Disjoint-Set Forests

Thanks for Showing Up!

Outline for Today

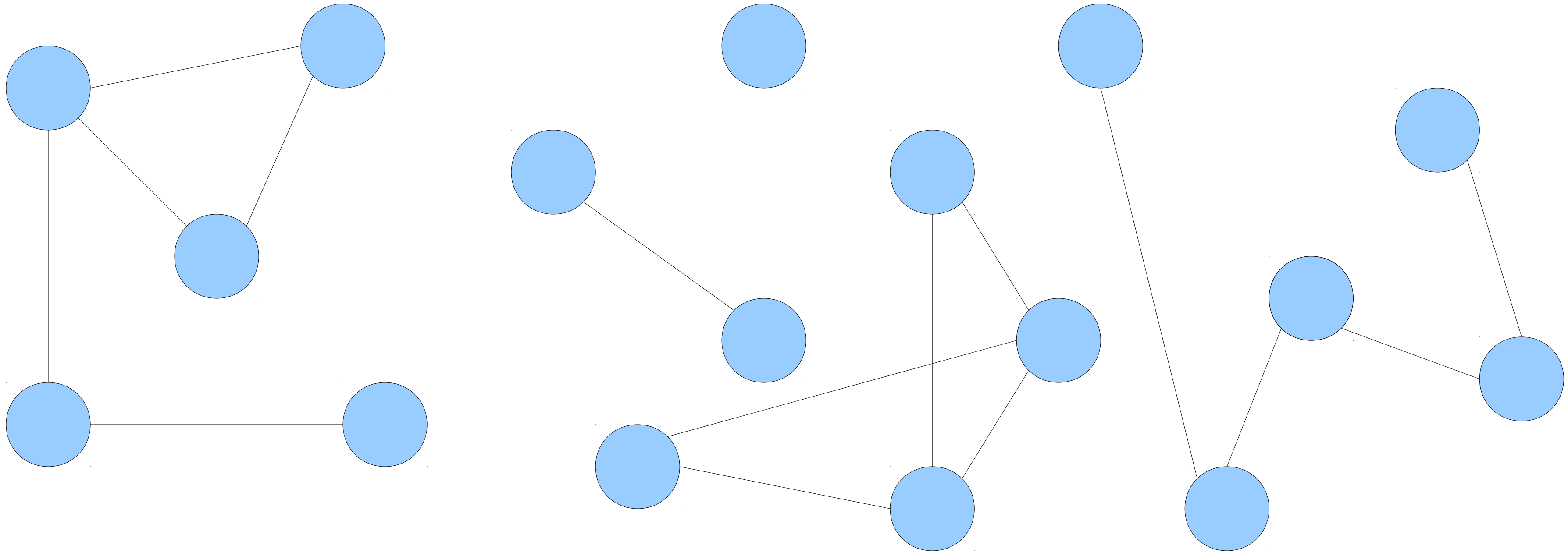
- **Incremental Connectivity**
  - Maintaining connectivity as edges are added to a graph.
- **Disjoint-Set Forests**
  - A simple data structure for incremental connectivity.
- **Union-by-Rank and Path Compression**
  - Two improvements over the basic data structure.
- **Forest Slicing**
  - A technique for analyzing these structures.
- **The Ackermann Inverse Function**
  - An unbelievably slowly-growing function.

The Dynamic Connectivity Problem

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?"

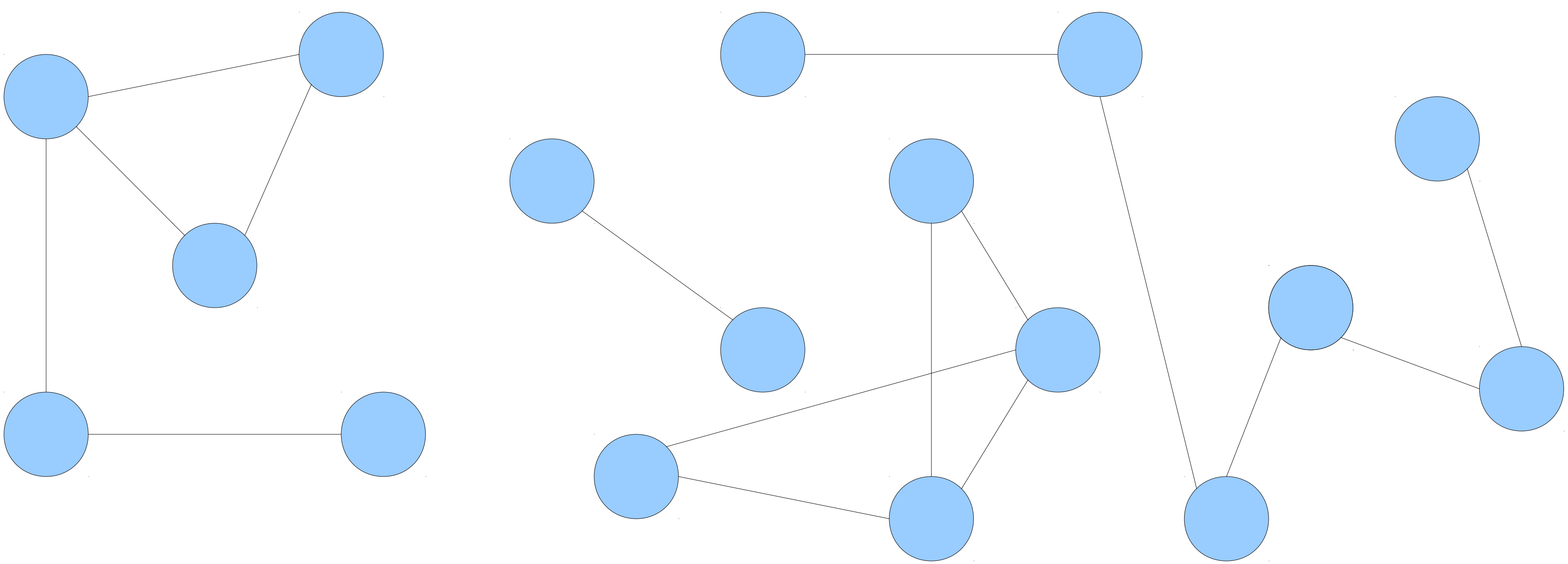


Dynamic Connectivity

- 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

- Euler tour trees solve dynamic connectivity in forests.
- Today, we'll focus on the ***incremental dynamic connectivity problem:*** maintaining connectivity when edges can only be added, not deleted.
- Applications to Kruskal's MST algorithm.
- Next Monday, we'll see how to achieve full dynamic connectivity in polylogarithmic amortized time.

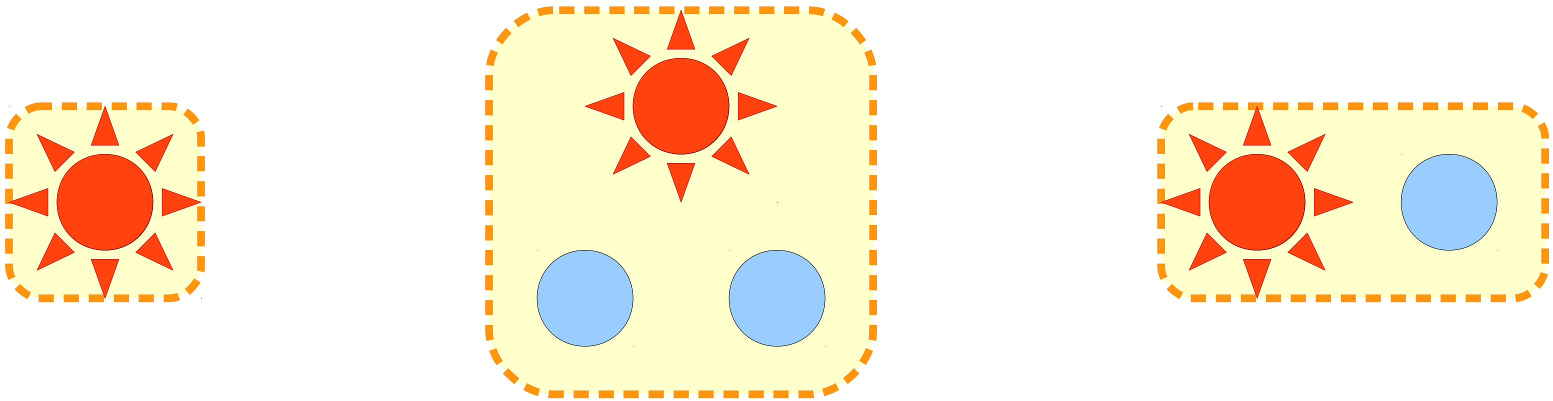
Incremental Connectivity and Partitions

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.
- **Goal:** Maintain a set partition while supporting the ***union*** and ***in-same-set*** operation.

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.



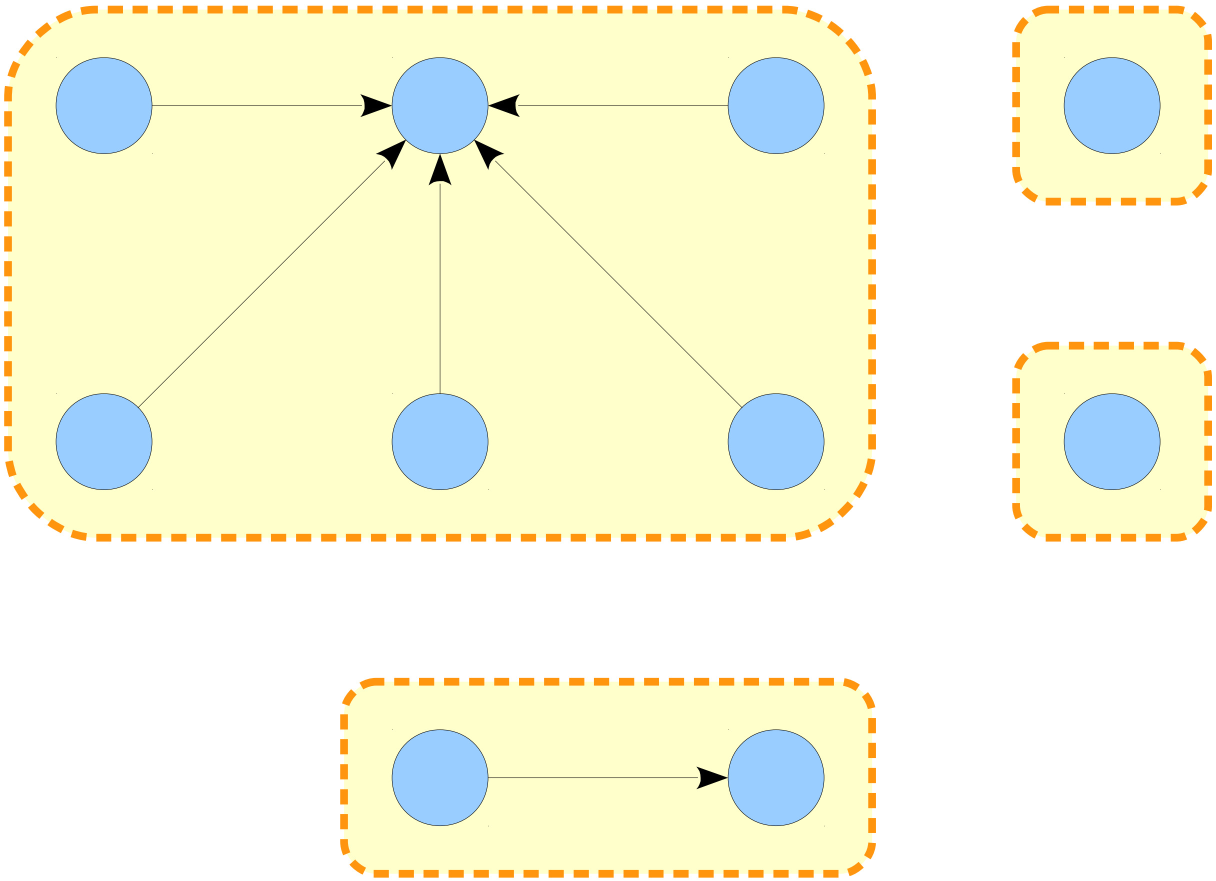
Union-Find Structures

- A ***union-find structure*** is a data structure supporting the following operations:
  - ***find***(x), which returns the representative of 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:** Associate each element in a set with a representative from that set.
- 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.

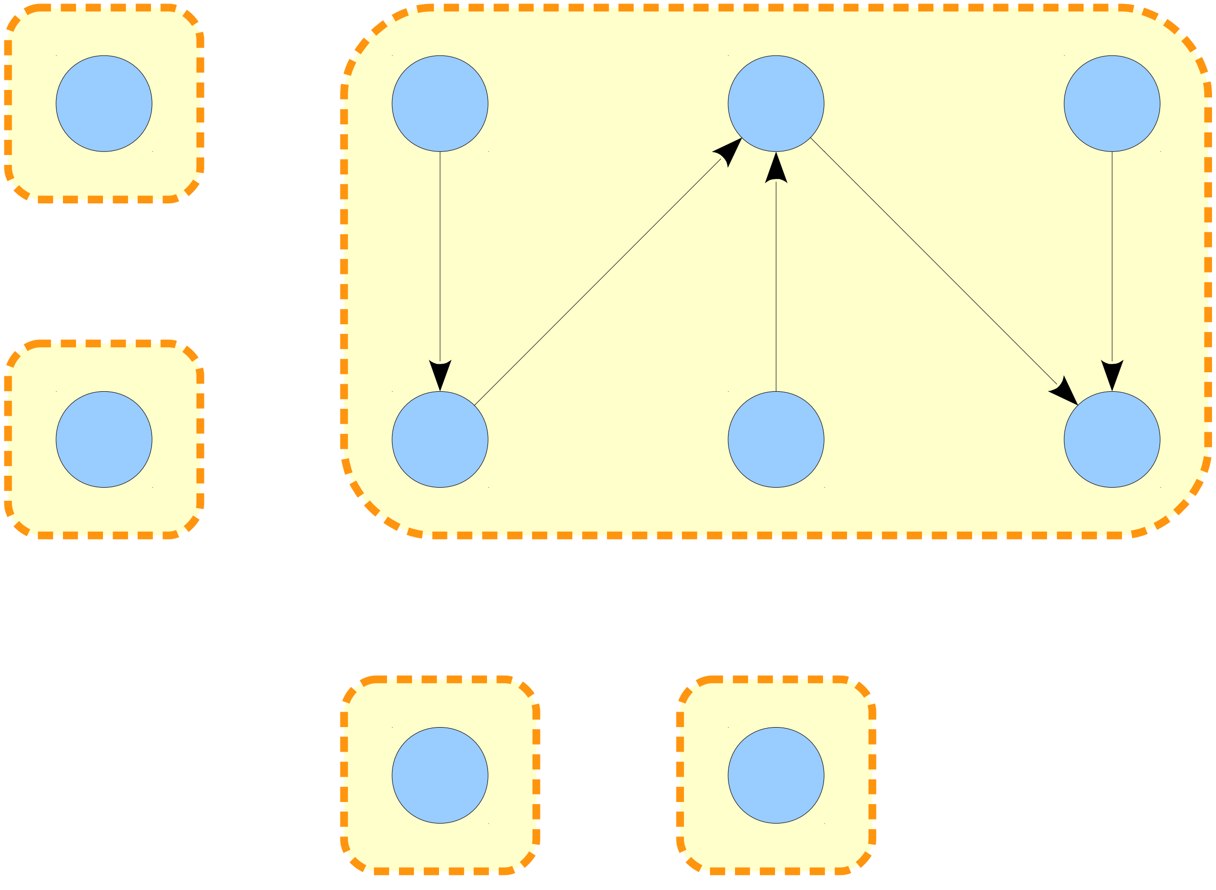
Using Representatives



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.
- Can we avoid paying this cost?

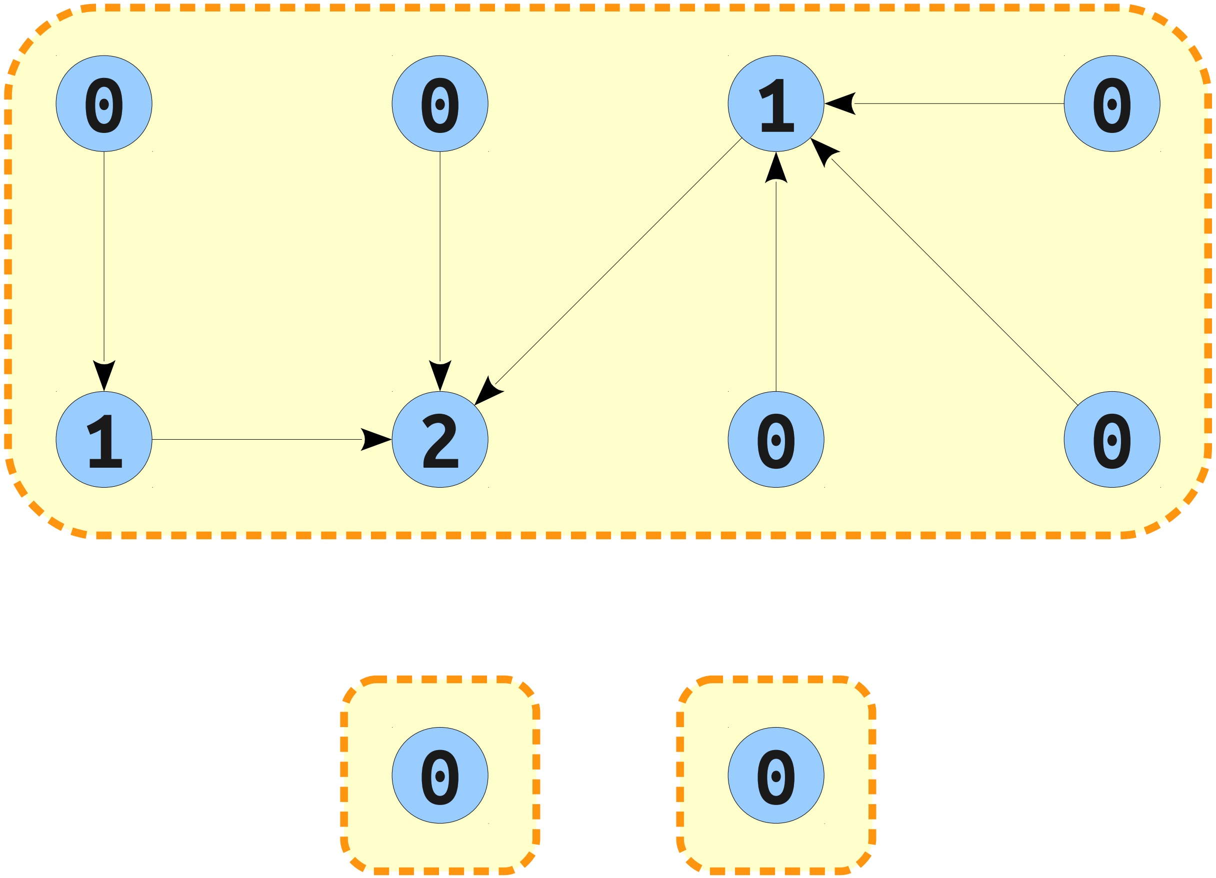
Hierarchical Representatives



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



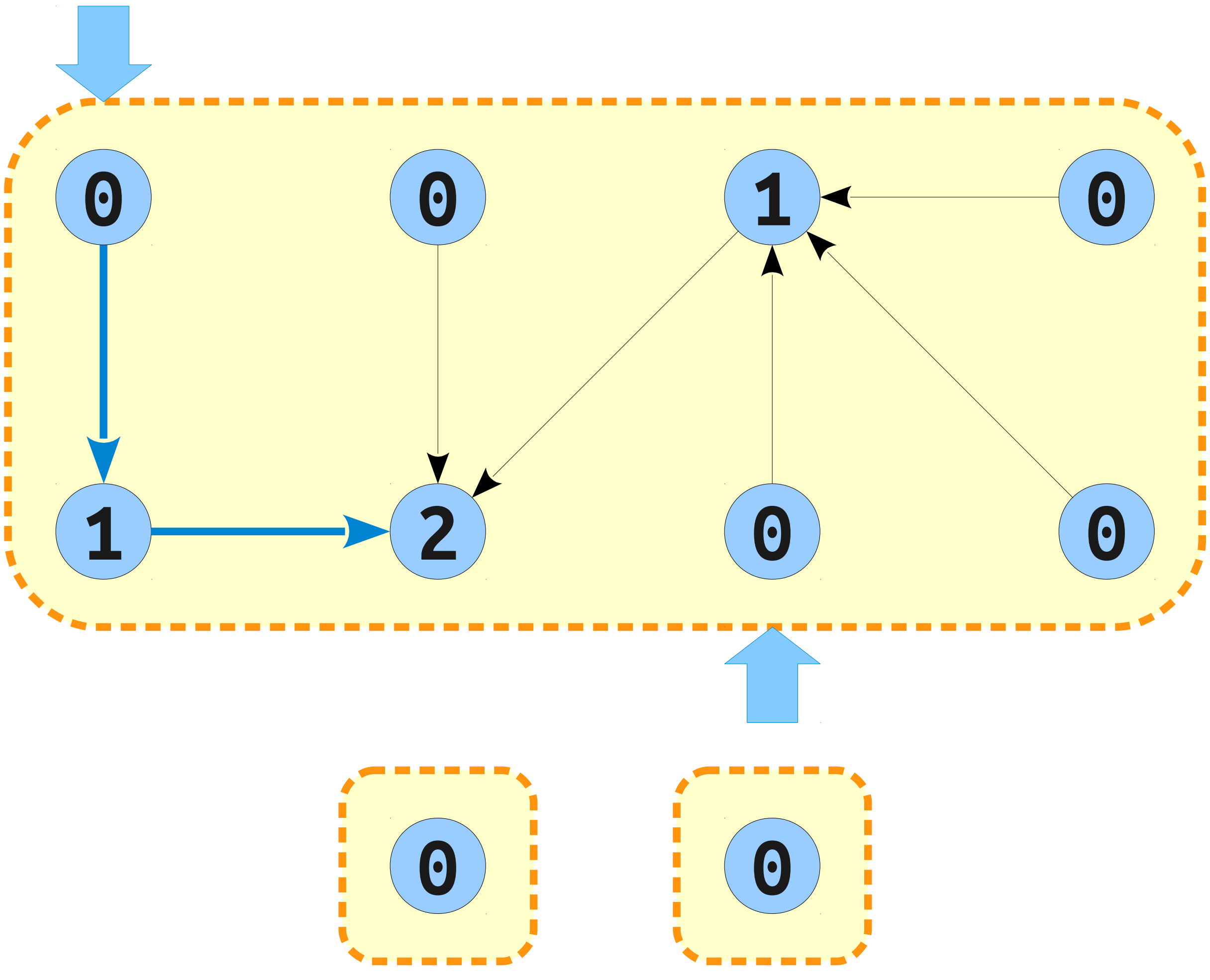
Union by Rank

- 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.

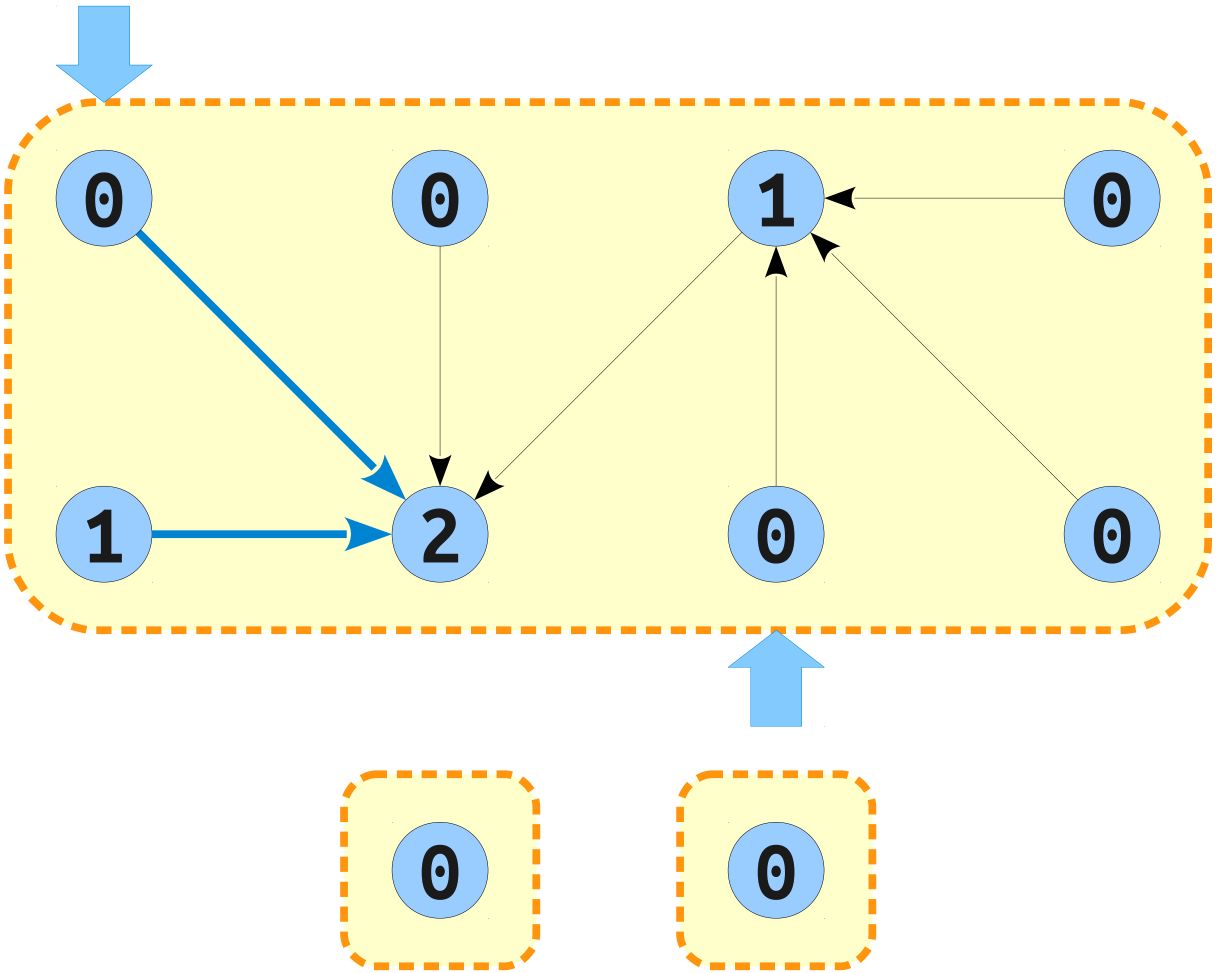
Union by Rank

- **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.
- **Claim:** Maximum rank of a node in a graph with n nodes is $O(\log n)$.
- Runtime for ***union*** and ***find*** is now $O(\log n)$.

Path Compression



Path Compression



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.
- When combined with union-by-rank, the runtime is O(log n).
- Intuitively, it seems like this shouldn't be tight, since repeated ***find*** operations will end up taking less time.

The Claim

- **Claim:** The runtime of ***union*** and ***find*** when using path compression and union-by-rank is amortized O(α(*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 a proof due to Seidel and Sharir based on a forest-slicing approach.

Where We're Going

- ***This analysis is nontrivial.***
- First, we're going to define our cost model so we know how to analyze the structure.
- Next, we'll introduce the forest-slicing approach and use it to prove a key lemma.
- Finally, we'll use that lemma to build recurrence relations that analyze the runtime.

Our Cost Model

- The cost of a ***union*** or ***find*** is O(1) plus Θ(#ptr-changes-made)
- Therefore, the cost of *m* operations is

$$\Theta(m + \#\text{ptr-changes-made})$$

- We will analyze the number of pointers changed across the life of the data structure to bound the overall cost.

Some Accounting Tricks

- To perform a ***union*** operation, we need to first perform two ***find***s.
- After that, only O(1) time is required to perform the ***union*** operation.
- Therefore, we can replace each ***union***(x, y) with three operations:
  - A call to ***find***(x).
  - A call to ***find***(y).
  - A linking step between the nodes found this way.
- Going forward, we will assume that each ***union*** operation will take worst-case time O(1).

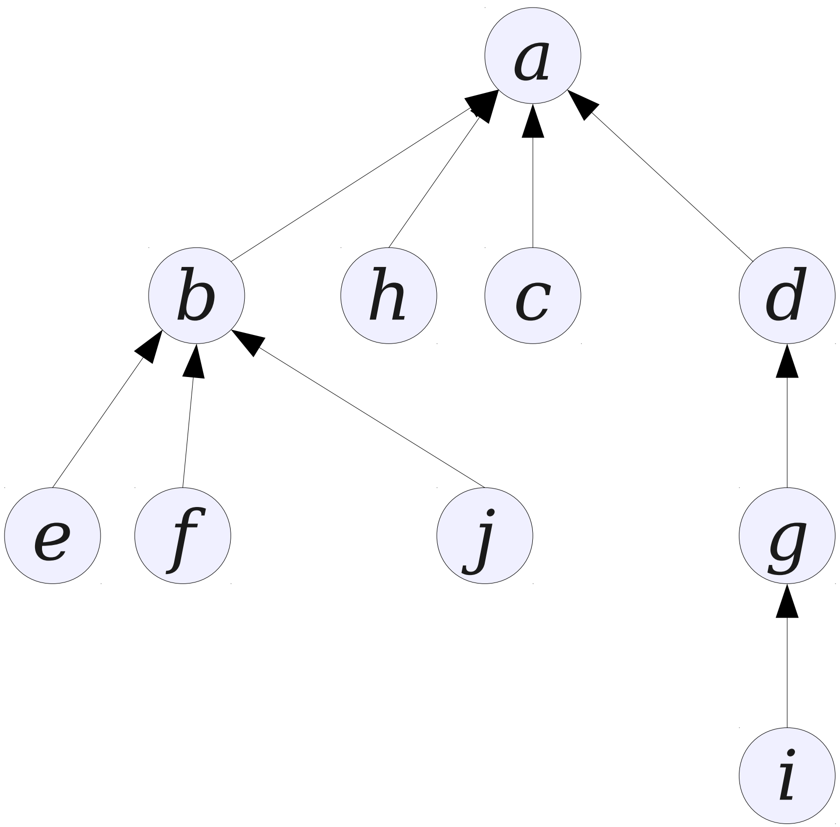
A Slight Simplification

- Currently, ***find***(x) compresses from x up to its ancestor.
- For mathematical simplicity, we'll introduce an operation ***compress***(x, y) that compresses from x upward to y, assuming that y is an ancestor of x.
- Our analysis will then try to bound the total cost of the ***compress*** operations.

Removing the Interleaving

- We will run into some trouble in our analysis because ***union***s and ***compress***es can be interleaved.
- To address this, we will will remove the interleaving by pretending that all ***compress***es come before all ***compress***es.
- This does not change the overall work being done.

Removing the Interleaving



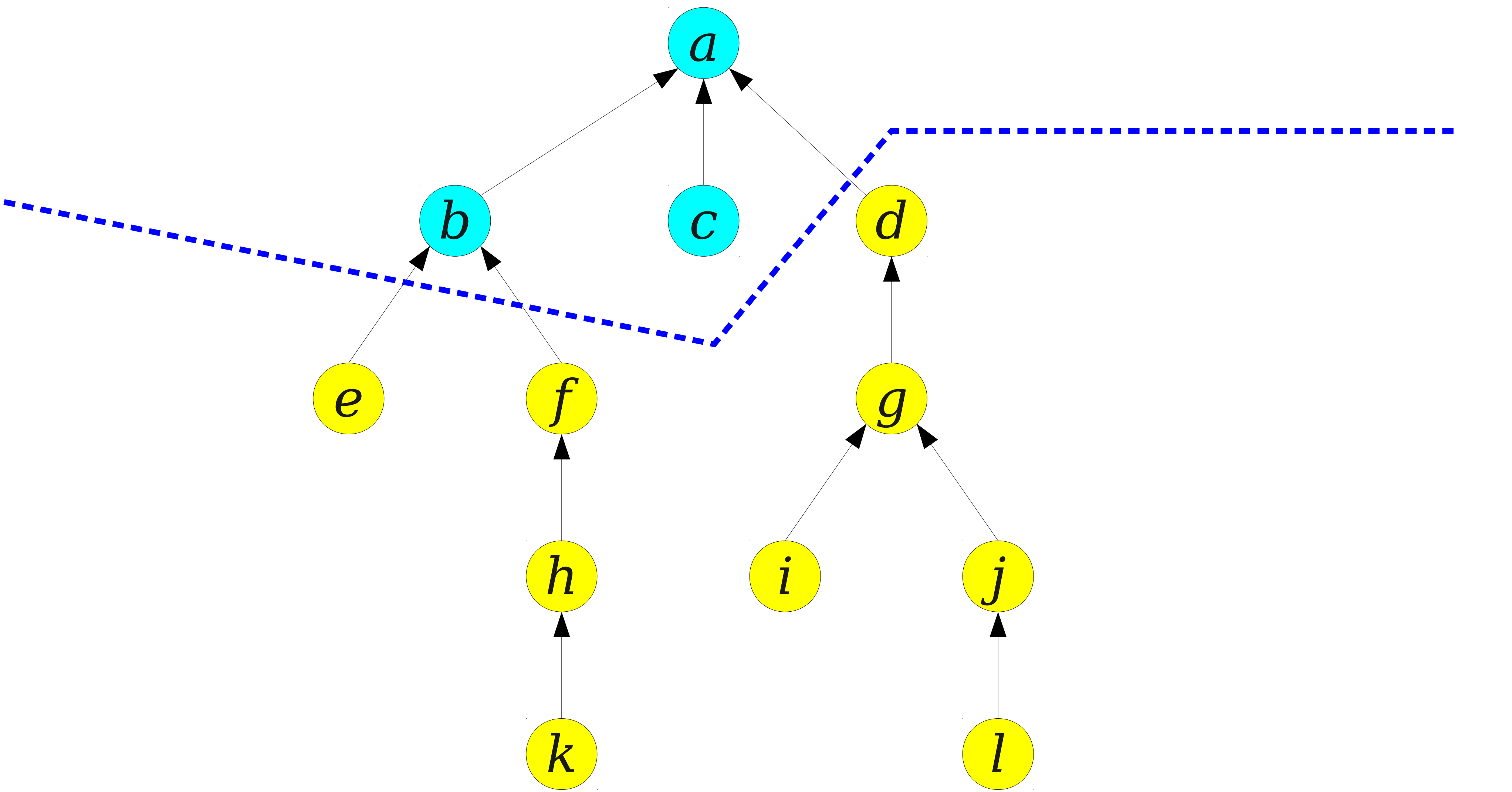
compress(j, b) ***union***(b, a) ***compress***(h, a)	***union***(b, a) ***compress***(j, b) ***compress***(h, a)
$f \to b$ $h \to b$ $j \to b$ $b \to a$ $h \to a$	$b \to a$ $f \to b$ $h \to b$ $j \to b$ $h \to a$

Recap: The Setup

- Transform any sequence of ***union***s and ***find***s as follows:
  - Replace all ***union*** operations with two ***find***s and a ***union*** on the ancestors.
  - Replace each ***find*** operation with a ***compress*** operation indicating its start and end nodes.
  - Move all ***union*** operations to the front.
- Since all ***union***s are at the front, we build the entire forest before we begin compressing.
- Can analyze ***compress*** assuming the forest has already been created for us.

The Forest-Slicing Approach

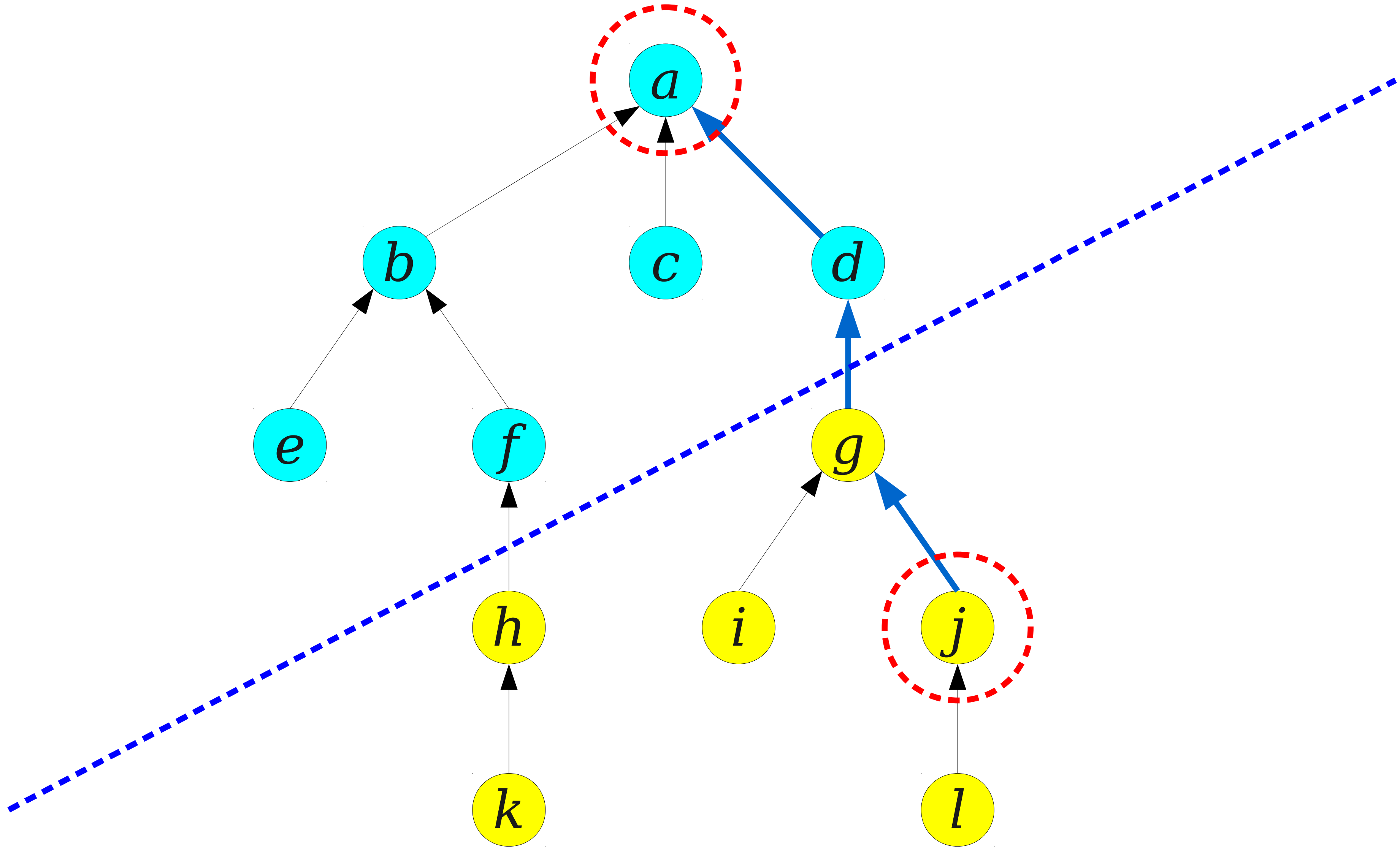
Forest-Slicing



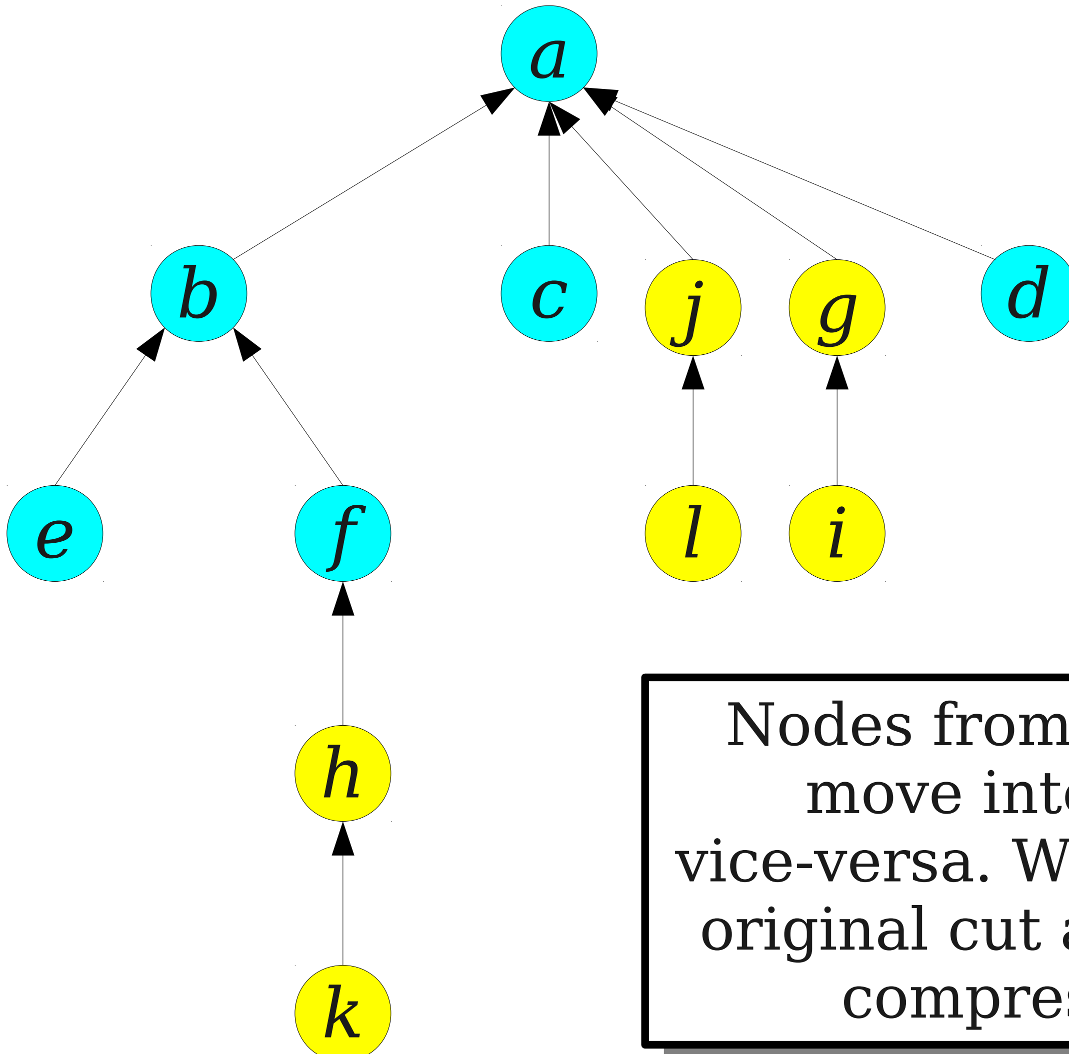
Forest-Slicing

- Let $\mathscr{F}$ be a disjoint-set forest.
- Consider splitting $\mathscr{F}$ into two forests $\mathscr{F}_+$ and $\mathscr{F}_-$ with the following properties:
  - $\mathscr{F}_+$ is ***upward-closed***: if $x \in \mathscr{F}_+$, then any ancestor of x is also in $\mathscr{F}_+$.
  - $\mathscr{F}_-$ is ***downward-closed***: if $x \in \mathscr{F}_-$, then any descendant of x is also in $\mathscr{F}_-$.
- We'll call $\mathscr{F}_+$ the ***top forest*** and $\mathscr{F}_-$ the ***bottom forest***.

Forest-Slicing



Forest-Slicing

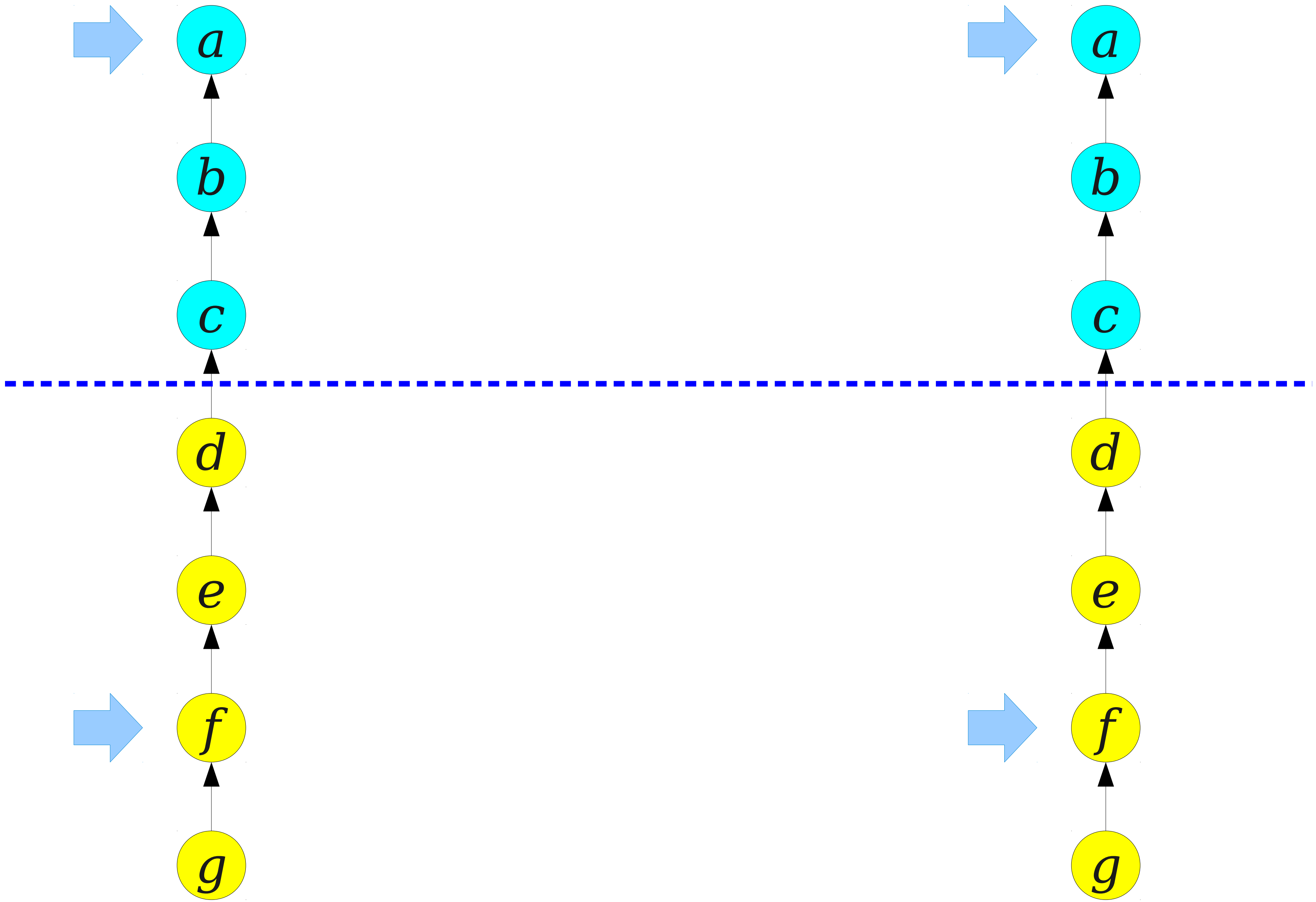


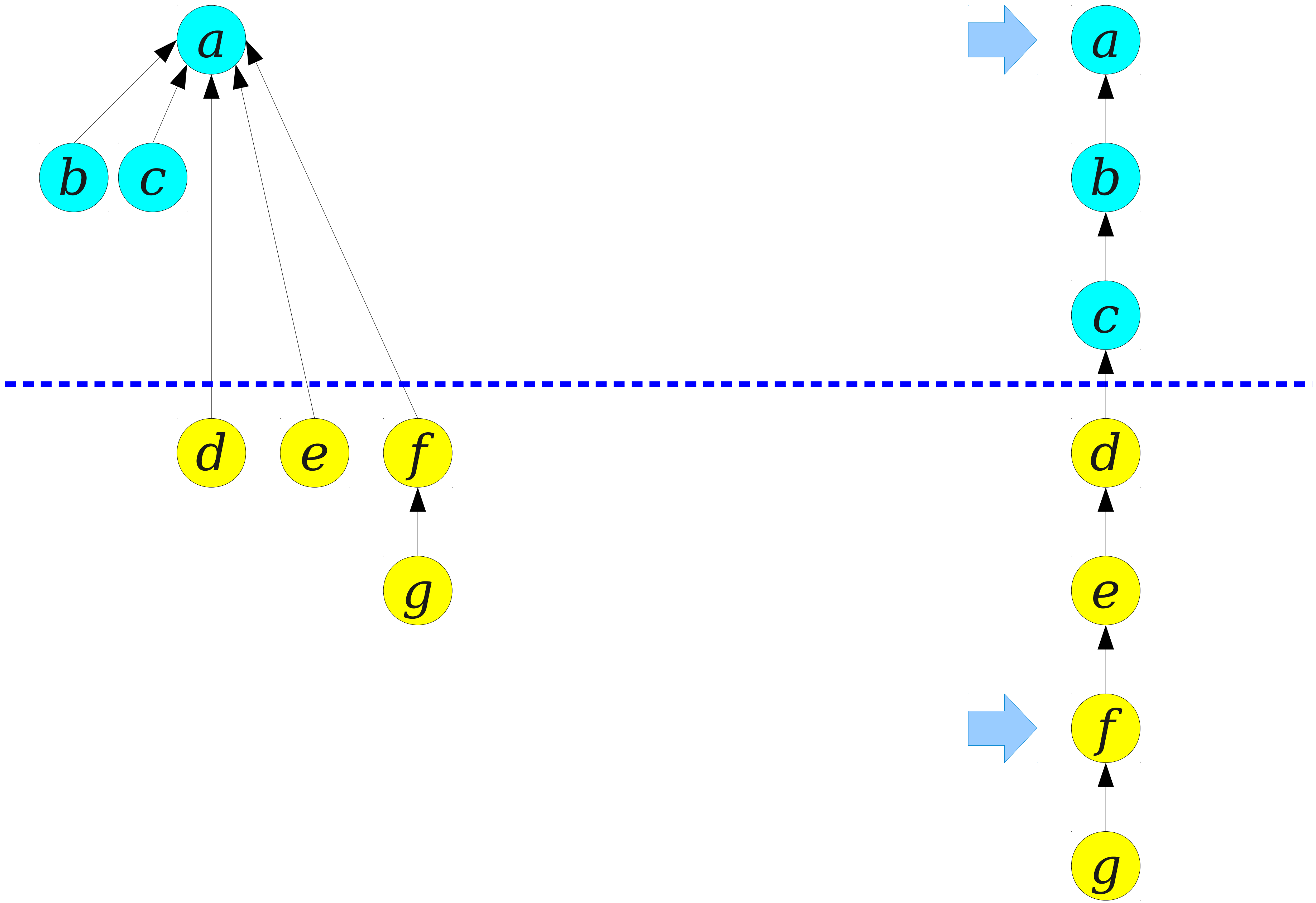
Nodes from $\mathscr{F}_-$ never move into $\mathscr{F}_+$ or vice-versa. We retain the original cut after doing compressions.

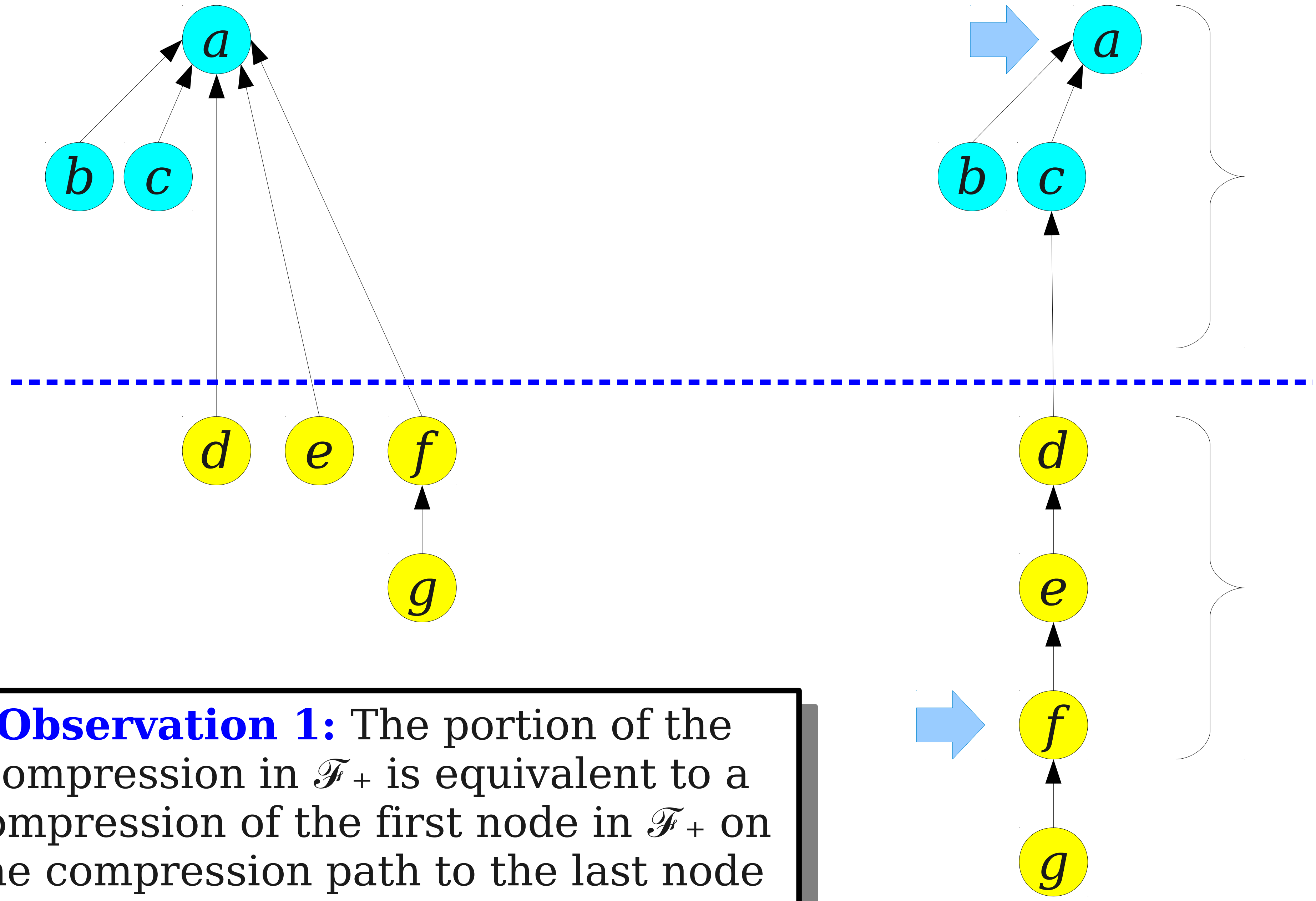
Why Slice Forests?

Forest-Slicing

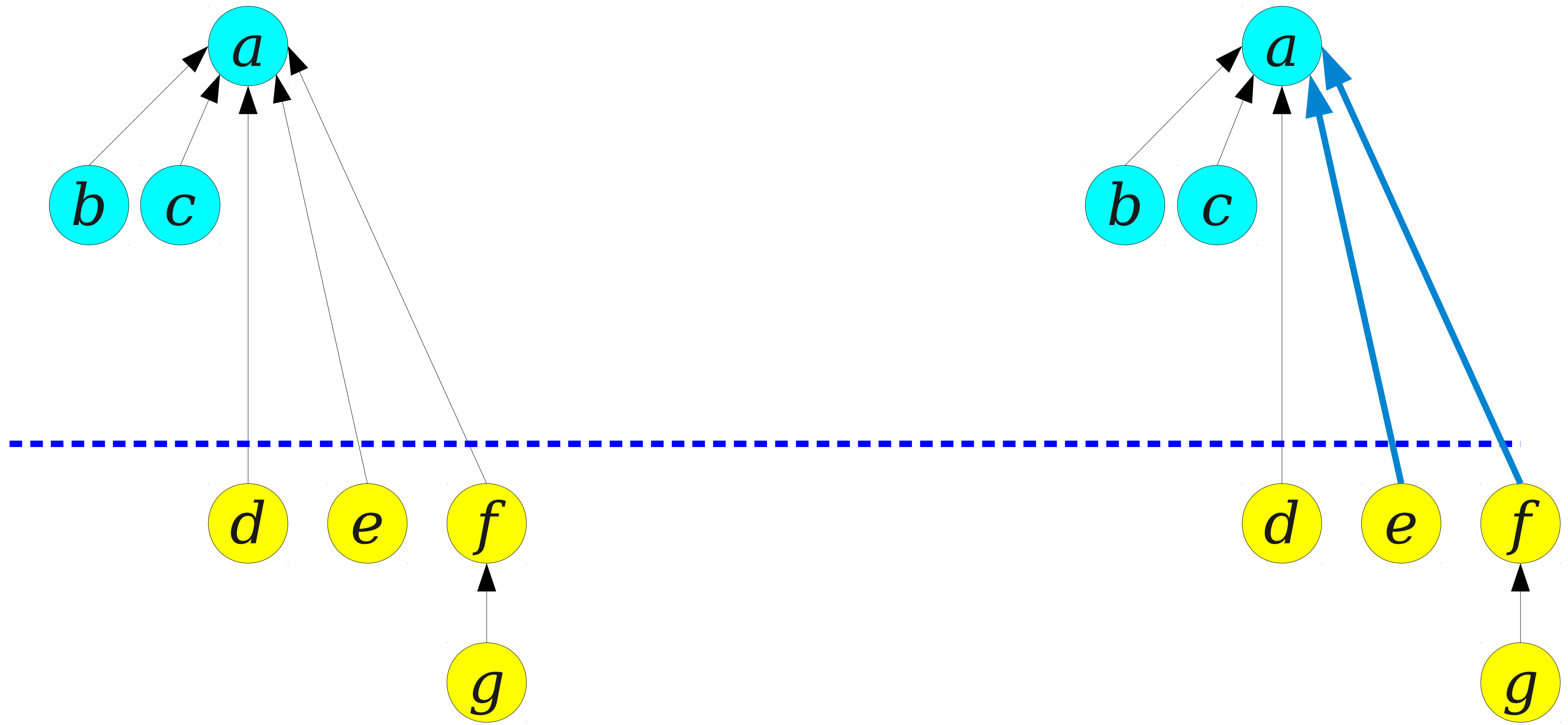
- **Key insight:** Each ***compress*** operation is either
  - purely in $\mathscr{F}_+$,
  - purely in $\mathscr{F}_-$, or
  - crosses from $\mathscr{F}_-$ into $\mathscr{F}_+$.
- Analyze the runtime of a series of compressions using a divide-and-conquer approach:
  - Analyze the compressions purely in $\mathscr{F}_+$ and $\mathscr{F}_-$ recursively.
  - Bound the cost of the compressions crossing from $\mathscr{F}_+$ to $\mathscr{F}_-$ separately.


a
b
c
d
e
f
g
a
b
c
d
e
f
g

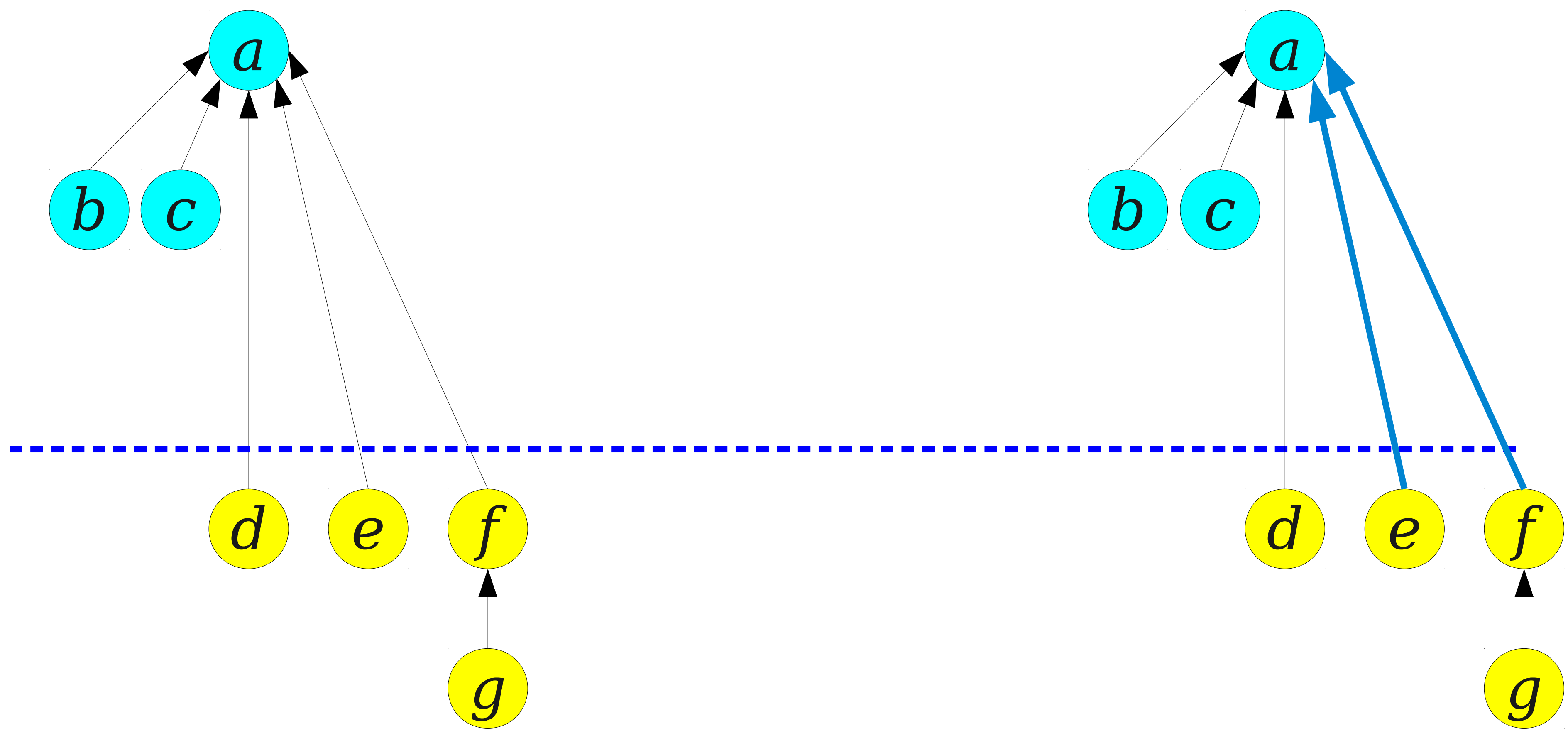

a
b
c
d
e
f
g
a
b
c
d
e
f
g



Observation 1: The portion of the compression in $\mathscr{F}_+$ is equivalent to a compression of the first node in $\mathscr{F}_+$ on the compression path to the last node in $\mathscr{F}_+$ on the compression path.



Observation 2: The effect of the compression on $\mathscr{F}_-$ is *not* the same as the effect of compressing from the first node in $\mathscr{F}_-$ to the last node in $\mathscr{F}_-$.



Observation 3: The cost of the compress in $\mathscr{F}_-$ is the number of nodes in $\mathscr{F}_-$ that got a parent in $\mathscr{F}_+$, plus (possibly) one more for the topmost node in $\mathscr{F}_-$ on the compression path.

The Cost of Crossing Compressions

- Suppose we do k compressions crossing from $\mathscr{F}_-$ into $\mathscr{F}_+$.
- We can upper bound the cost of these compressions as the sum of the following:
  - The cost of all the tops of those compressions, which occur purely in $\mathscr{F}_+$.
  - k (one per compression).
  - The number of nodes in $\mathscr{F}_-$, since each node in $\mathscr{F}_-$ gets a parent in $\mathscr{F}_+$ for the first time at most once.

Theorem: Let $\mathscr{F}$ be a disjoint-set forest and let $\mathscr{F}_+$ and $\mathscr{F}_-$ be a partition of $\mathscr{F}$ into top and bottom forests.

Then for any series of m compressions C, there exist compression sequences C_+ in $\mathscr{F}_+$ and C_- in $\mathscr{F}_-$ such that

- $\text{cost}(C) \leq \text{cost}(C_+) + \text{cost}(C_-) + n + m_+$
- $m_+ + m_- = m$

Here, $m_+ = |C_+|$ and $m_- = |C_-|$.

Paths purely in $\mathscr{F}_+$ or $\mathscr{F}_-$, plus the tops of paths crossing them.

Nodes in $\mathscr{F}_-$ getting their first parent in $\mathscr{F}_+$

Nodes in $\mathscr{F}_-$ having their parent in $\mathscr{F}_+$ change.

Time-Out for Announcements!

Midterms Graded

- Midterms are graded and available for pickup.
- Solutions and statistics released in hardcopy up front.
- Regrades accepted until next Monday at 3:15PM; just let us know what problem(s) you'd like us to review.

Presentation Schedule

- We've posted the presentation schedule to the course website.
- You're welcome to attend any presentations you'd like as long as they're not on the same data structure that you chose.

Writeup Logistics

- Writeup will be due electronically as a PDF exactly 24 hours before you present.
- Your writeup should include
  - background on the data structure,
  - how the data structure works,
  - a correctness and runtime analysis, and
  - what your “interesting” addition is.

Presentation Logistics

- Presentation should be 15 – 20 minutes; we'll cut you off after 20 minutes.
- We'll have up to five minutes of questions afterwards.
- Please arrive five minutes early so you have time to get set up.
- Don't try to present everything – you won't have time!

Your Questions

“Can we have some opportunity to practice our presentations in whatever room they're being presented in?”

Absolutely! The room assignments are posted online. Feel free to stop by those locations to try out your presentation, and feel free to invite friends along with!

"How long do you plan on teaching? Do you think you'll ever want to leave Stanford, either to pursue options at other universities or to do research for a company? Do you have any desire to start your own company?"

Wow, that's a hard one.

Back to CS166!

The Main Analysis

Where We Are

- We now have a divide-and-conquer-style result for evaluating the runtime of a series of ***compress***es.
- We'll now combine that analysis with some Clever Math to get the overall runtime bound.

Rank Forests

- The result we proved about compression costs works even if we don't use union-by-rank.
- If we do use union-by-rank, the following results hold:
  - The maximum rank of any node is O(log n).
  - For any rank r, there are at most n / 2^r nodes of rank greater than r.

Some Terminology

- Let's denote by $T(m, n)$ the maximum possible cost of performing m ***compress***es in a rank forest of n nodes.
- Define $T(m, n, r)$ to be the maximum possible cost of performing m ***compress***es in a rank forest of at most n nodes and with maximum rank r.
- Note that $T(m, n) = T(m, n, \mathrm{O}(\log n))$.

Functional Iteration

- Let $f : \mathbb{N} \to \mathbb{N}$ be a nondecreasing function where $f(0) = 0$ and $f(n) < n$ for $n > 0$.
- The ***iterated function of f***, denoted ***f****, is defined as follows:

$$f^*(n) = \begin{cases} 0 & \text{if } f(n) \leq 2 \\ 1 + f^*(f(n)) & \text{otherwise} \end{cases}$$

- Intuitively, $f^*(n)$ is the number of times that f has to be applied to n for n to drop down to 2.
  - (The choice of 2 here is arbitrary; we just need a nice, small constant.)

Functional Iteration

- As an example, consider the function lg n, assuming that we round down.
- Notice that
  - lg 137 = 7
  - lg 7 = 2
- Therefore, lg* 137 = 2.

Iterated Logarithms

- For any k, define

$$\log^{*(k)} n = \log^{***\ldots*} n \ (k \text{ times})$$

- These functions are *extremely* slowly-growing.
- $\log^* n \leq 4$ for all $n < 2^{65,536}$, for example.
- **Fun exercise:** What is the inverse function of $\log^* n$? How about $\log^{**} n$?

Where We're Going

- We're going to show that

$$T(m, n) = O(n \lg^{*(k)} \lg n + m)$$

  for any constant $k \geq 1$

- From there, we'll define a function $\alpha(n)$ that grows slower than $\lg^{*(k)} n$ for any k and prove that

$$T(m, n) = O(m\alpha(n) + n)$$

Our Approach

- Our result will rely on a “feedback” technique used to build stronger results out of weaker ones.
- We'll find an initial proof that

$$T(m, n) = O(n \lg^* \lg n + m).$$

- Then, we'll prove that if we know that $T(m, n) = O(n \lg^{*(k)} \lg n + m)$, then we can prove $T(m, n) = O(n \lg^{*(k+1)} \lg n + m)$

Proving $T(m, n) = \mathrm{O}(n \lg^* \lg n + m)$

A Starting Point

- ***Lemma:*** $T(m, n, r) \leq nr$.
- ***Proof:*** Since the maximum possible rank is r, each node can have its parent change at most r times. Therefore, the number of pointer changes made is at most nr.
- (Remember that we've defined the cost to be the number of pointer changes.)

Getting a Recurrence

- Let $\mathscr{F}$ be a rank forest of maximum rank r and let C be a worst-case series of m compresses performed in $\mathscr{F}$.
- Split $\mathscr{F}$ into $\mathscr{F}_-$ and $\mathscr{F}_+$ by putting all nodes of rank at most lg r into $\mathscr{F}_-$ and all other nodes into $\mathscr{F}_+$.
- By our earlier theorem, there exist C_+ and C_- such that

$$cost(C) \leq cost(C_+) + cost(C_-) + n + m_+$$

- Therefore

$$T(m, n, r) \leq cost(C_+) + cost(C_-) + n + m_+$$

- Let's see if we can simplify this expression, starting with $cost(C_+)$.

An Observation

- The forest $\mathscr{F}_+$ consists of all nodes whose rank is greater than lg r.
- Therefore, the ranks go from lg $r + 1$ up through and including r.
- By our earlier result, the number of nodes in $\mathscr{F}_+$ is at most $n / 2^{\lg r} = n / r$.
- If we subtract lg $r + 1$ from the ranks of all of the nodes, we end up with a rank forest whose maximum rank is at most r.
- Therefore, by our earlier lemma, we get that $cost(C_+) \le r\ (n / r) = n$.

The Recurrence

- We had

$$T(m, n, r) \le cost(C_+) + cost(C_-) + n + m_+$$

- We now have

$$T(m, n, r) \le cost(C_-) + 2n + m_+$$

- Notice that C_- is a set of compressions in a rank forest of maximum rank lg r.
- There are at most n nodes in $\mathscr{F}_-$ and the number of compresses in C_- is m_-.
- Therefore, we have

$$T(m, n, r) \le T(m_-, n, \lg r) + 2n + m_+$$

Solving the Recurrence

- We have

$$T(m, n, r) \leq T(m_-, n, \lg r) + 2n + m_+$$

- As our base cases:

$$\mathrm{T}(0, n, r) = 0$$

$$T(m, n, 2) \leq 2n$$

- As the recursion unwinds:
  - The $2n$ term gets multiplied by the number of layers in the recursion.
  - The m_+ term sums across the layers to at most m.
- The solution is $\boldsymbol{T(m, n, r) \leq 2nL + m}$, where L is the total number of layers in the recursion.

Solving the Recurrence

- The solution is $T(m, n, r) \leq 2nL + m$, where L is the total number of layers in the recursion.
- At each layer, we shrink r from r to lg r.
- The maximum number of times you can do this before r gets to 2 is at most lg* r.
- Therefore, $T(m, n, r) \leq 2n$ lg* $r + m$.
- Since $r = O(\log n)$, this is **$O(n \lg^* \lg n + m)$**.

Adding Extra Stars

The Feedback Lemma

- ***Lemma***: If

$$T(m, n, r) \leq 2n \log^{*(k)} r + km$$

then

$$T(m, n, r) \leq 2n \log^{*(k+1)} r + (k+1)m$$

- This will enable us to place as many stars as we'd like on the runtime.

What We'll Prove

- ***Lemma***: If

$$T(m, n, r) \leq 2n \log^* r + m$$

  then

$$T(m, n, r) \leq 2n \log^{**} r + 2m$$

- This is a special case of the theorem with $k = 1$, but uses the same basic approach.
- **Fun exercise:** Update the proof to the general case.

The Recurrence

- Let $\mathscr{F}$ be a rank forest of maximum rank r and let C be a worst-case series of m compressions performed in $\mathscr{F}$.
- Split $\mathscr{F}$ into $\mathscr{F}_-$ and $\mathscr{F}_+$ by putting all nodes of depth at most lg* r into $\mathscr{F}_-$ and all other nodes into $\mathscr{F}_+$.
- There exist C_+ and C_- such that

$$cost(C) \leq cost(C_+) + cost(C_-) + n + m_+$$

- Therefore

$$T(m, n, r) \leq cost(C_+) + cost(C_-) + n + m_+$$

- Let's see if we can simplify this expression.

An Observation

- The forest $\mathscr{F}_+$ consists of all nodes whose rank is at least $\lg^* r$.
- Therefore, the ranks go from $\lg^* r + 1$ up through and including r.
- The number of nodes in $\mathscr{F}_+$ is at most $n / 2^{\lg^* r}$
- If we subtract $\lg^* r + 1$ from the ranks of all of the nodes, we end up with a rank forest with ranks going up to at most r.
- Then $cost(C_+) \le 2(n / 2^{\lg^* r}) \lg^* r + m_+$.
- Therefore, $cost(C_+) \le 2n + m_+$.

The Recurrence

- We had

$$T(m, n, r) \leq cost(C_+) + cost(C_-) + n + m_+$$

- We now have

$$T(m, n, r) \leq cost(C_-) + 2n + 2m_+$$

- Notice that C_- is a set of compressions in a rank forest of maximum rank lg* r.
- There are at most n nodes in $\mathscr{F}_-$ and the number of compresses in C_- is m_-.
- Therefore, we have

$$T(m, n, r) \leq T(m_-, n, \lg^* r) + 2n + 2m_+$$

Solving the Recurrence

- We have
    - $T(m, n, r) \leq T(m_-, n, \lg^* r) + 2n + 2m_+$
- As our base cases:

$$T(0, n, r) = 0$$

$$T(m, n, 2) \leq 2n$$

- As the recursion unwinds:
    - The $2n$ term gets multiplied by the number of layers in the recursion.
    - The $2m_+$ term sums across the layers to $2m$.
- The solution is $\boldsymbol{T(m, n, r) \leq 2nL + 2m}$, where L is the total number of layers in the recursion.

Solving the Recurrence

- The solution is $T(m, n, r) \leq 2nL + 2m$, where L is the total number of layers in the recursion.
- At each layer, we shrink r from r to $\lg^* r$.
- The maximum number of times you can do this before r gets to 2 is $\lg^{**} r$.
- Thus $\boldsymbol{T(m, n, r) \leq 2n \lg^{**} r + 2m}$.

The Optimal Approach

- We know that for any $k > 0$, that

$$T(m, n, r) \leq 2n \lg^{*(k)} r + km$$

- Since $r = O(\log n)$, this means that for any $k > 0$, we have

$$T(m, n) = O(n \lg^{*(k)} \lg n + km)$$

- What is the optimal value of k?
- The ***Ackermann inverse function*** $\alpha(n)$ is defined as follows:

$$\alpha(m, n) = \min \{ k \mid \lg^{*(k)} \lg n \leq 1 + m / n \}$$

- Therefore:

$$T(m, n) = O(n + m + \alpha(m, n)) = \mathbf{O(n + m\alpha(m, n))}$$

Completing the Analysis

- In a forest of n nodes, if we do m ***union*** and ***find*** operations, the total runtime will be

$$O(m + m\alpha(m, n)) = O(n + m\alpha(m, n)).$$

- Assuming that $m \geq n$, the amortized cost per operation is $\boldsymbol{O(\alpha(m, n))}$.

For Perspective

- Consider $2^{65,536}$.
- Then
  - $\lg 2^{65,536} = 65{,}536 = 2^{16}$
  - $\lg 2^{16} = 16 = 2^4$
  - $\lg 2^4 = 4 = 2^2$
  - $\lg 2^2 = 2$
- So $\lg^* 2^{65,536} = 4$.

For Perspective

- Recall that lg* $2^{65,656} = 4$.
- Let z be 2 raised to the $2^{65,656}$th power.
- Then lg* $z = 5$.
- If you let $z' = 2^z$, then lg* $z' = 6$.
- Since lg** z' counts the number of times you have to apply lg* to z' to drop it down to two, this means that lg** z' is about three.
- Therefore, if $m \geq n$, then $\alpha(m, n) \leq 3$ as long as $n \geq z'$.

Next Time

- **Fully-Dynamic Connectivity**
  - How to maintain full connectivity information in a dynamic graph.