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.

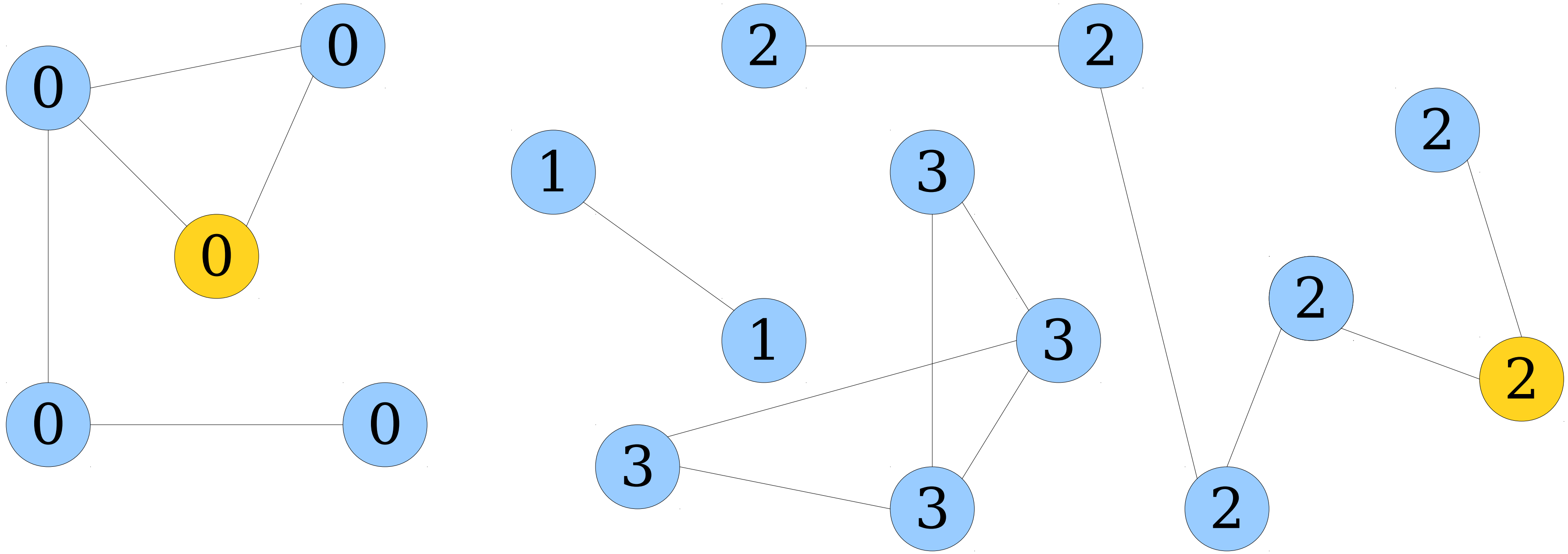
The Dynamic Connectivity Problem

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?"

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

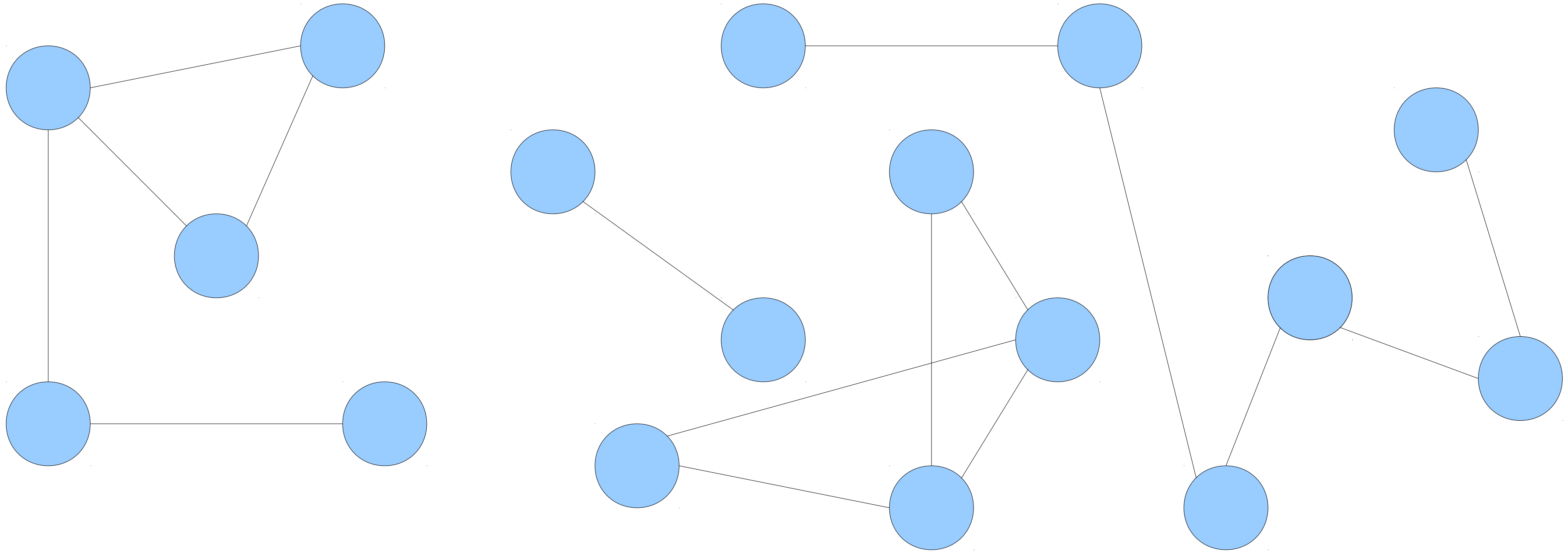


Dynamic Connectivity

- 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

- 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.

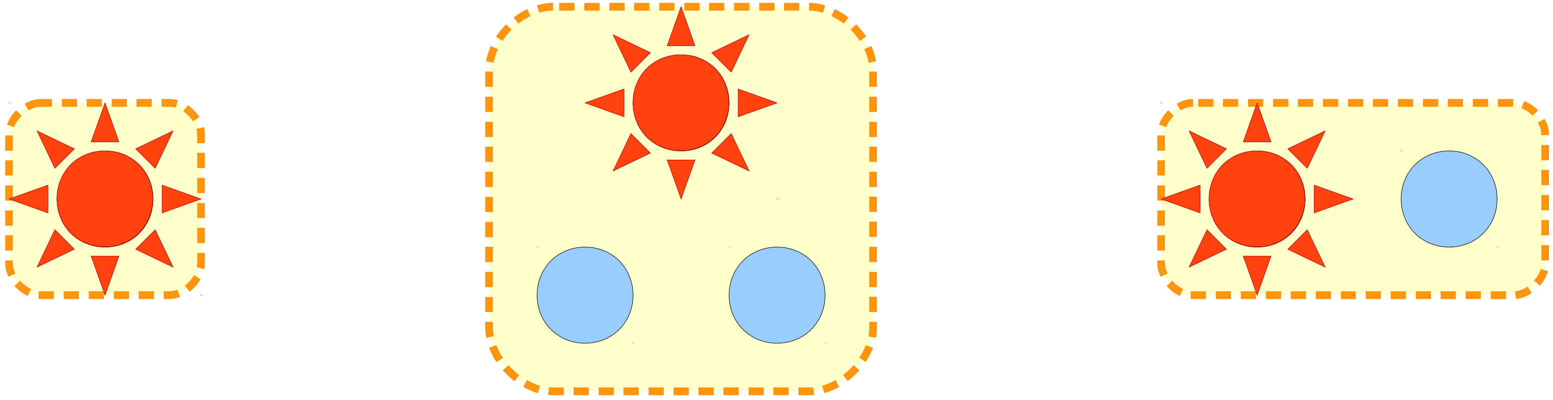
Incremental Connectivity and Partitions

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.



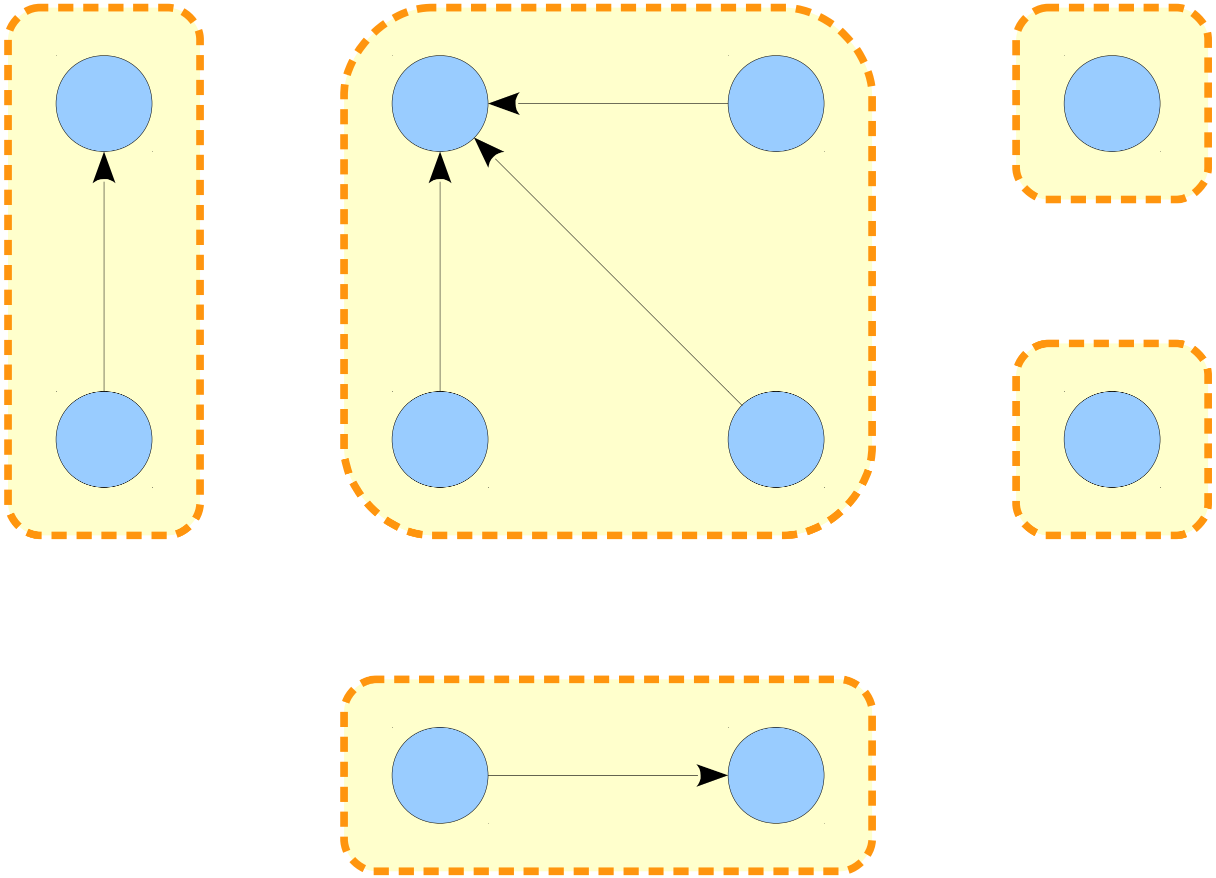
Union-Find Structures

- 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.

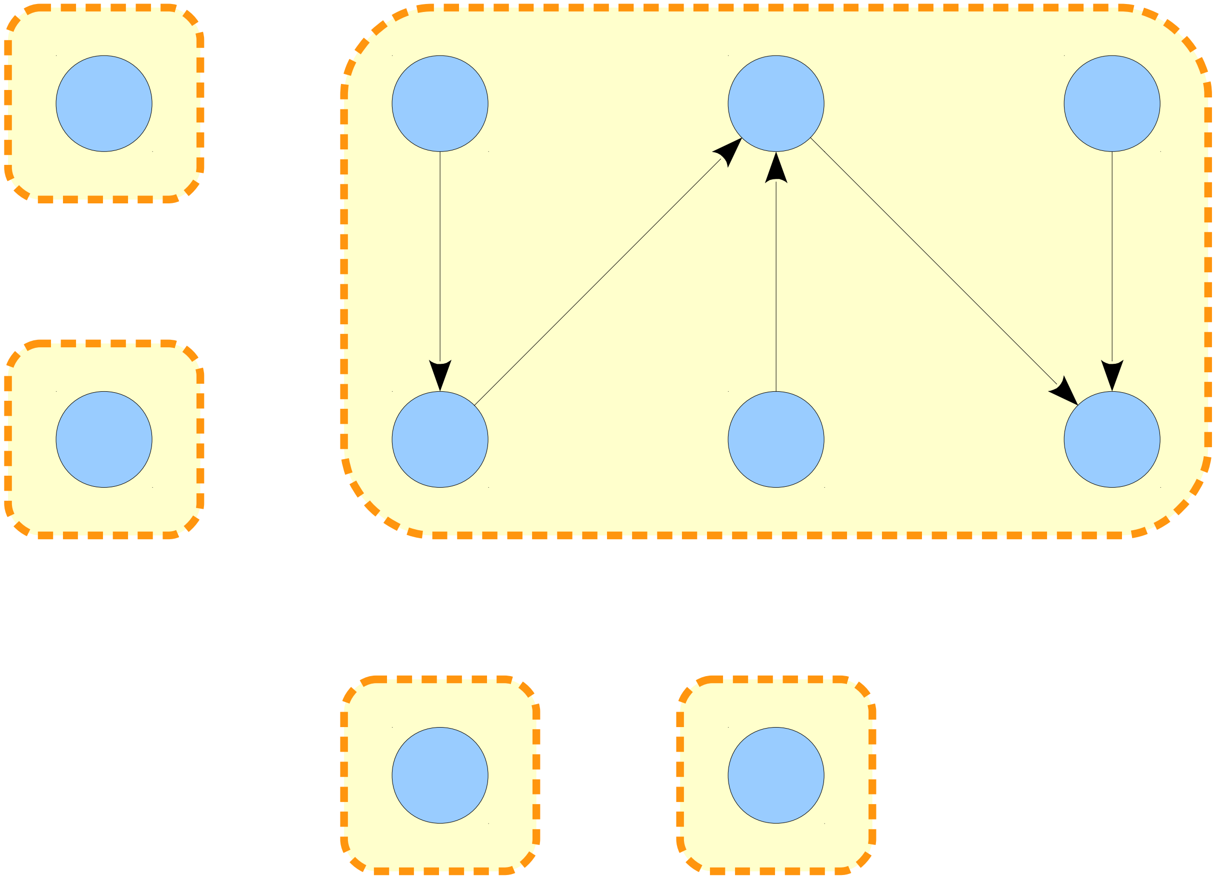
Using Representatives



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?

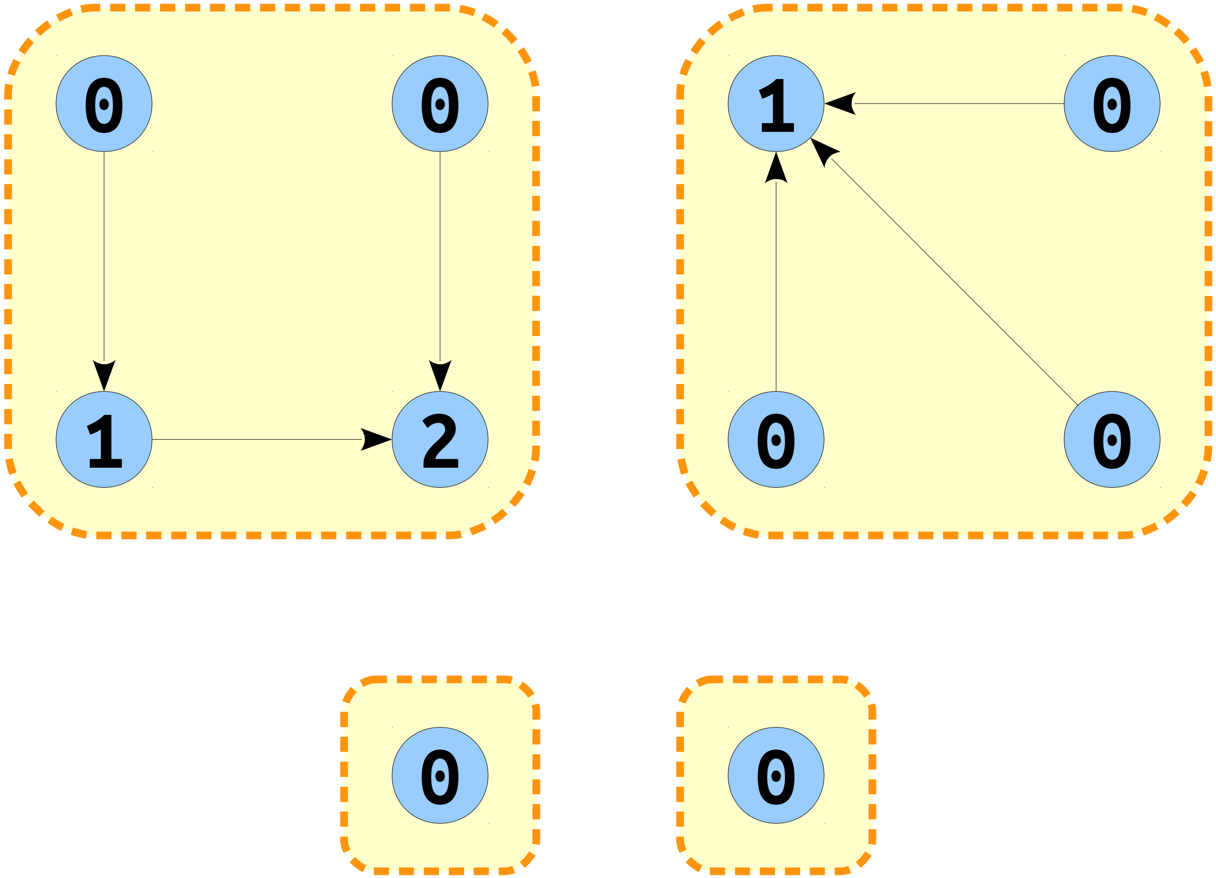
Hierarchical Representatives



Hierarchical Representatives

- 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



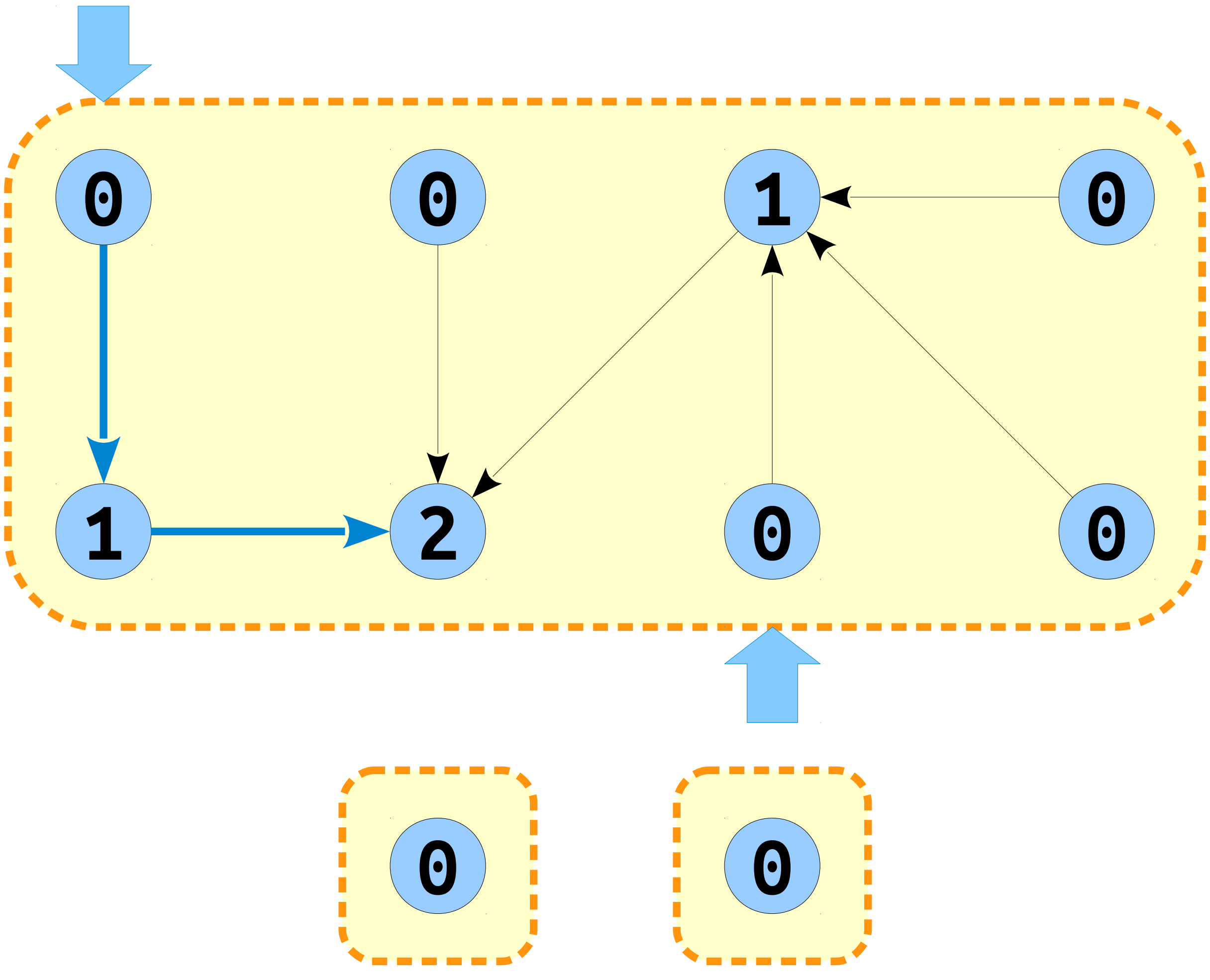
Union by Rank

- 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.

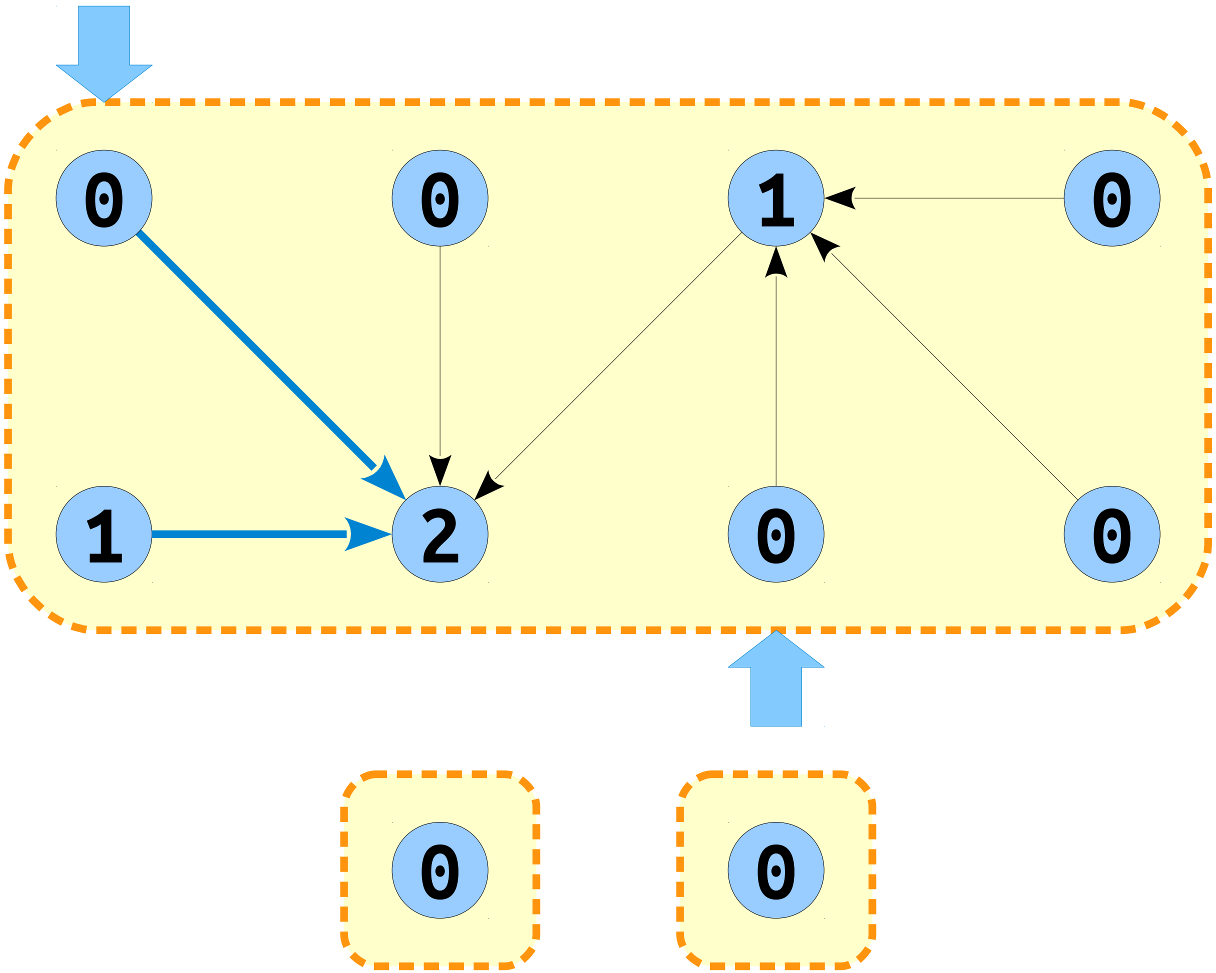
Union by Rank

- ***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.
  - 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



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?

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***.

Where We're Going

- 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 performing a ***union*** or ***find*** depends on the length of the paths followed.
- The cost of any one operation is

  $$\Theta(1 + \#\text{ptr-changes-made})$$

  because each time we visit a node that doesn't immediately point to its representative, we change where it points.
- 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).

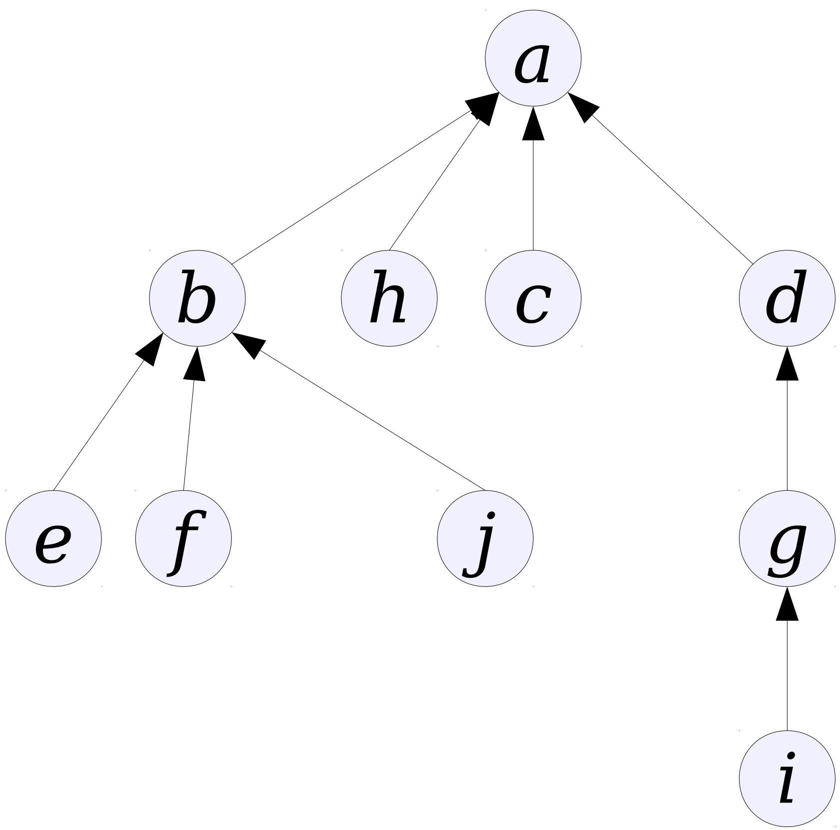
A Slight Simplification

- 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 ***union***s 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)
$f \to b$ $h \to b$ $j \to b$ $b \to a$ $h \to a$	$b \to a$ $f \to b$ $h \to b$ $j \to b$ $h \to a$

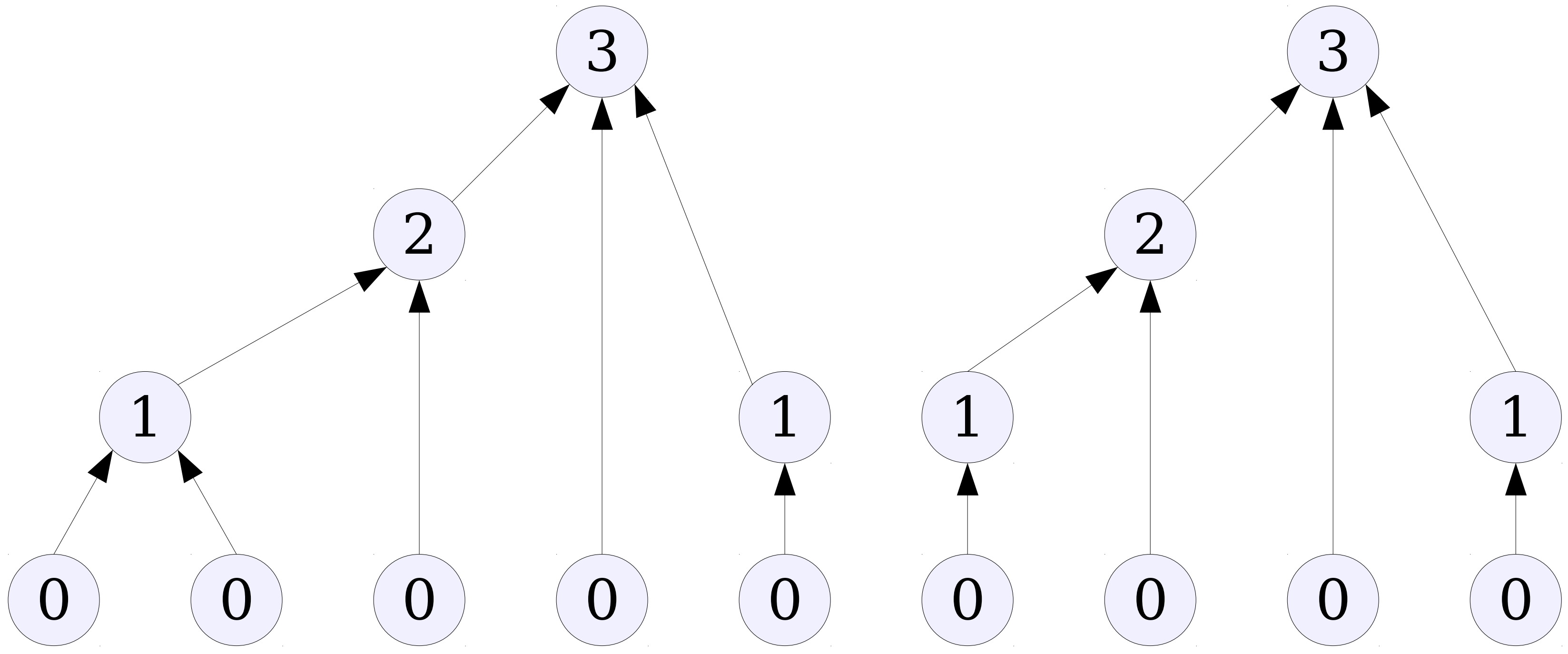
Recap: The Setup

- 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.

A Quick Initial Analysis

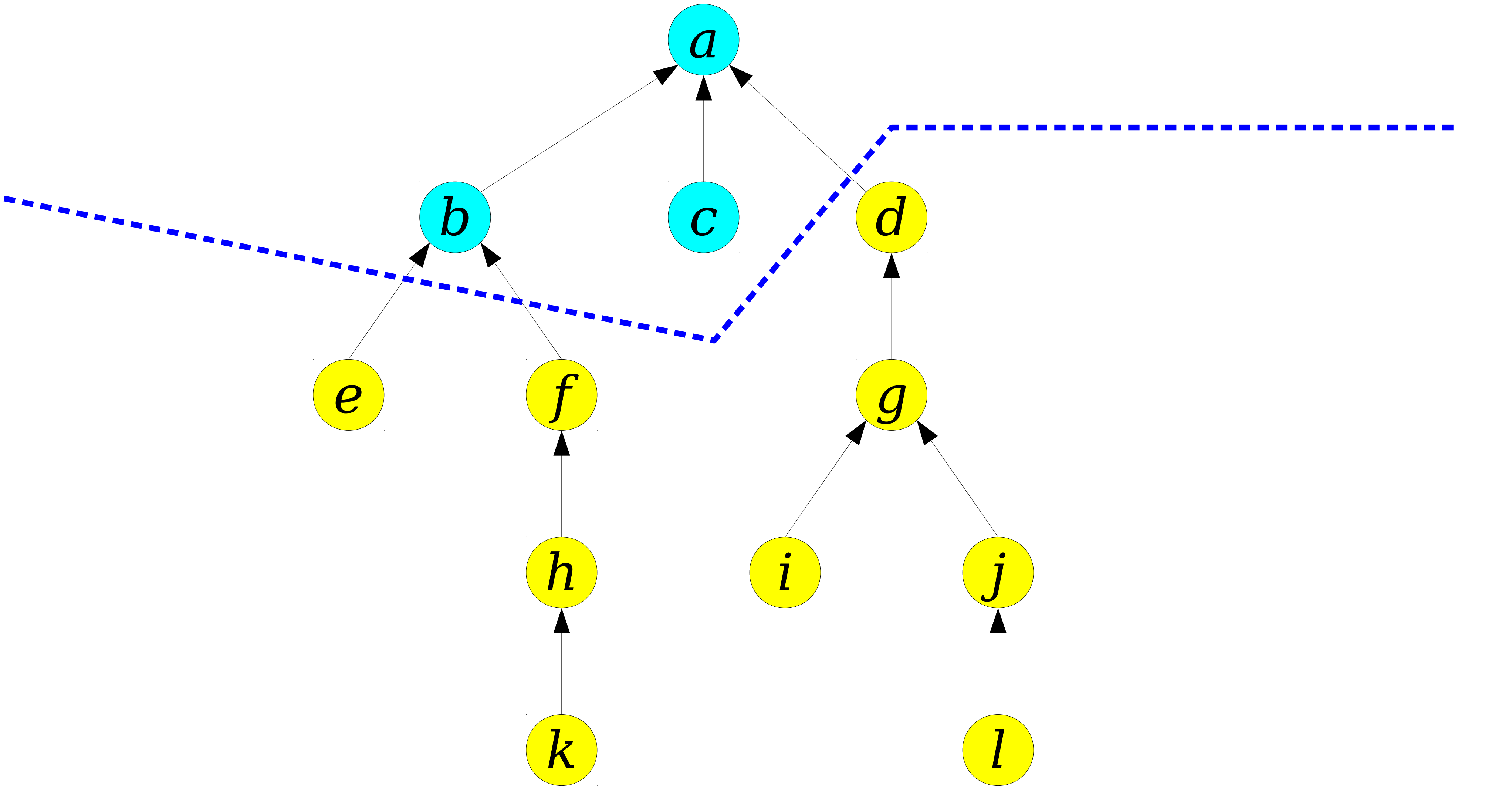
An Initial Analysis

- ***Lemma:*** Any series of m ***compress*** operation on a forest $\mathscr{F}$ with n nodes and maximum rank r makes at most nr pointer changes.
- ***Proof:*** Every time a node's representative change, the rank of that representative increases. The maximum number of times this can happen per node is r, giving an upper bound of nr. ■



The Forest-Slicing Approach

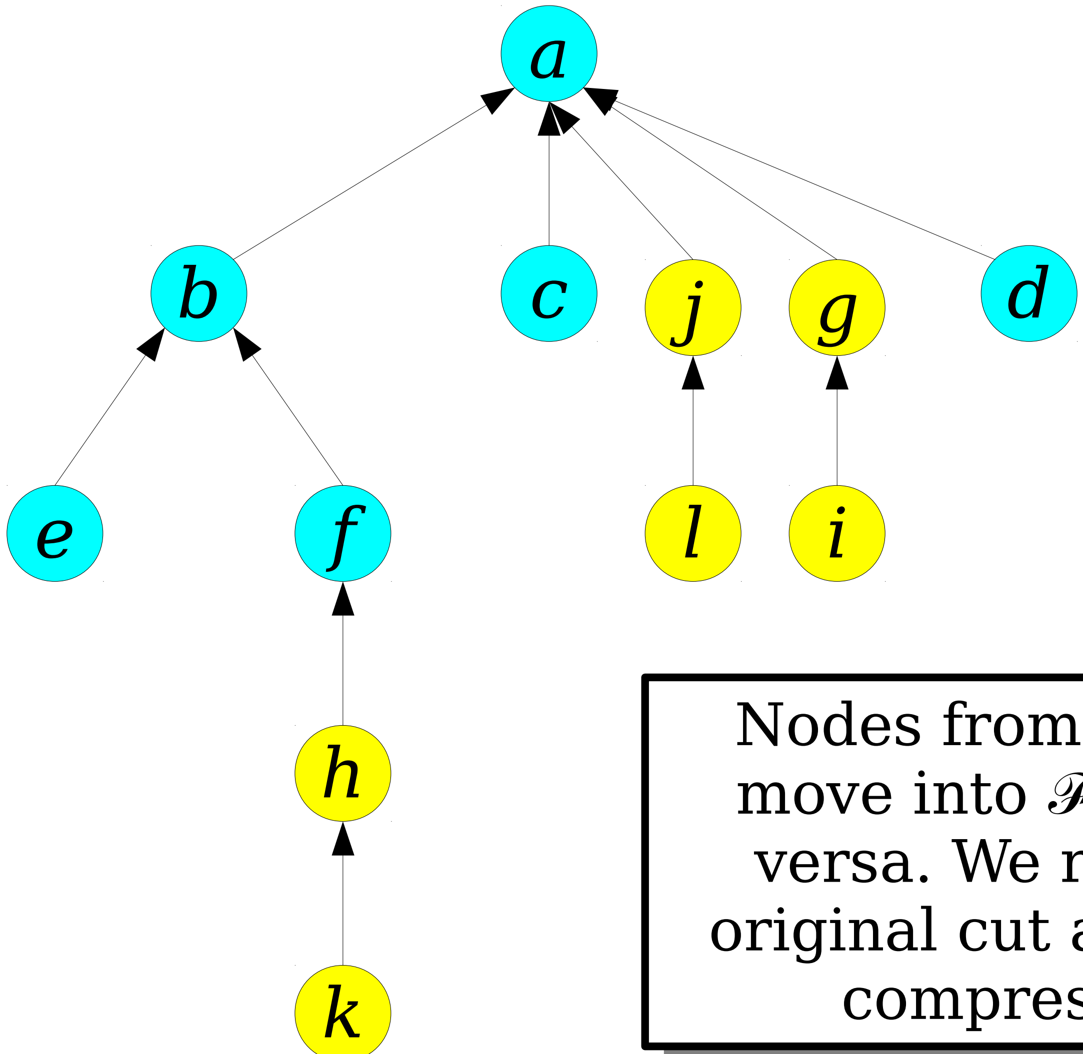
Forest-Slicing



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

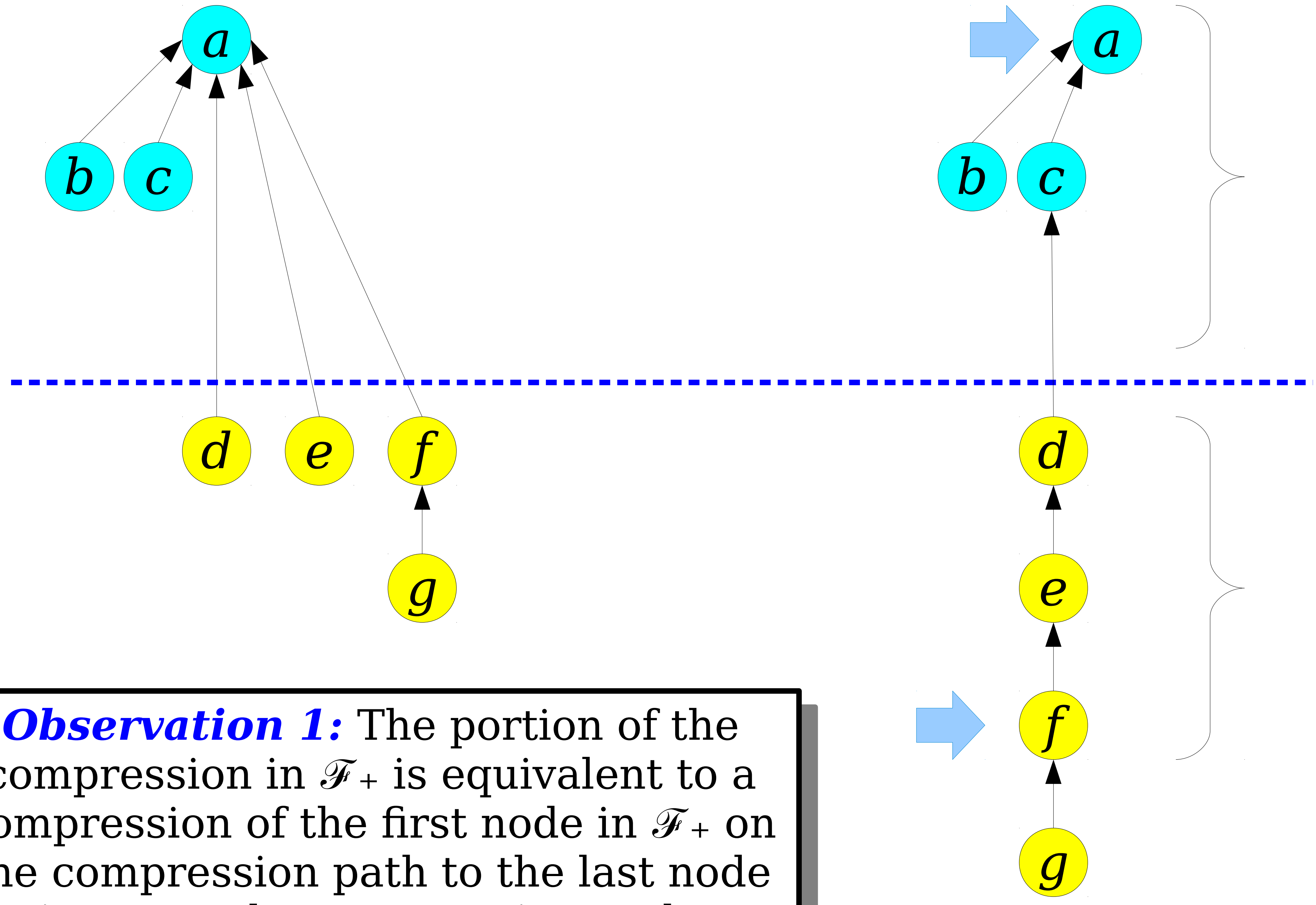


Nodes from $\mathscr{F}_-$ never move into $\mathscr{F}_+$ or vice-versa. We retain the original cut after doing compressions.

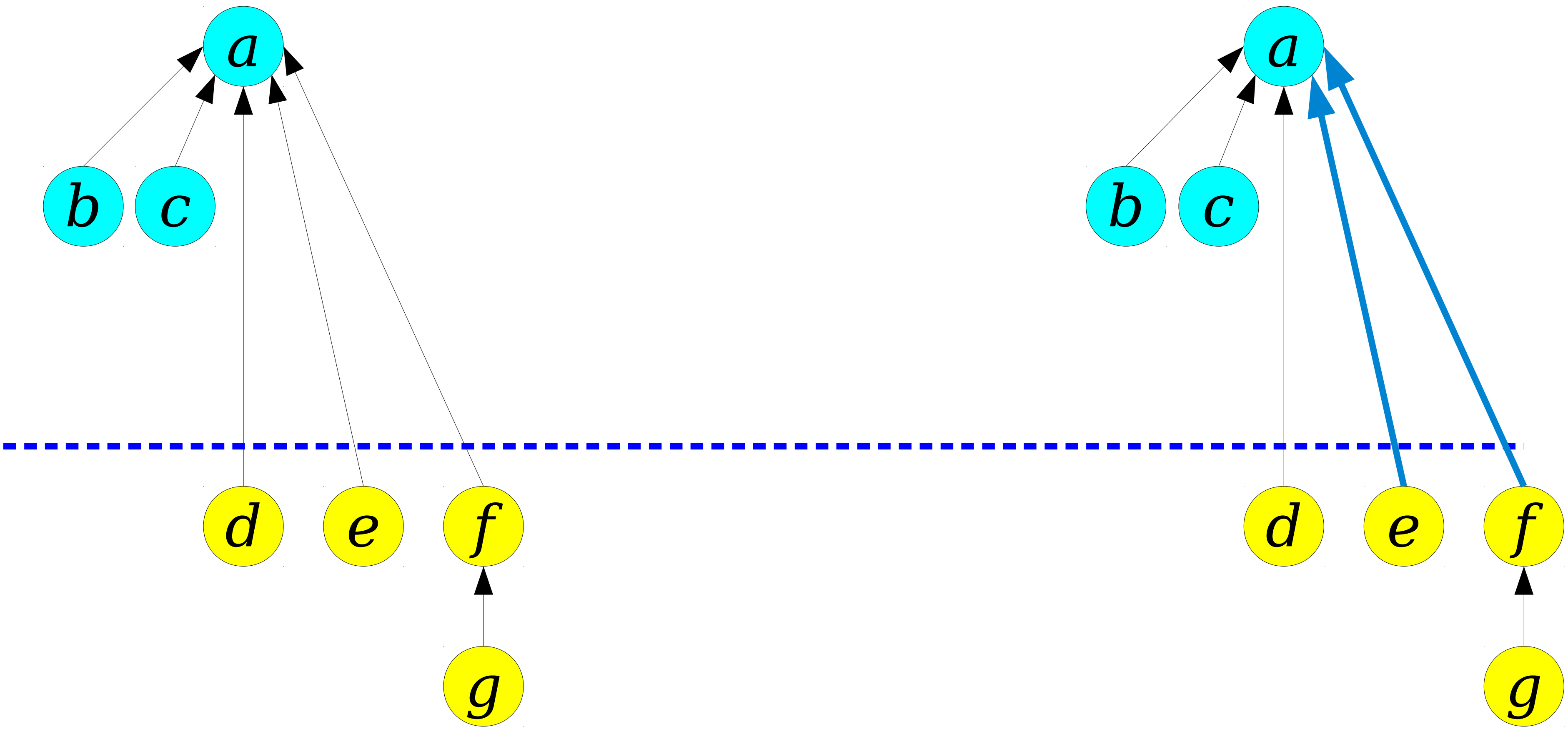
Why Slice Forests?

Forest-Slicing

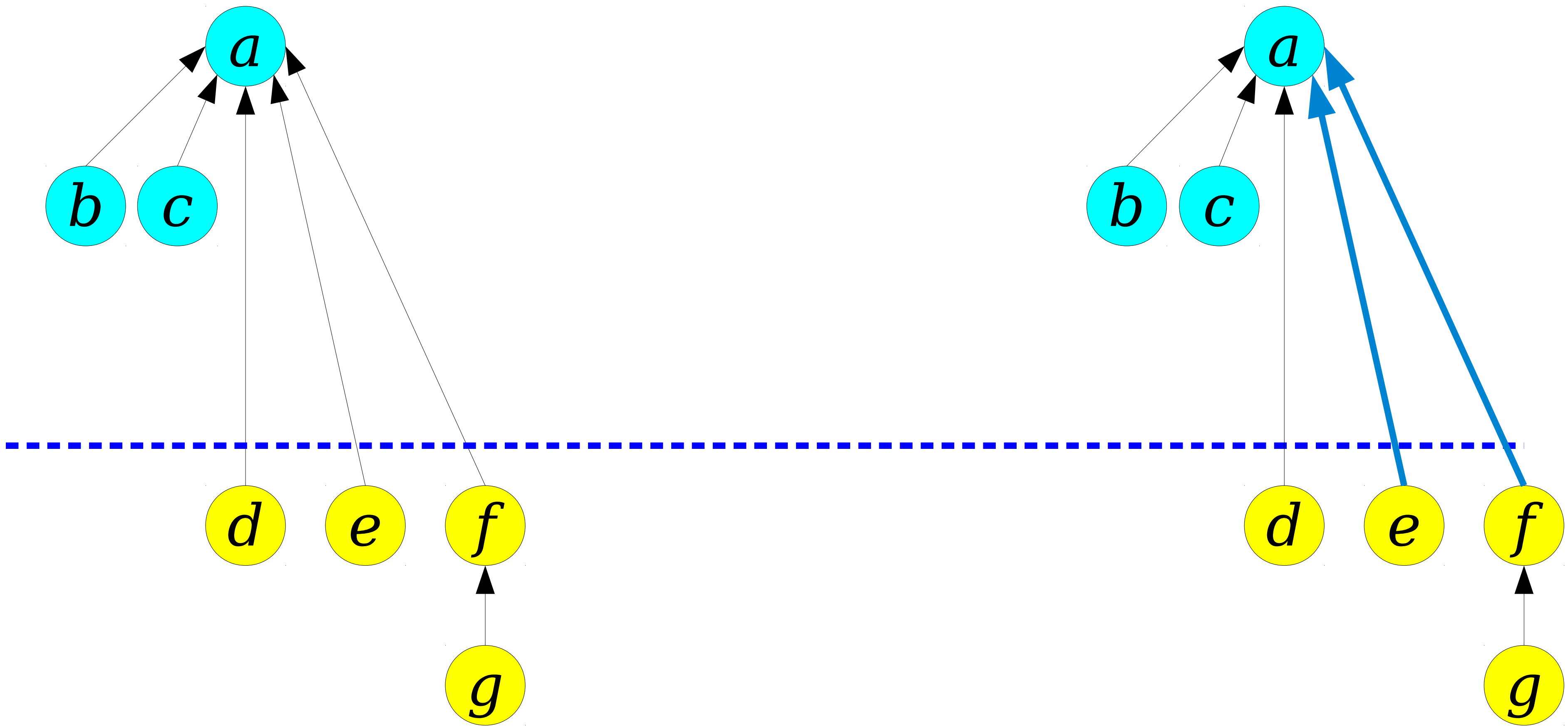
- ***Key insight:*** Each ***compress*** operation is either
  - purely in $\mathscr{F}_+$,
  - purely in $\mathscr{F}_-$, or
  - crosses from $\mathscr{F}_-$ into $\mathscr{F}_+$.
- If we can bound the cost of ***compress*** operations that cross from $\mathscr{F}_-$ to $\mathscr{F}_+$, we can try to set up a recurrence relation to analyze the cost of those ***compress***es.



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 m compressions, of which m_+ of them cross 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 $\mathscr{F}_+$;
  - 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; and
  - m_+, since each compression may change the pointer of the topmost node on the path in $\mathscr{F}_-$.

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 two sequences of compressions

- C_+, a series of m_+ compressions purely in $\mathscr{F}_+$; and
- C_-, a series of m_- compressions purely in $\mathscr{F}_-$,

such that

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

Compressions that appear purely in $\mathscr{F}_+$ or purely in $\mathscr{F}_-$, plus the tops of crossing compressions.

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!

The midterm is tonight
from 7PM – 10PM
in room 320-105.

Good luck!

Back to CS166!

The Main Analysis

Where We Are

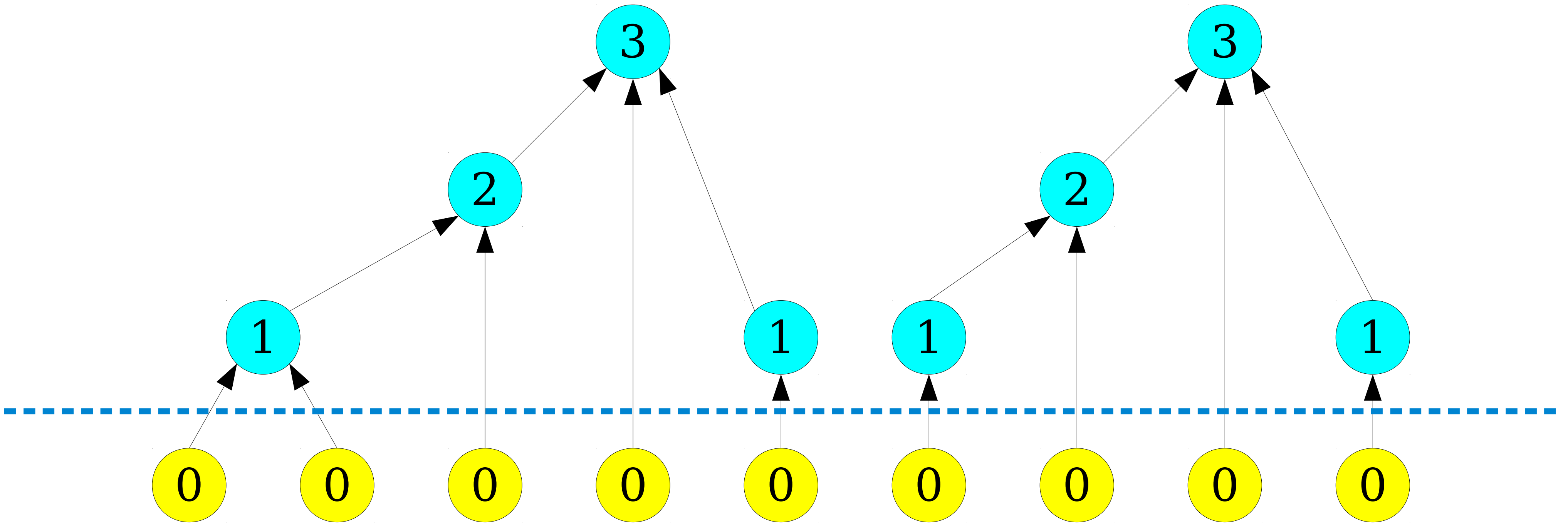
- We now have a sort of recurrence relation for evaluating the runtime of a series C of m ***compress***es on an n-node forest $\mathscr{F}$ sliced into $\mathscr{F}_+$ and $\mathscr{F}_-$:

$$\text{cost}(C) \le \text{cost}(C_+) + \text{cost}(C_-) + n + m_+$$

- This recurrence relation assumes that we already know how we've sliced $\mathscr{F}$ into $\mathscr{F}_+$ and $\mathscr{F}_-$.
- To complete the analysis, we're going to need to precisely quantify what happens if we slice the forest in a number of different ways.

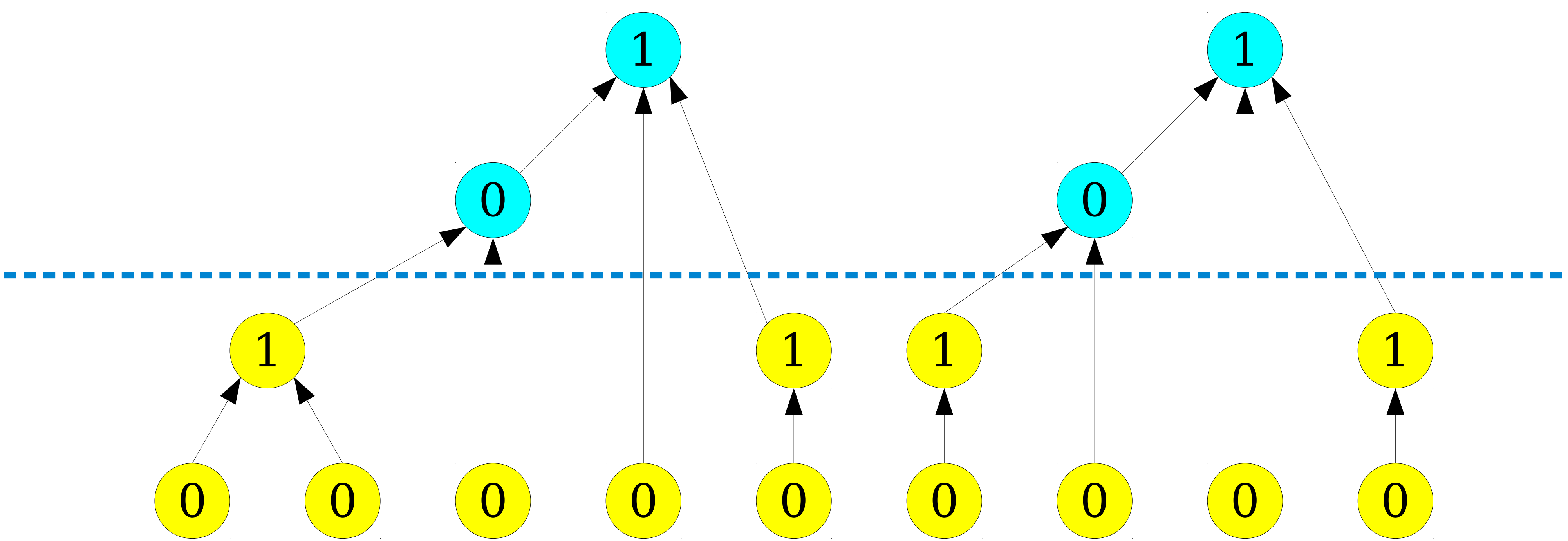
Natural Slices

- One "natural" way to slice a forest $\mathscr{F}$ into $\mathscr{F}_+$ and $\mathscr{F}_-$ is to pick some threshold rank. We then choose $\mathscr{F}_+$ to be all the nodes whose rank is above the threshold and $\mathscr{F}_-$ to be all the other nodes.



Natural Slices

- If our initial forest has maximum rank r and we slice the forest at rank r', the bottom forest has maximum rank r' and the top forest is (essentially) a forest of rank $r - r'$.



Slicing our Forest

- Imagine that we have our forest $\mathscr{F}$ of maximum rank r.
- Suppose we cut slice the forest into $\mathscr{F}_+$ and $\mathscr{F}_-$ at some rank r'.
- We know that

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

- Let's investigate $\text{cost}(C_+)$ and $\text{cost}(C_-)$ independently.

The Top Forest

- Let's begin by thinking about cost(C_+), the cost of compresses in the top forest $\mathscr{F}_+$.
- ***Recall:*** $\mathscr{F}_+$ consists of all nodes of rank r' or higher.
- Intuitively, we'd expect there to not be "too many" nodes in the top forest, since it's exponentially harder to get nodes of progressively harder orders.
- Using our lemma from before, we know that there can be at most $\boldsymbol{n / 2^{r'}}$ nodes in $\mathscr{F}_+$.
- Therefore, using our (weak) bound from before, we see that

$$\text{cost}(C_+) \leq \boldsymbol{nr / 2^{r'}}.$$

Slicing our Forest

- Imagine that we have our forest $\mathscr{F}$ of maximum rank r.
- Suppose we cut slice the forest into $\mathscr{F}_+$ and $\mathscr{F}_-$ at some rank r'.
- We know that

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

- Therefore

$$\text{cost}(C) \leq nr \,/\, 2^{r'} + \text{cost}(C_-) + n + m_+.$$

- Let's now go investigate $\text{cost}(C_-)$.

Improving our Recurrence

$$\text{cost}(C) \leq nr \,/\, 2^{r'} + \text{cost}(C_-) + n + m_+.$$

- Notice that $\text{cost}(C)$ is the cost of
  - doing m ***compress***es,
  - in an n-node forest, with
  - maximum rank r.
- We now have $\text{cost}(C_-)$, which is the cost of
  - doing m_- ***compress***es,
  - in a forest with at most n nodes, with
  - maximum rank r'.
- Let's make these dependencies more explicit.

Improving our Recurrence

$$\text{cost}(C) \le nr \,/\, 2^{r'} + \text{cost}(C_-) + n + m_+.$$

- Define $\text{T}(m, n, r)$ to be the cost of
  - performing m ***compress*** operations,
  - in a forest of at most n nodes, where
  - the maximum rank is r.
- The above recurrence can be rewritten as

$$\text{T}(m, n, r) \le \text{T}(m_-, n, r') + nr \,/\, 2^{r'} + n + m_+$$

- Now, we "just" need to solve this recurrence. Don't worry… it's not too bad!

Finalizing our Recurrence

$$T(m, n, r) \le T(m_-, n, \boldsymbol{r'}) + \boldsymbol{nr / 2^{r'}} + n + m_+$$

- The above recurrence is dependent on having a choice of r' based on our choice of r.
- If we make r' too large, then the recurrence relation takes too long to bottom out and we'll expect a higher runtime.
- If we make r' too small, the $nr / 2^{r'}$ term will be too large and our analysis won't be tight.
- How do we balance these terms out?

Finalizing our Recurrence

$$T(m, n, r) \le T(m_-, n, r') + nr / 2^{r'} + n + m_+$$

- ***Idea:*** Choose $r' = \lg r$. Then

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

- Imagine that this recurrence expands out L times before it bottoms out. Think about what happens:
  - The $2n$ term gets summed in L times.
  - The m_+ term – the number of compresses in the top forest – sums up to at most m across all compressions.
- Overall, we get $T(m, n, r) \le \mathbf{2nL + m}$.

Iterated Logarithms

- We now have

$$T(m, n, r) \leq 2nL + m.$$

- The quantity L represents the number of layers in the recurrence, and at each step we have r dropping to lg r.
- The ***iterated logarithm***, denoted **lg* *n***, is the number of times we can apply lg to n before it drops to some constant (say, 2). Therefore:

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

- And since the maximum rank is at most lg n, we see that the cost of performing m operations on an n-node forest is $\mathbf{O(n \lg^* n + m)}$.

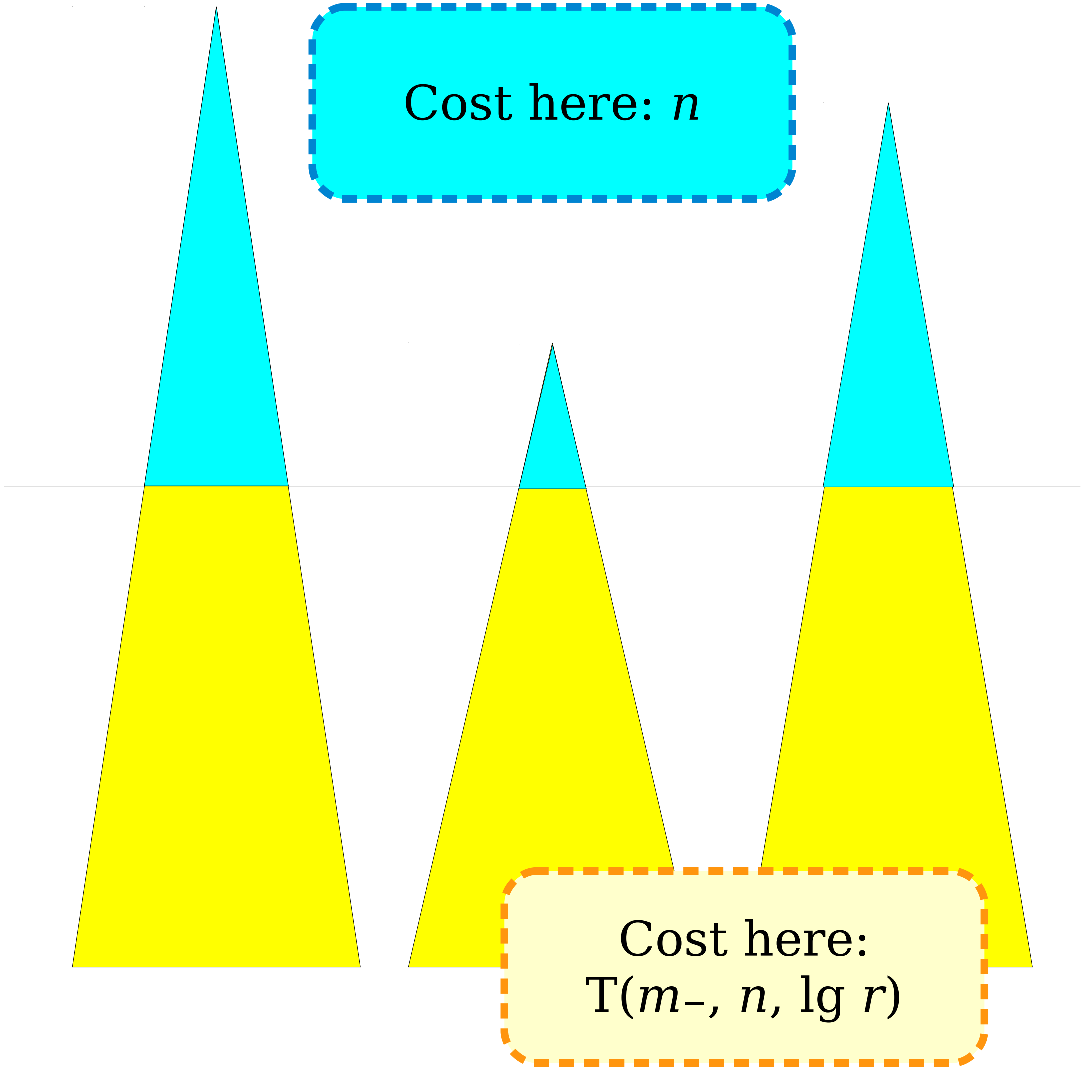
Iterated Logarithms

- The function lg n is the inverse of the function 2^n; that is, $2 \times 2 \times \ldots \times 2$, n times.
- The ***tetration*** operation, denoted $^{n}2$, is given by $^{n}2 = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$, with n copies of 2 in the tower of exponents. It grows *extremely* quickly!
- The function lg* n is the inverse of tetration. It grows *extremely* slowly!
- ***Useful fact:*** lg* $n \leq 5$ for any n less than or equal to the number of atoms in the universe.

Our Strategy

- Let's recap, how we got here.
- We begin with a forest $\mathscr{F}$ of maximum rank r.
- We sliced $\mathscr{F}$ at rank lg r.
- We (directly) obtained a weak bound on the cost of the compressions in the (small) forest $\mathscr{F}_+$.
- We recursively obtained a (good) bound on the cost of the compressions in the (larger) forest $\mathscr{F}_-$.
- We solved the recurrence to get the bound

$$\mathbf{T(\mathit{m}, \mathit{n}, \mathit{r}) \leq 2\mathit{n} \lg^* \mathit{r} + \mathit{m}}.$$

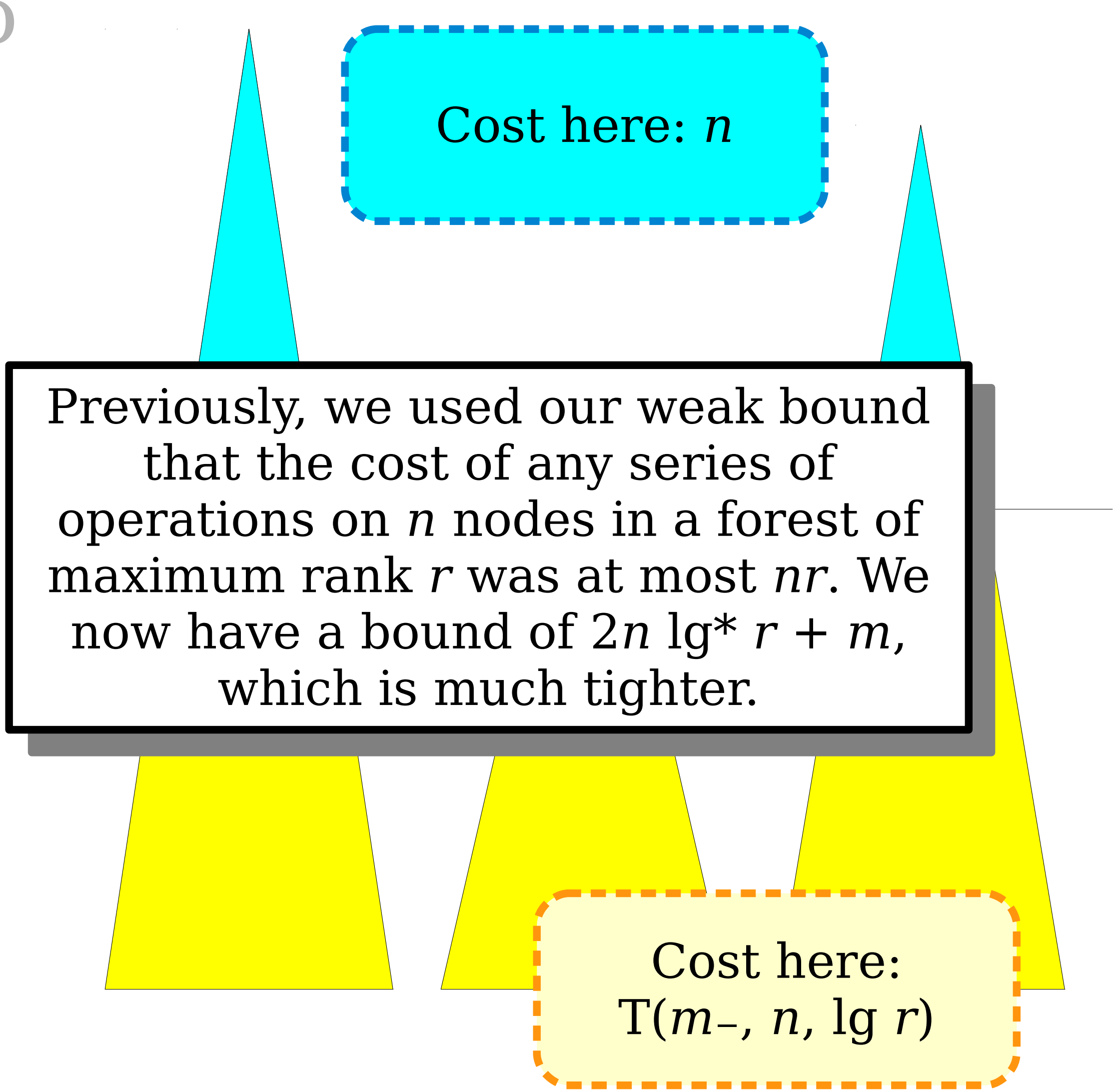


Our Strategy

What could we do to tighten the runtime bound?

- ***Option 1:*** Tighten the bound on the cost of the top forest.

Option 2: Slice the forest even lower to make the recursion tree shorter.

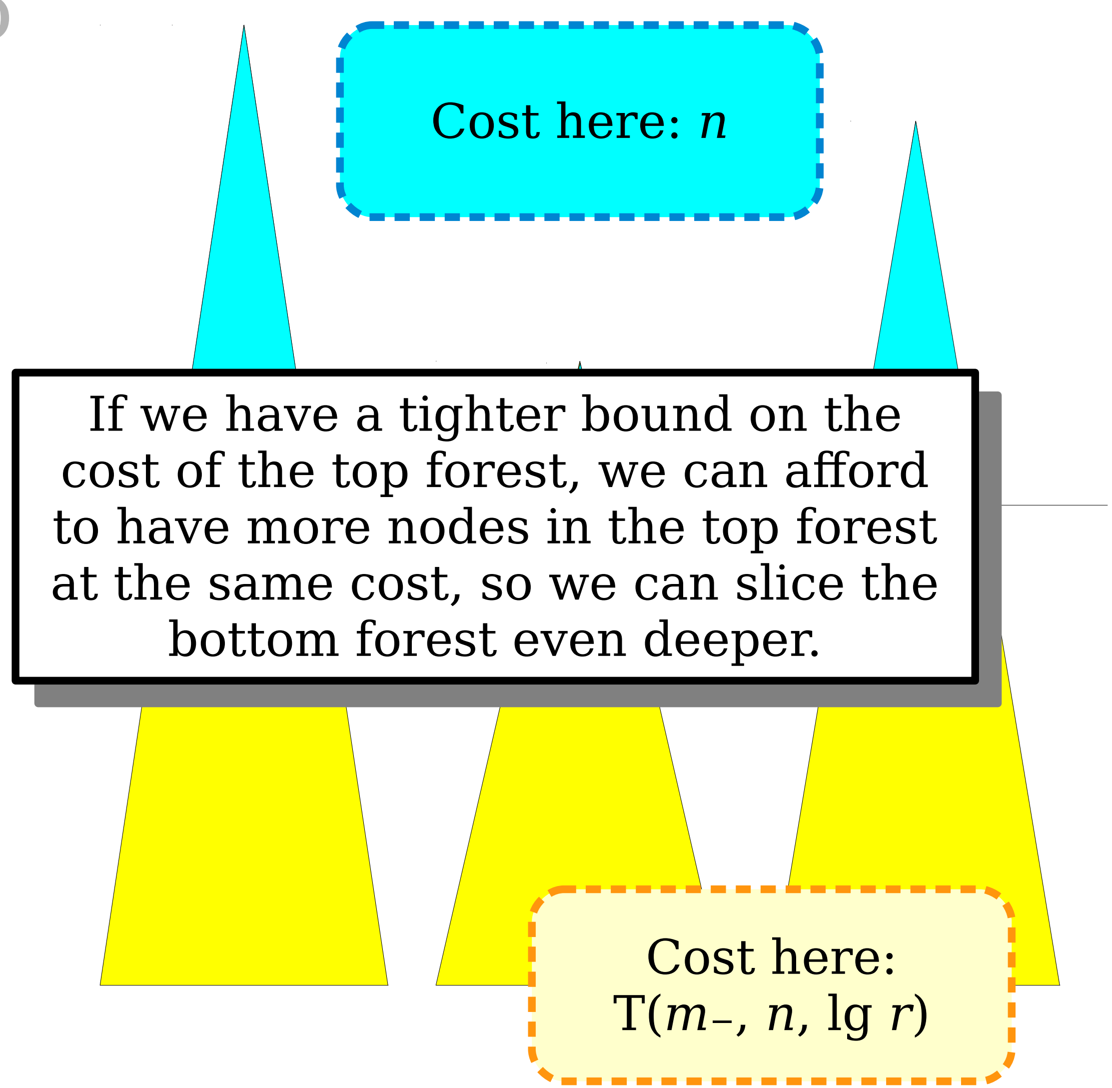


Our Strategy

What could we do to tighten the runtime bound?

Option 1: Tighten the bound on the cost of the top forest.

- ***Option 2:*** Slice the forest even lower to make the recursion tree shorter.



Slicing our Forest, Again

- Imagine that we have a forest $\mathscr{F}$ of maximum rank r.
- Suppose we cut slice the forest into $\mathscr{F}_+$ and $\mathscr{F}_-$ at some rank r'.
- We know that

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

- Therefore

$$\text{T}(m, n, r) \leq \text{cost}(C_+) + \text{T}(m, n, r') + n + m_+.$$

- Let's investigate $\text{cost}(C_+)$ using our previous analysis.

The Top Forest

- ***Lemma:*** In an n-node forest $\mathscr{F}$ of maximum rank r, if we split $\mathscr{F}$ into $\mathscr{F}_+$ and $\mathscr{F}_-$ by cutting the forest at rank r', then cost(C_+) $\leq 2n \lg^* r / 2^{r'} + m_+$.
- ***Proof:*** There are $n / 2^{r'}$ nodes in this forest and the maximum rank is at most r. The cost of performing m_+ compress operations here is therefore

$$2(n / 2^{r'}) \lg^* r + m_+.$$

- ***Observation:*** Our previous bound was

$$rn / 2^{r'}.$$

  We previously set $r' = \lg r$ because that was as low as we could go without cost(C_+) being too high. With our new bound, we can afford to make r' much lower.

Our Recurrence

- We had

$$T(m, n, r) \leq \text{cost}(C_+) + T(m_-, n, r') + n + m_+.$$

- So we now have

$$T(m, n, r) \leq T(m_-, n, r') + 2n \lg^* r \,/\, 2^{r'} + n + 2m_+.$$

- Previously, we picked $r' = \lg r$ and ended up with a bound in terms of $\lg^* r$.
- Now, we pick $r' = \lg^* r$. Then we have

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

- Using a similar analysis as before, if L is the number of layers in the recurrence, this solves to

$$T(m, n, r) \leq \mathbf{2nL + 2m}.$$

Iterated Iteration

- We have

$$T(m, n, r) \le \mathbf{2\boldsymbol{nL} + 2\boldsymbol{m}},$$

  where L is the number of layers in the iteration.
- At each step, we shrink r to lg* r. The maximum number of times we can do this is denoted **lg** *r***, so we have

$$T(m, n, r) \le 2n \lg^{**} r + 2m.$$

- So the cost of any m operations is $\mathbf{O(\boldsymbol{n} \lg^{**} \boldsymbol{n} + \boldsymbol{m})}$.

Iterated Iterated Logarithms

- The ***pentation*** operation is next in the family of fast-growing functions.
- Just as tetration is iterated exponentiation, pentation is iterated tetration, so 2 pentated to the *n*th power, denoted ${}_n2$, is

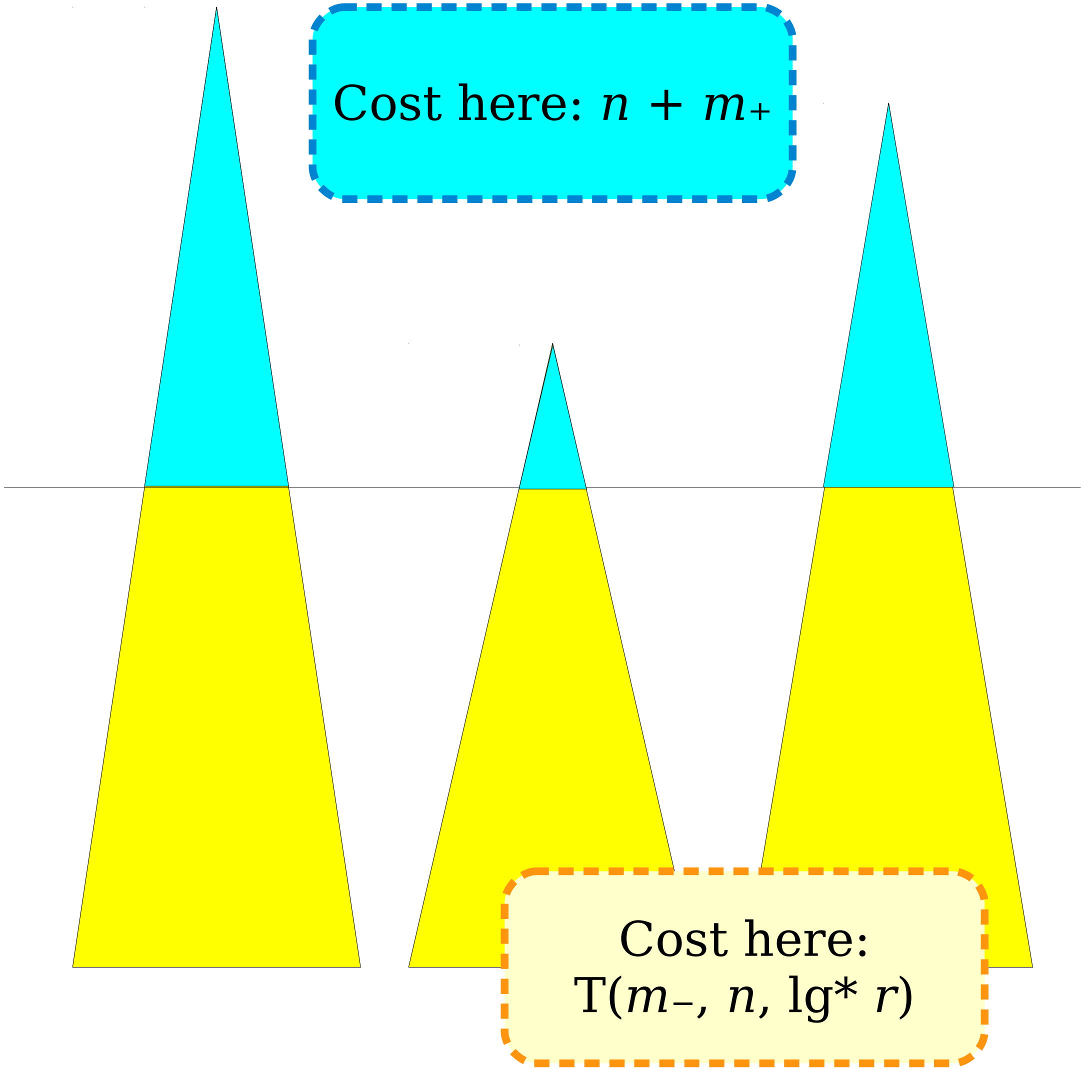
$$\left(2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\right)^{\cdot^{\cdot^{\cdot^{\left(2^{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\right)}}}}$$

  where there are ${}^n2$ copies of the exponential towers.
- The function lg** n is the inverse of pentation. It grows *unbelievably* slowly!

Our Strategy

- Let's recap, how we got here.
- We begin with a forest $\mathscr{F}$ of maximum rank r.
- We sliced $\mathscr{F}$ at rank lg* r.
- We (directly) obtained a weak bound on the cost of the compressions in the (small) forest $\mathscr{F}_+$.
- We recursively obtained a (good) bound on the cost of the compressions in the (larger) forest $\mathscr{F}_-$.
- We solved the recurrence to get the bound

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

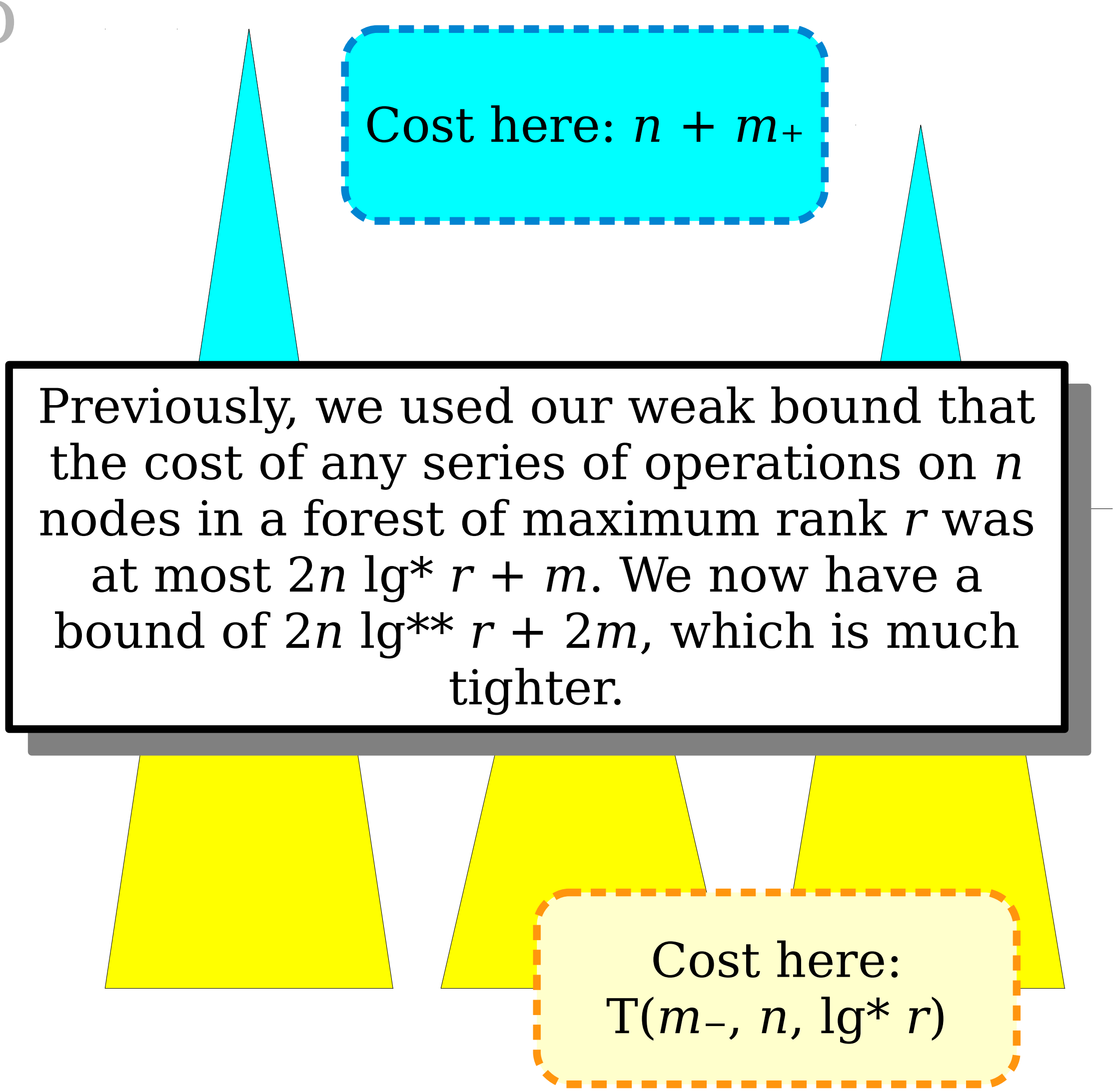


Our Strategy

What could we do to tighten the runtime bound?

- ***Option 1:*** Tighten the bound on the cost of the top forest.

Option 2: Slice the forest even lower to make the recursion tree shorter.

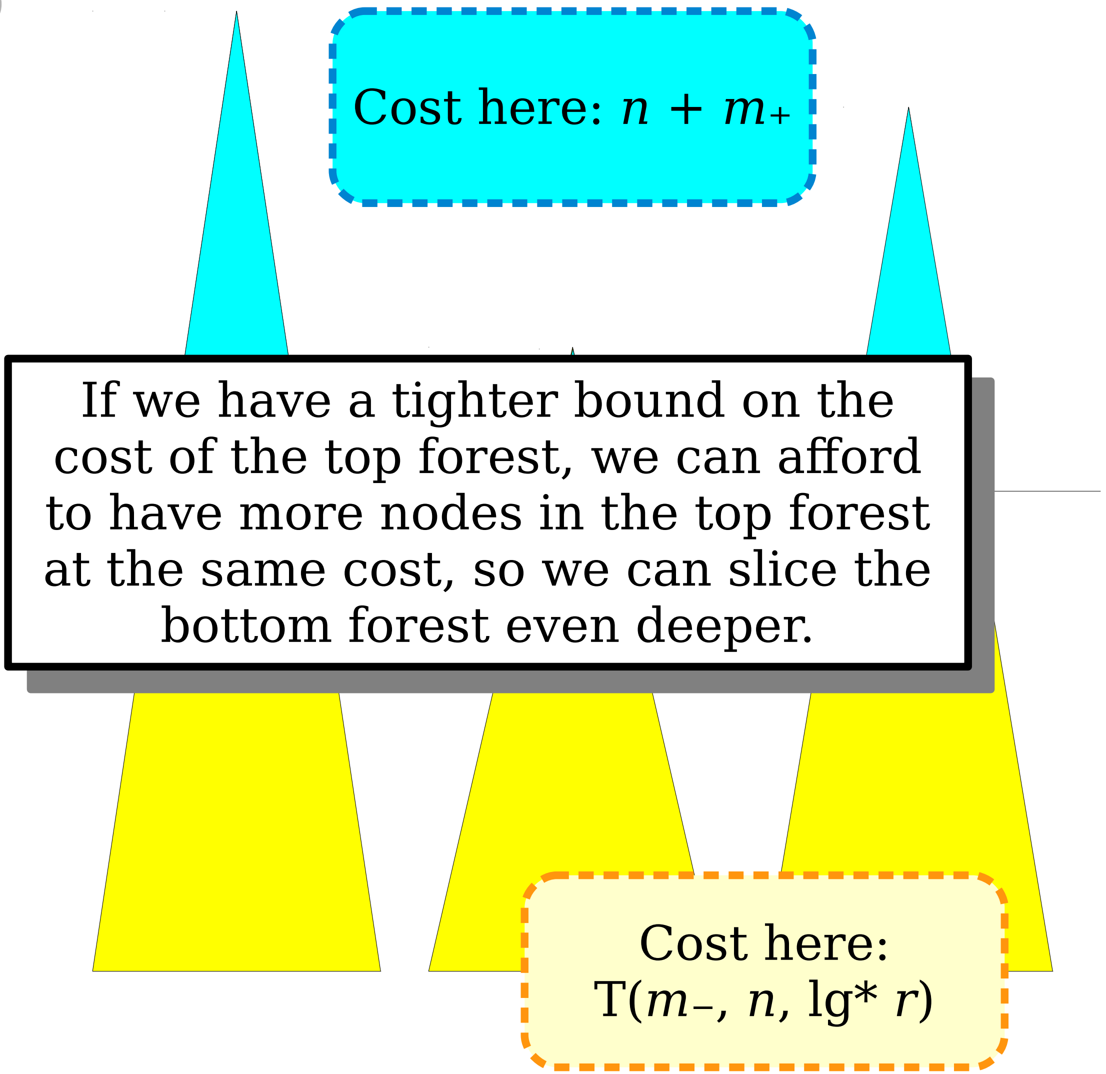


Our Strategy

What could we do to tighten the runtime bound?

Option 1: Tighten the bound on the cost of the top forest.

- ***Option 2:*** Slice the forest even lower to make the recursion tree shorter.



The Feedback Lemma

- ***Lemma:*** Suppose we know that

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

  Then

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

- ***Proof:*** Induction! Use the previous proof as a template: split the forest at rank $\lg^{*(k)} r$, use the known bound to bound the cost of the top forest, and use recursion to bound the cost of the bottom forest. ■

The Final Steps

- For any $k \in \mathbb{N}$, we have

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

- We can upper-bound r at $\log n$, so we have

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

- As n gets larger and larger, we can increase the value of k to make the $\lg^{*(k)} n$ term at most some constant value.
- ***Question:*** What is that k, as a function of n?
- The ***Ackermann inverse function***, denoted $\boldsymbol{\alpha(n)}$, is

$$\alpha(n) = \min\{\ k \in \mathbb{N} \mid \lg^{*(k)} n \leq 3\ \}$$

- ***Theorem:*** The cost of performing any m operations on any n-node disjoint set forest using union-by-rank and path compression is $\mathbf{O(n + m\alpha(n))}$.

Intuiting α(n)

- Imagine we want to define some function A such that
  - $A(n, 0) = 2$
  - $A(n, 1) = 2 + 2 + \ldots + 2 = 2n$
  - $A(n, 2) = 2 \times 2 \times \ldots \times 2 = 2^n$.
  - $A(n, 3) = 2^{2^{\cdots^{2}}} = {}^{n}2$. (tetration)
  - $A(n, 4) = {}^{{}^{2\cdots 2}}2 = {}_{n}2$. (pentation)
  - $A(n, 5)$ doesn't have a name, but scares children.
- The function A is called an ***Ackermann-type function***. There are a number of different functions in this family, but they all (fundamentally) apply higher and higher orders of functions to the arguments.

Intuiting α(n)

- ***Theorem:*** Asymptotically, the function α(n) is the inverse of $A(n, n)$, hence the name “Ackermann inverse”.
- ***Intuition:***
  - lg n is the inverse of 2^n, which is $A(n, 2)$.
  - lg* n is the inverse of ${}^{n}2$ (tetration), which is $A(n, 3)$.
  - lg** is the inverse of ${}_{n}2$ (pentation), which is $A(n, 4)$.
  - α(n) tells you how many stars you need to make $\lg^{*(k)} n$ drop to a constant, which essentially asks for which essentially asks for what order of operation you need to invert.
- This function grows more slowly than *any* of the iterated logarithm families. It's so slowly-growing that an input to it that would make it more than, say, 10 can't even be expressed without inventing special notation for fast-growing numbers.

Intuiting α(n)

- If you keep dividing by two, you should expect a log term.
- If you keep taking logs, you should expect a log* term.
- If you keep taking log*s, you should expect a log** term.
- If you keep adding stars to your logs, you should expect an α term.

Some Notes on $\alpha(n)$

- The term $\alpha(n)$ arises in many different algorithms:
  - Range semigroup queries: there's a lower bound of $\alpha(n)$ on the cost of a query under certain algebraic assumptions.
  - Minimum spanning trees: the fastest known deterministic MST algorithm runs in time $O(m\alpha(n))$ due to a connection to the above topic.
  - Splay trees: imagine you treat a splay tree as a deque. Hilariously, the best bound we have on the runtime of performing n deque operations is $O(n\alpha^*(n))$. It's suspected to be $O(n)$, but this hasn't been proven.
- $\alpha(n)$ and its variants are the slowest-growing functions that are routinely encountered in algorithms and data structures. ***And now you know where it comes from!***

Next Time

- ***Euler Tour Trees***
  - Fully dynamic connectivity in forests.
- ***Dynamic Graphs***
  - Fully dynamic connectivity in general graphs (ITA).