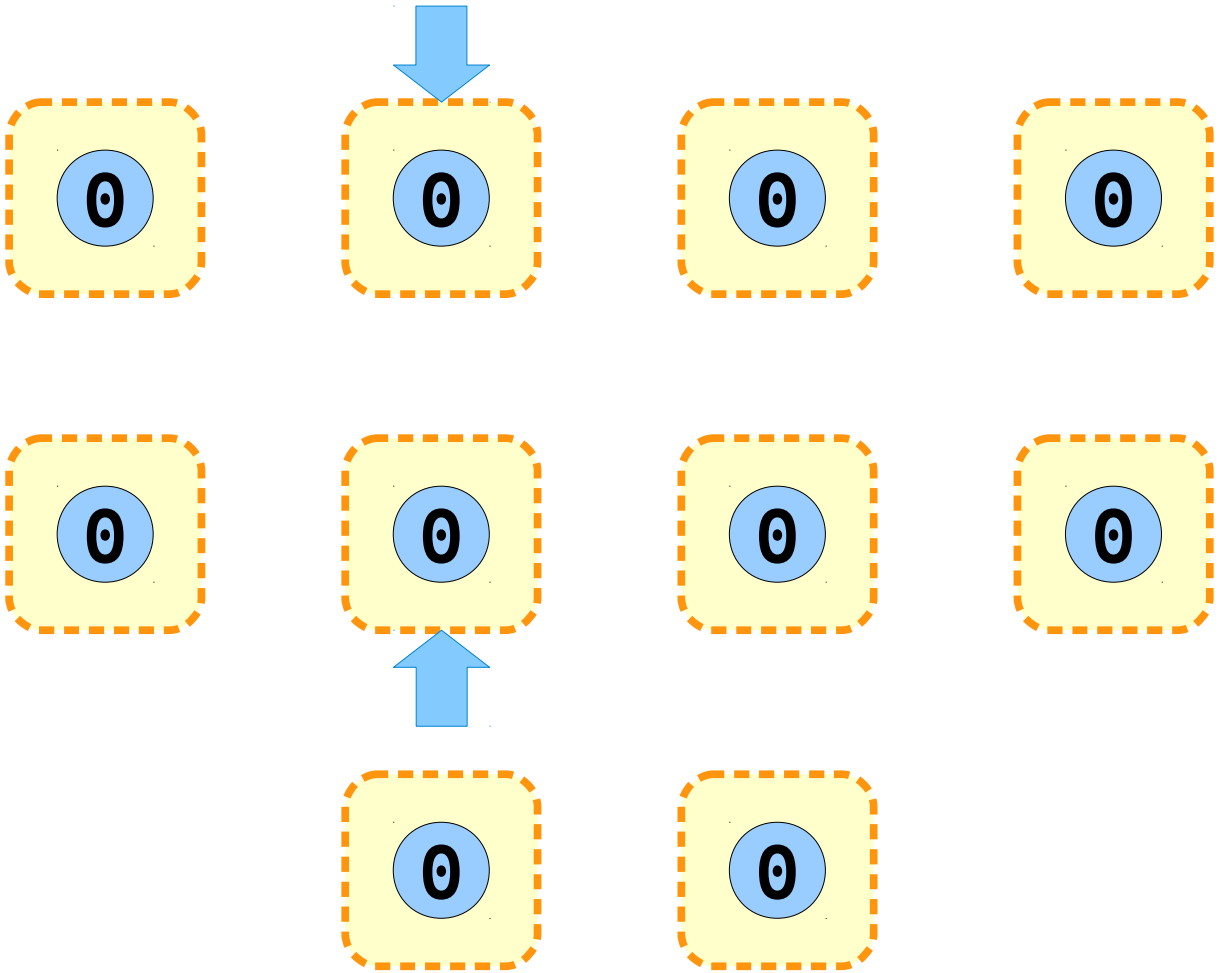
Union by Rank



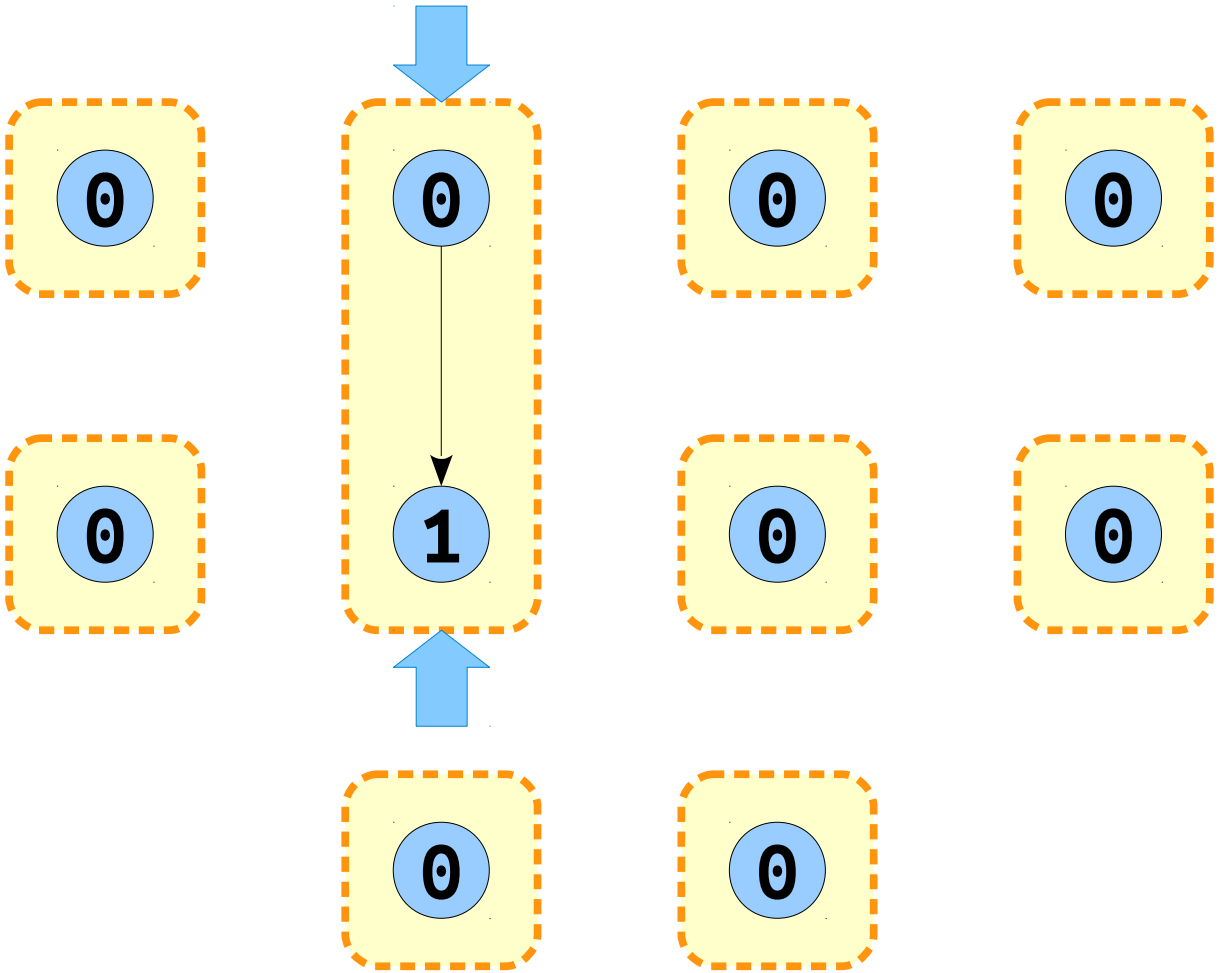
Union by Rank



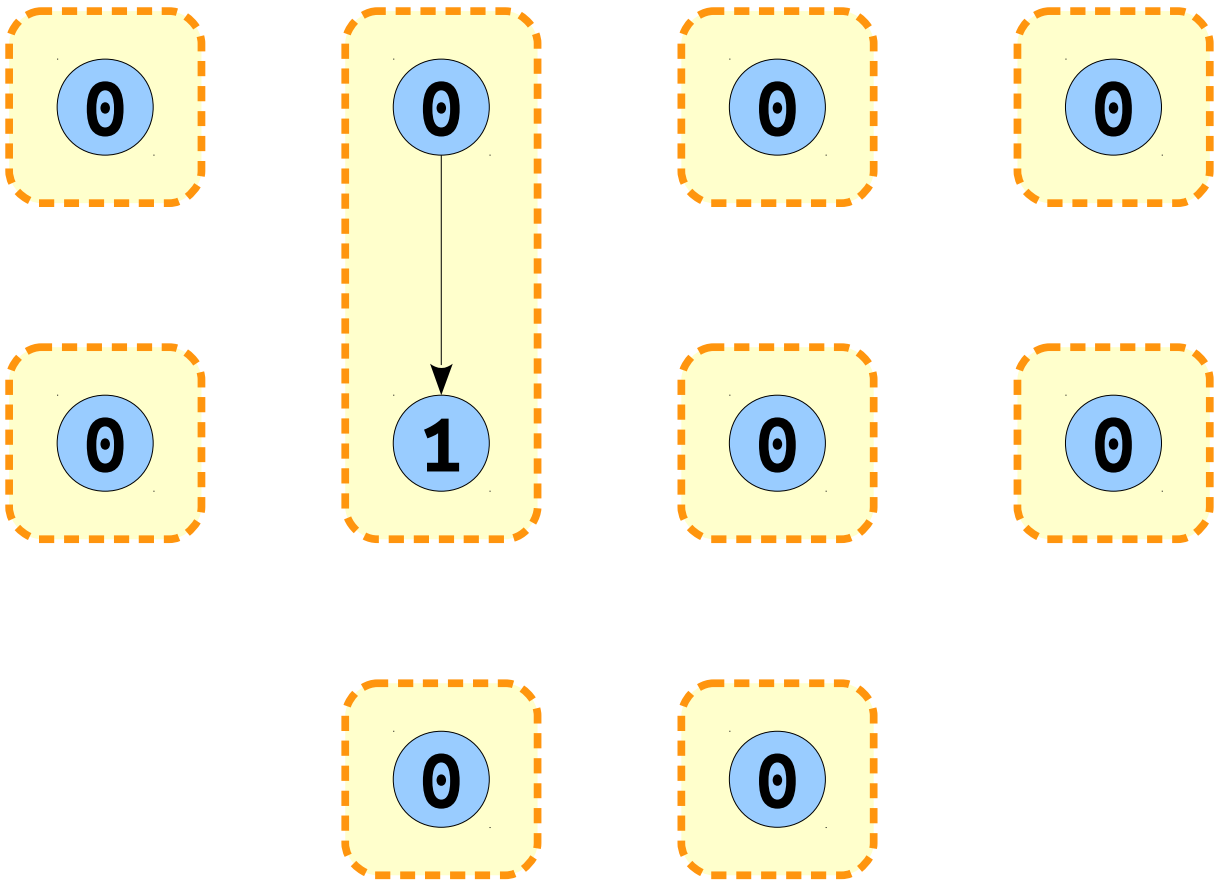
Union by Rank



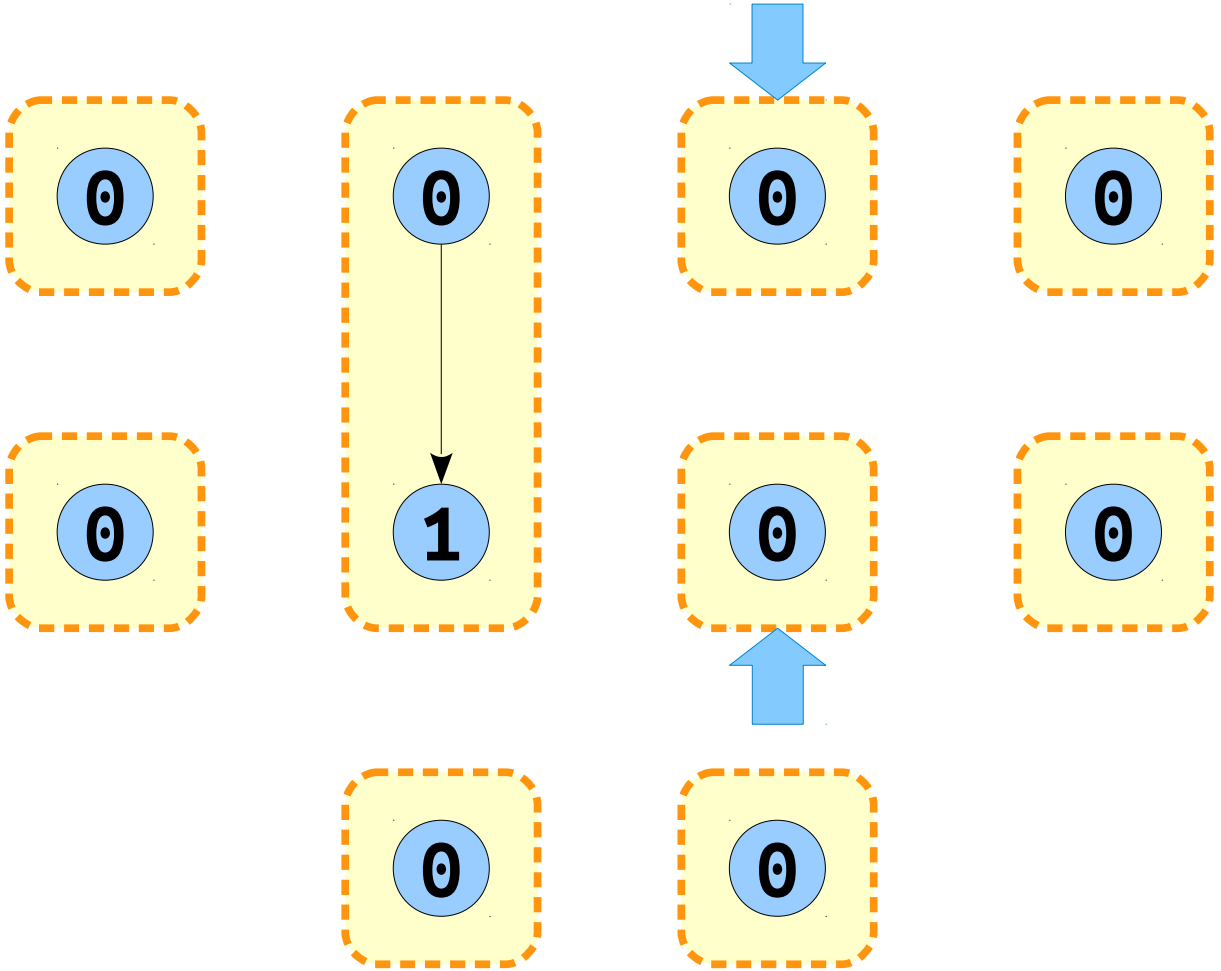
Union by Rank



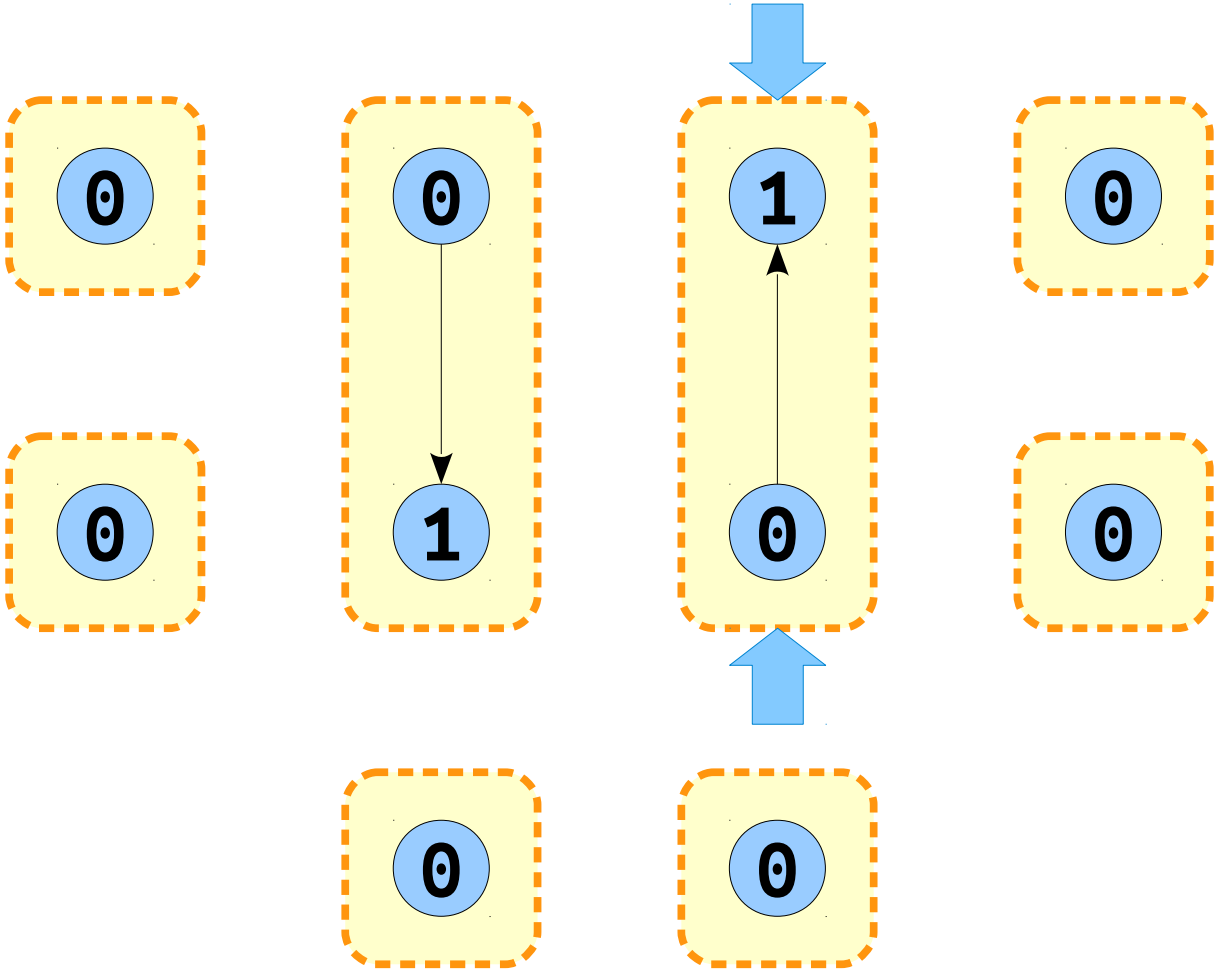
Union by Rank



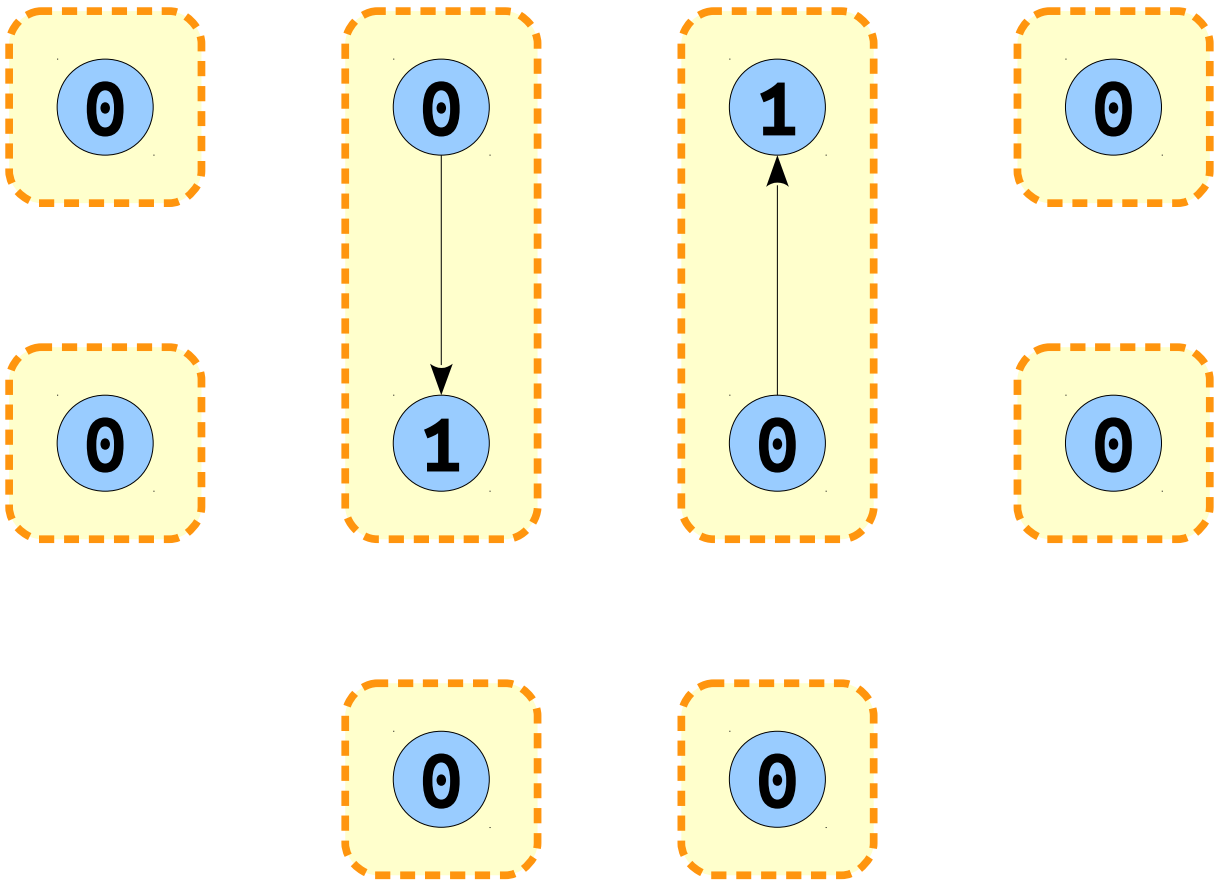
Union by Rank



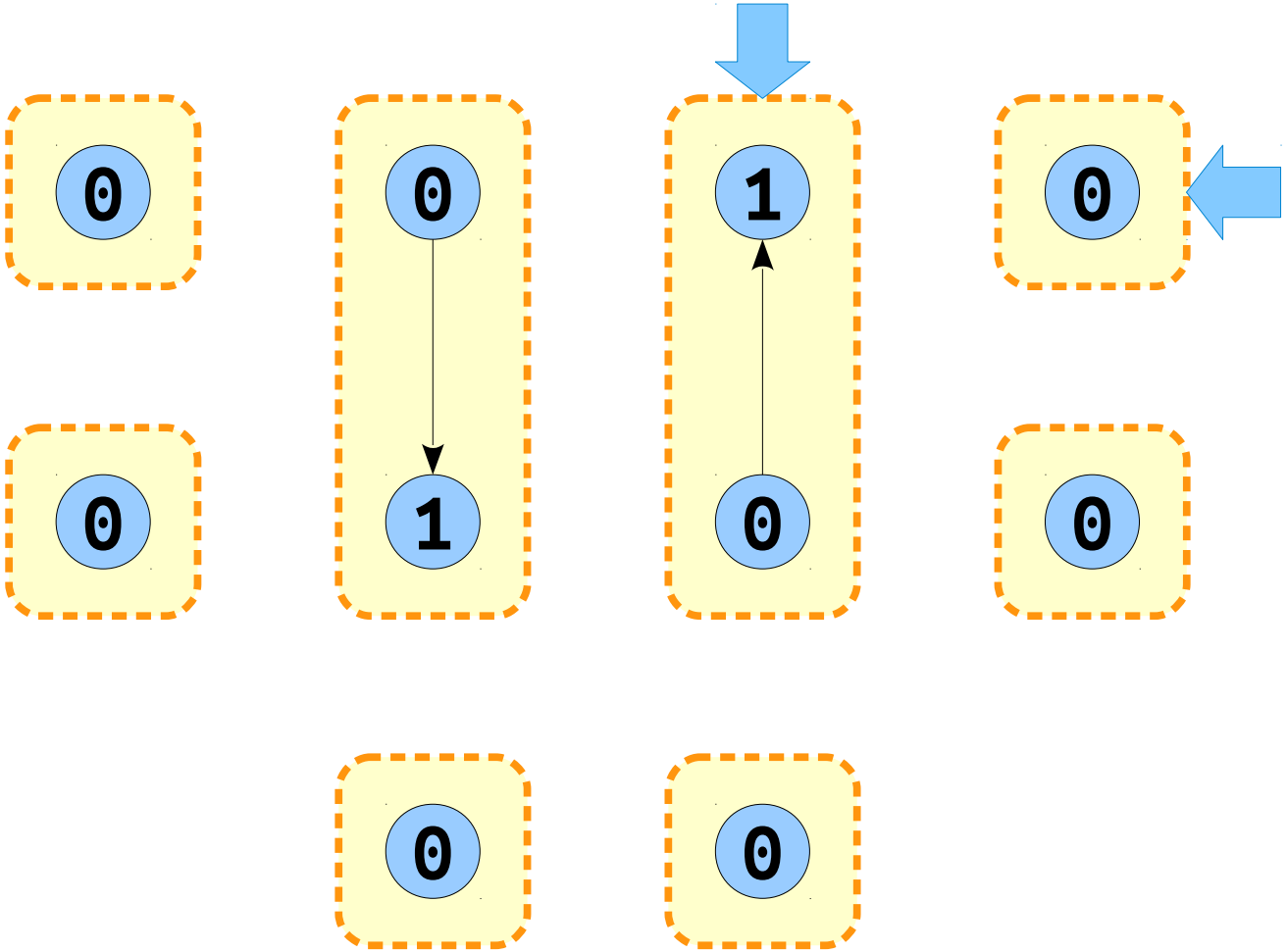
Union by Rank



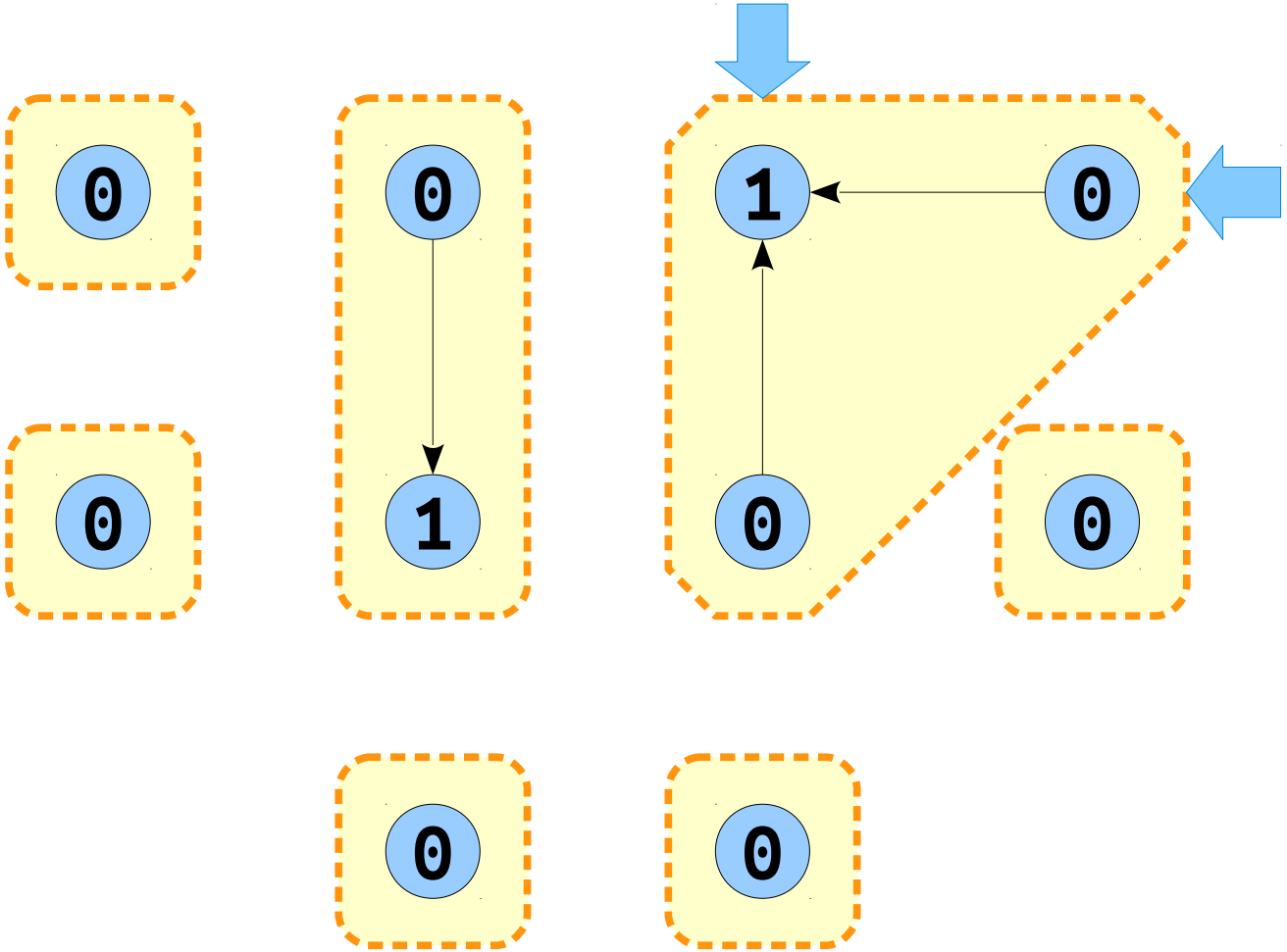
Union by Rank



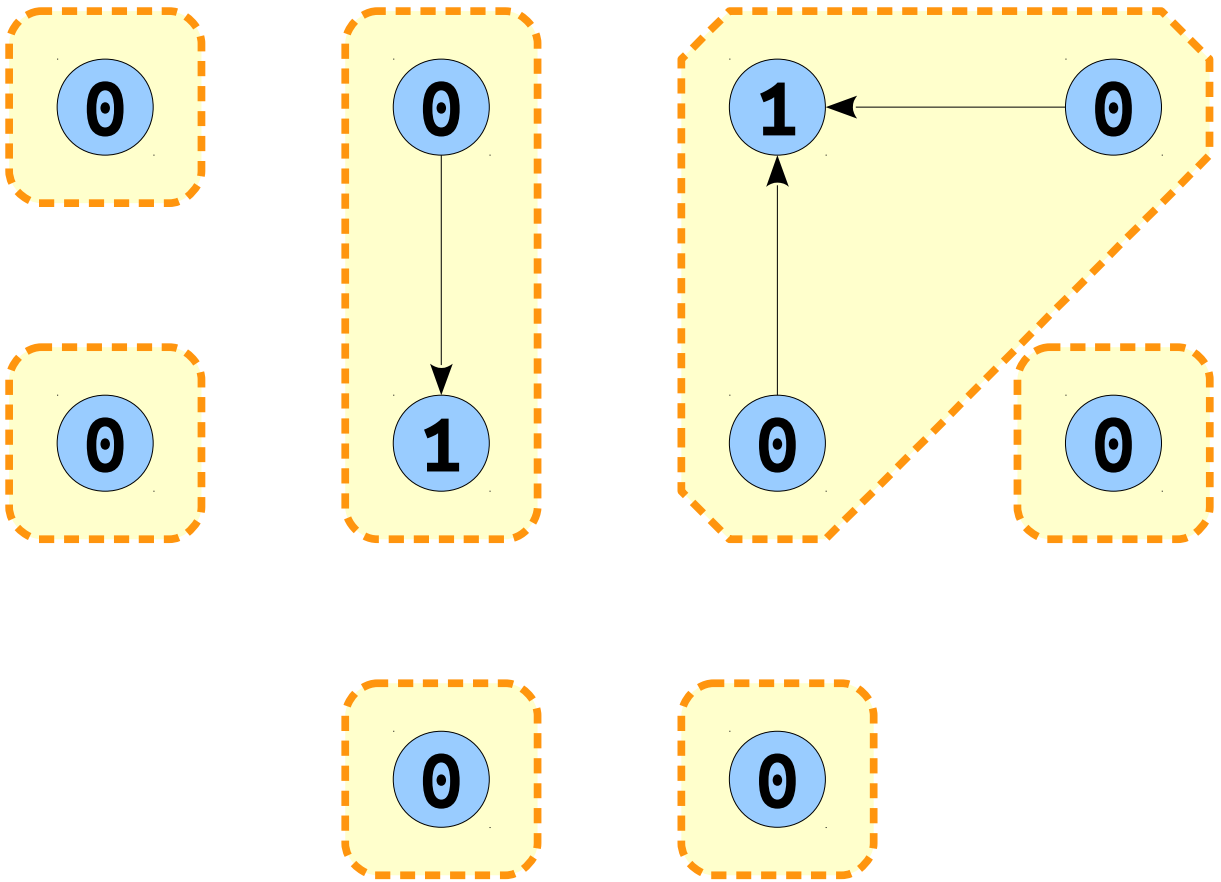
Union by Rank



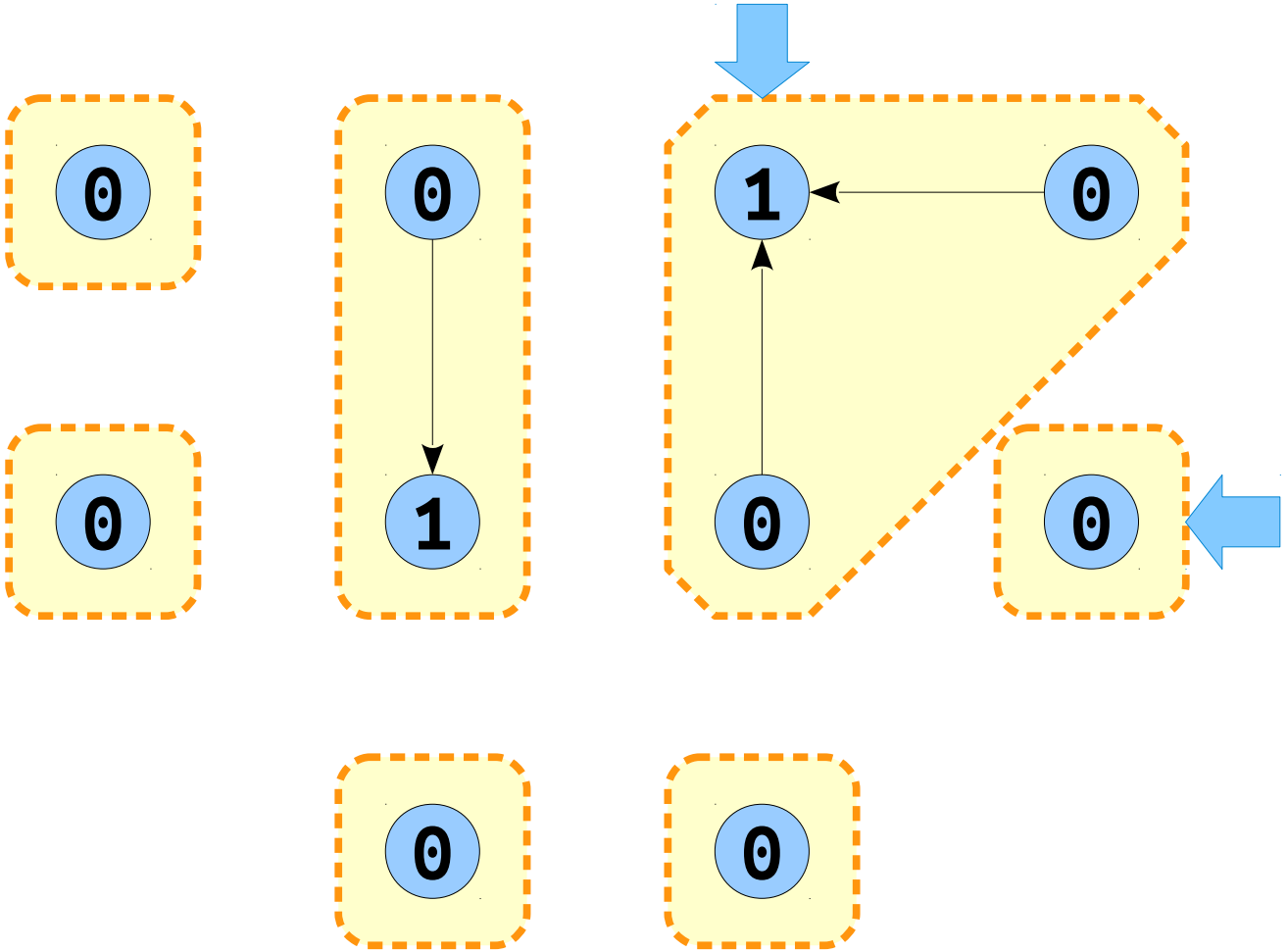
Union by Rank



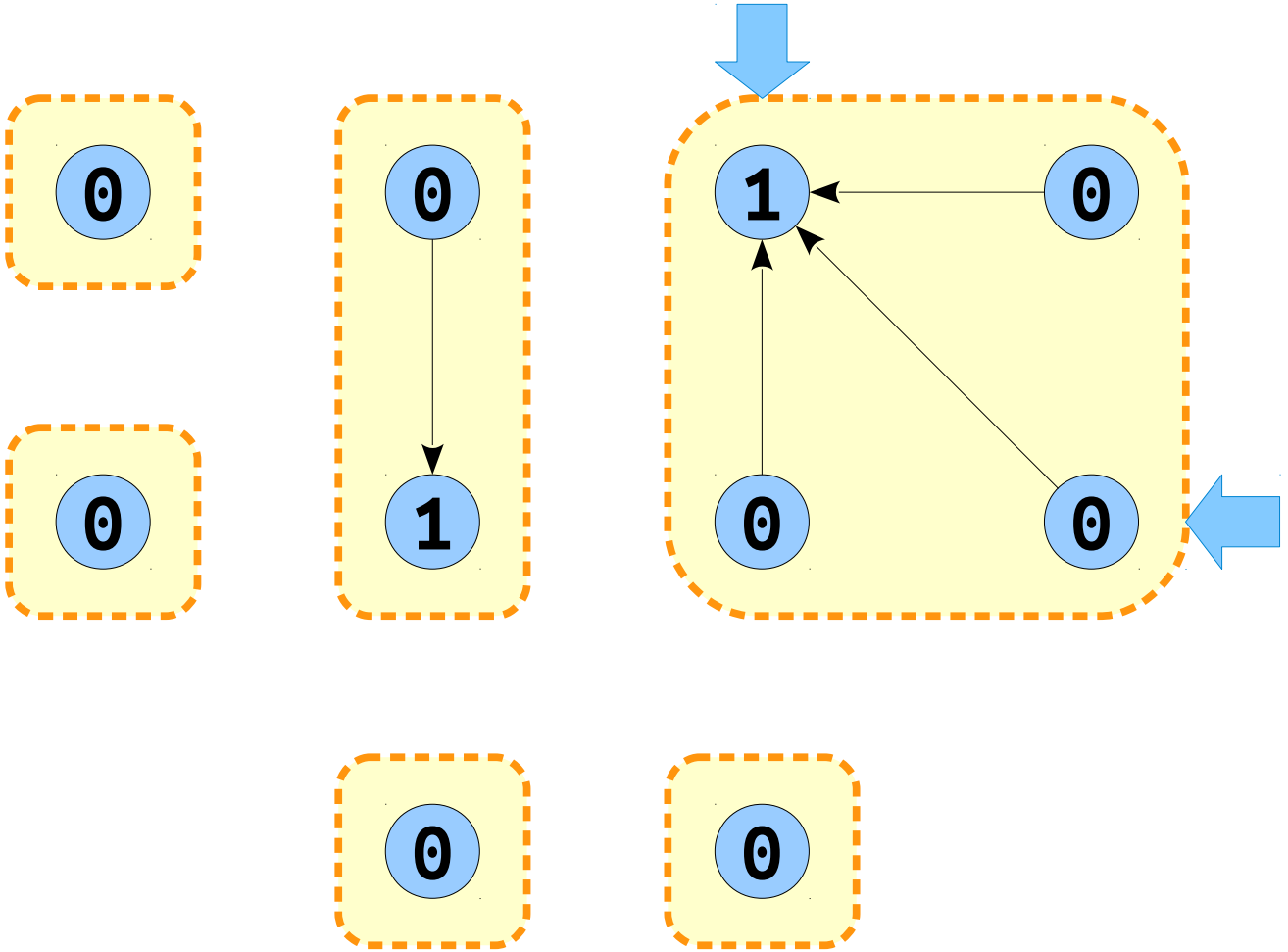
Union by Rank



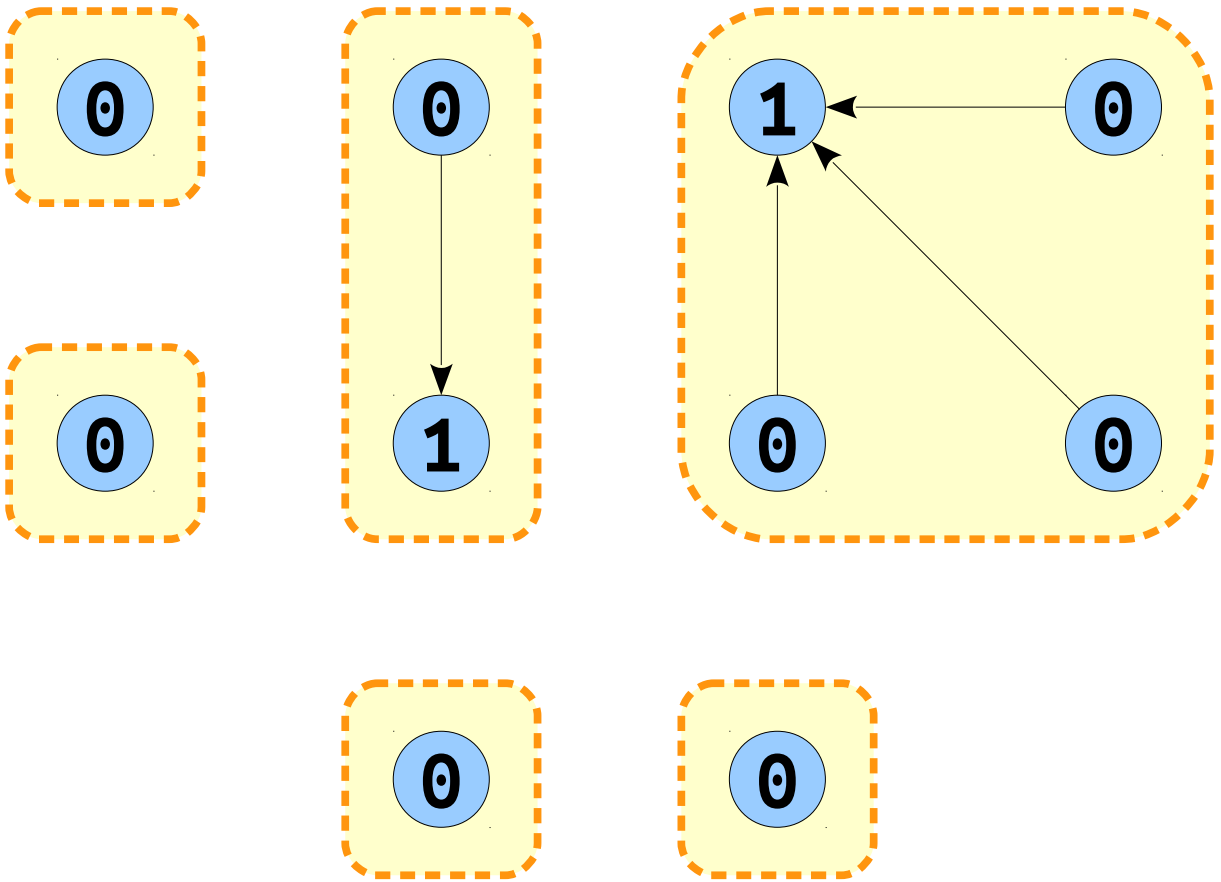
Union by Rank



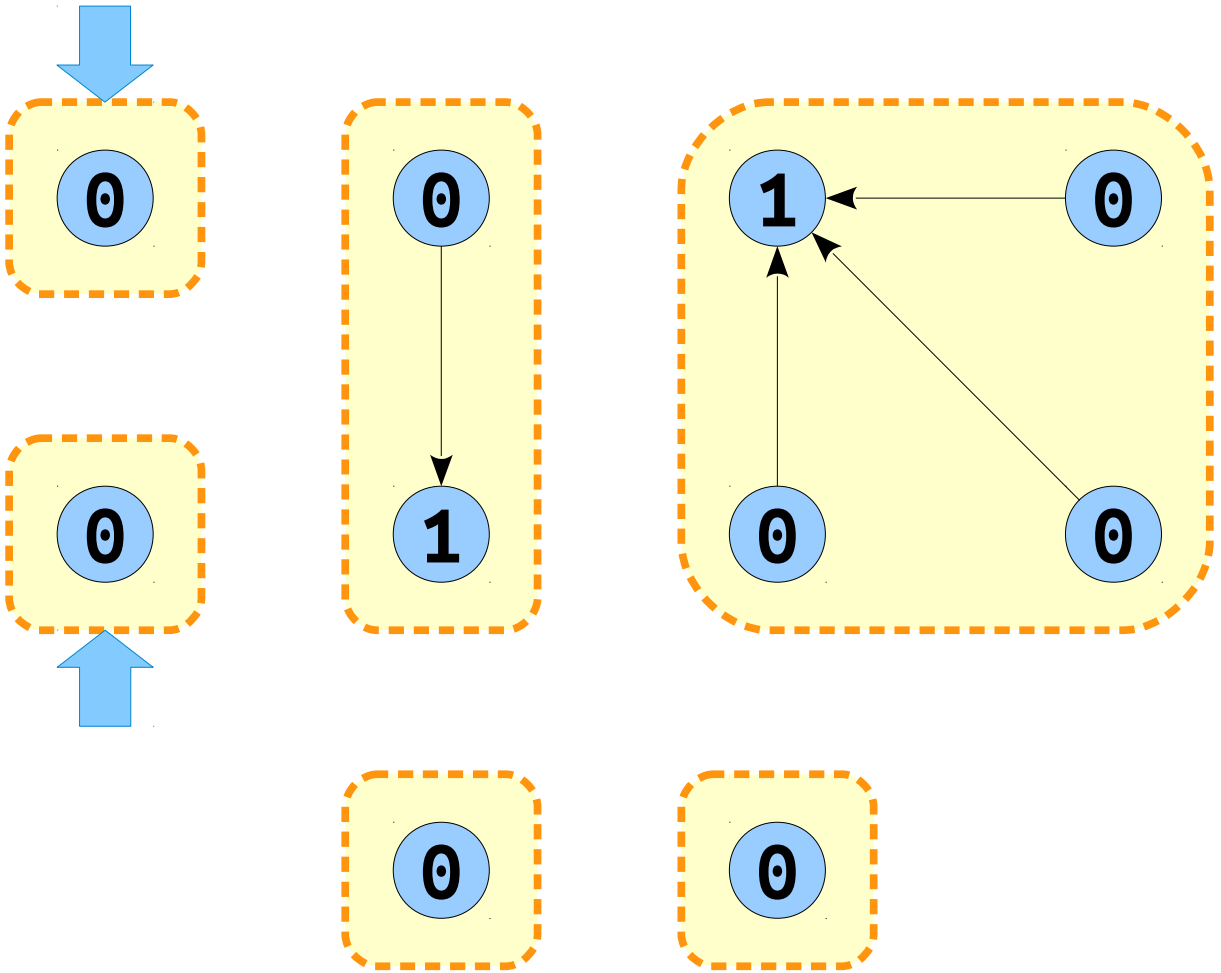
Union by Rank



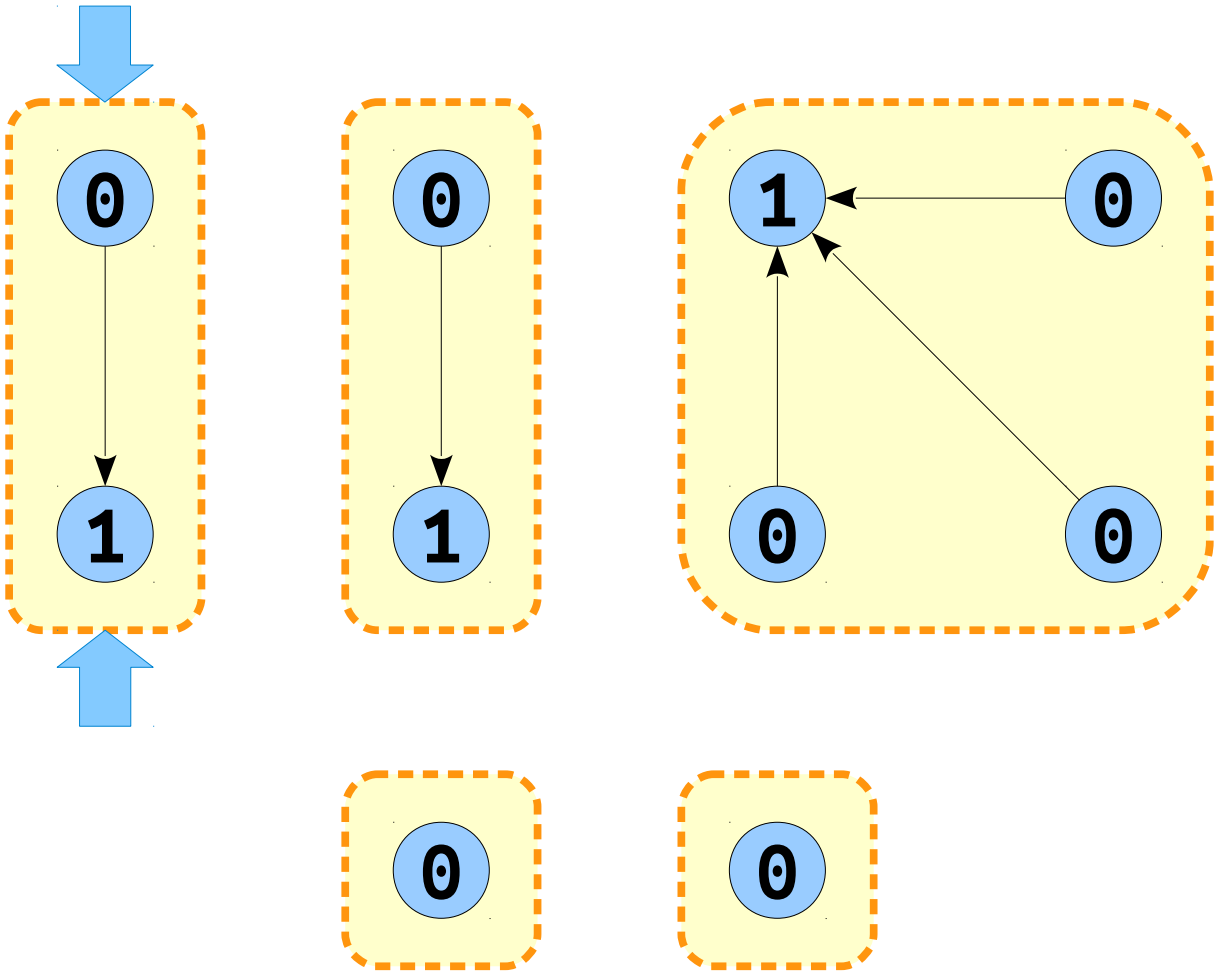
Union by Rank



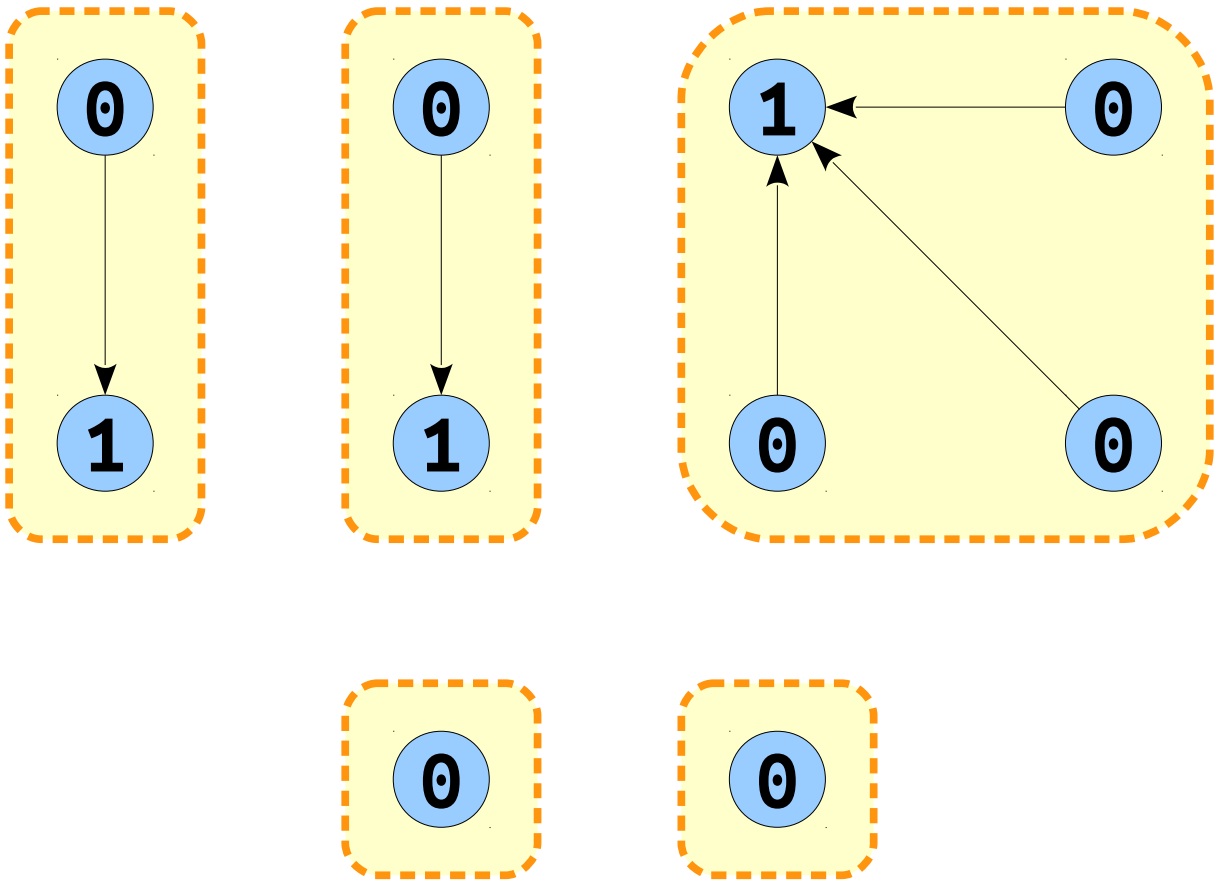
Union by Rank



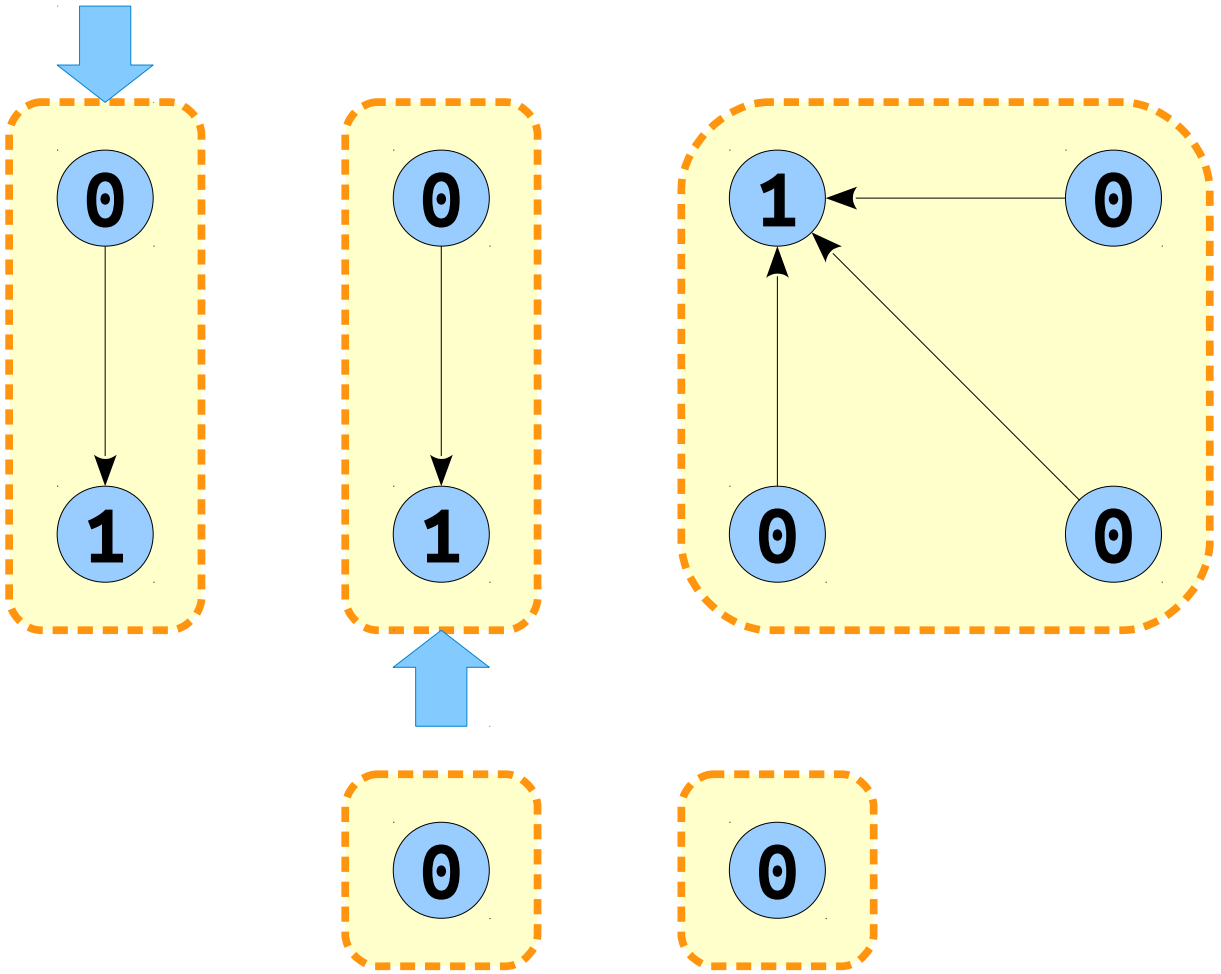
Union by Rank



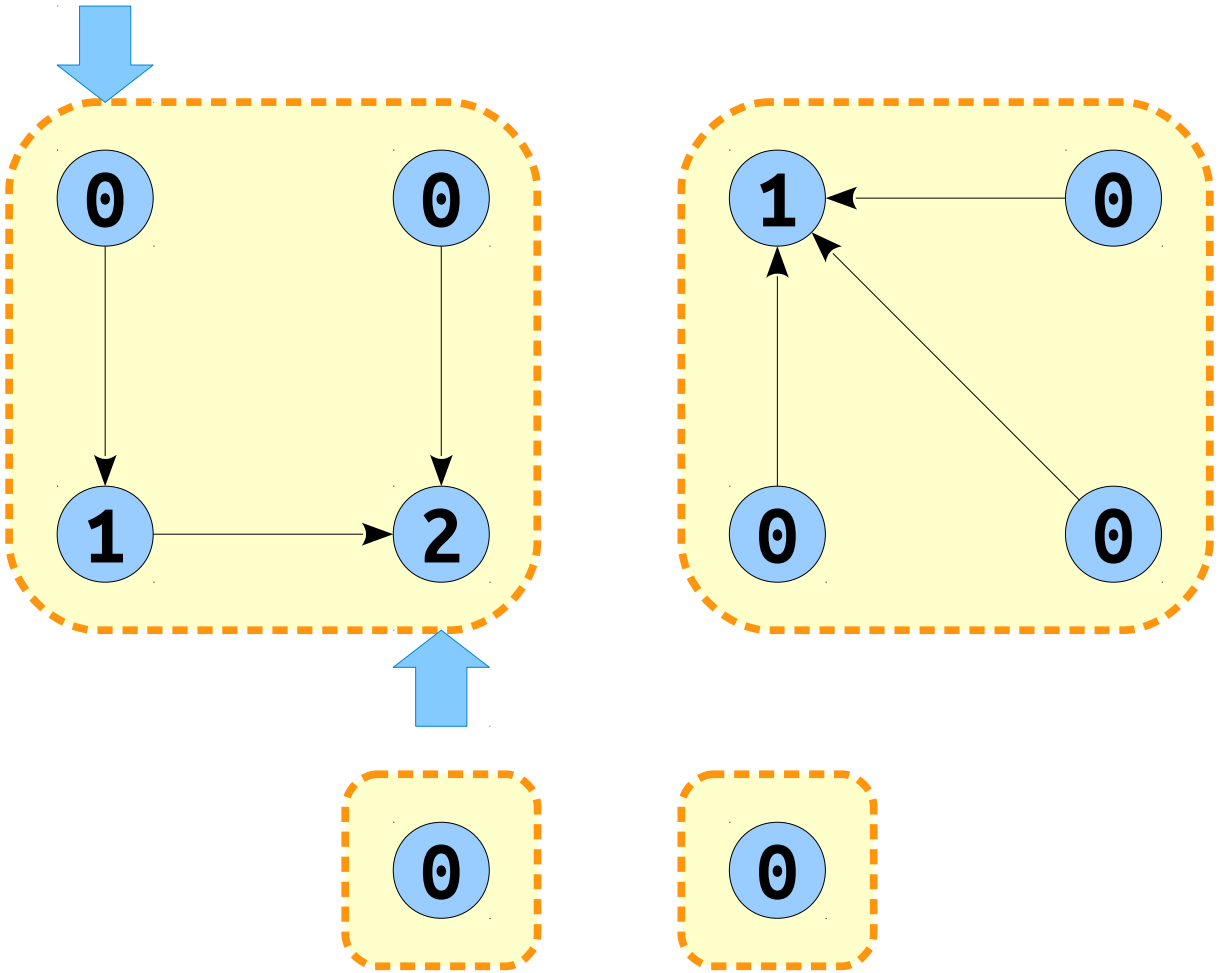
Union by Rank



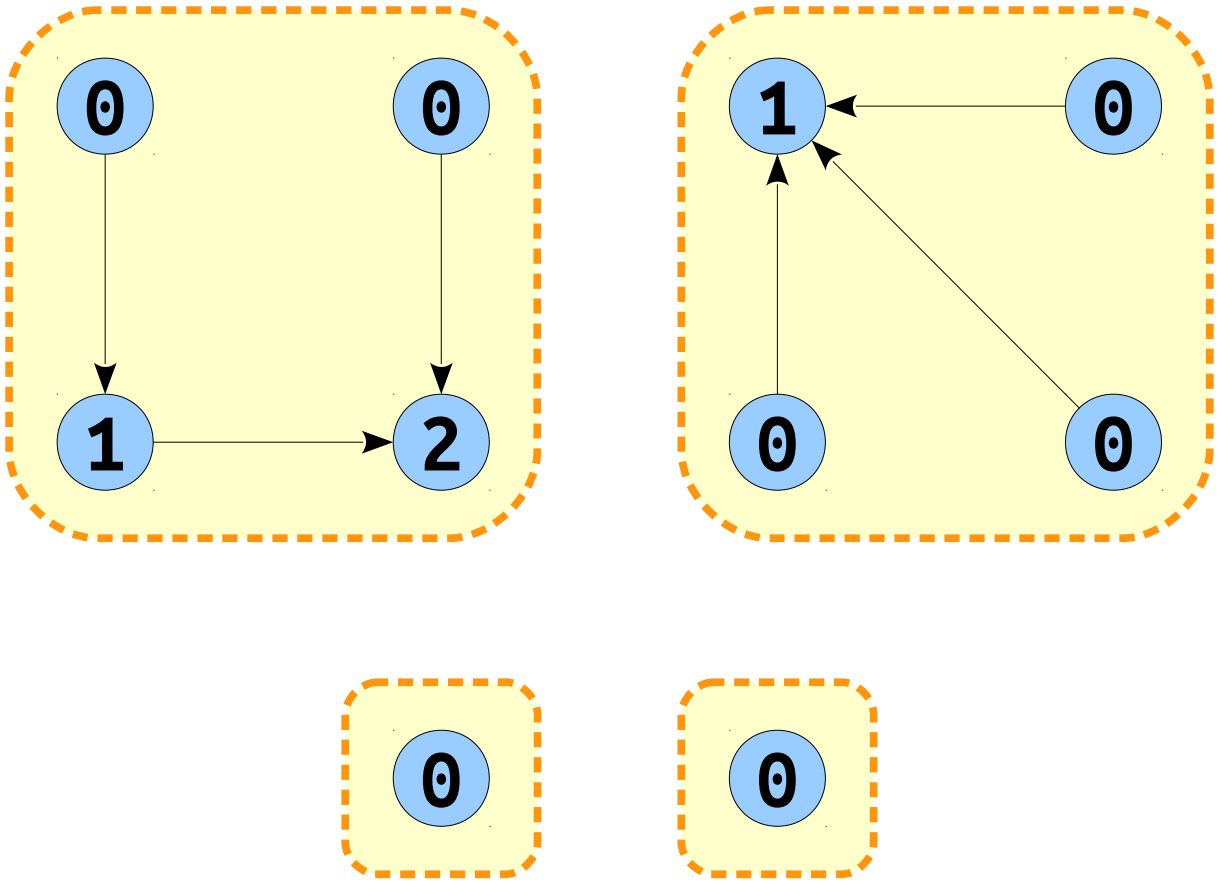
Union by Rank



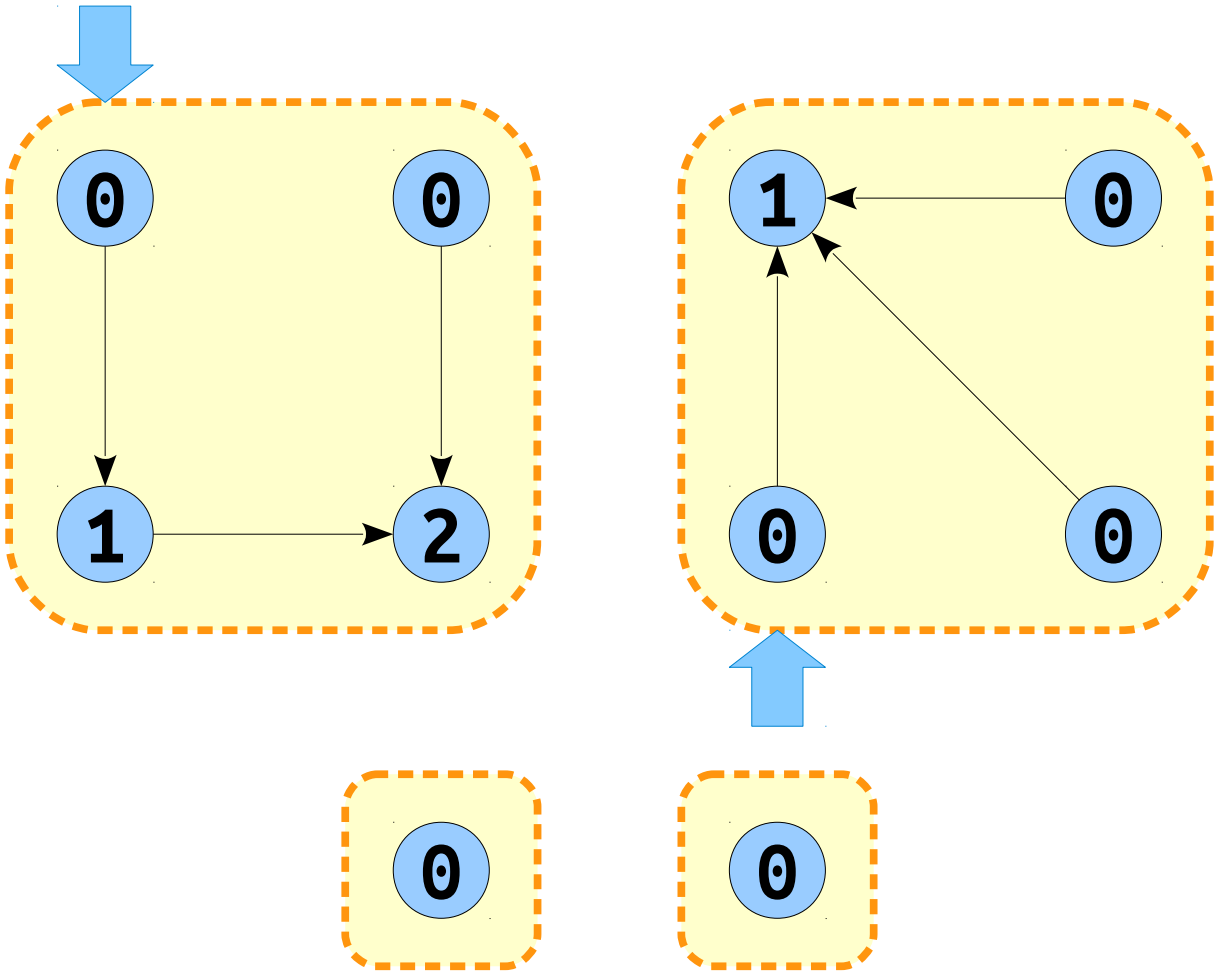
Union by Rank



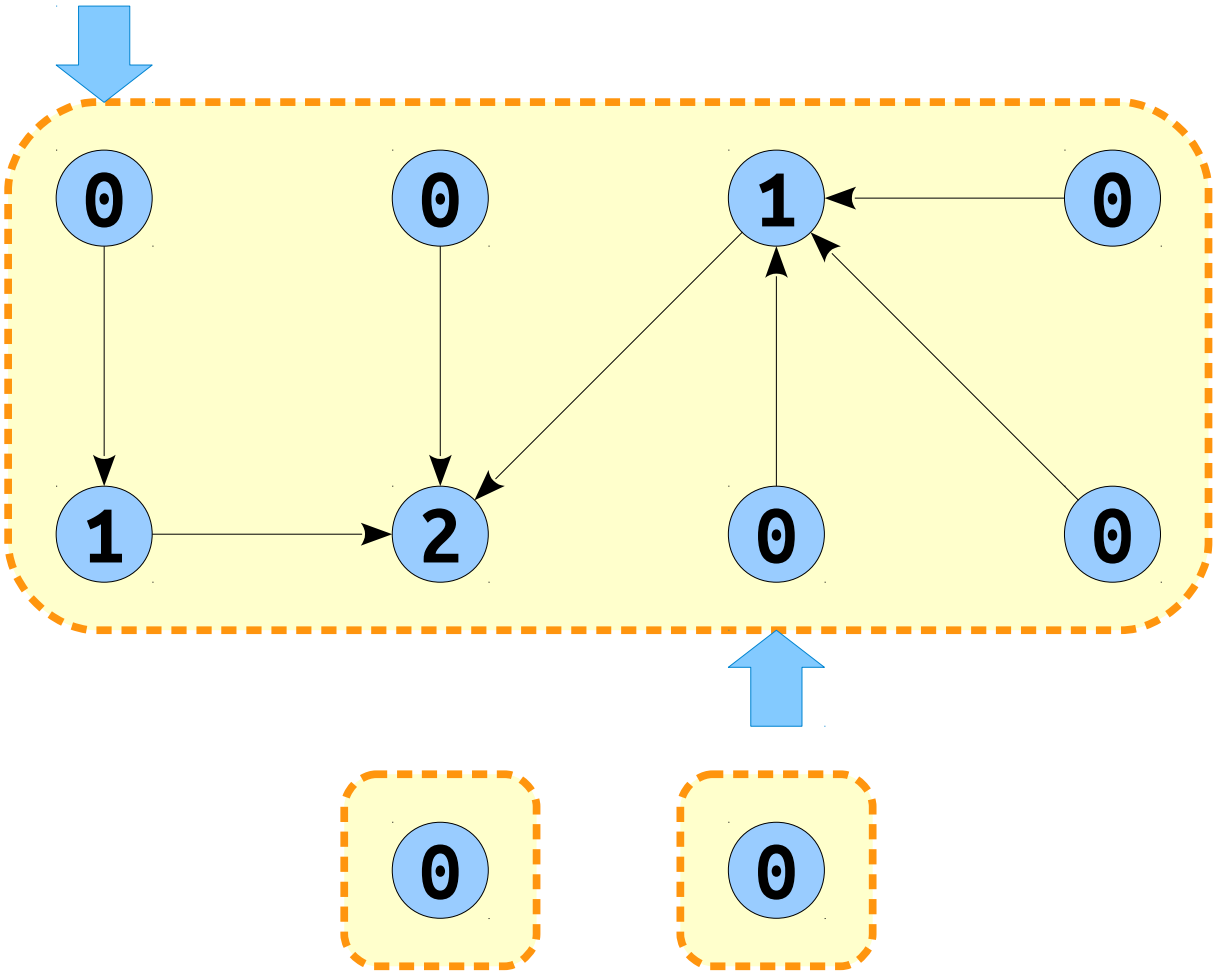
Union by Rank



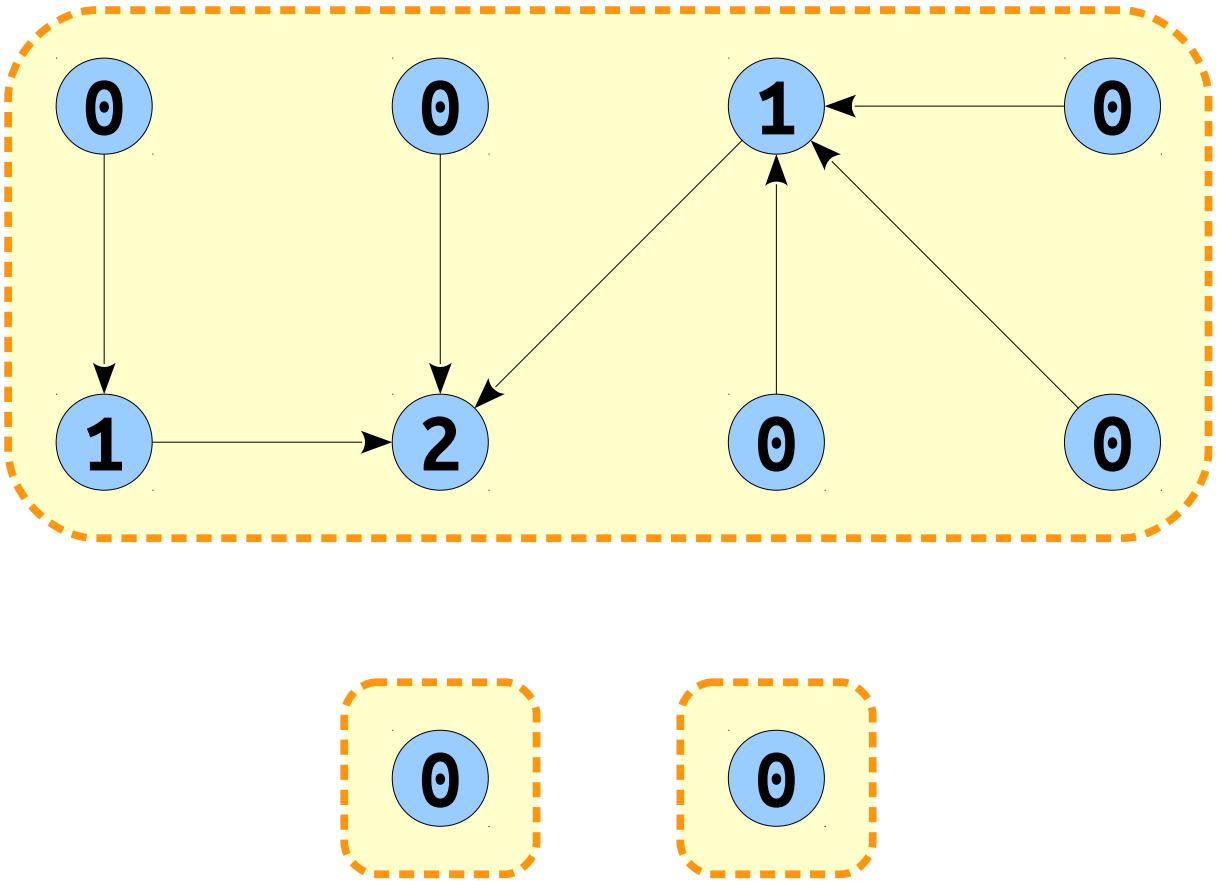
Union by Rank



Union by Rank



Union by Rank



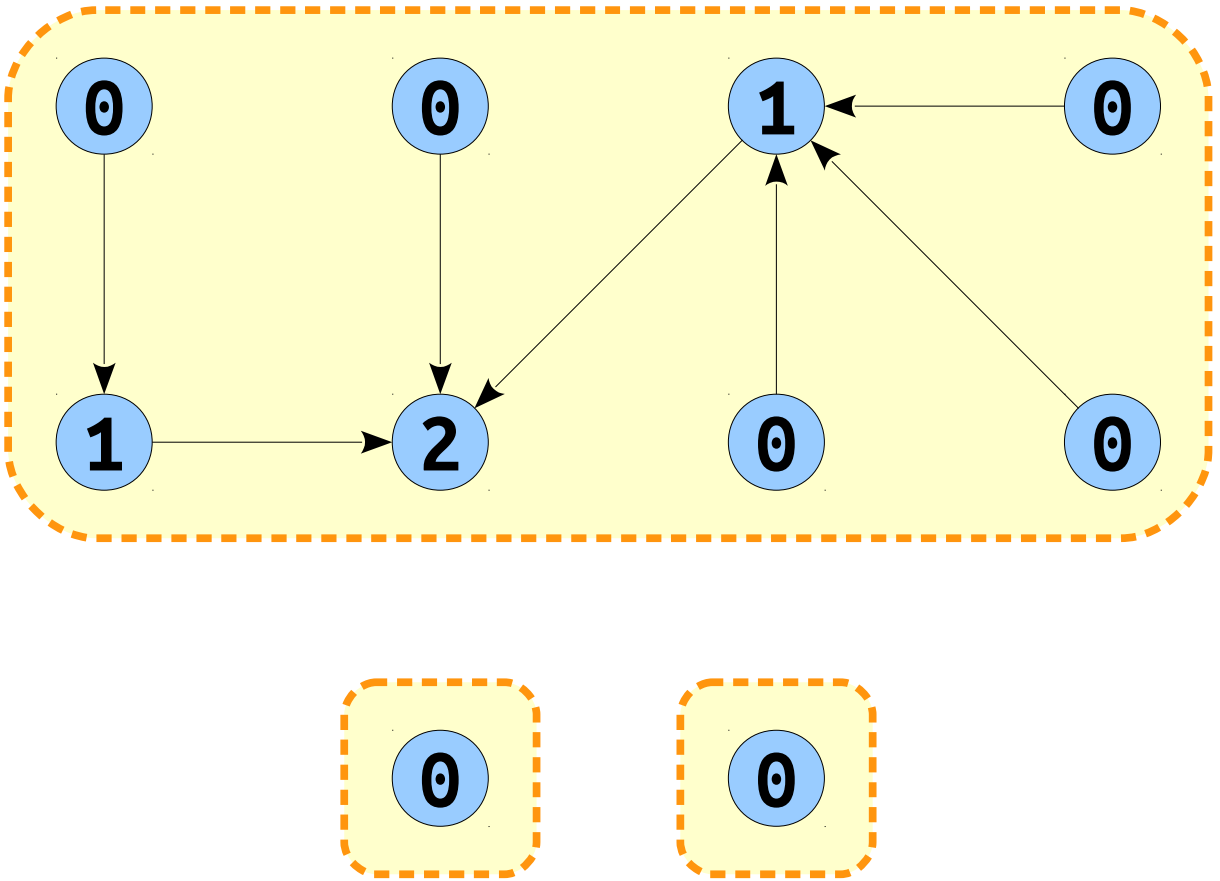
Union by Rank

- ***Claim:*** The number of nodes in a tree of rank r is at least 2^r.
  - Proof is by induction; intuitively, need to double the size to get to a tree of the next order.
  - Fun fact: the smallest tree with a root of rank r is a binomial tree of order r. Crazy!
- ***Claim:*** Maximum rank of a node in a graph with n nodes is O(log n).
- Runtime for ***union*** and ***find*** is now O(log n).
- ***Useful fact for later on:*** The number of nodes of rank r or higher in a disjoint set forest with n nodes is at most n / 2^r.

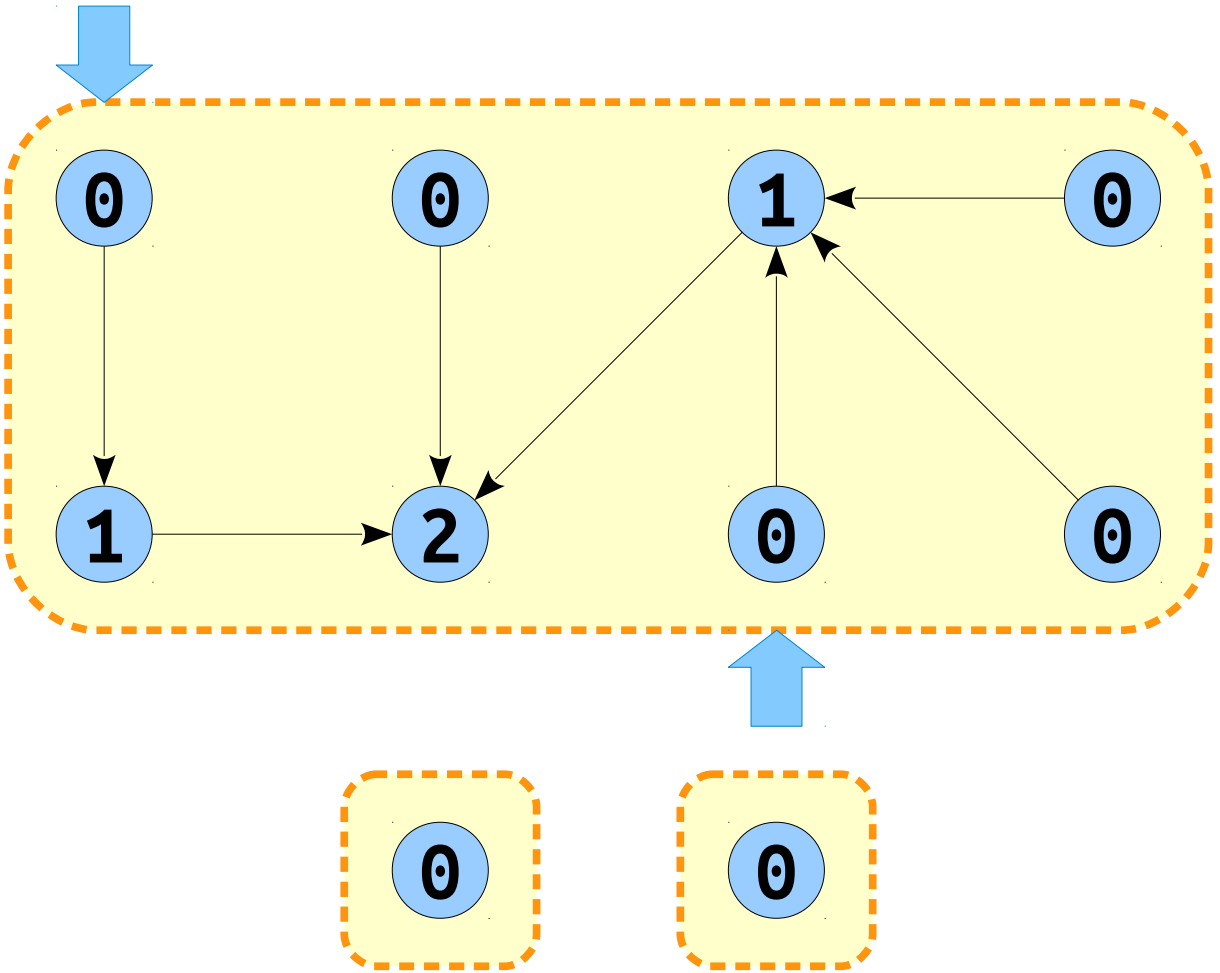
Path Compression

- ***Path compression*** is an optimization to the standard disjoint-set forest.
- When performing a ***find***, change the parent pointers of each node found along the way to point to the representative.
- Purely using path compression, each operation has amortized cost O(log n).
- What happens if we combine this with union-by-rank?

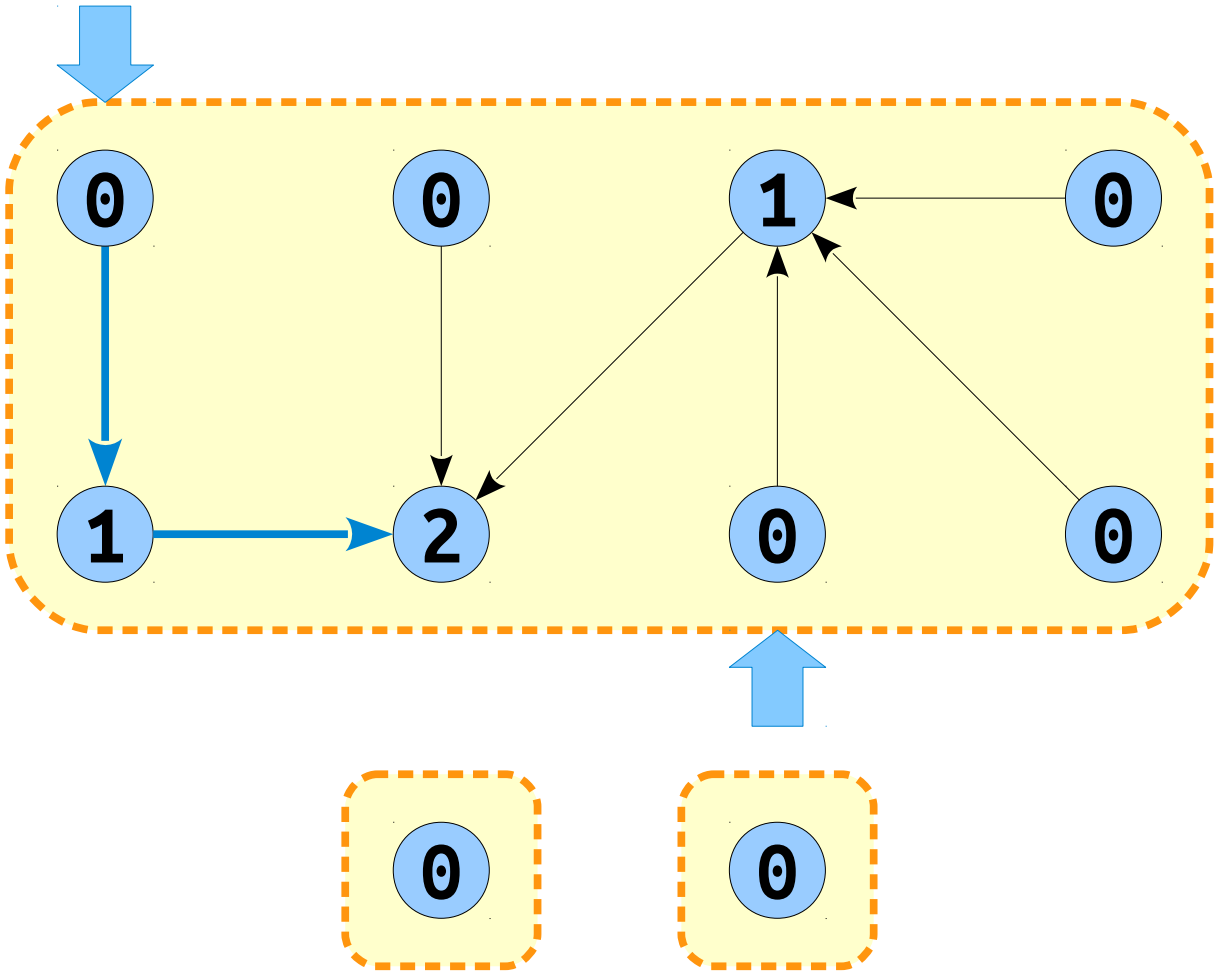
Path Compression



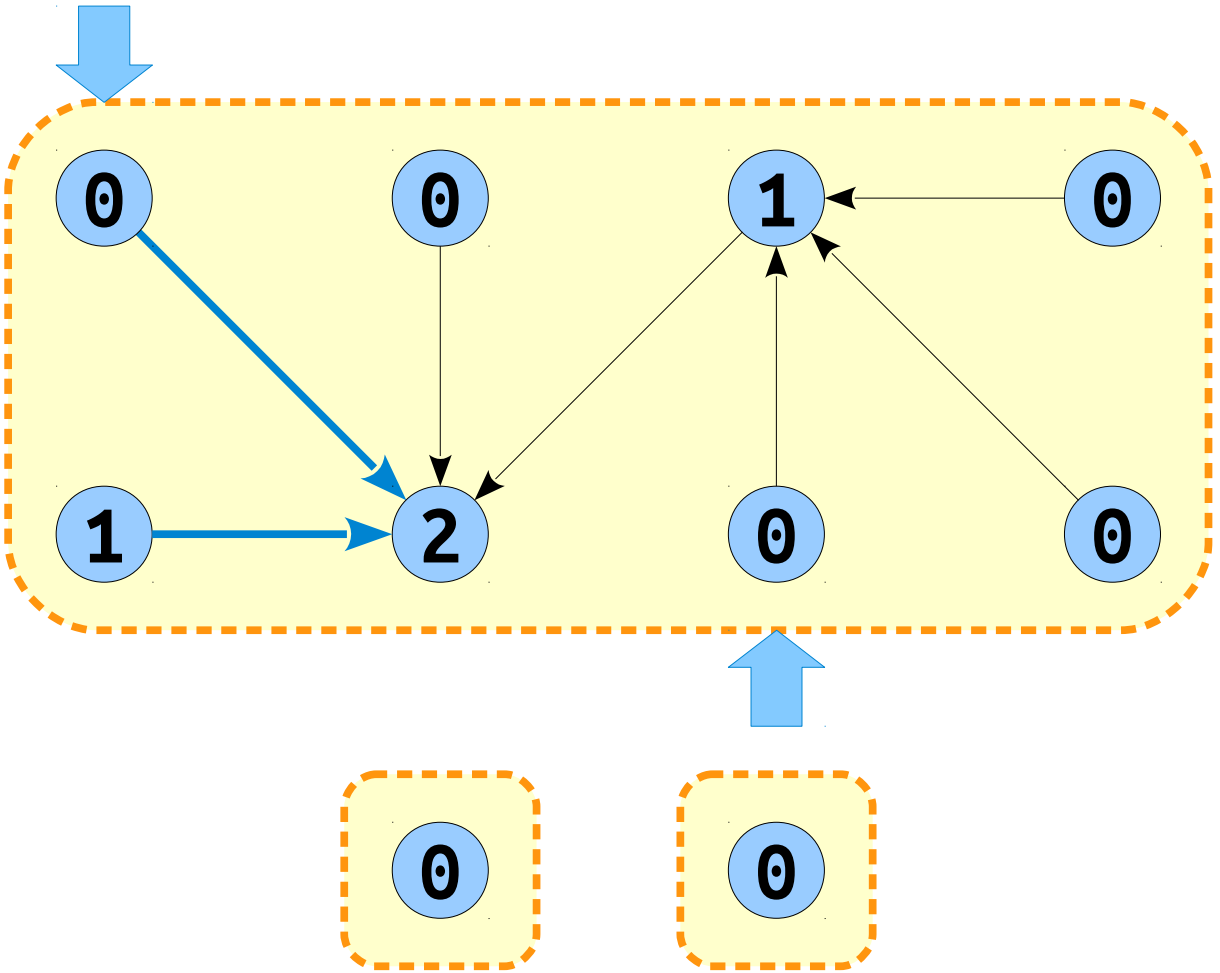
Path Compression



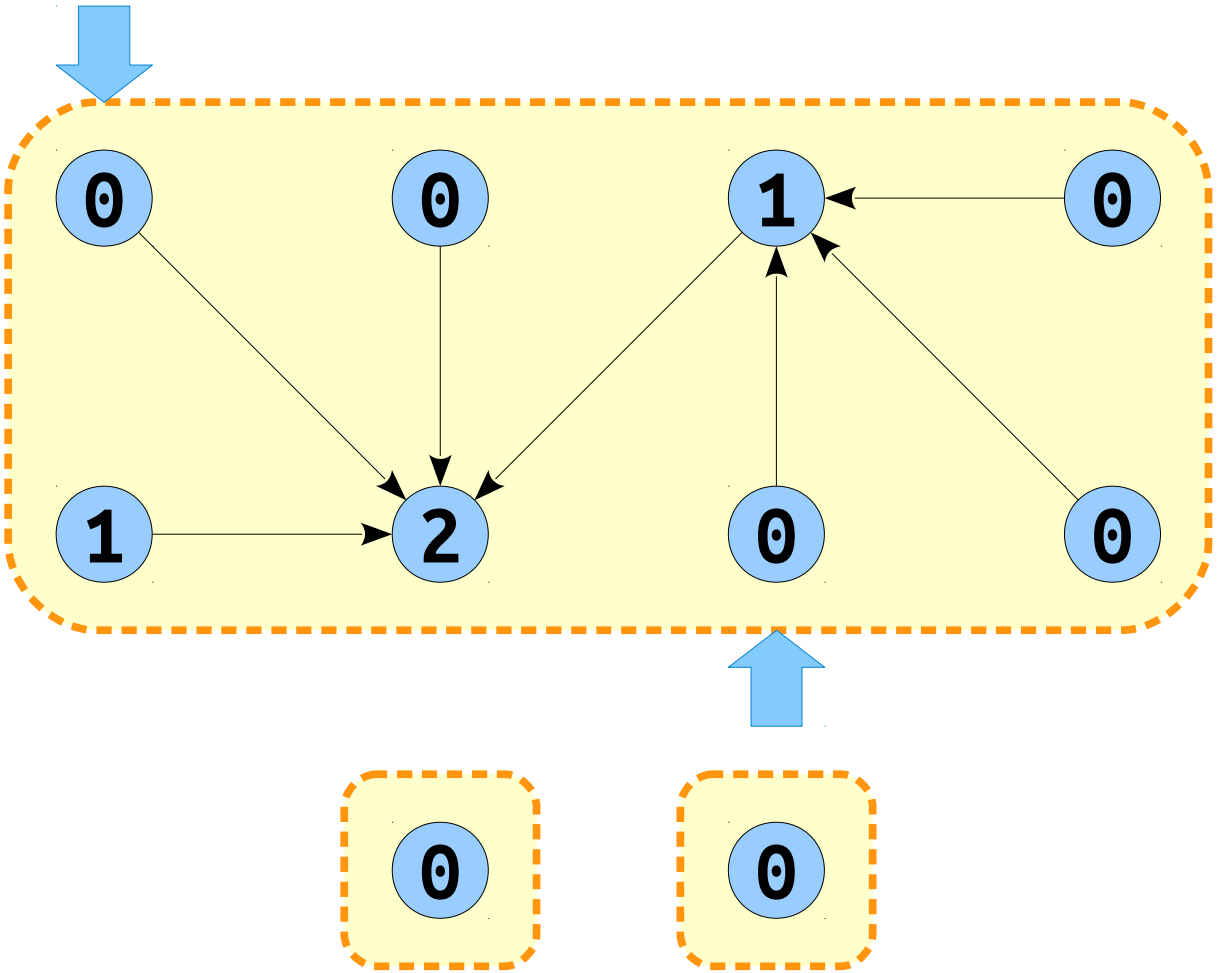
Path Compression



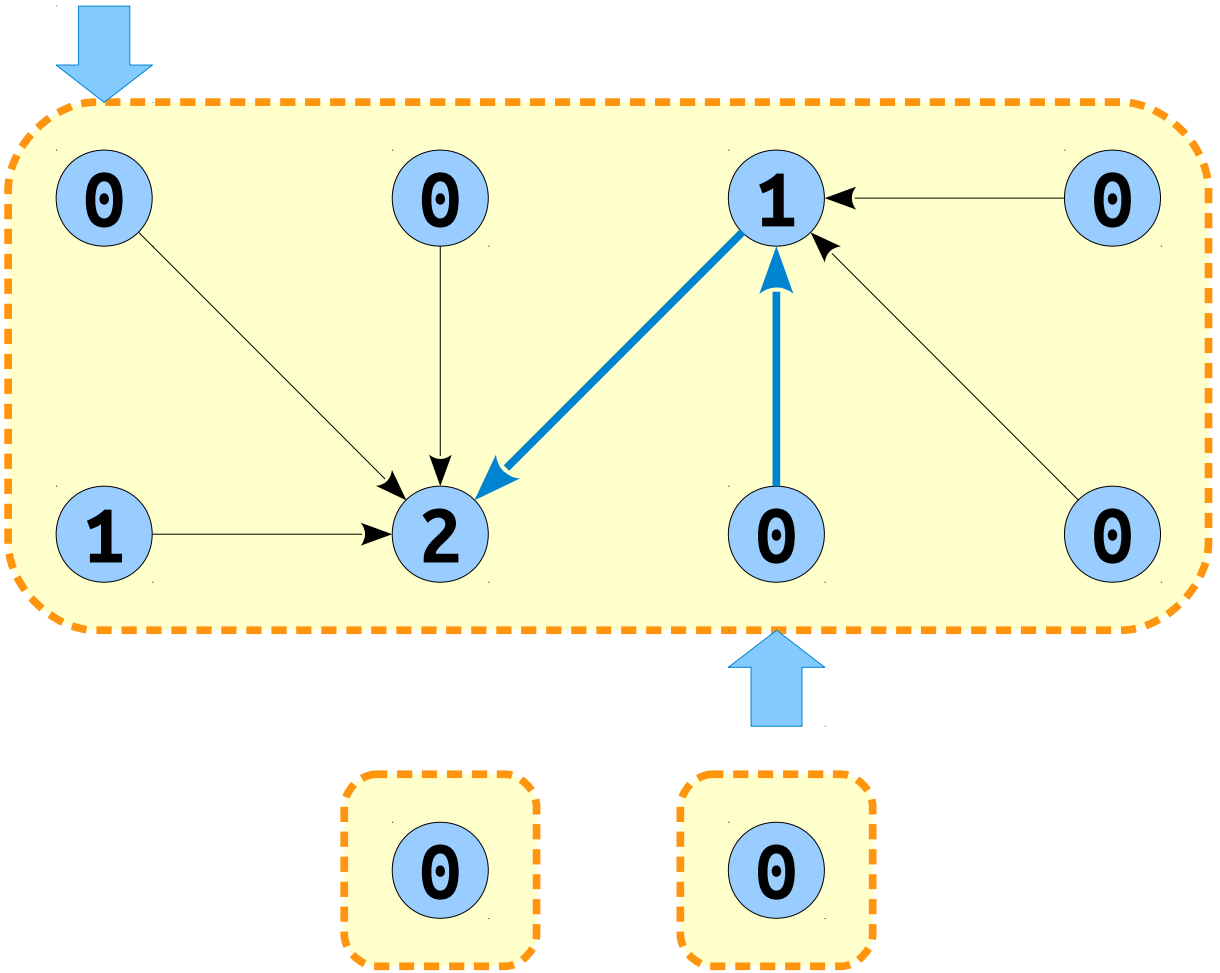
Path Compression



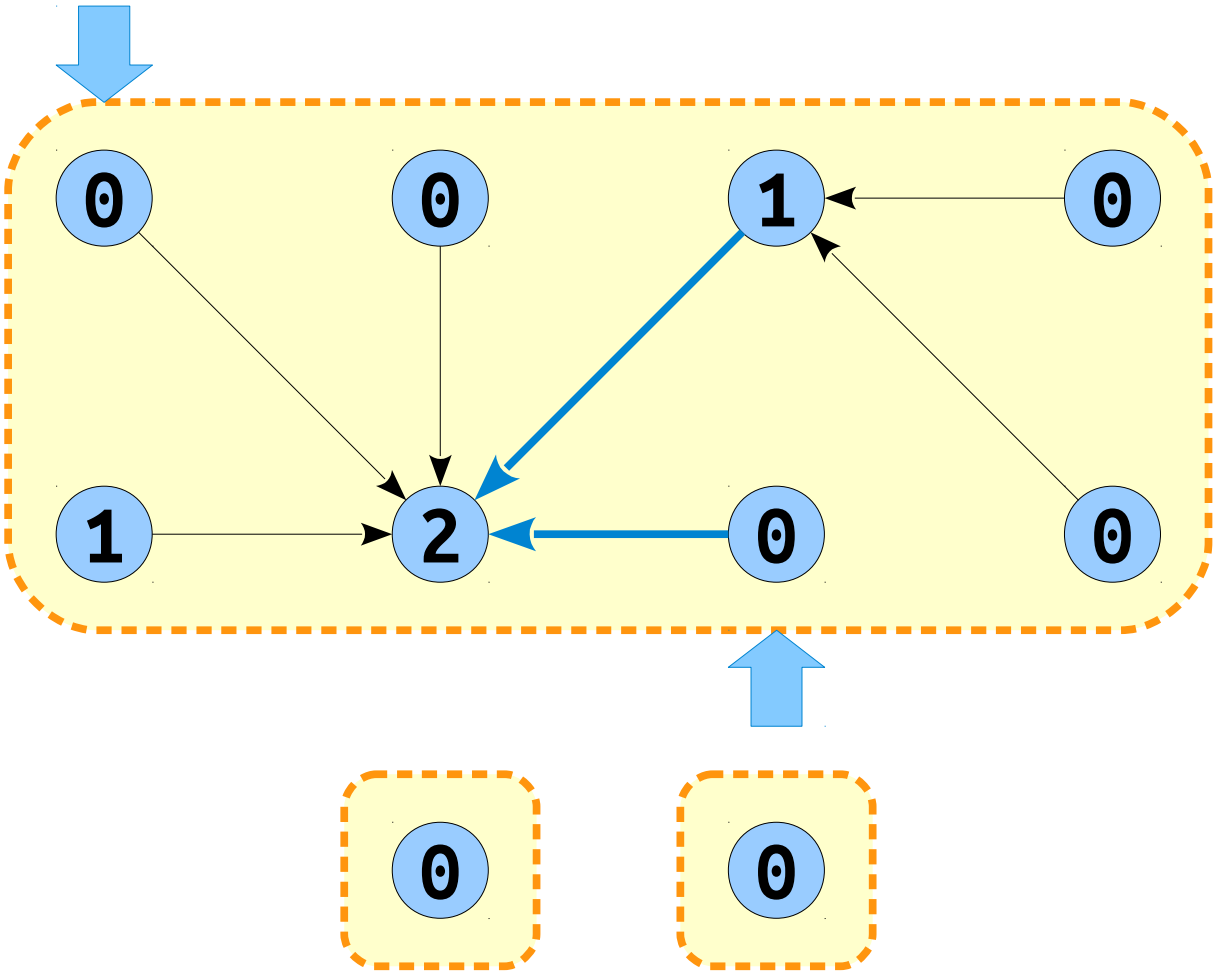
Path Compression



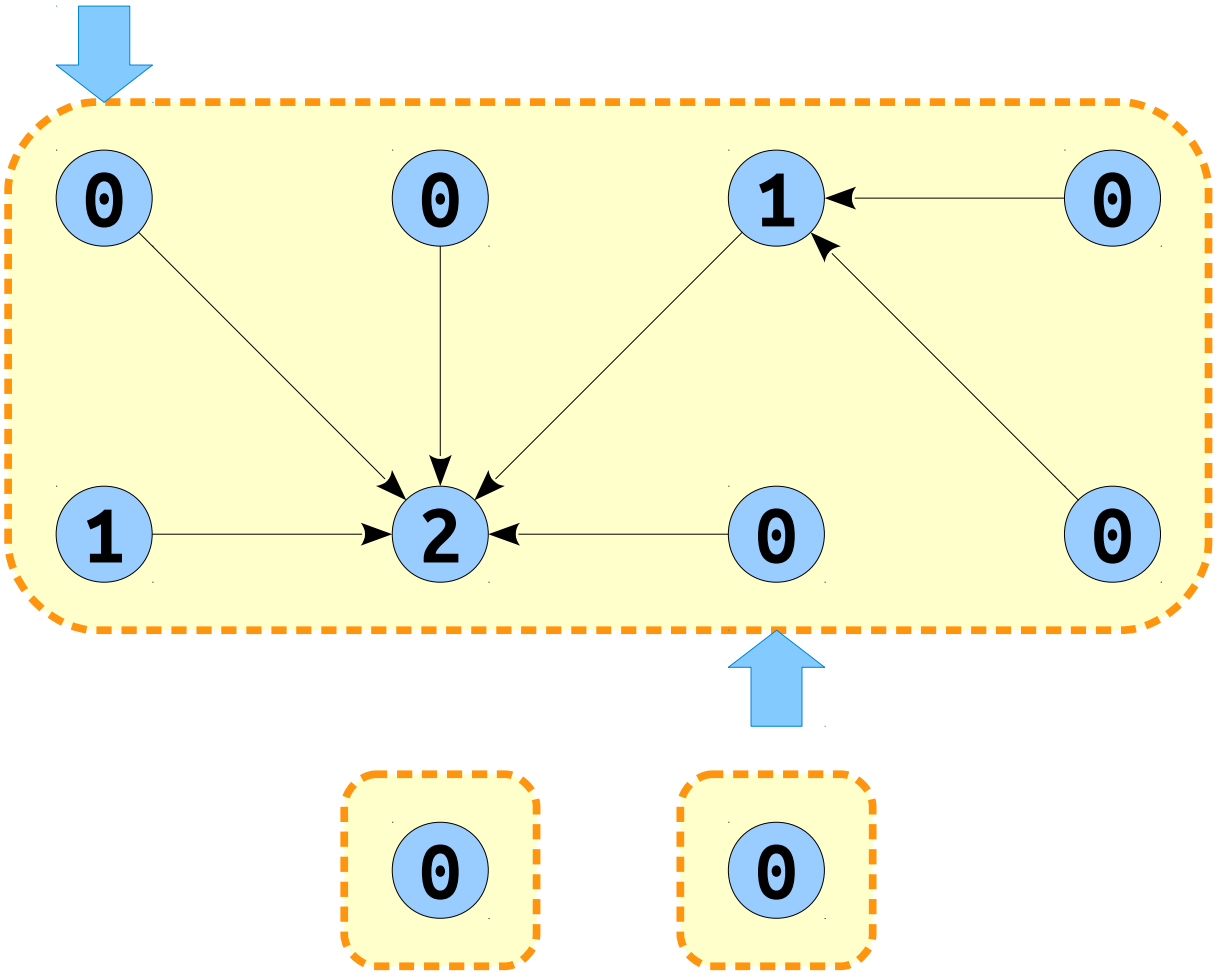
Path Compression



Path Compression

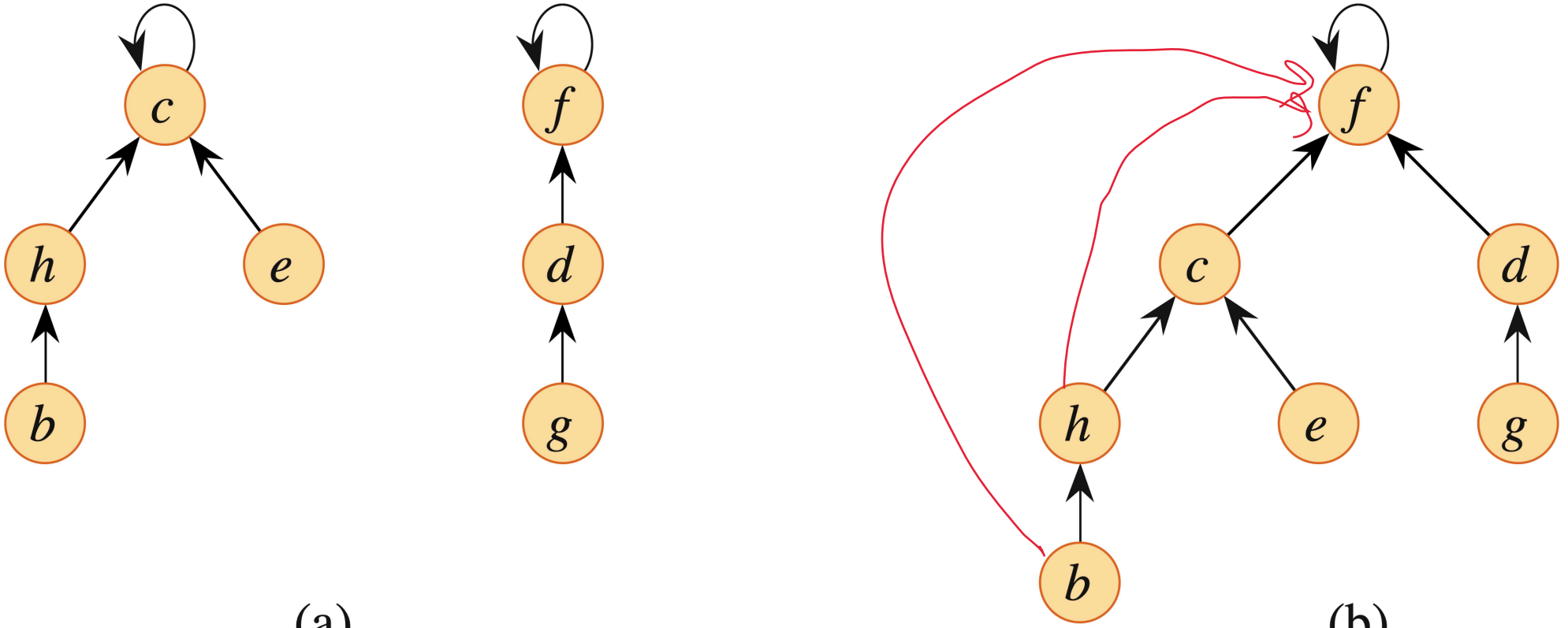


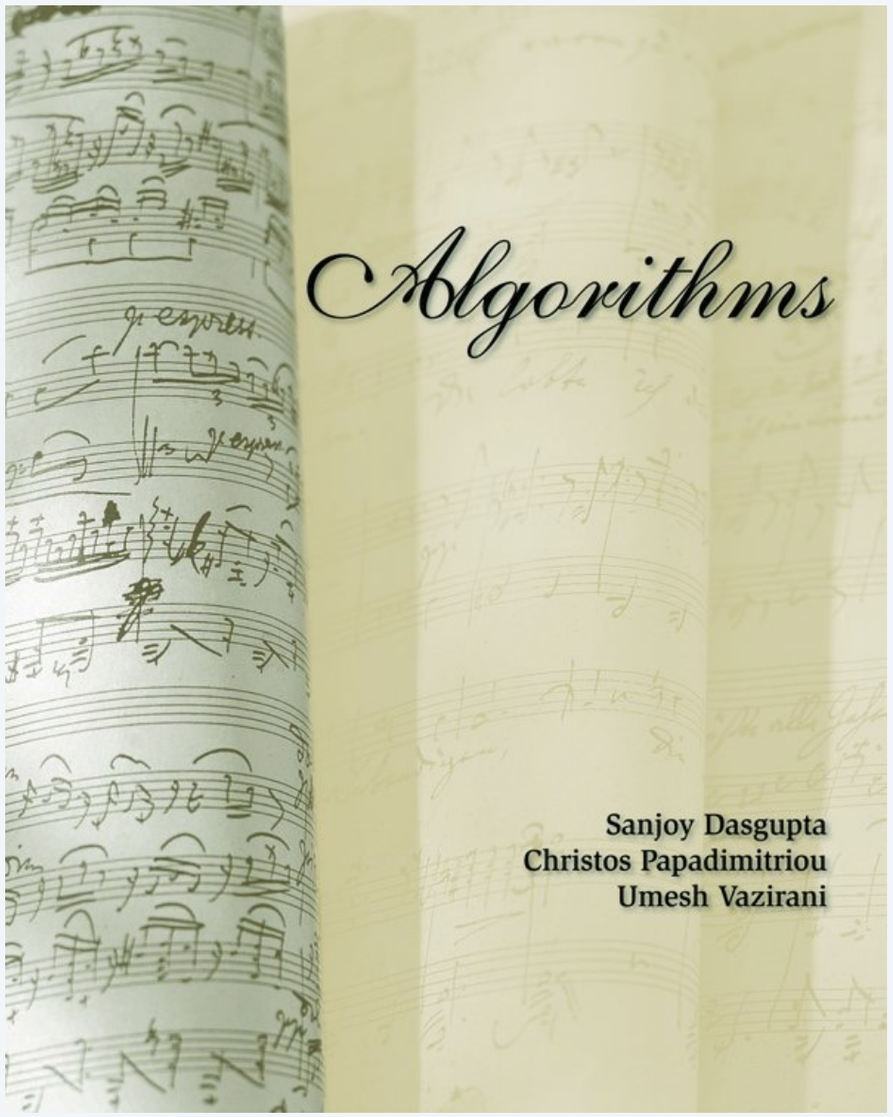
Path Compression



The Claim

- ***Claim:*** The runtime of performing m ***union*** and ***find*** operations on an n-node disjoint-set forest using path compression and union-by-rank is $O(n + m\alpha(n))$, where α is an *extremely* slowly-growing function.
- The original proof of this result (which is included in CLRS) is due to Tarjan and uses a complex amortized charging scheme.
- Today, we'll use an an aggregate analysis due to Seidel and Sharir based on a technique called ***forest-slicing***.


c
h
e
b
f
d
g
(a)
f
c
d
h
e
g
b
(b)



UNION–FIND

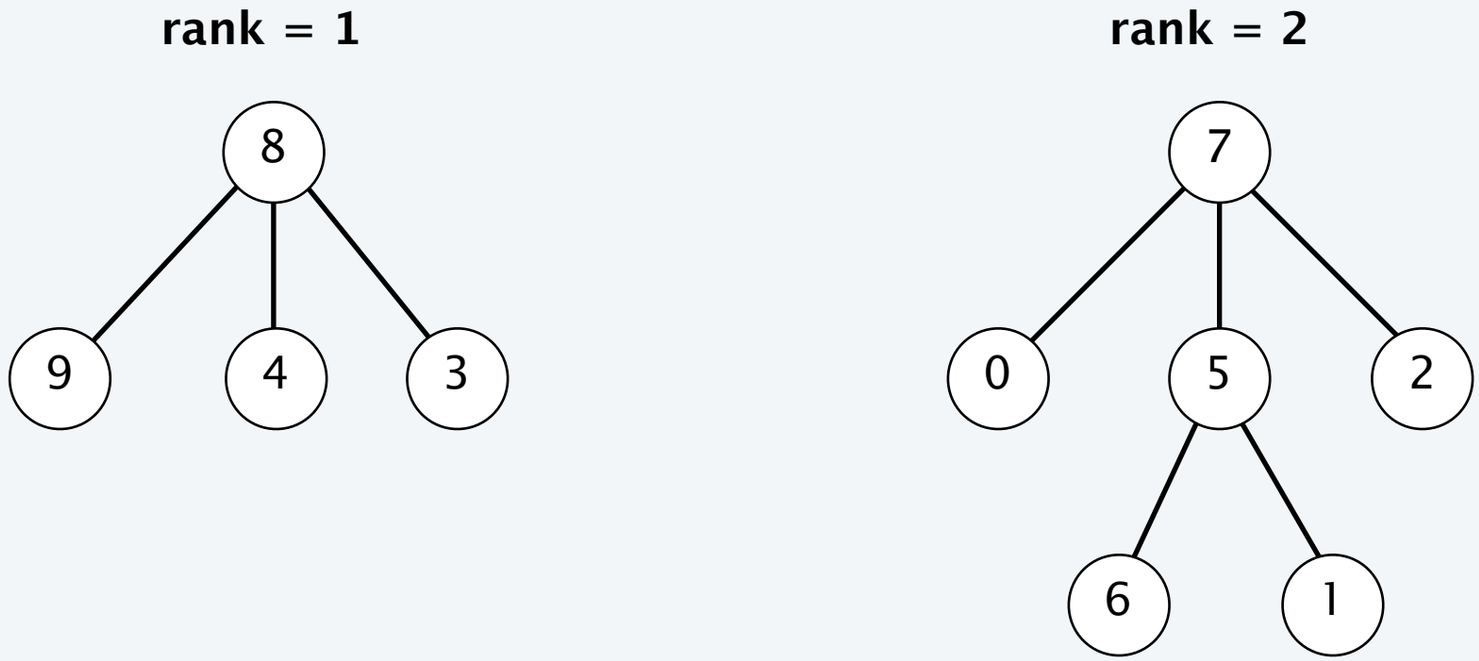
- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

SECTION 5.1.4

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.

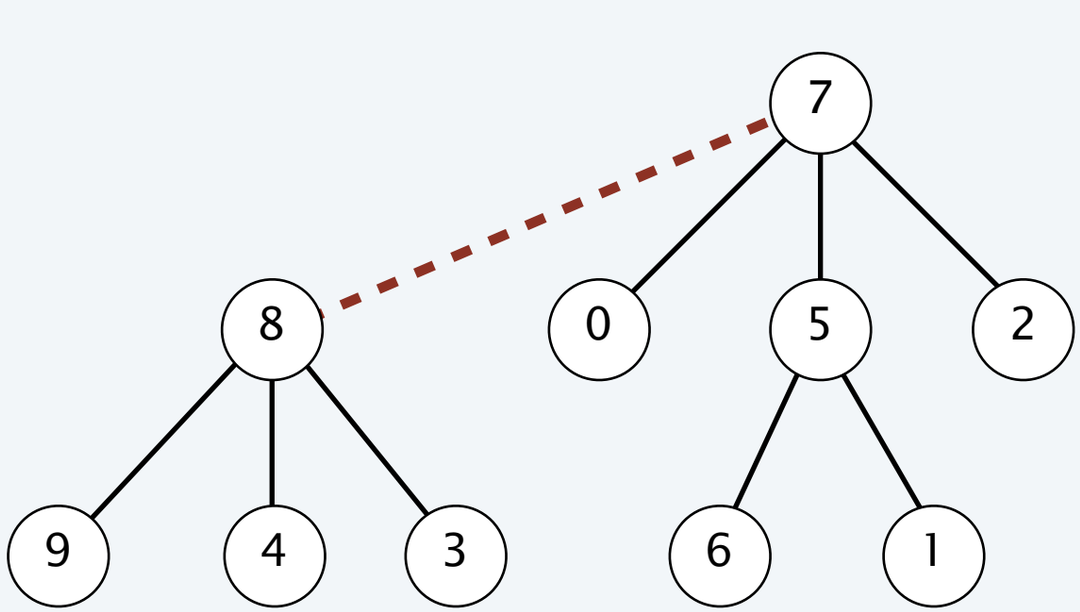
UNION(5, 3)



Note. For now, rank = height.

Link-by-rank

Link-by-rank. Maintain an integer rank for each node, initially 0. Link root of smaller rank to root of larger rank; if tie, increase rank of larger root by 1.



Note. For now, rank = height.

Link-by-rank: properties

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.

Pf. A node of rank k is created only by linking two roots of rank $k-1$. ▪

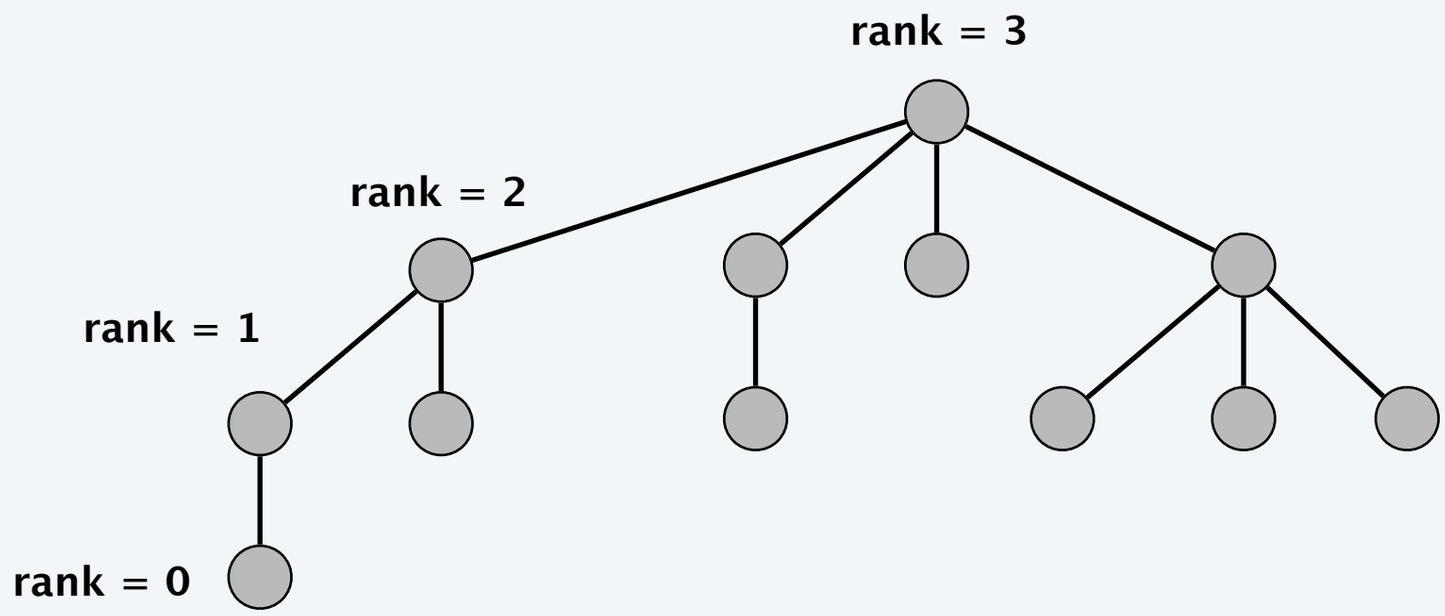
PROPERTY 2. If x is not a root node, then $rank[x]$ will never change again.

Pf. Rank changes only for roots; a nonroot never becomes a root. ▪

PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.

Pf. The parent can change only for a root, so before linking $parent[x] = x$.

After x is linked-by-rank to new root r we have $rank[r] > rank[x]$. ▪



Link-by-rank: properties

PROPERTY 4. Any root node of rank k has $\geq 2^k$ nodes in its tree.

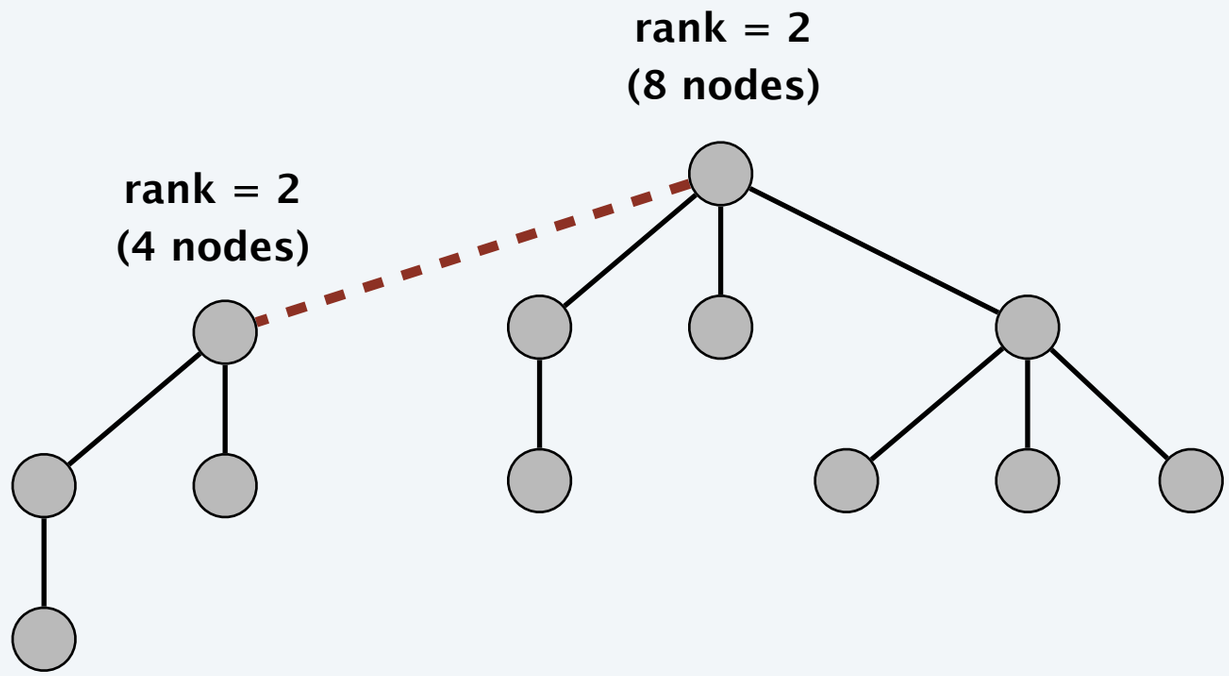
Pf. [by induction on k]

- Base case: true for $k = 0$.
- Inductive hypothesis: assume true for $k - 1$.
- A node of rank k is created only by linking two roots of rank $k - 1$.
- By inductive hypothesis, each subtree has $\geq 2^{k-1}$ nodes

  $\Rightarrow$ resulting tree has $\geq 2^k$ nodes. ▪

PROPERTY 5. The highest rank of a node is $\leq \lfloor \lg n \rfloor$.

Pf. Immediate from PROPERTY 1 and PROPERTY 4. ▪

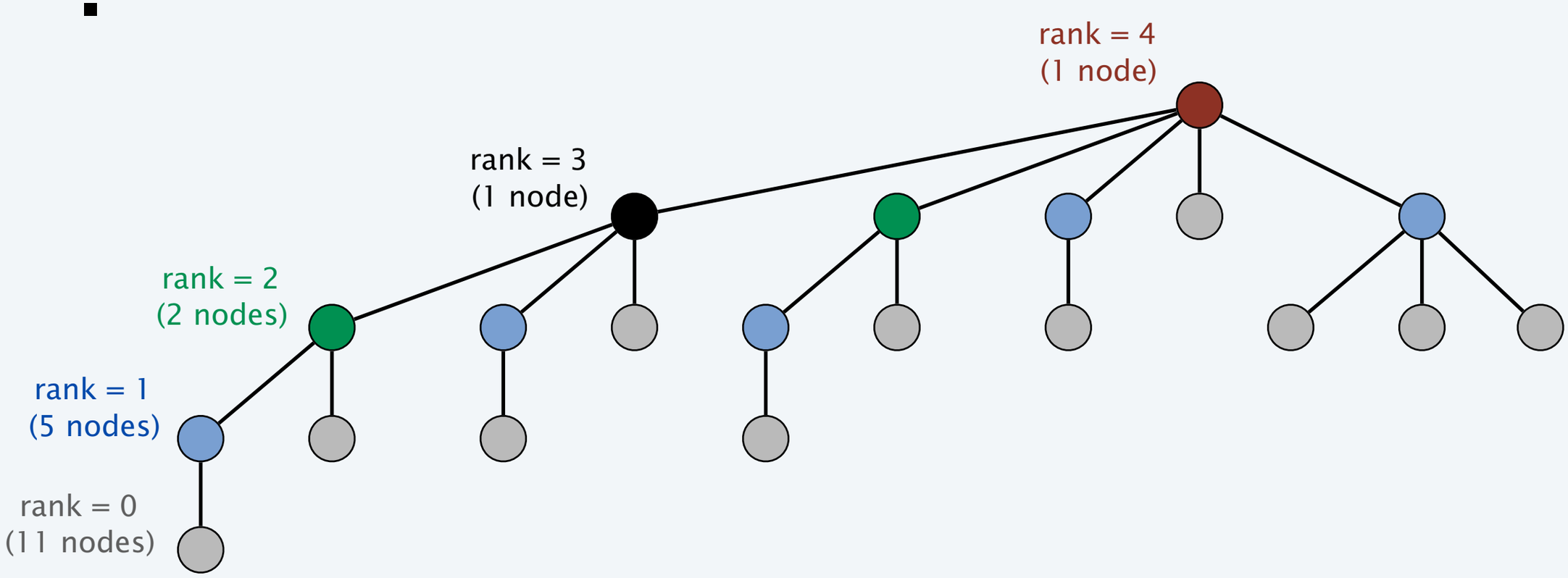


Link-by-rank: properties

PROPERTY 6. For any integer $k \geq 0$, there are $\leq n / 2^k$ nodes with rank k.

Pf.

- Any root node of rank k has $\geq 2^k$ descendants. [PROPERTY 4]
- Any nonroot node of rank k has $\geq 2^k$ descendants because:
  - it had this property just before it became a nonroot [PROPERTY 4]
  - its rank doesn't change once it became a nonroot [PROPERTY 2]
  - its set of descendants doesn't change once it became a nonroot
- Different nodes of rank k can't have common descendants. [PROPERTY 1]
  - ∎

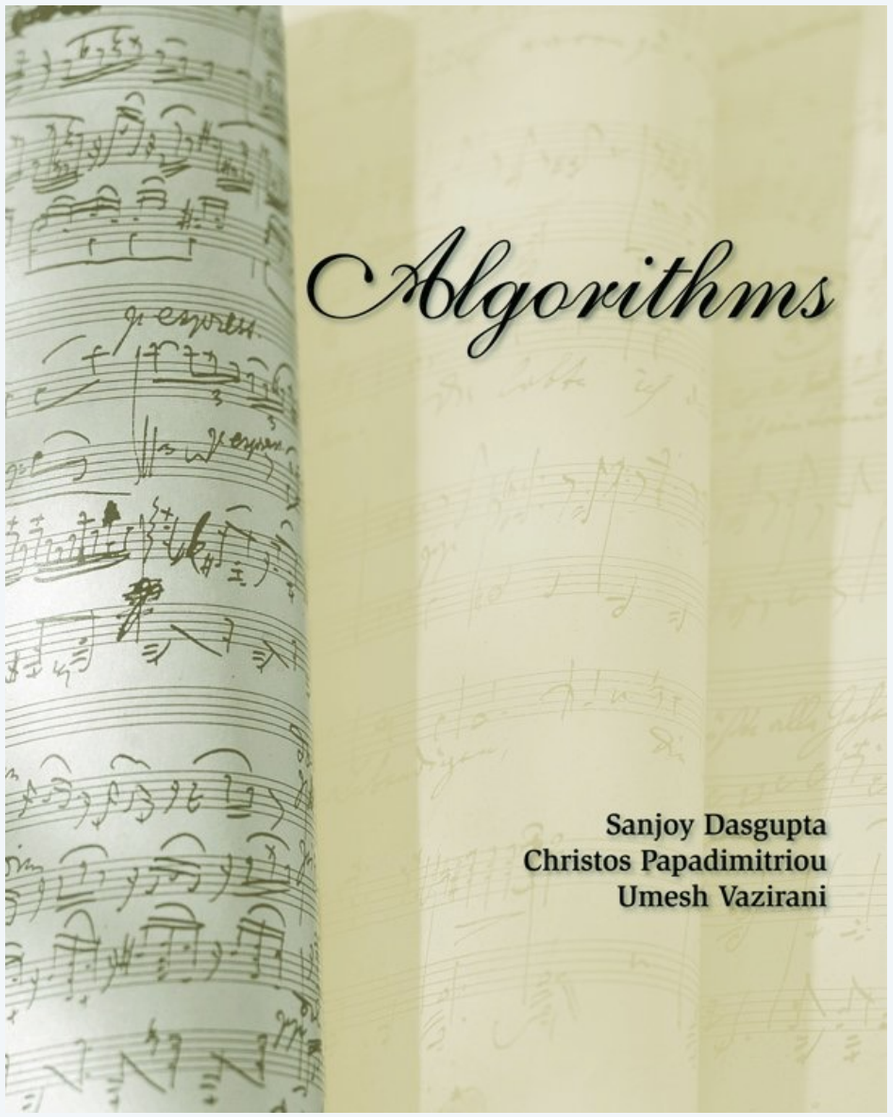


Link-by-rank: analysis

Theorem. Using link-by-rank, any UNION or FIND operation takes $O(\log n)$ time in the worst case, where n is the number of elements.

Pf.

- The running time of UNION and FIND is bounded by the tree height.
- By PROPERTY 5, the height is $\leq \lfloor \lg n \rfloor$. ▪



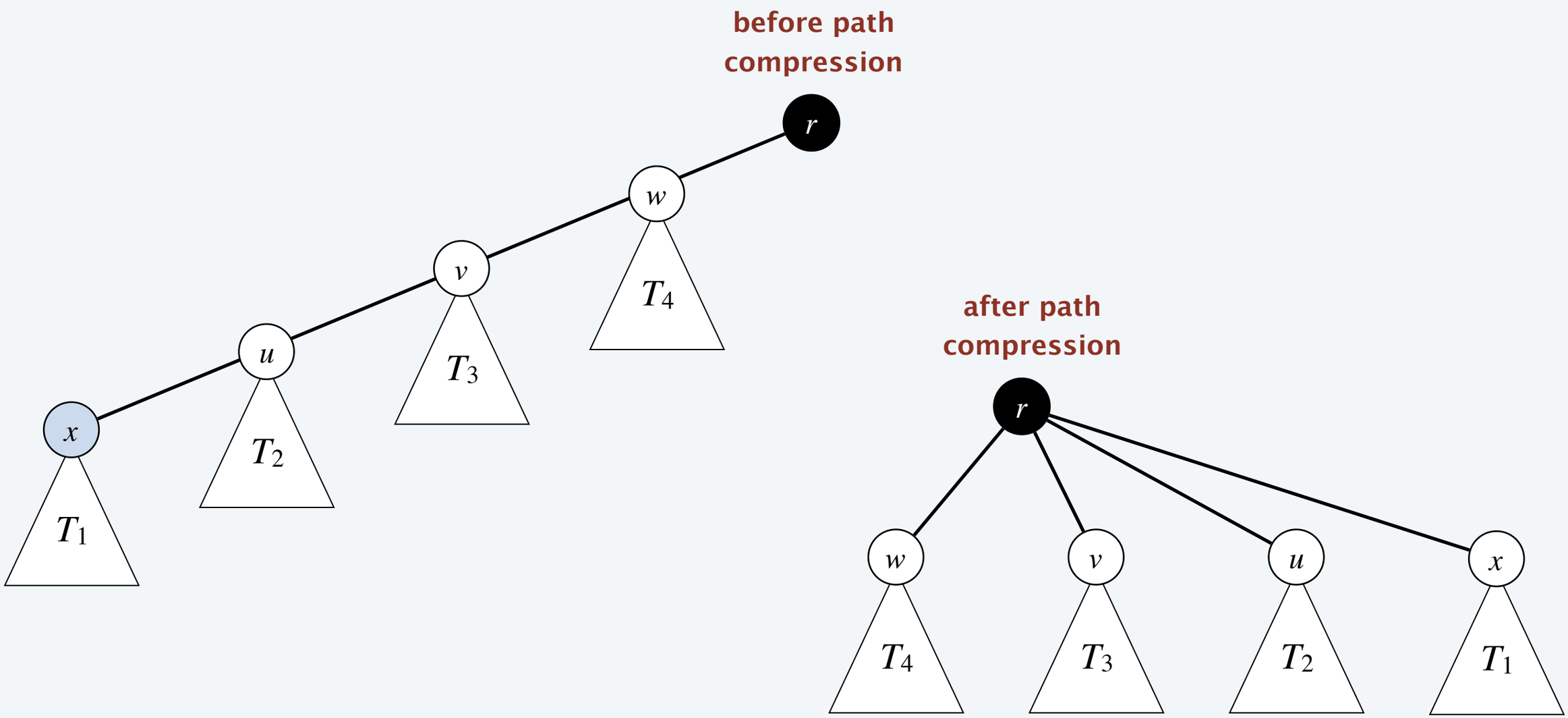
UNION–FIND

- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

SECTION 5.1.4

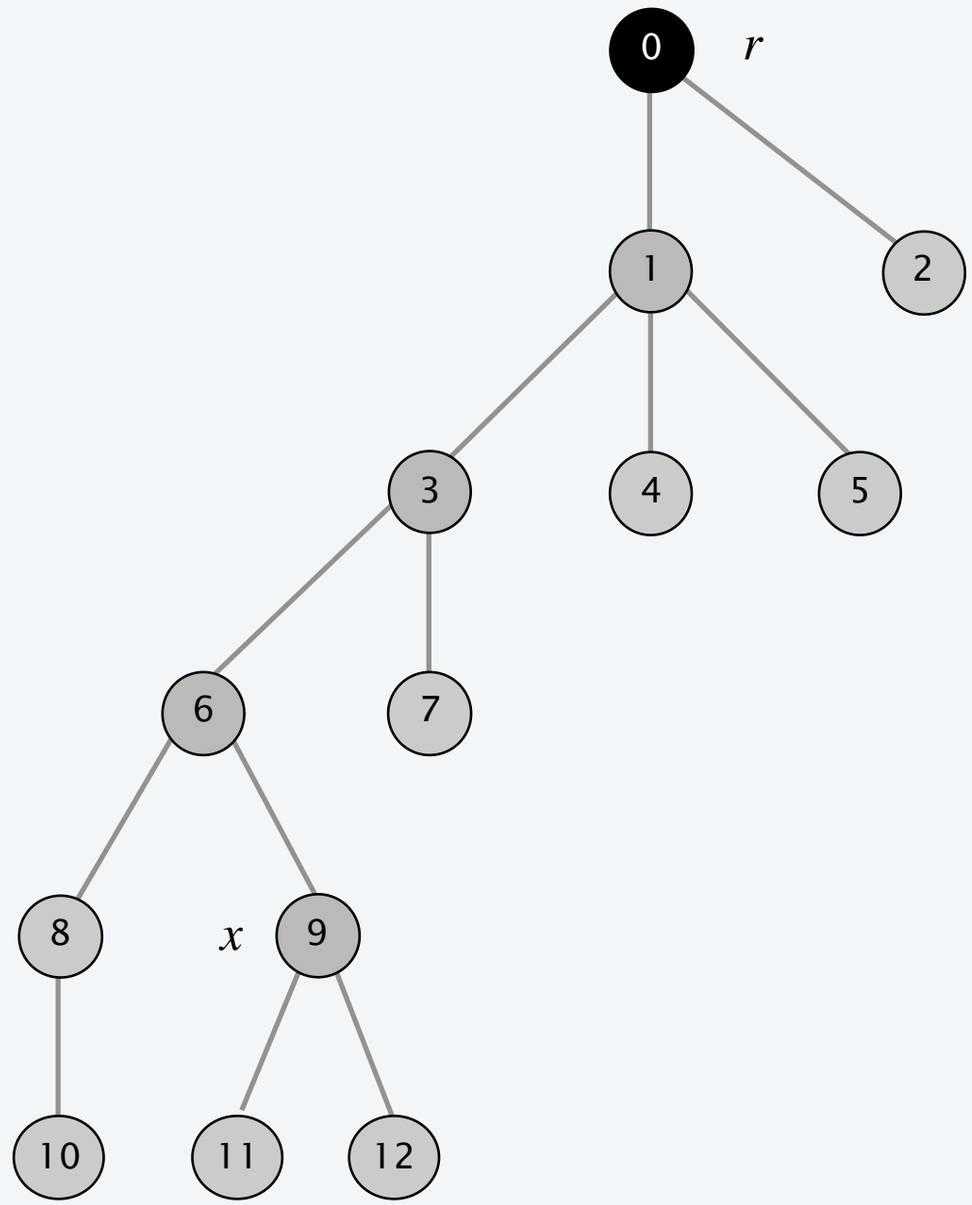
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



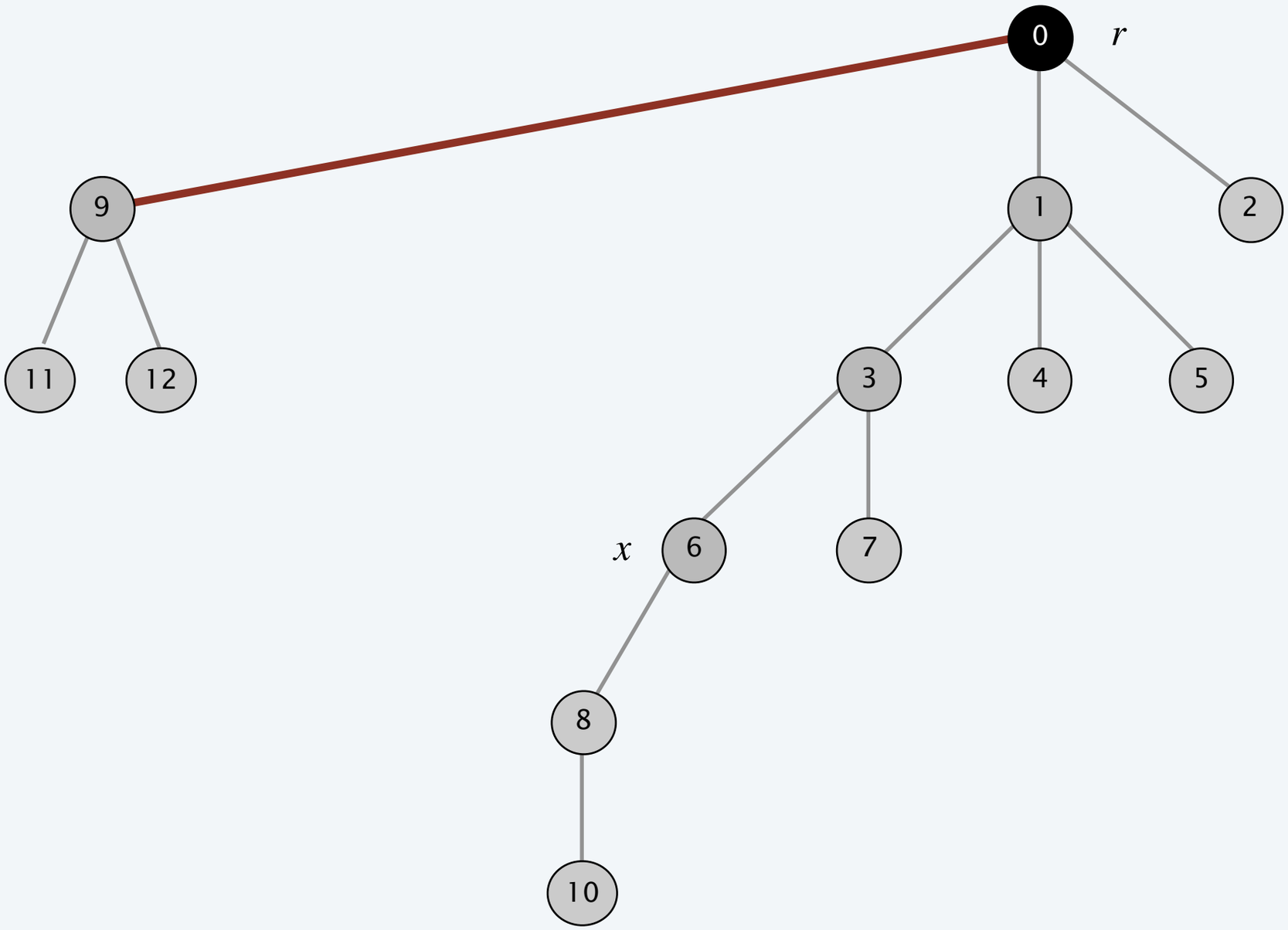
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



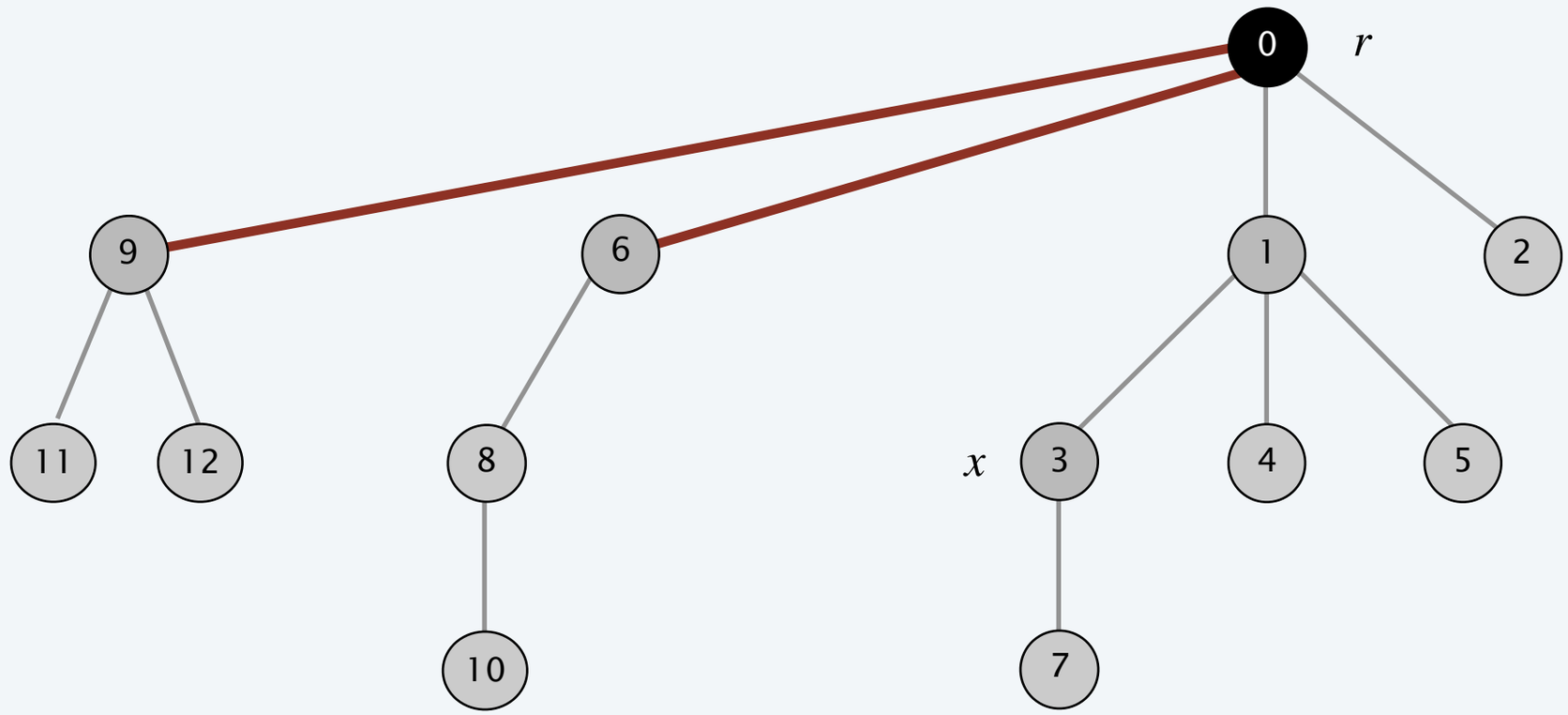
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



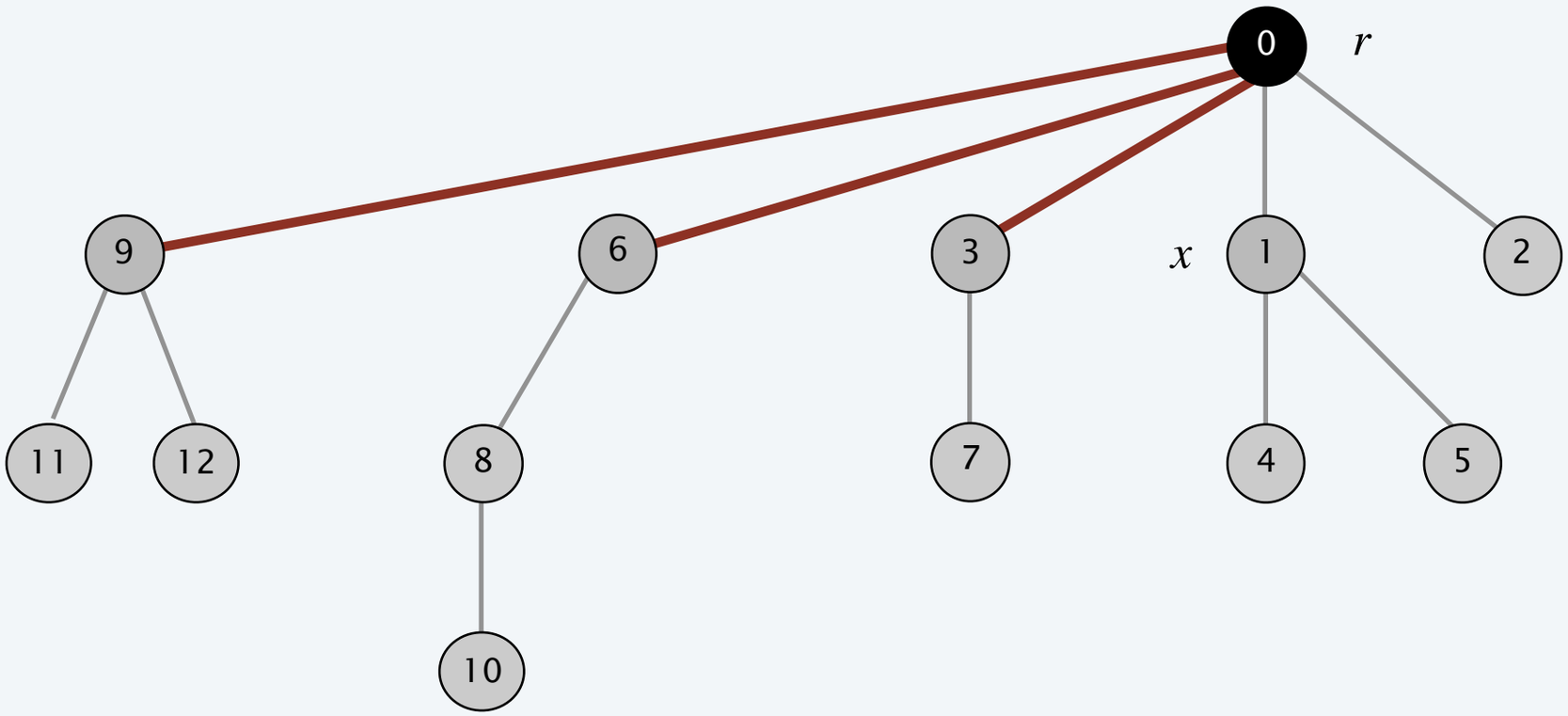
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



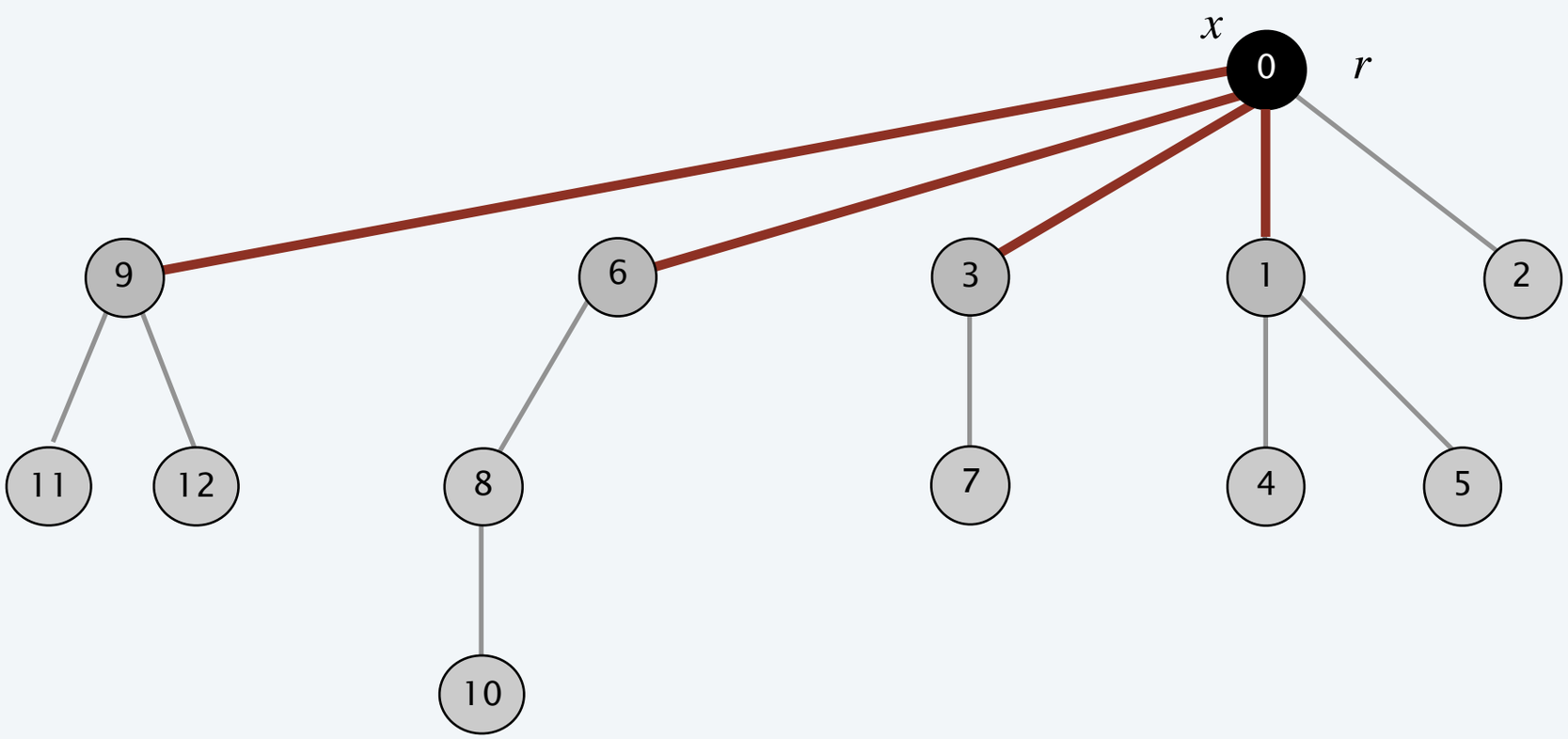
Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.



Path compression

Path compression. When finding the root r of the tree containing x, change the parent pointer of all nodes along the path to point directly to r.

FIND(x)

IF ($x \neq parent[x]$)

$parent[x] \leftarrow$ FIND($parent[x]$).

RETURN $parent[x]$.

this FIND implementation changes the tree structure (!)

Note. Path compression does not change the rank of a node; so $height(x) \leq rank[x]$ but they are not necessarily equal.

Path compression

Fact. Path compression with naïve linking can require $\Omega(n)$ time to perform a single UNION or FIND operation, where n is the number of elements.

Pf. The height of the tree is $n - 1$ after the sequence of union operations:

UNION$(1, 2)$, UNION$(2, 3)$, …, UNION$(n - 1, n)$. ▪

naïve linking: link root of first tree to root of second tree

Theorem. [Tarjan–van Leeuwen 1984] Starting from an empty data structure, path compression with naïve linking performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log n)$ time.

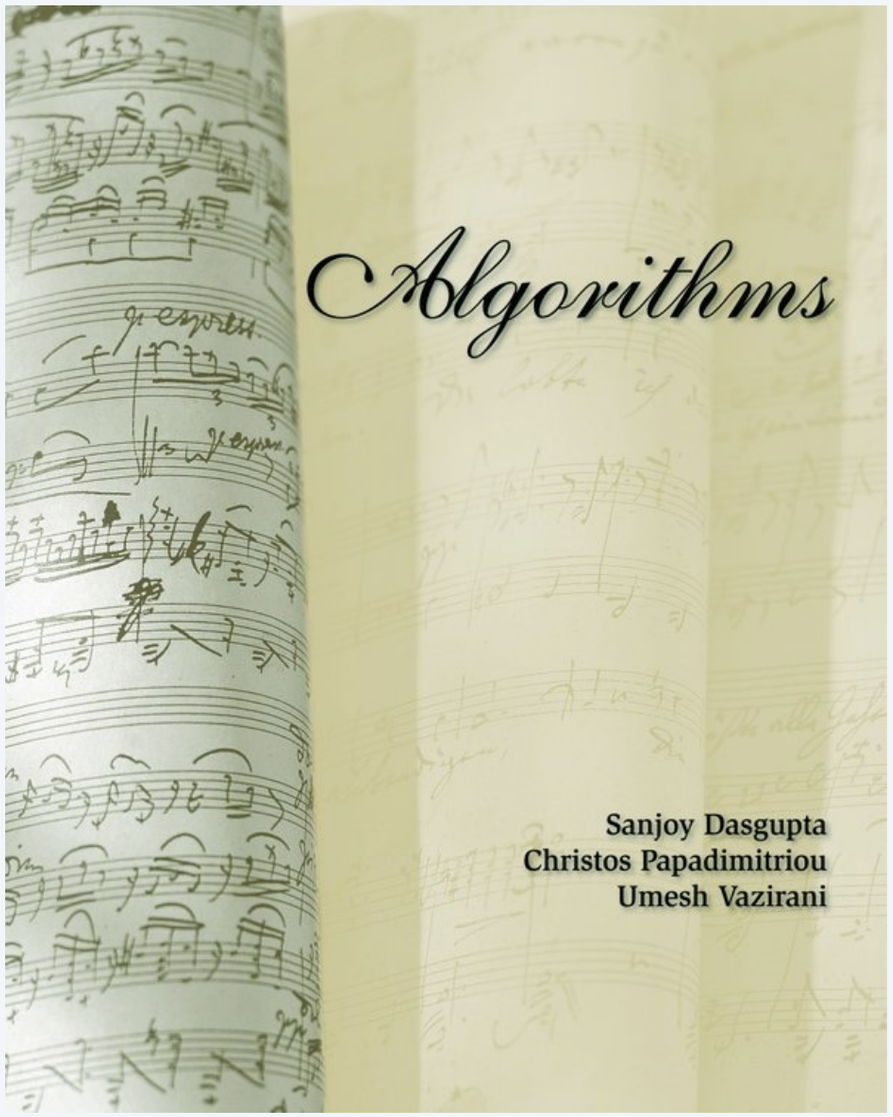
Pf. Nontrivial (but omitted).

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2        y.p = x
3  else x.p = y
4        if x.rank == y.rank
5              y.rank = y.rank + 1

FIND-SET(x)
1  if x ≠ x.p                          // not the root?
2        x.p = FIND-SET(x.p)           // the root becomes the parent
3  return x.p                          // return the root
```



UNION–FIND

- *naïve linking*
- *link-by-size*
- *link-by-rank*
- *path compression*
- *link-by-rank with path compression*
- *context*

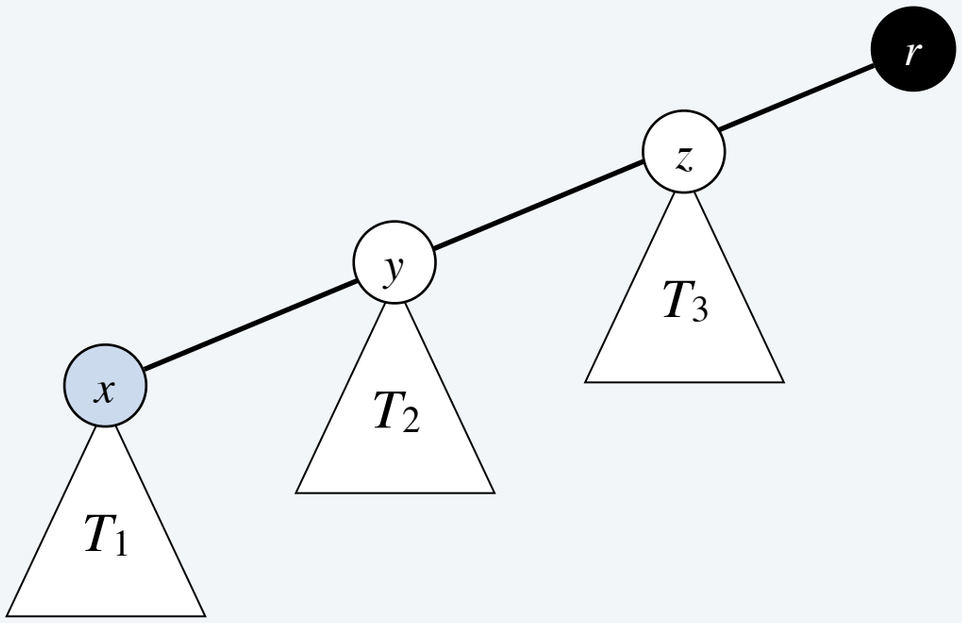
SECTION 5.1.4

Link-by-rank with path compression: properties

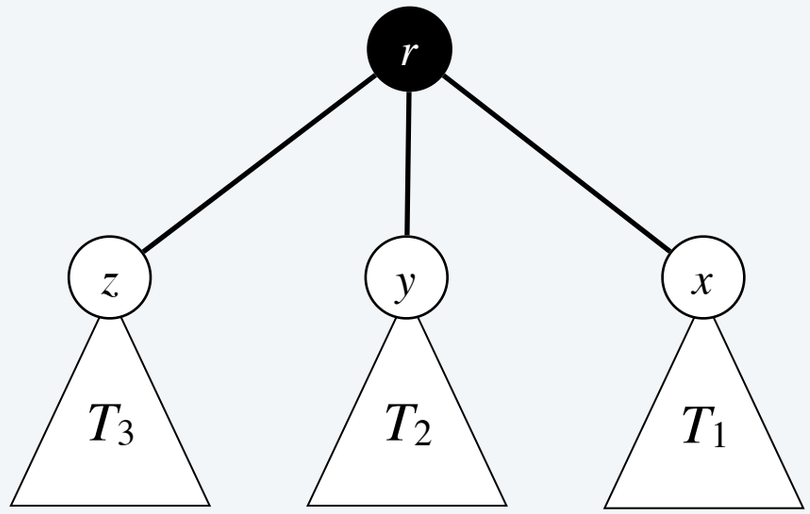
PROPERTY. The tree roots, node ranks, and elements within a tree are the same with or without path compression.

Pf. Path compression does not create new roots, change ranks, or move elements from one tree to another. ▪

before path compression



after path compression



Link-by-rank with path compression: properties

PROPERTY. The tree roots, node ranks, and elements within a tree are the same with or without path compression.

COROLLARY. PROPERTY 2, 4–6 hold for link-by-rank with path compression.

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.
PROPERTY 2. If x is not a root node, then $rank[x]$ will never change again.
PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.
PROPERTY 4. Any root node of rank k has $\geq 2^k$ nodes in its tree.
PROPERTY 5. The highest rank of a node is $\leq \lfloor \lg n \rfloor$.
PROPERTY 6. For any integer $k \geq 0$, there are $\leq n / 2^k$ nodes with rank k.

Bottom line. PROPERTY 1–6 hold for link-by-rank with path compression.
(but we need to recheck PROPERTY 1 and PROPERTY 3)

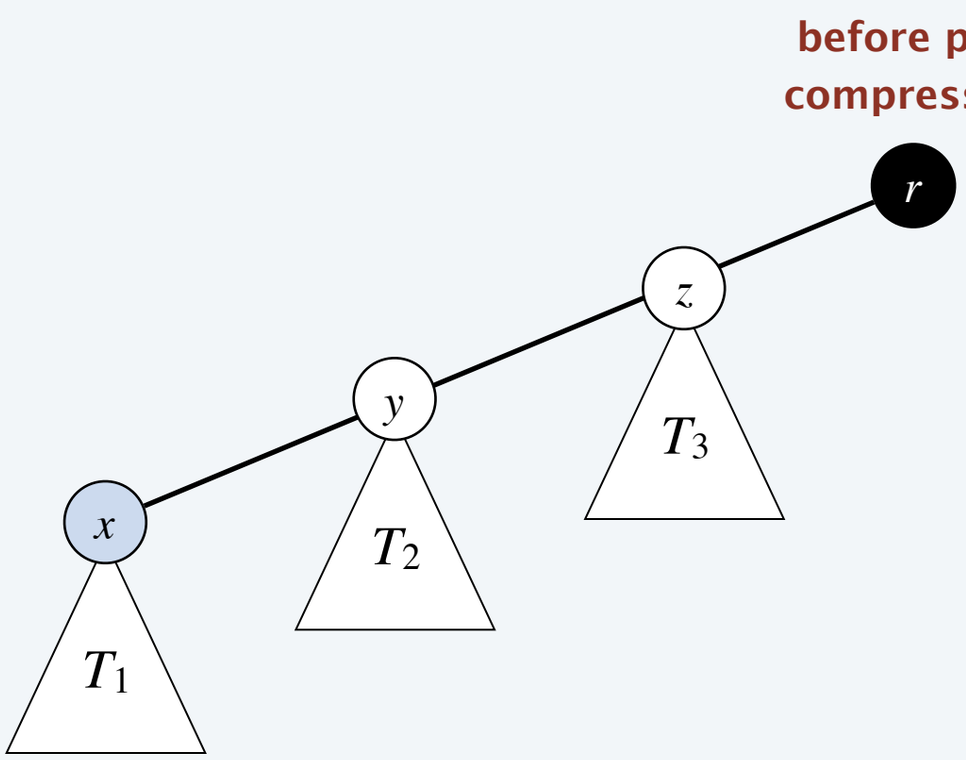
Link-by-rank with path compression: properties

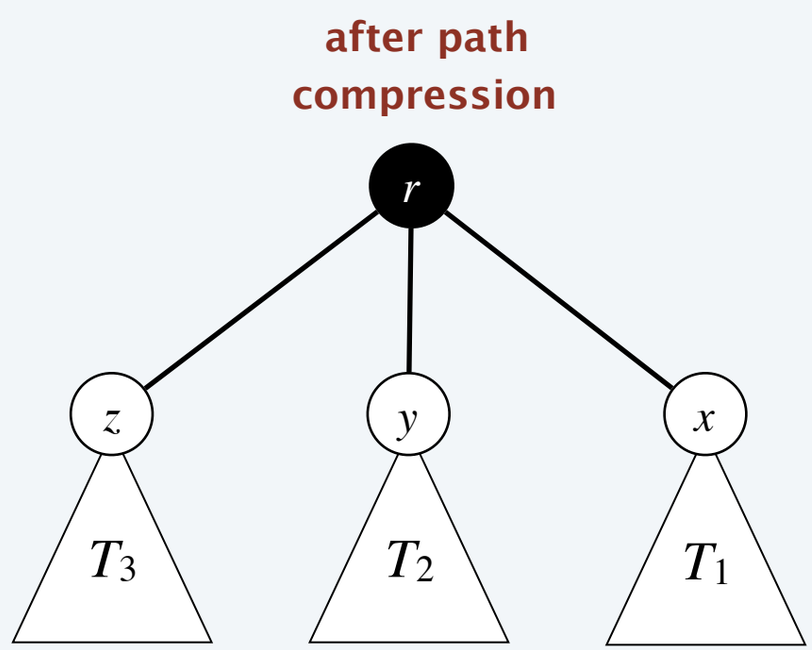
PROPERTY 3. If $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.

Pf. Path compression can make x point to only an ancestor of $parent[x]$.

PROPERTY 1. If x is not a root node, then $rank[x] < rank[parent[x]]$.

Pf. Path compression doesn't change any ranks, but it can change parents. If $parent[x]$ doesn't change during a path compression, the inequality continues to hold; if $parent[x]$ changes, then $rank[parent[x]]$ strictly increases.





Iterated logarithm function

Def. The iterated logarithm function is defined by:

$$\lg^* n = \begin{cases} 0 & \text{if } n \leq 1 \\ 1 + \lg^*(\lg n) & \text{otherwise} \end{cases}$$

n	$\lg^* n$
1	0
2	1
$[3, 4]$	2
$[5, 16]$	3
$[17, 65536]$	4
$[65537, 2^{65536}]$	5

iterated lg function

Note. We have $\lg^* n \leq 5$ unless n exceeds the # atoms in the universe.

Analysis

Divide nonzero ranks into the following groups:

- $\{ 1 \}$
- $\{ 2 \}$
- $\{ 3, 4 \}$
- $\{ 5, 6, \ldots, 16 \}$
- $\{ 17, 18, \ldots, 2^{16} \}$
- $\{ 65537, 65538, \ldots, 2^{65536} \}$
- ...

Property 7. Every nonzero rank falls within one of the first $\lg^* n$ groups.

Pf. The rank is between 0 and $\lfloor \lg n \rfloor$. [PROPERTY 5]

Creative accounting

Credits. A node receives credits as soon as it ceases to be a root.

If its rank is in the interval $\{ k+1, k+2, \ldots, 2^k \}$, we give it 2^k credits.

group k

Proposition. Number of credits disbursed to all nodes is $\leq n \lg^* n$.

Pf.

- All nodes in group k have rank $\geq k+1$.
- By PROPERTY 6, the number of nodes with rank $\geq k+1$ is at most

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \cdots \leq \frac{n}{2^k}$$

- Thus, nodes in group k need at most n credits in total.
- There are $\leq \lg^* n$ groups. [PROPERTY 7] ▪

Running time of FIND

Running time of FIND. Bounded by number of parent pointers followed.

- Recall: the rank strictly increases as you go up a tree. [PROPERTY 1]
- Case 0: $parent[x]$ is a root $\Rightarrow$ only happens for one link per FIND.
- Case 1: $rank[parent[x]]$ is in a higher group than $rank[x]$.
- Case 2: $rank[parent[x]]$ is in the same group as $rank[x]$.

Case 1. At most $\lg^* n$ nodes on path can be in a higher group. [PROPERTY 7]

Case 2. These nodes are charged 1 credit to follow parent pointer.

- Each time x pays 1 credit, $rank[parent[x]]$ strictly increases. [PROPERTY 1]
- Therefore, if $rank[x]$ is in the group $\{ k+1, \ldots, 2^k \}$, the rank of its parent will be in a higher group before x pays 2^k credits.
- Once $rank[parent[x]]$ is in a higher group than $rank[x]$, it remains so because:
  - $rank[x]$ does not change once it ceases to be a root. [PROPERTY 2]
  - $rank[parent[x]]$ does not decrease. [PROPERTY 3]
  - thus, x has enough credits to pay until it becomes a Case 1 node. ▪

Link-by-rank with path compression

Theorem. Starting from an empty data structure, link-by-rank with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log^* n)$ time.

Link-by-size with path compression

Theorem. [Hopcroft–Ullman 1973] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m \log^* n)$ time.

SIAM J. COMPUT.
Vol. 2, No. 4, December 1973

SET MERGING ALGORITHMS*

J. E. HOPCROFT† AND J. D. ULLMAN‡

Abstract. This paper considers the problem of merging sets formed from a total of n items in such a way that at any time, the name of a set containing a given item can be ascertained. Two algorithms using different data structures are discussed. The execution times of both algorithms are bounded by a constant times $nG(n)$, where $G(n)$ is a function whose asymptotic growth rate is less than that of any finite number of logarithms of n.

Key words. algorithm, algorithmic analysis, computational complexity, data structure, equivalence algorithm, merging, property grammar, set, spanning tree

Link-by-size with path compression

Theorem. [Tarjan 1975] Starting from an empty data structure, link-by-size with path compression performs any intermixed sequence of $m \geq n$ MAKE-SET, UNION, and FIND operations on a set of n elements in $O(m\,\alpha(m, n))$ time, where $\alpha(m, n)$ is a functional inverse of the Ackermann function.

Efficiency of a Good But Not Linear Set Union Algorithm

ROBERT ENDRE TARJAN

University of California, Berkeley, California

ABSTRACT. Two types of instructions for manipulating a family of disjoint sets which partition a universe of n elements are considered $FIND(x)$ computes the name of the (unique) set containing element x $UNION(A, B, C)$ combines sets A and B into a new set named C. A known algorithm for implementing sequences of these instructions is examined It is shown that, if $t(m, n)$ is the maximum time required by a sequence of $m \geq n$ *FIND*s and $n - 1$ intermixed *UNION*s, then $k_1 m\alpha(m, n) \leq t(m, n) \leq k_2 m\alpha(m, n)$ for some positive constants k_1 and k_2, where $\alpha(m, n)$ is related to a functional inverse of Ackermann's function and is *very* slow-growing.

Inverse Ackermann function

Definition.

$$\alpha_k(n) = \begin{cases} \lceil n/2 \rceil & \text{if } k = 1 \\ 0 & \text{if } n = 1 \text{ and } k \geq 2 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases}$$

Ex.

- $\alpha_1(n) = \lceil n / 2 \rceil$.
- $\alpha_2(n) = \lceil \lg n \rceil$ = # of times we divide n by 2, until we reach 1.
- $\alpha_3(n) = \lg^* n$ = # of times we apply the lg function to n, until we reach 1.
- $\alpha_4(n)$ = # of times we apply the iterated lg function to n, until we reach 1.

$$2 \uparrow 65536 = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}} \quad \text{(65536 times)}$$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...	2^{16}	...	2^{65536}	...	$2 \uparrow 65536$
$\alpha_1(n)$	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	...	2^{15}	...	2^{65535}	...	*huge*
$\alpha_2(n)$	0	1	2	2	3	3	3	3	4	4	4	4	4	4	4	4	...	16	...	65536	...	$2 \uparrow 65535$
$\alpha_3(n)$	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	...	4	...	5	...	65536
$\alpha_4(n)$	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	...	3	...	3	...	4

Inverse Ackermann function

Definition.

$$\alpha_k(n) = \begin{cases} \lceil n/2 \rceil & \text{if } k = 1 \\ 0 & \text{if } n = 1 \text{ and } k \geq 2 \\ 1 + \alpha_k(\alpha_{k-1}(n)) & \text{otherwise} \end{cases}$$

Property. For every $n \geq 5$, the sequence $\alpha_1(n), \alpha_2(n), \alpha_3(n), \ldots$ converges to 3.

Ex. $[n = 9876!]$ $\alpha_1(n) \geq 10^{35163}$, $\alpha_2(n) = 116812$, $\alpha_3(n) = 6$, $\alpha_4(n) = 4$, $\alpha_5(n) = 3$.

One-parameter inverse Ackermann. $\alpha(n) = \min \{ k : \alpha_k(n) \leq 3 \}$.

Ex. $\alpha(9876!) = 5$.

Two-parameter inverse Ackermann. $\alpha(m, n) = \min \{ k : \alpha_k(n) \leq 3 + m / n \}$.

$$
\begin{aligned}
A_3(1) &= A_2^{(2)}(1) \\
&= A_2(A_2(1)) \\
&= A_2(7) \\
&= 2^8 \cdot 8 - 1 \\
&= 2^{11} - 1 \\
&= 2047
\end{aligned}
$$

$$
\begin{aligned}
A_4(1) &= A_3^{(2)}(1) \\
&= A_3(A_3(1)) \\
&= A_3(2047) \\
&= A_2^{(2048)}(2047) \\
&\gg A_2(2047) \\
&= 2^{2048} \cdot 2048 - 1 \\
&= 2^{2059} - 1 \\
&> 2^{2056} \\
&= (2^4)^{514} \\
&= 16^{514} \\
&\gg 10^{80} ,
\end{aligned}
$$

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2\,, \\ 1 & \text{for } n = 3\,, \\ 2 & \text{for } 4 \leq n \leq 7\,, \\ 3 & \text{for } 8 \leq n \leq 2047\,, \\ 4 & \text{for } 2048 \leq n \leq A_4(1)\,. \end{cases}$$

INFORMATION ONLY

Lemma 19.2

For any integer $j \geq 1$, we have $A_1(j) = 2j + 1$.

Lemma 19.3

For any integer $j \geq 1$, we have $A_2(j) = 2^{j+1}(j + 1) - 1$.

Lemma 19.4

For all nodes x, we have $x.rank \leq x.p.rank$, with strict inequality if $x \neq x.p$ (x is not a root). The value of $x.rank$ is initially 0, increases through time until $x \neq x.p$, and from then on, $x.rank$ does not change. The value of $x.p.rank$ monotonically increases over time.

Corollary 19.5

On the simple path from any node going up toward a root, node ranks strictly increase. ■

Lemma 19.6

Every node has rank at most $n - 1$.

Lemma 19.7

Suppose that we convert a sequence S' of m' MAKE-SET, UNION, and FIND-SET operations into a sequence S of m MAKE-SET, LINK, and FIND-SET operations by turning each UNION into two FIND-SET operations followed by one LINK. Then, if sequence S runs in $O(m\,\alpha(n))$ time, sequence S' runs in $O(m'\,\alpha(n))$ time.

INFORMATION ONLY

$$\text{level}(x) = \max\{k : x.p.rank \geq A_k(x.rank)\} \ . \tag{19.3}$$

$$0 \leq \text{level}(x) < \alpha(n) \ , \tag{19.4}$$

$$\text{iter}(x) = \max\left\{i : x.p.rank \geq A^{(i)}_{\text{level}(x)}(x.rank)\right\} \ . \tag{19.5}$$

$$1 \leq \text{iter}(x) \leq x.rank \ , \tag{19.6}$$

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot x.rank & \text{if } x \text{ is a root or } x.rank = 0 \ , \\ (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) & \\ & \text{if } x \text{ is not a root and } x.rank \geq 1 \ . \end{cases} \tag{19.7}$$

INFORMATION ONLY

Lemma 19.8

For every node x, and for all operation counts q, we have

$$0 \le \phi_q(x) \le \alpha(n) \cdot x.rank\ .$$

Corollary 19.9

If node x is not a root and $x.rank > 0$, then $\phi_q(x) < \alpha(n) \cdot x.rank$.

Lemma 19.10

Let x be a node that is not a root, and suppose that the qth operation is either a LINK or a FIND-SET. Then after the qth operation, $\phi_q(x) \le \phi_{q-1}(x)$. Moreover, if $x.rank \ge 1$ and either level(x) or iter(x) changes due to the qth operation, then $\phi_q(x) \le \phi_{q-1}(x) - 1$. That is, x's potential cannot increase, and if it has positive rank and either level(x) or iter(x) changes, then x's potential drops by at least 1.

INFORMATION ONLY

INFORMATION ONLY

Lemma 19.12

The amortized cost of each LINK operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is LINK(x, y). The actual cost of the LINK operation is $O(1)$. Without loss of generality, suppose that the LINK makes y the parent of x.

To determine the change in potential due to the LINK, note that the only nodes whose potentials may change are x, y, and the children of y just prior to the operation. We'll show that the only node whose potential can increase due to the LINK is y, and that its increase is at most $\alpha(n)$:

- By Lemma 19.10, any node that is y's child just before the LINK cannot have its potential increase due to the LINK.
- From the definition (19.7) of $\phi_q(x)$, note that, since x was a root just before the qth operation, $\phi_{q-1}(x) = \alpha(n) \cdot x.rank$ at that time. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$. Otherwise,

  $$\begin{aligned}\phi_q(x) &< \alpha(n) \cdot x.rank \quad \text{(by Corollary 19.9)} \\ &= \phi_{q-1}(x)\,,\end{aligned}$$

  and so x's potential decreases.
- Because y is a root prior to the LINK, $\phi_{q-1}(y) = \alpha(n) \cdot y.rank$. After the LINK operation, y remains a root, so that y's potential still equals $\alpha(n)$ times its rank after the operation. The LINK operation either leaves y's rank alone or increases y's rank by 1. Therefore, either $\phi_q(y) = \phi_{q-1}(y)$ or $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

INFORMATION ONLY

Lemma 19.13

The amortized cost of each FIND-SET operation is $O(\alpha(n))$.

Proof Suppose that the qth operation is a FIND-SET and that the find path contains s nodes. The actual cost of the FIND-SET operation is $O(s)$. We will show that no node's potential increases due to the FIND-SET and that at least $\max\{0, s - (\alpha(n) + 2)\}$ nodes on the find path have their potential decrease by at least 1.

We first show that no node's potential increases. Lemma 19.10 takes care of all nodes other than the root. If x is the root, then its potential is $\alpha(n) \cdot x.rank$, which does not change due to the FIND-SET operation.

Now we show that at least $\max\{0, s - (\alpha(n) + 2)\}$ nodes have their potential decrease by at least 1. Let x be a node on the find path such that $x.rank > 0$ and x is followed somewhere on the find path by another node y that is not a root, where $\text{level}(y) = \text{level}(x)$ just before the FIND-SET operation. (Node y need not *immediately* follow x on the find path.) All but at most $\alpha(n) + 2$ nodes on the find path satisfy these constraints on x. Those that do not satisfy them are the first node on the find path (if it has rank 0), the last node on the path (i.e., the root), and the last node w on the path for which $\text{level}(w) = k$, for each $k = 0, 1, 2, \ldots, \alpha(n) - 1$.

Consider such a node x. It has positive rank and is followed somewhere on the find path by nonroot node y such that $\text{level}(y) = \text{level}(x)$ before the path compression occurs. We claim that the path compression decreases x's potential by at least 1. To prove this claim, let $k = \text{level}(x) = \text{level}(y)$ and $i = \text{iter}(x)$ before the path compression occurs. Just prior to the path compression caused by the FIND-SET, we have

$$\begin{aligned}
x.p.rank &\geq A_k^{(i)}(x.rank) && \text{(by the definition (19.5) of iter}(x)\text{)}, \\
y.p.rank &\geq A_k(y.rank) && \text{(by the definition (19.3) of level}(y)\text{)}, \\
y.rank &\geq x.p.rank && \text{(by Corollary 19.5 and because } y \text{ follows } x \text{ on the find path)}.
\end{aligned}$$

Putting these inequalities together gives

$$\begin{aligned}
y.p.rank &\geq A_k(y.rank) \\
&\geq A_k(x.p.rank) && \text{(because } A_k(j) \text{ is strictly increasing)} \\
&\geq A_k(A_k^{(i)}(x.rank)) \\
&= A_k^{(i+1)}(x.rank) && \text{(by the definition (3.30) of functional iteration)}.
\end{aligned}$$

INFORMATION ONLY

Because path compression makes x and y have the same parent, after path compression we have $x.p.rank = y.p.rank$. The parent of y might change due to the path compression, but if it does, the rank of y's new parent compared with the rank of y's parent before path compression is either the same or greater. Since $x.rank$ does not change, $x.p.rank = y.p.rank \geq A_k^{(i+1)}(x.rank)$ after path compression. By the definition (19.5) of the iter function, the value of $\text{iter}(x)$ increases from i to at least $i + 1$. By Lemma 19.10, $\phi_q(x) \leq \phi_{q-1}(x) - 1$, so that x's potential decreases by at least 1.

The amortized cost of the FIND-SET operation is the actual cost plus the change in potential. The actual cost is $O(s)$, and we have shown that the total potential decreases by at least $\max\{0, s - (\alpha(n) + 2)\}$. The amortized cost, therefore, is at most $O(s) - (s - (\alpha(n) + 2)) = O(s) - s + O(\alpha(n)) = O(\alpha(n))$, since we can scale up the units of potential to dominate the constant hidden in $O(s)$. (See Exercise 19.4-6.) ■

INFORMATION ONLY