Dynamic Connectivity

Outline for Today

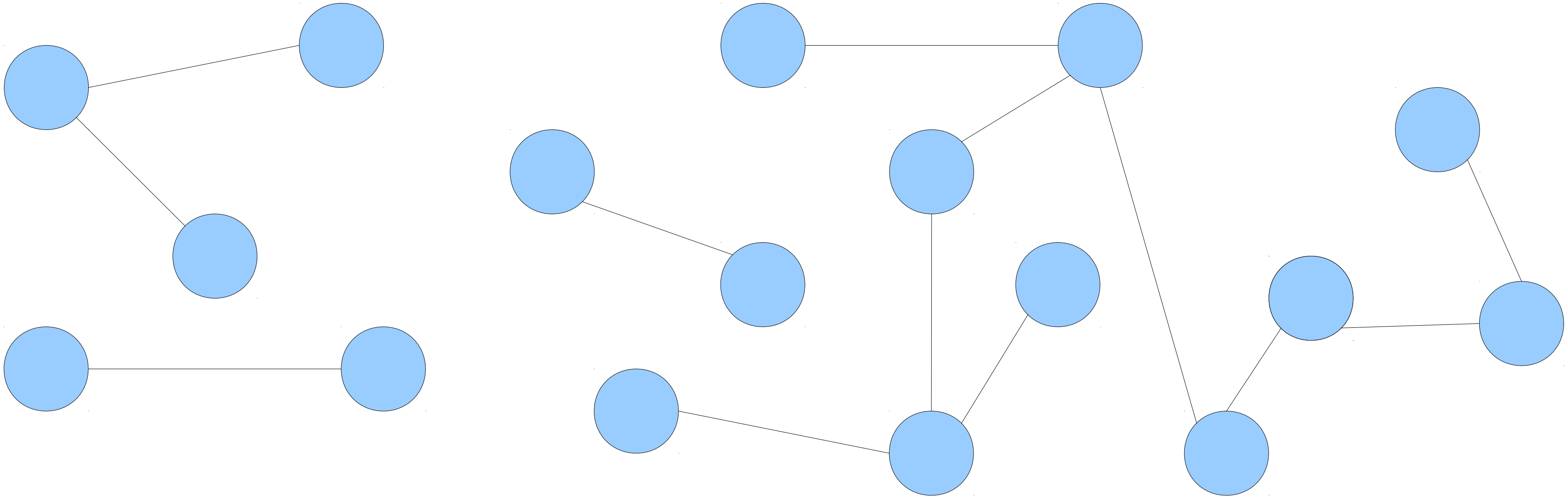
- ***Euler Tour Trees***
  - A data structure for dynamic connectivity in forests.
- ***Dynamic Graphs***
  - A data structure for dynamic connectivity in arbitrary undirected graphs.

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?"

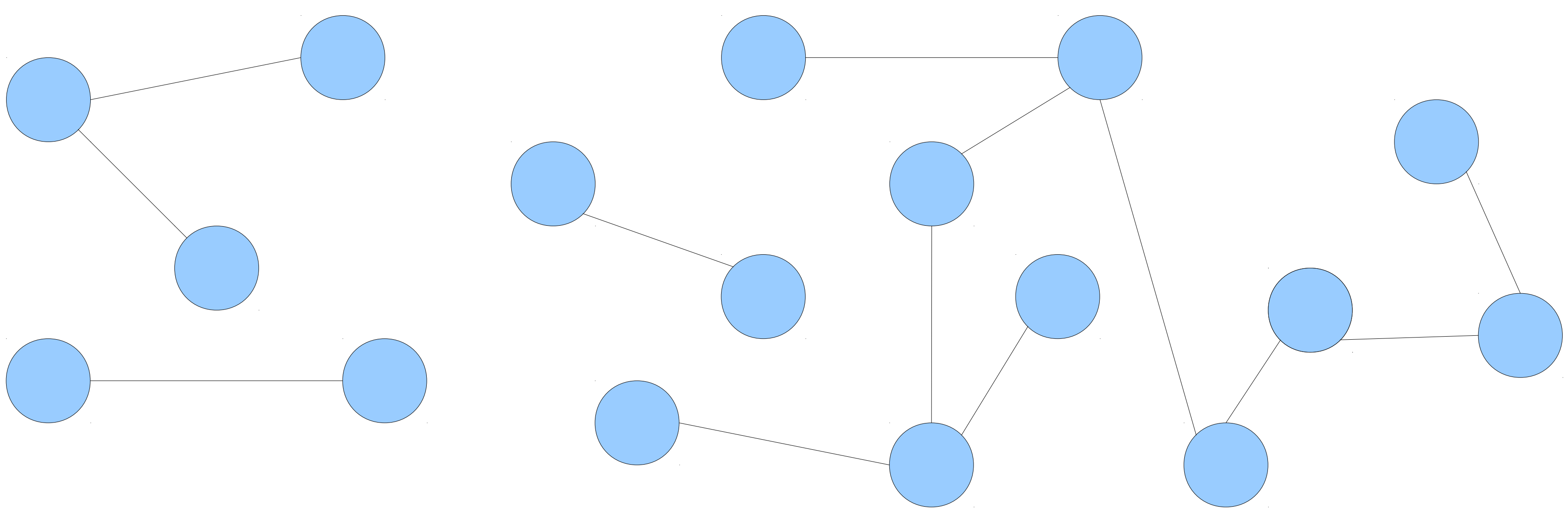


Dynamic Connectivity

- 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!

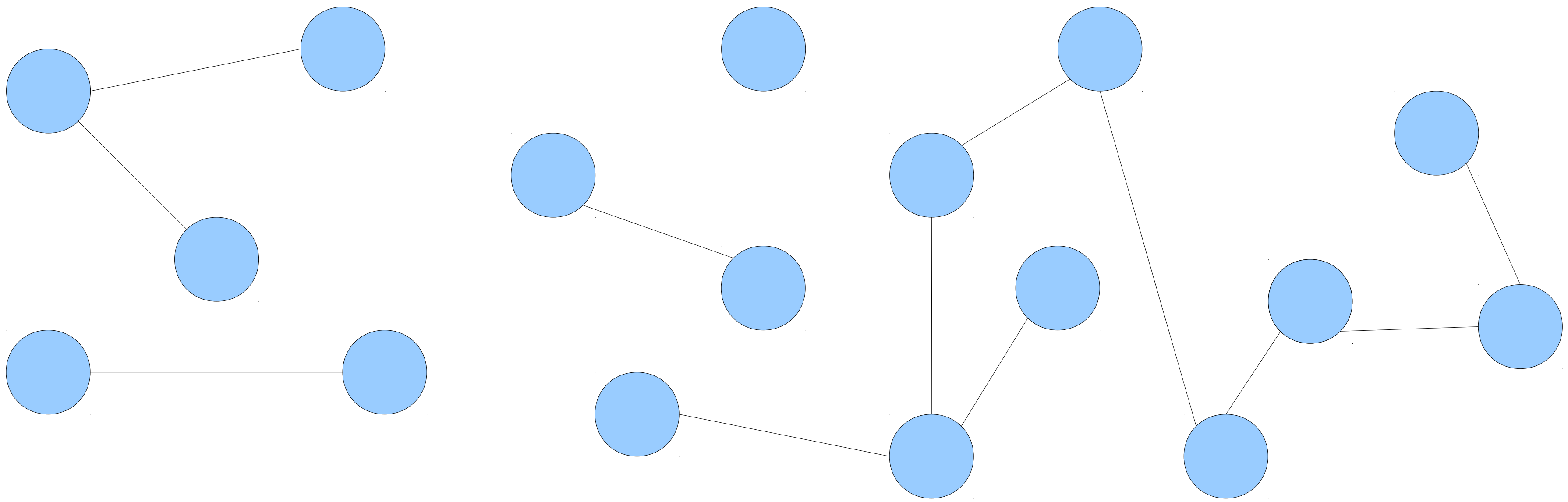


Dynamic Connectivity in Forests

- Consider the following special-case of the dynamic connectivity problem:

Maintain an undirected ***forest*** *F* so that edges may be inserted an deleted and connectivity queries may be answered efficiently.

- Each deleted edge splits a tree in two; each added edge joins two trees and never closes a cycle.



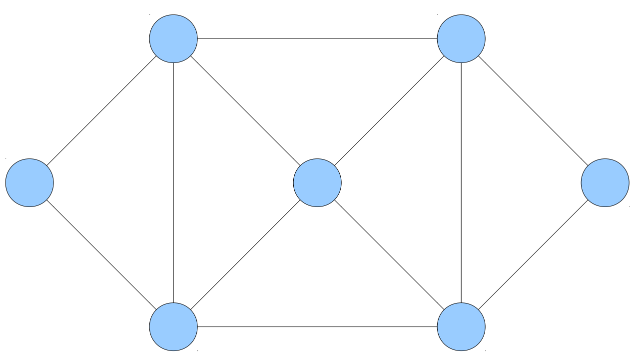
Dynamic Connectivity in Forests

- ***Goal***: Support these three operations:
  - ***link***(u, v): Add in edge {u, v}. The assumption is that u and v are in separate trees.
  - ***cut***(u, v): Cut the edge {u, v}. The assumption is that the edge exists in the tree.
  - ***are-connected***(u, v): Return whether u and v are connected.
- The data structure we'll develop can perform these operations time **O(log *n*)** each.

Euler Tours

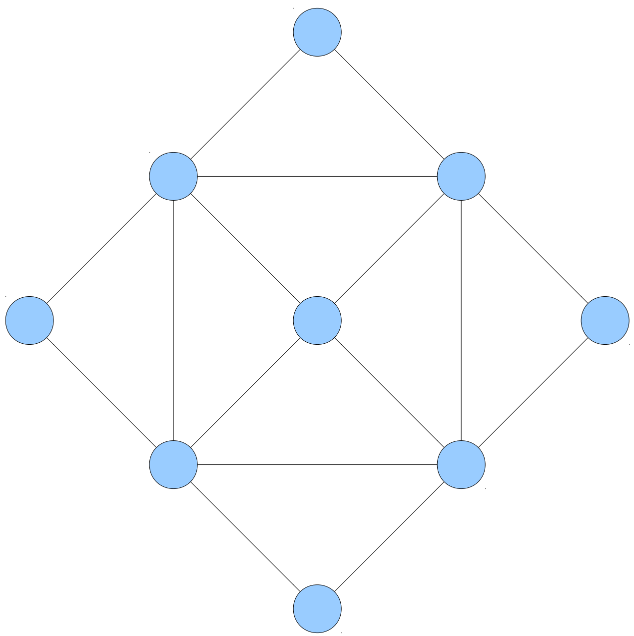
Euler Tours

- In a graph *G*, an ***Euler tour*** is a path through the graph that visits every edge exactly once.
- Mathematically formulates the “trace this figure without picking up your pencil or redrawing any lines” puzzles.



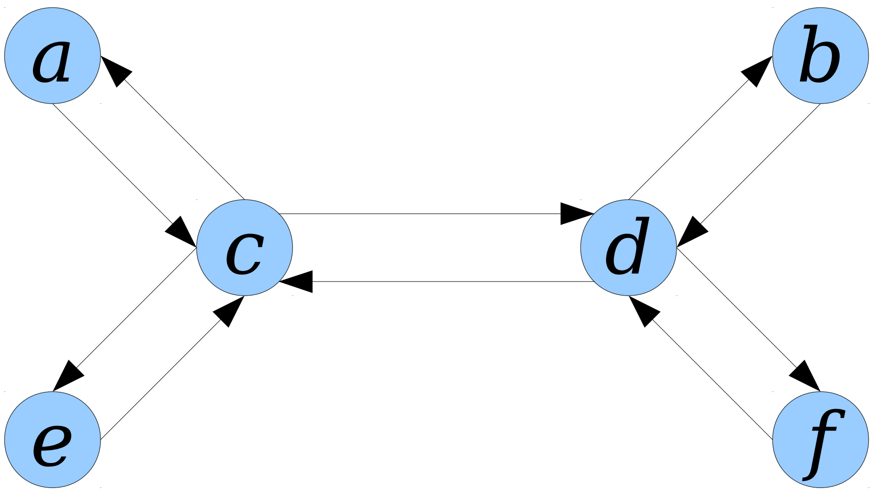
Euler Tours

- In a graph G, an ***Euler tour*** is a path through the graph that visits every edge exactly once.
- Mathematically formulates the “trace this figure without picking up your pencil or redrawing any lines” puzzles.



Euler Tours on Trees

- Trees do not have Euler tours.



a c d b d f d c e c a

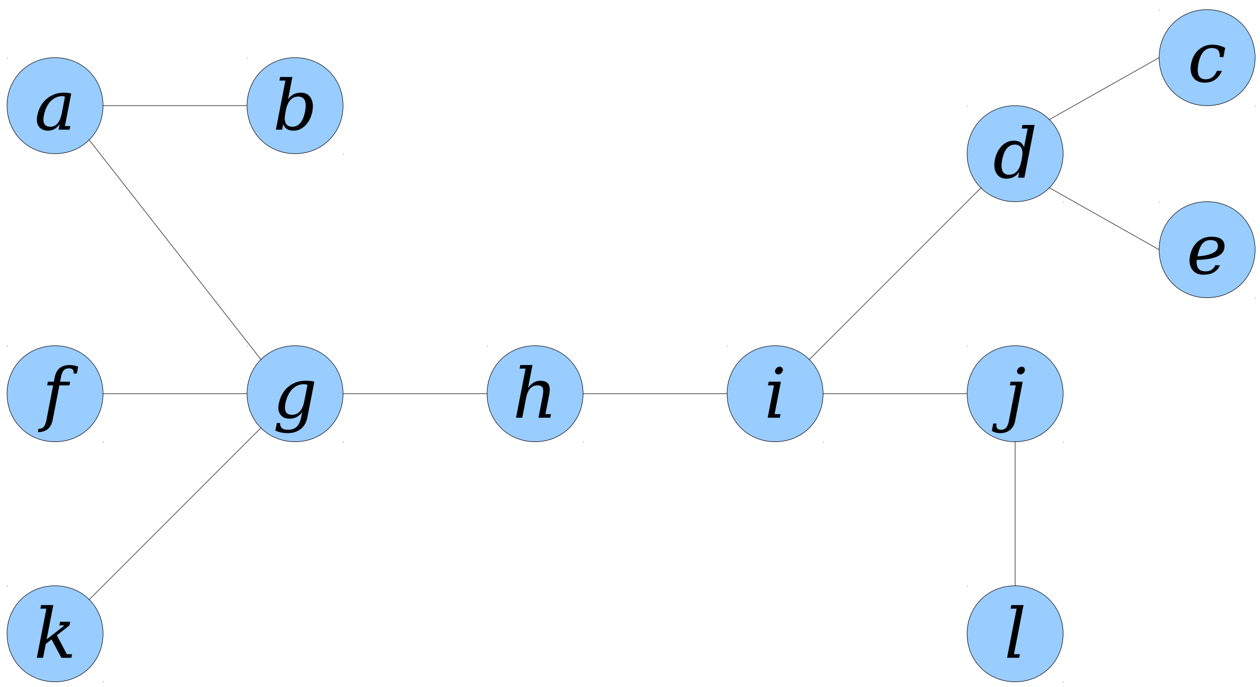
- ***Technique:*** replace each edge $\{u, v\}$ with two edges (u, v) and (v, u).
- Resulting graph has an Euler tour.

Euler Tour Trees

- The first data structure we'll design today is called an ***Euler tour tree***. It solves the dynamic connectivity problem in forests.
- ***High-level idea:*** Instead of storing the trees in the forest, store their Euler tours.
- Each edge insertion or deletion translates into a set of manipulations on the Euler tours of the trees in the forest.
- Checking whether two nodes are connected can be done by checking if they're in the same Euler tour.

Properties of Euler Tours

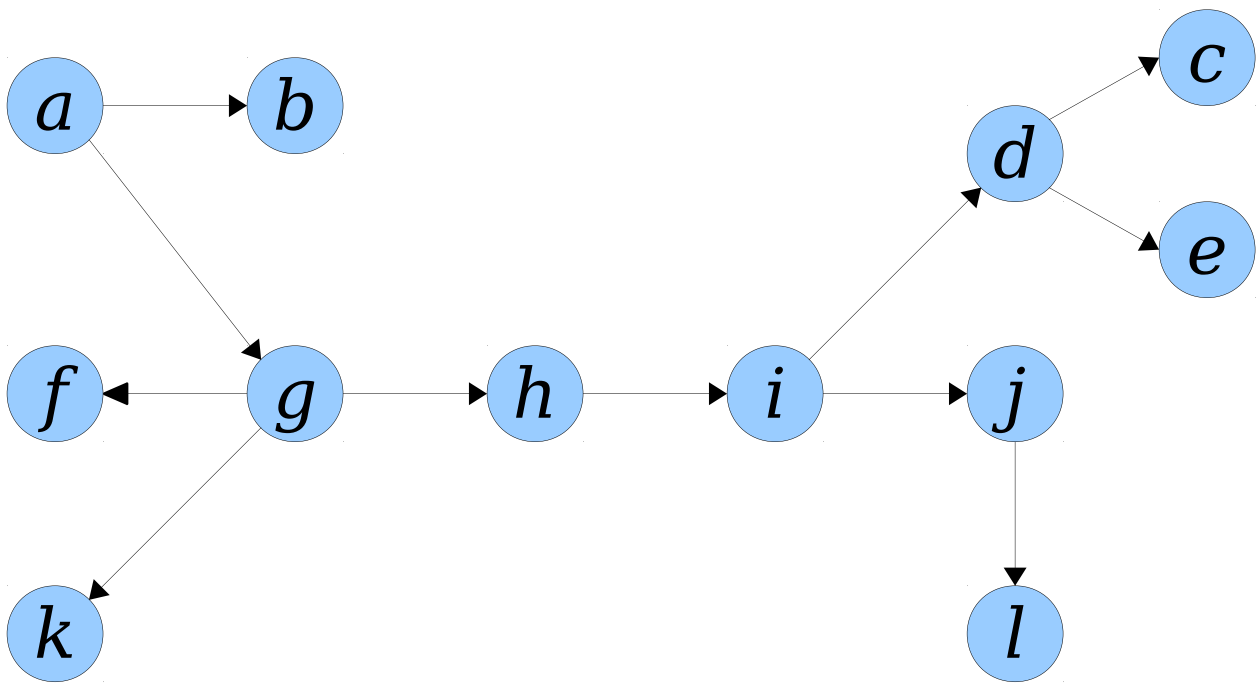
- The sequence of nodes visited in an Euler tour of a tree is closely connected to the structure of the tree.



a b a g h i d c d e d i j l j i h g f g k g a

Properties of Euler Tours

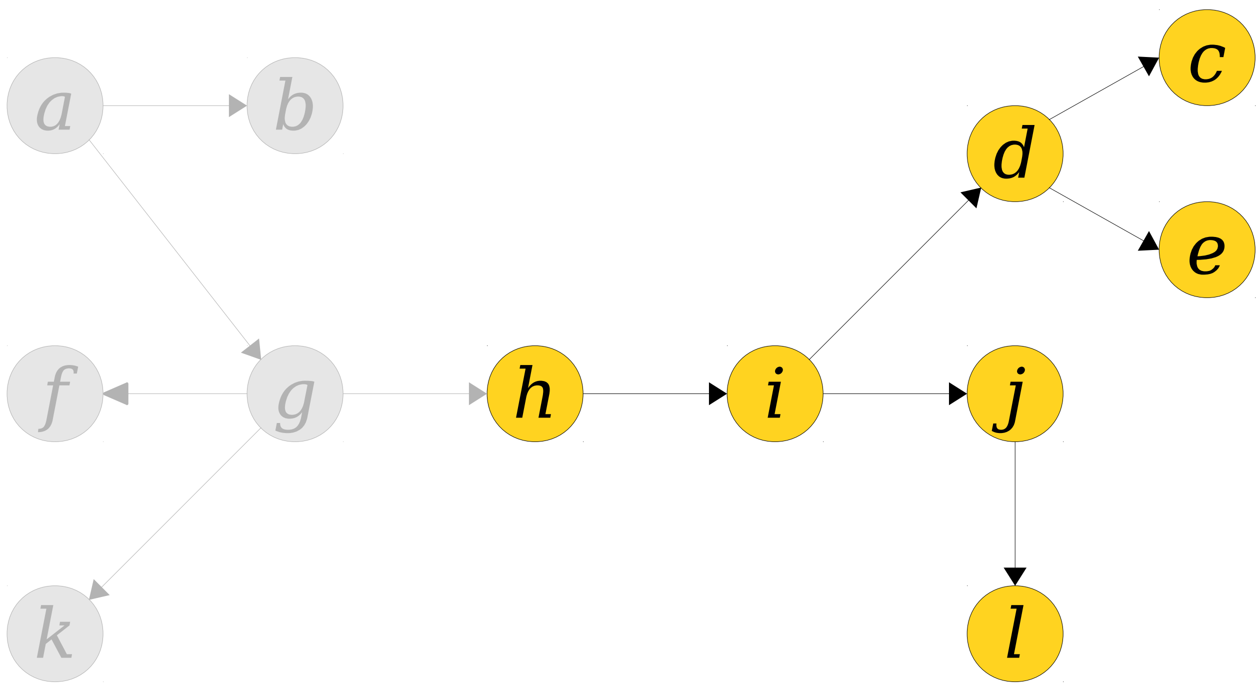
- The sequence of nodes visited in an Euler tour of a tree is closely connected to the structure of the tree.



a b a g h i d c d e d i j l j i h g f g k g a

Properties of Euler Tours

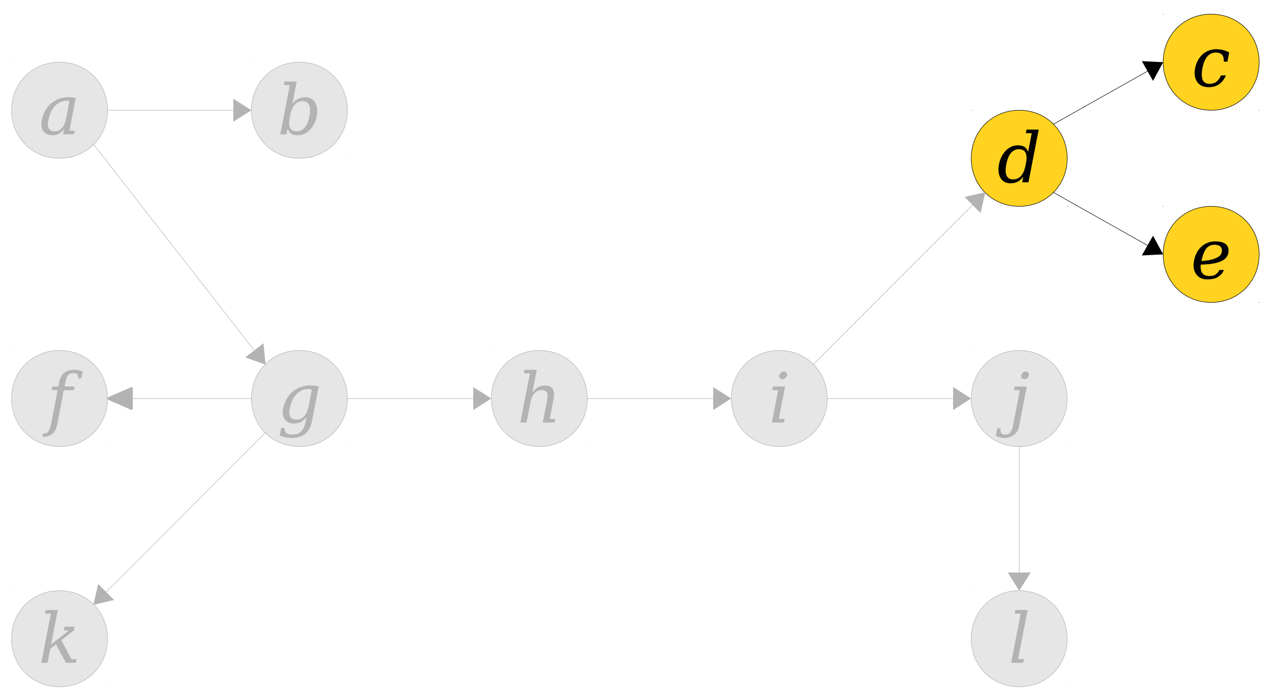
- The sequence of nodes visited in an Euler tour of a tree is closely connected to the structure of the tree.



a b a g ***h i d c d e d i j l j i h*** ***g f g k g a***

Properties of Euler Tours

- The sequence of nodes visited in an Euler tour of a tree is closely connected to the structure of the tree.



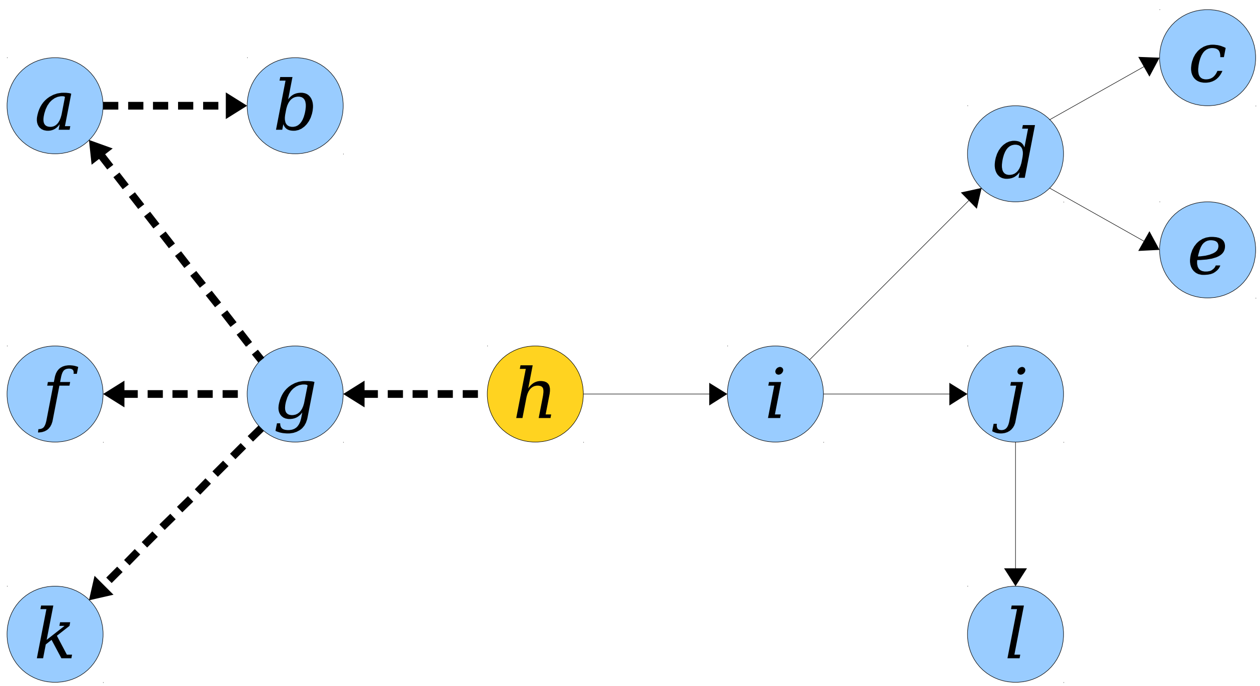
*a b a g h i **d c d e d** i j l j i h g f g k g a*

Properties of Euler Tours

- The sequence of nodes visited in an Euler tour of a tree is closely connected to the structure of the tree.
- Begin by directing all edges away from the first node in the tour.
- ***Claim:*** The sequences of nodes visited between the first and last instance of a node v gives an Euler tour of the subtree rooted at v.

Rerooting a Tour

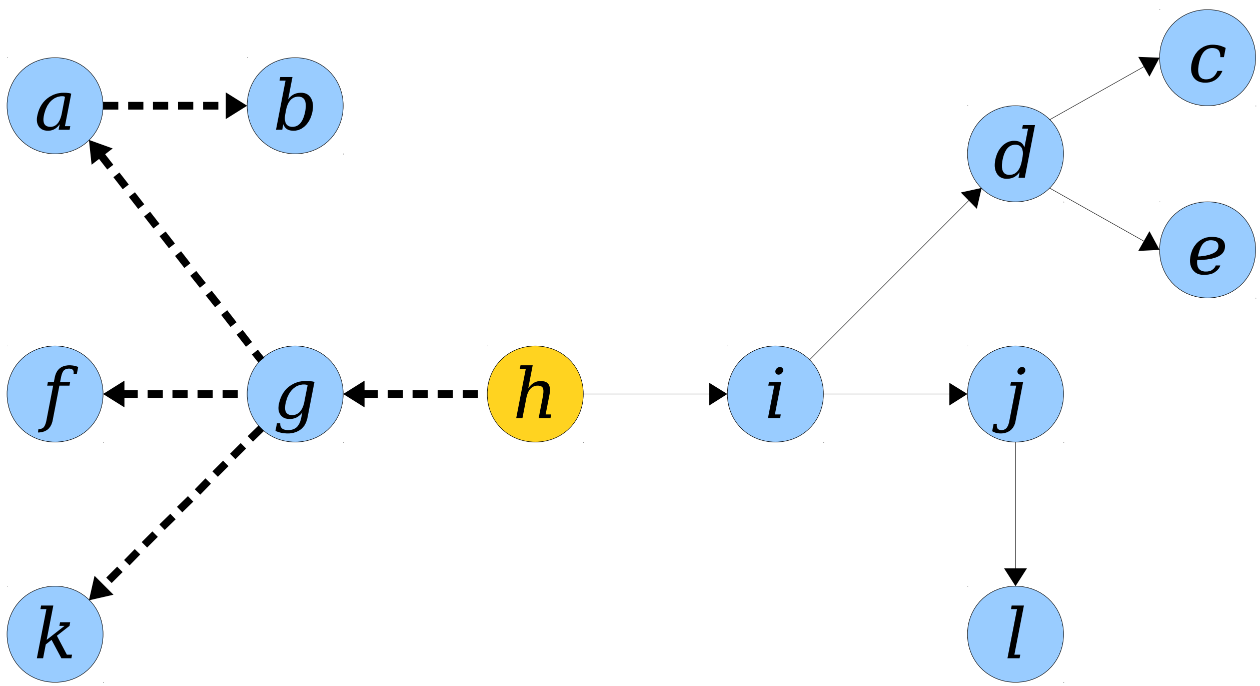
- The subtrees defined by ranges in Euler tours on trees depend on the root.
- In some cases, we will need to change the root of the tree.



a b a g h i d c d e d i j l j i h g f g k g a

Rerooting a Tour

- The subtrees defined by ranges in Euler tours on trees depend on the root.
- In some cases, we will need to change the root of the tree.



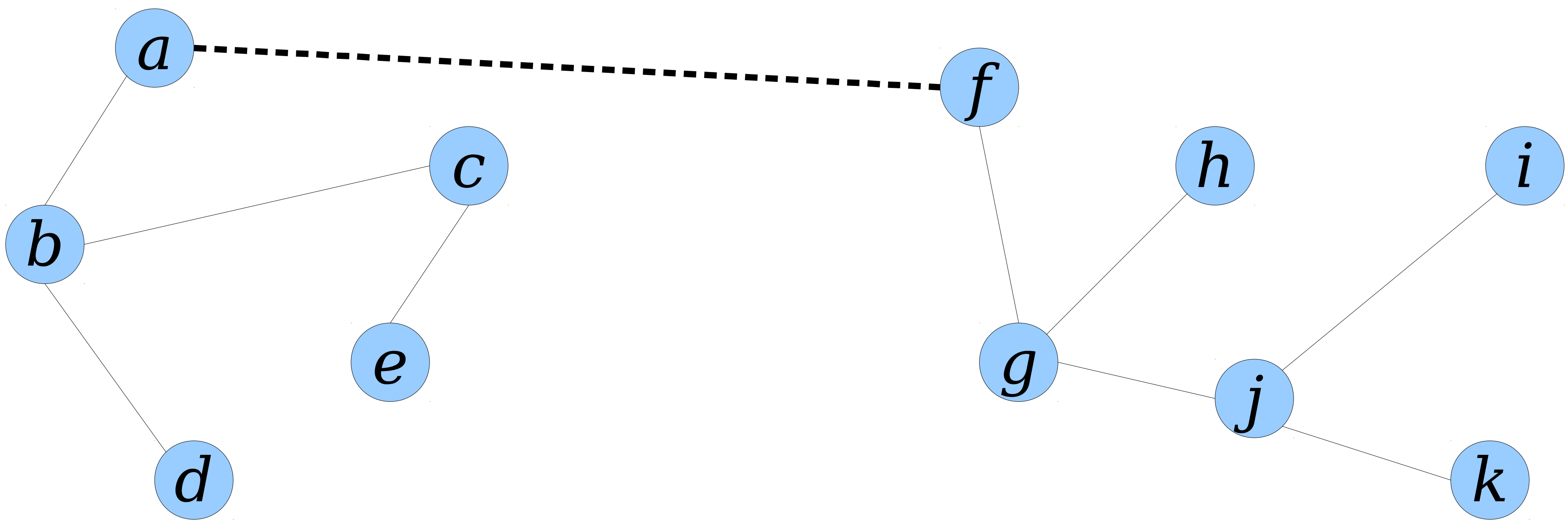
h i d c d e d i j l j i h g f g k g a b a g h

Rerooting a Tour

- ***Algorithm:***
  - Pick any occurrence of the new root r.
  - Split the tour into A and B, where B is the part of the tour before r.
  - Delete the first node of A and append r.
  - Concatenate B and A.

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:

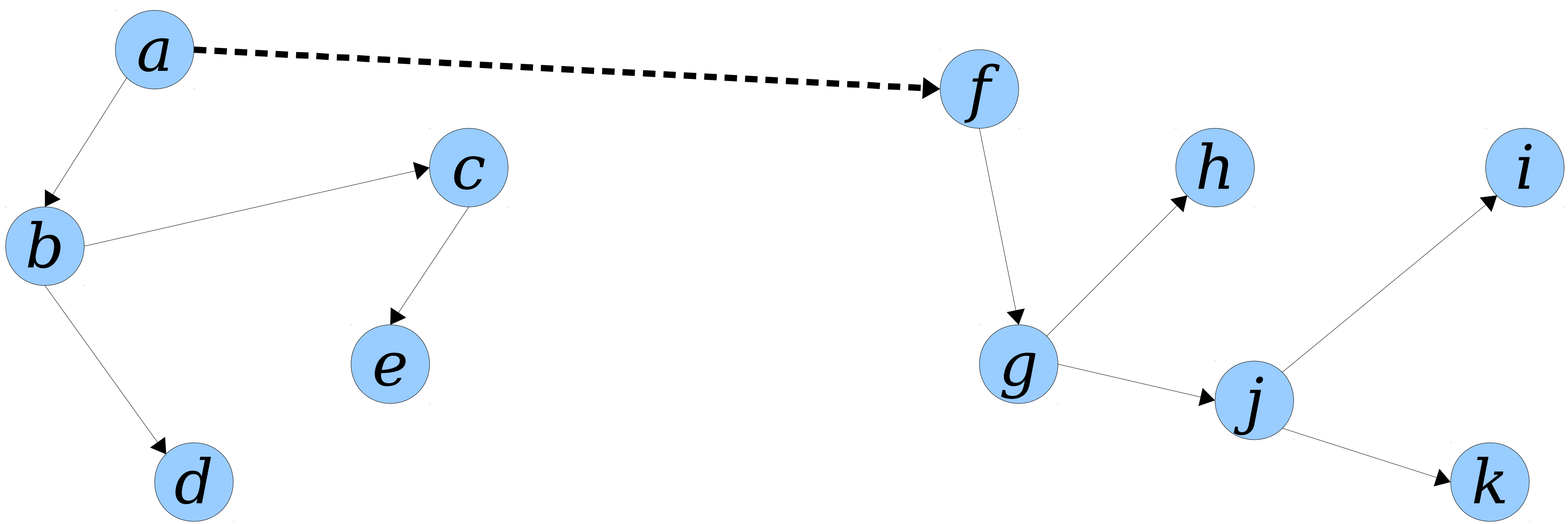


a b d b c e c b a

f g j k j i j g h g f

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:

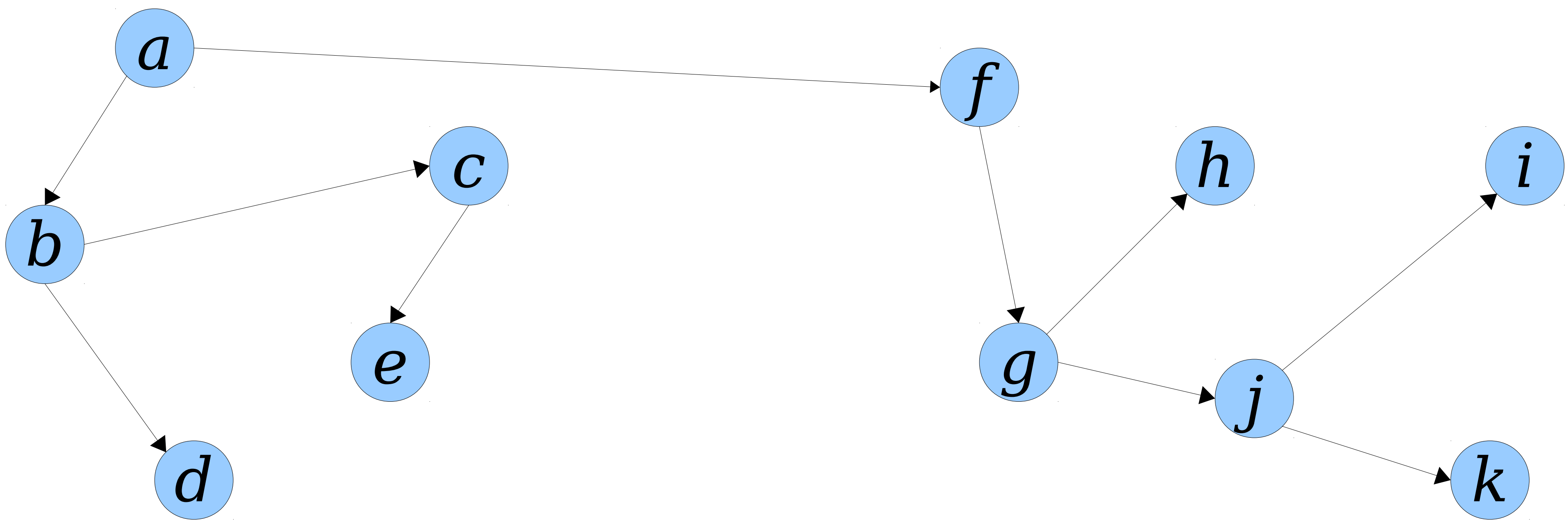


***a* b d b c e c b a**

***f* g j k j i j g h g f**

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge $\{u, v\}$.
- Watch what happens to the Euler tours:



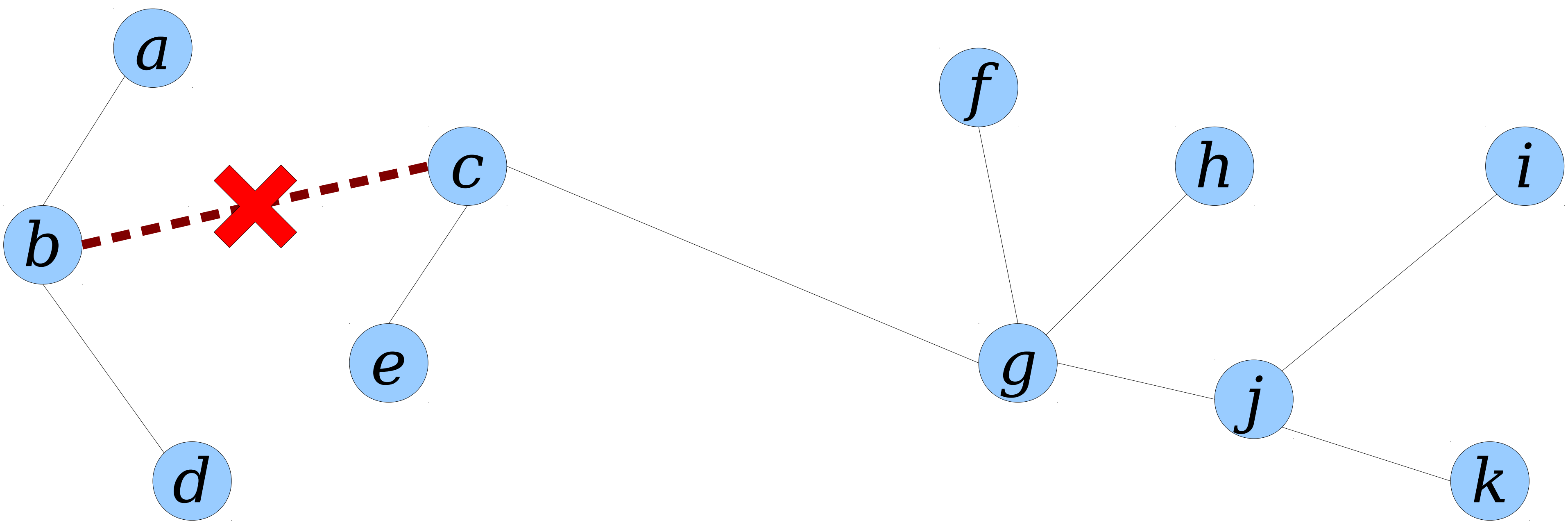
a b d b c e c b a f g j k j i j g h g f a

Euler Tours and Dynamic Trees

- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge $\{u, v\}$.
- To link T_1 and T_2 by adding $\{u, v\}$:
  - Let E_1 and E_2 be Euler tours of T_1 and T_2, respectively.
  - Rotate E_1 to root the tour at u.
  - Rotate E_2 to root the tour at v.
  - Concatenate E_1, E_2, $\{u\}$.

Euler Tours and Dynamic Trees

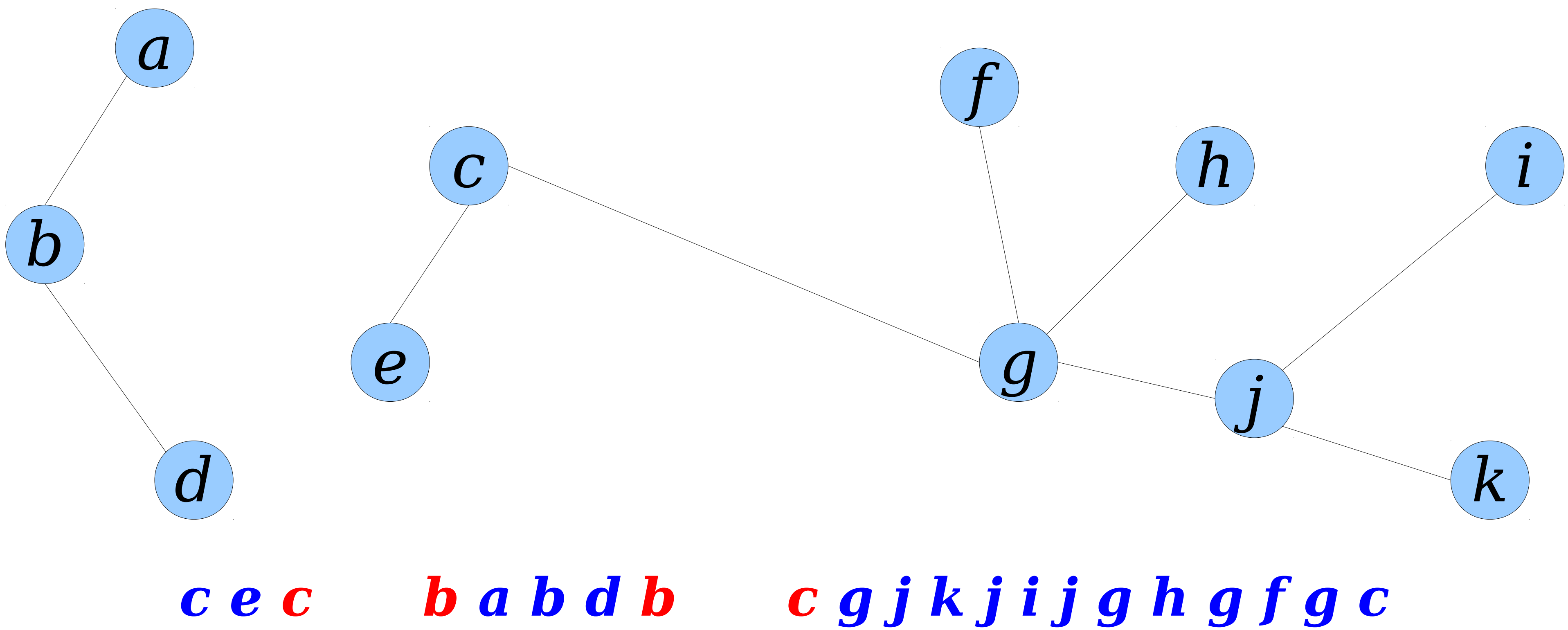
- Given a tree T, executing ***cut***(u, v) cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



c e c b a b d b c g j k j i j g h g f g c

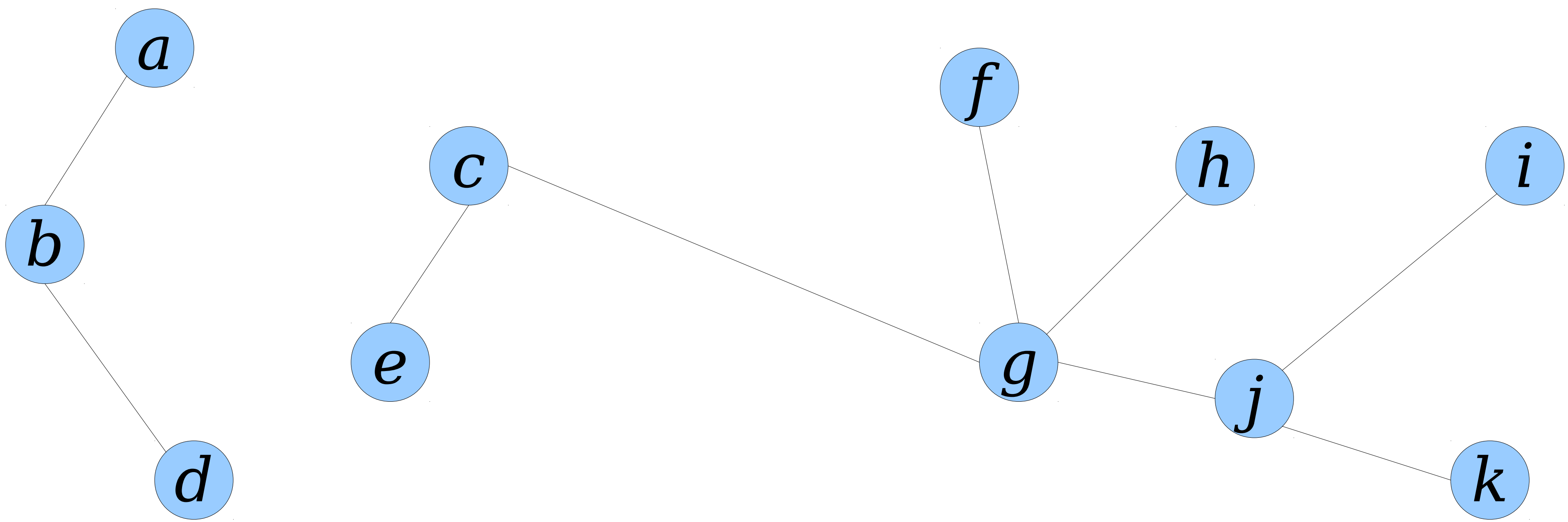
Euler Tours and Dynamic Trees

- Given a tree T, executing ***cut***(u, v) cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



Euler Tours and Dynamic Trees

- Given a tree T, executing ***cut***(u, v) cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



b a b d b c e c g j k j i j g h g f g c

Euler Tours and Dynamic Trees

- Given a tree T, executing ***cut***(u, v) cuts the edge $\{u, v\}$ from the tree (assuming it exists).
- To cut T into T_1 and T_2 by cutting $\{u, v\}$:
  - Let E be an Euler tour for T.
  - Split E at (u, v) and (v, u) to get J, K, L, in that order.
  - Delete the last entry of J.
  - Then $E_1 = K$.
  - Then $E_2 = J, L$

The Story So Far

- ***Goal:*** Implement ***link***, ***cut***, and ***are-connected*** as efficiently as possible.
- By representing trees via their Euler tours, can implement *link* and *cut* so that only O(1) joins and splits are necessary per operation.
- Questions to answer:
  - How do we efficiently implement these joins and splits?
  - Once we have the tours, how do we answer connectivity queries?

Representation Issues

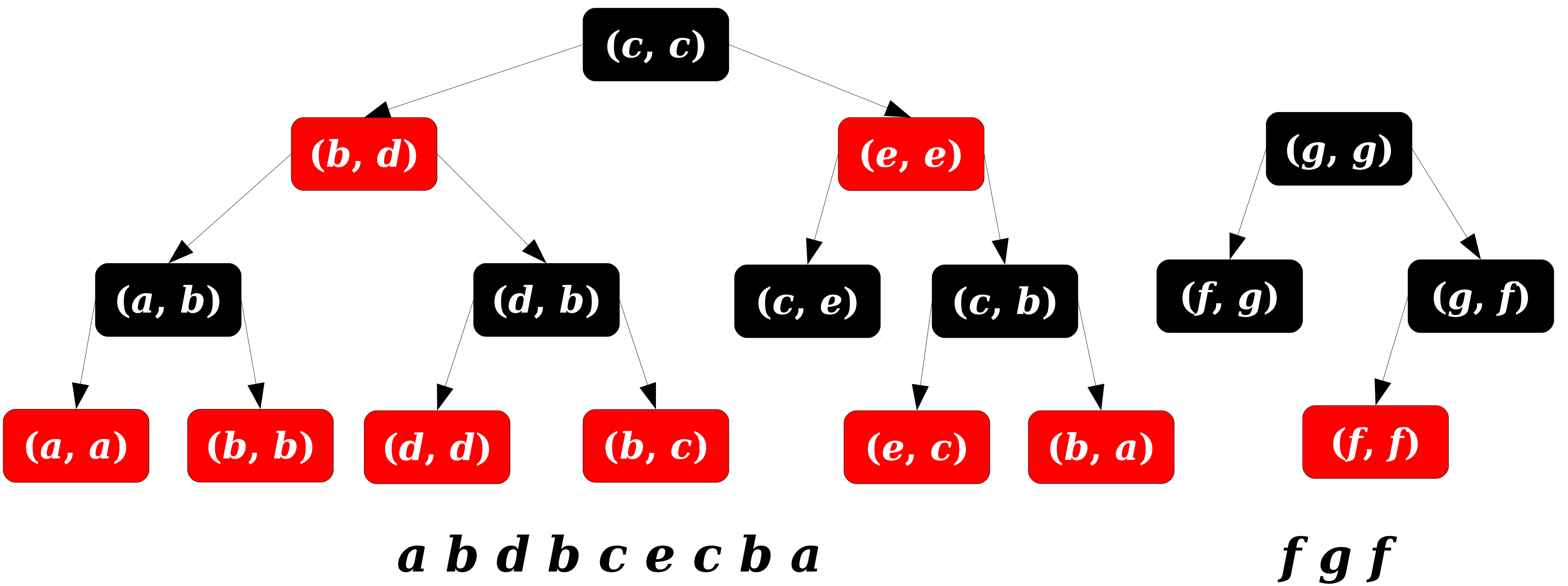
- We need a representation that lets us perform the following operations:
  - Determine if two nodes are in the same sequence.
  - Split a sequence at an arbitrary point.
  - Join a sequence at an arbitrary point.
  - Find where in a given sequence a particular edge is (for ***cut***)
  - Find where in a given sequence a particular node is (for rerooting at tour).

Representation Issues

- ***Idea:*** Rather than storing the tour as a series of *nodes*, store it as a series of *edges*.
  - This makes it easy to locate edges in the ***cut*** step.
- Add, for each node v, an edge (v, v) that's included in whatever tour contains v.
  - This gives us a way of identifying "some copy" of a node v when rerooting a tour.
- With this representation, each ***link*** or ***cut*** requires only O(1) sequence splits and sequence joins.

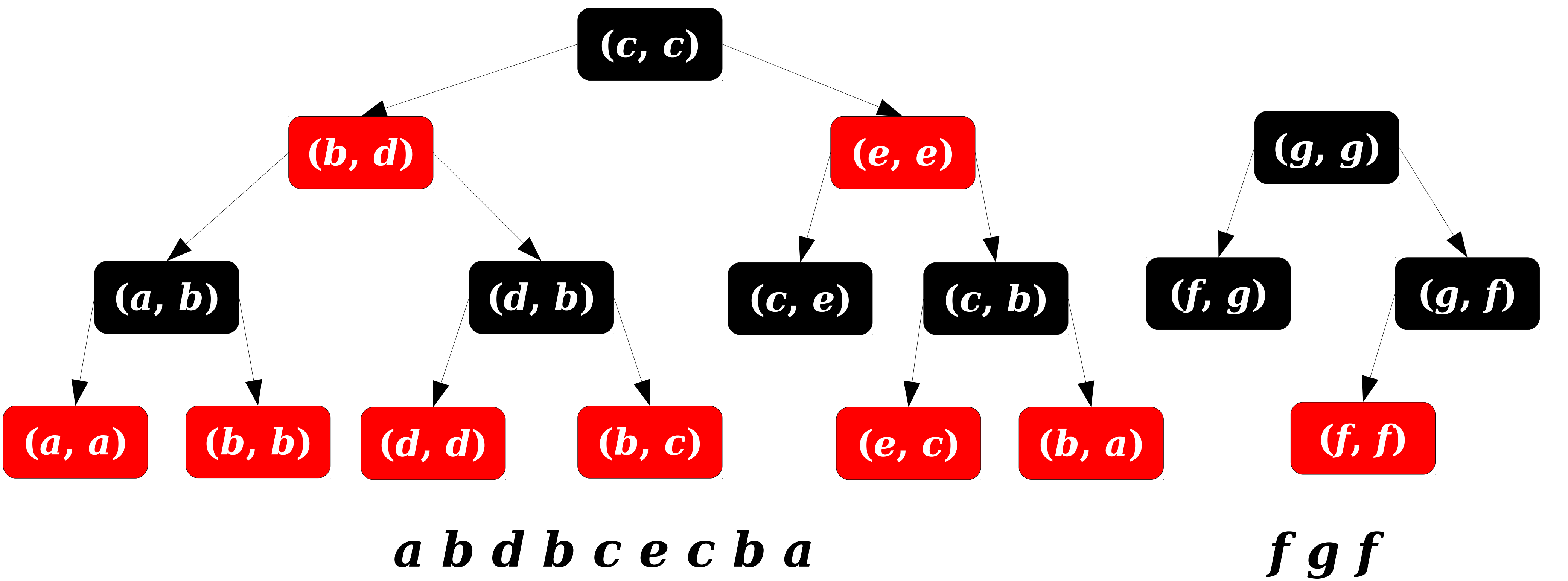
Representing Sequences

- ***Idea:*** Represent each sequence as a red/black tree augmented with order statistics information.
- These are not binary *search* trees. We're using the *shape* of a red/black tree to ensure balance.



Representing Sequences

- ***Observation:*** If nodes store pointers to their parents, can answer ***is-connected***(u, v) in time O(log n) by seeing if u and v are in the same tree.



Euler Tour Trees

- The data structure:
  - Represent each tree as an Euler tour.
  - Store those sequences as balanced binary trees.
  - Each node in the balanced trees stores a pointer to its parent.
  - Store an auxiliary BST holding pointers to each node and each edge so that they can be located efficiently.
- ***link***, ***cut***, and ***is-connected*** queries take time only **O(log *n*)** each.

Time-Out for Announcements!

Midterm Grading

- You're done with the midterm! Woohoo!
- We're going to be grading exams over the weekend. We'll release grades as soon as they're ready.
- Although we don't curve individual exam scores, we do curve raw total grades – and we anticipate having a pretty generous curve this quarter.

Final Project Logistics

- As a reminder, your final project paper is due 24 hours before your presentation.
- Your paper should be an accessible, engaging, and technically precise introduction to the data structure.
  - Give some background – why should we care about the data structure? Who invented it?
  - Describe it in as accessible a manner as possible. What are the key ideas driving it? Intuitively, why would you expect them to work? Then get more specific – how does each operation work?
  - Argue correctness and runtime, proving non-obvious results along the way and providing a good intuition.
- Then, describe your "interesting" component, and make it shine! Tell us why what you did was interesting and what you learned in the process.

Final Project Logistics

- Final project presentations start next week.
- Presentations should run around 15 minutes. We may have to cut you off if you run much more than this because we need to factor in setup and cleanup time.
- Your presentation won't be long enough to present everything from your paper, and you shouldn't try to do that. Instead, focus on what's important and interesting. Convey the major ideas, intuitions, and why the data structure is so cool!
- We'll ask a few questions at the end of the presentation, so be prepared to discuss things in a bit more detail.
- Please arrive around five minutes early so that you can get set up.

Back to CS166!

Fully-Dynamic Connectivity

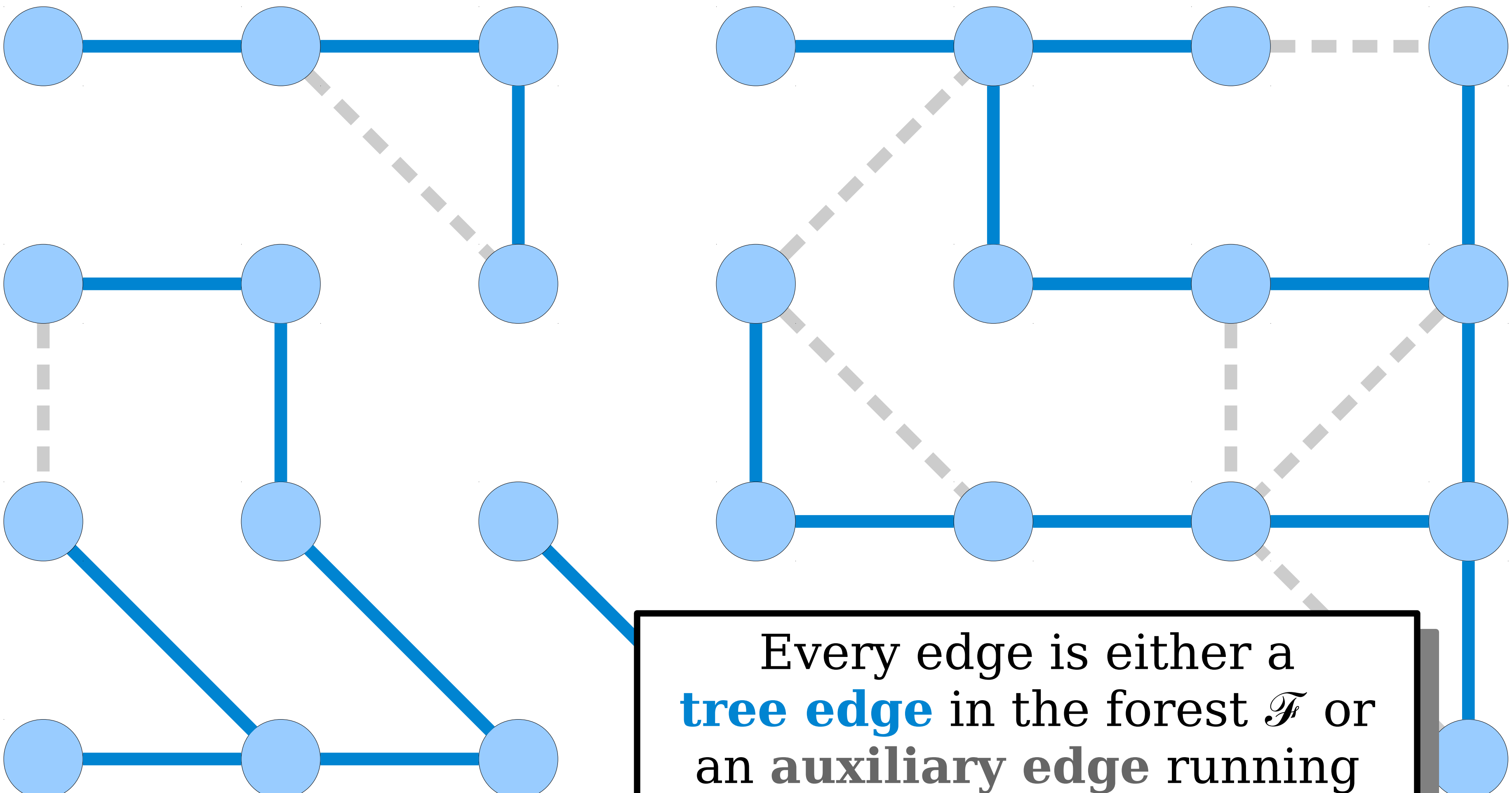
The Challenge

- Numerous issues arise in scaling up from forests to complete graphs:
  - In a forest, a ***link*** connects two distinct trees. In a general graph, the endpoints of a ***link*** might already be connected.
  - In a forest, a ***cut*** splits one tree into two. In a general graph, a ***cut*** might not change connectivity.
  - In a forest, there is a unique path between any two nodes in each tree. In a general graph, there can be many.
- As of 2016, there is no known Euler-tour-like approach for maintaining dynamic connectivity.

The Basic Idea

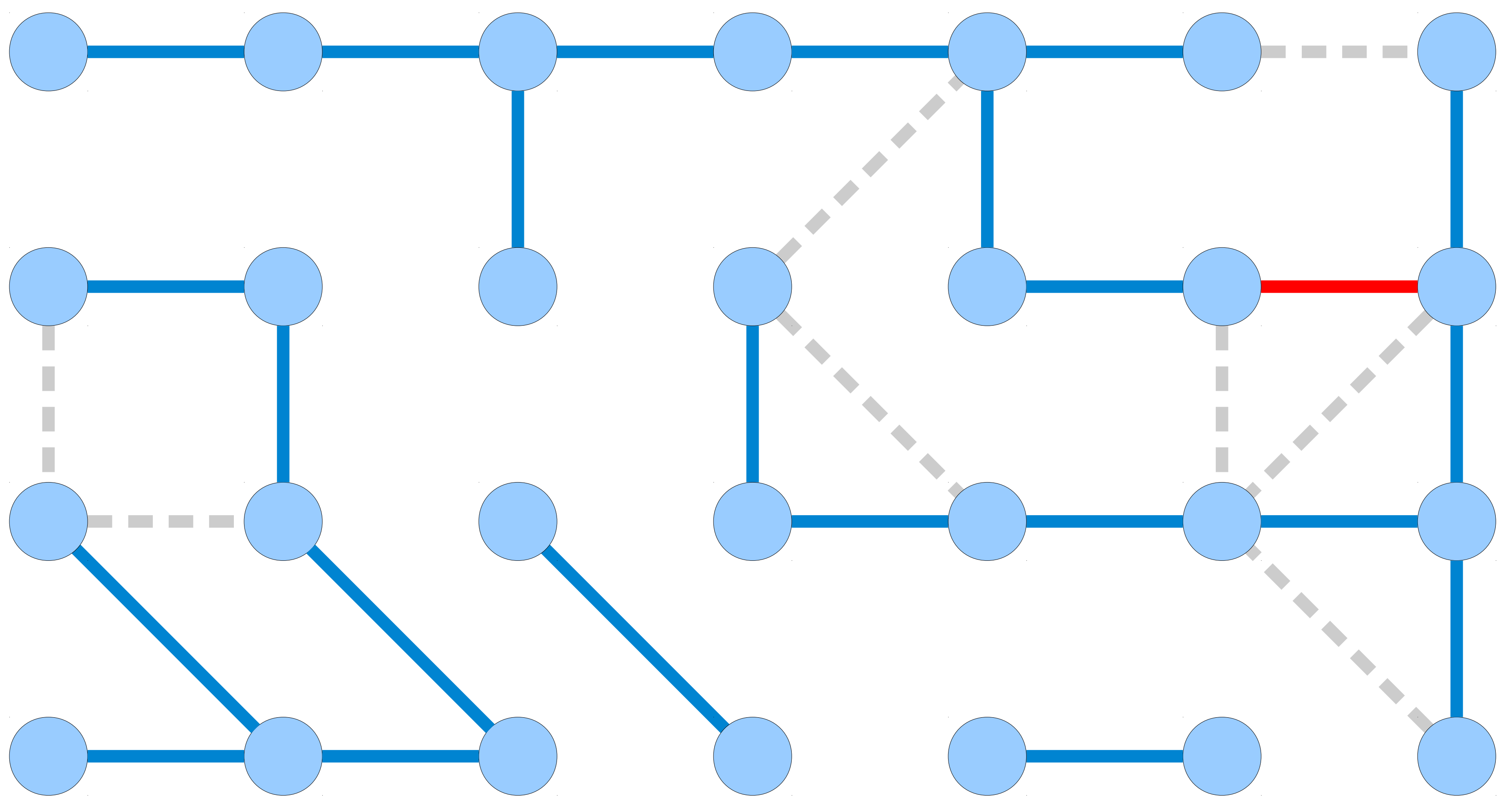
- Let G be an undirected graph and let $\mathscr{F}$ be a spanning forest for G.
- ***Observation:*** Two nodes u and v are connected in G iff they are connected in $\mathscr{F}$.
- ***Idea:*** Try to maintain a spanning forest $\mathscr{F}$ for G, represented as Euler tour trees.
- The challenge will be efficiently maintaining $\mathscr{F}$.

Maintaining a Forest

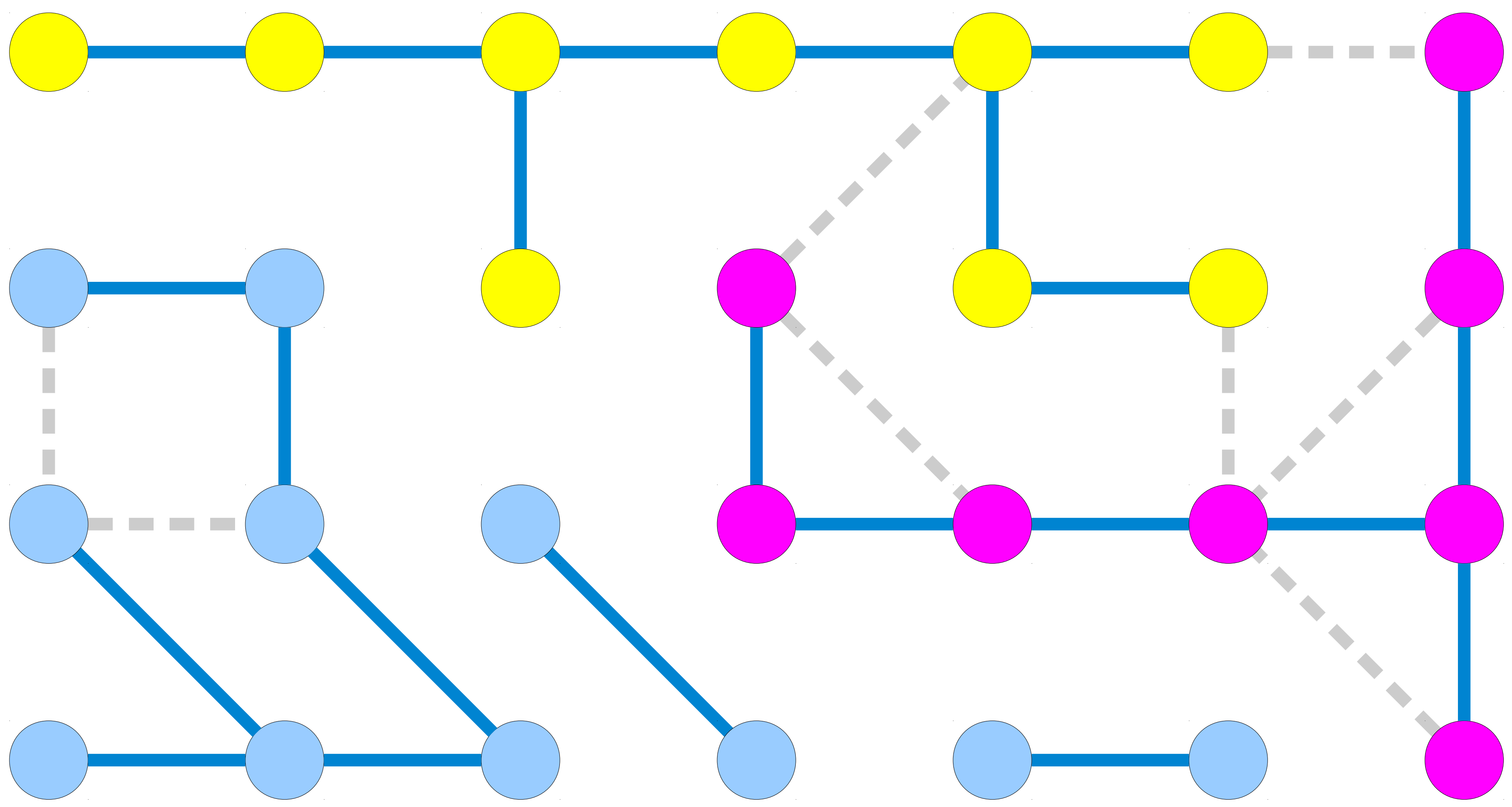


Every edge is either a **tree edge** in the forest $\mathscr{F}$ or an **auxiliary edge** running between two nodes in the same tree in $\mathscr{F}$.

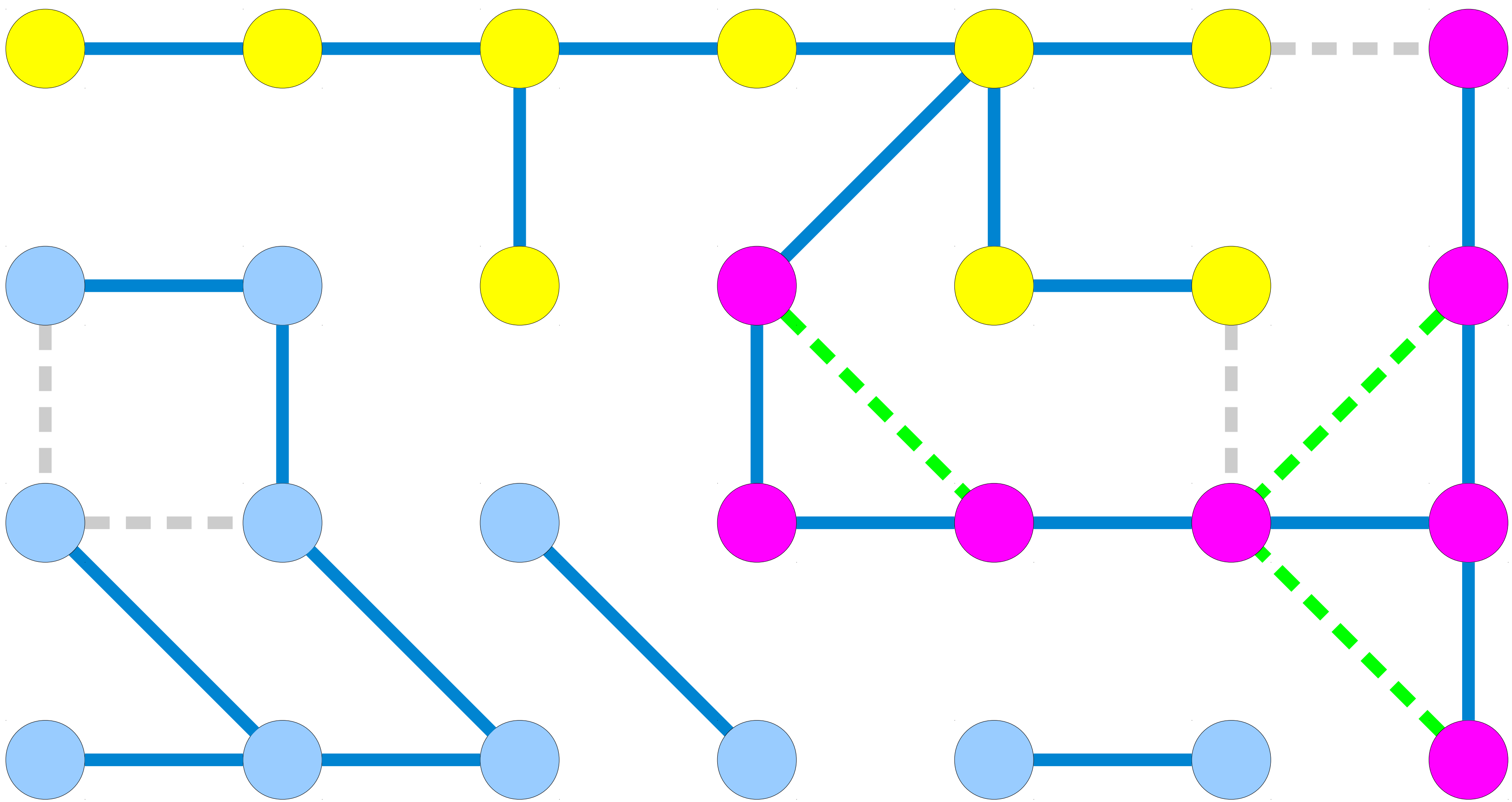
Maintaining a Forest



Maintaining a Forest



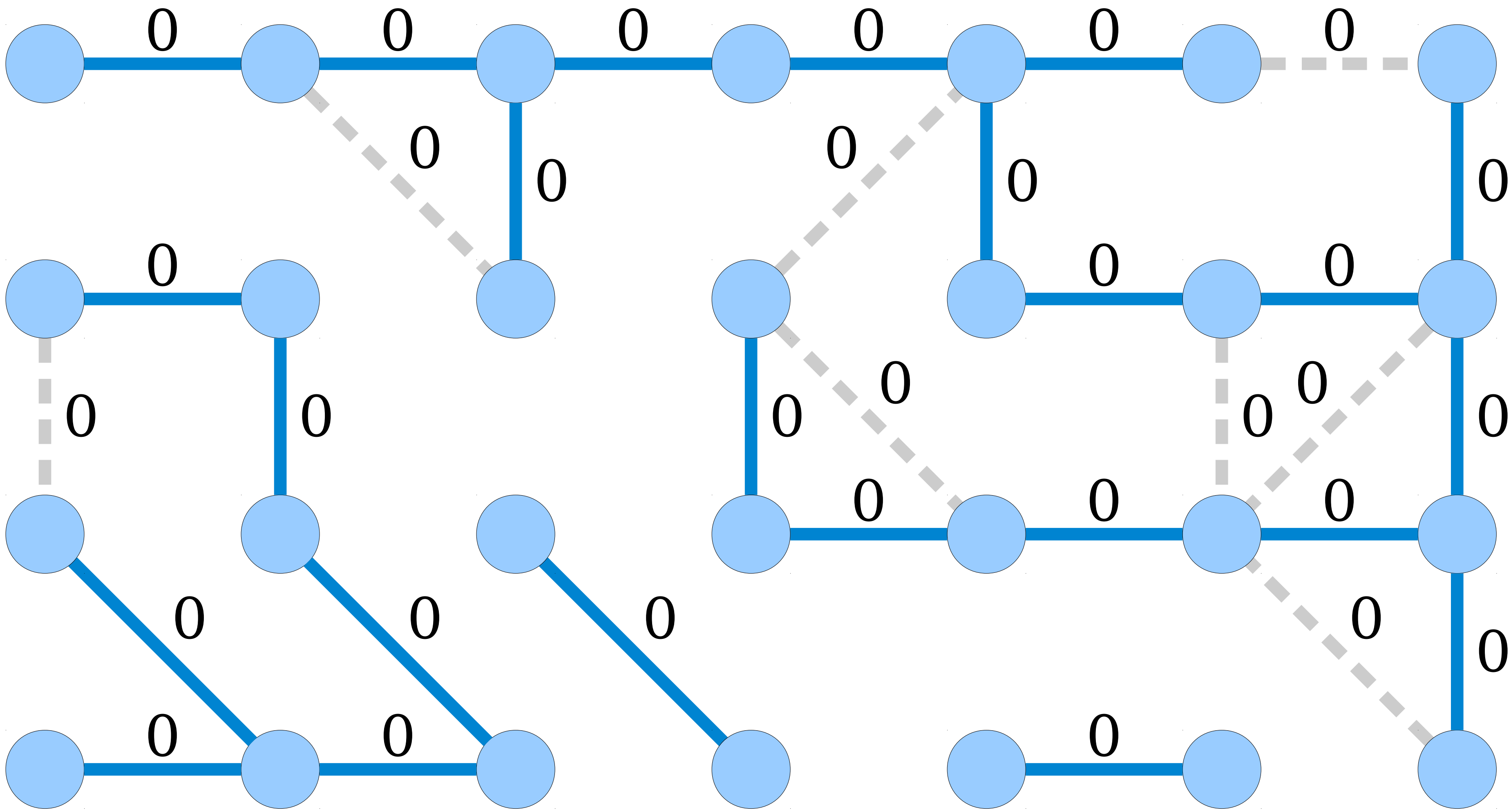
Maintaining a Forest



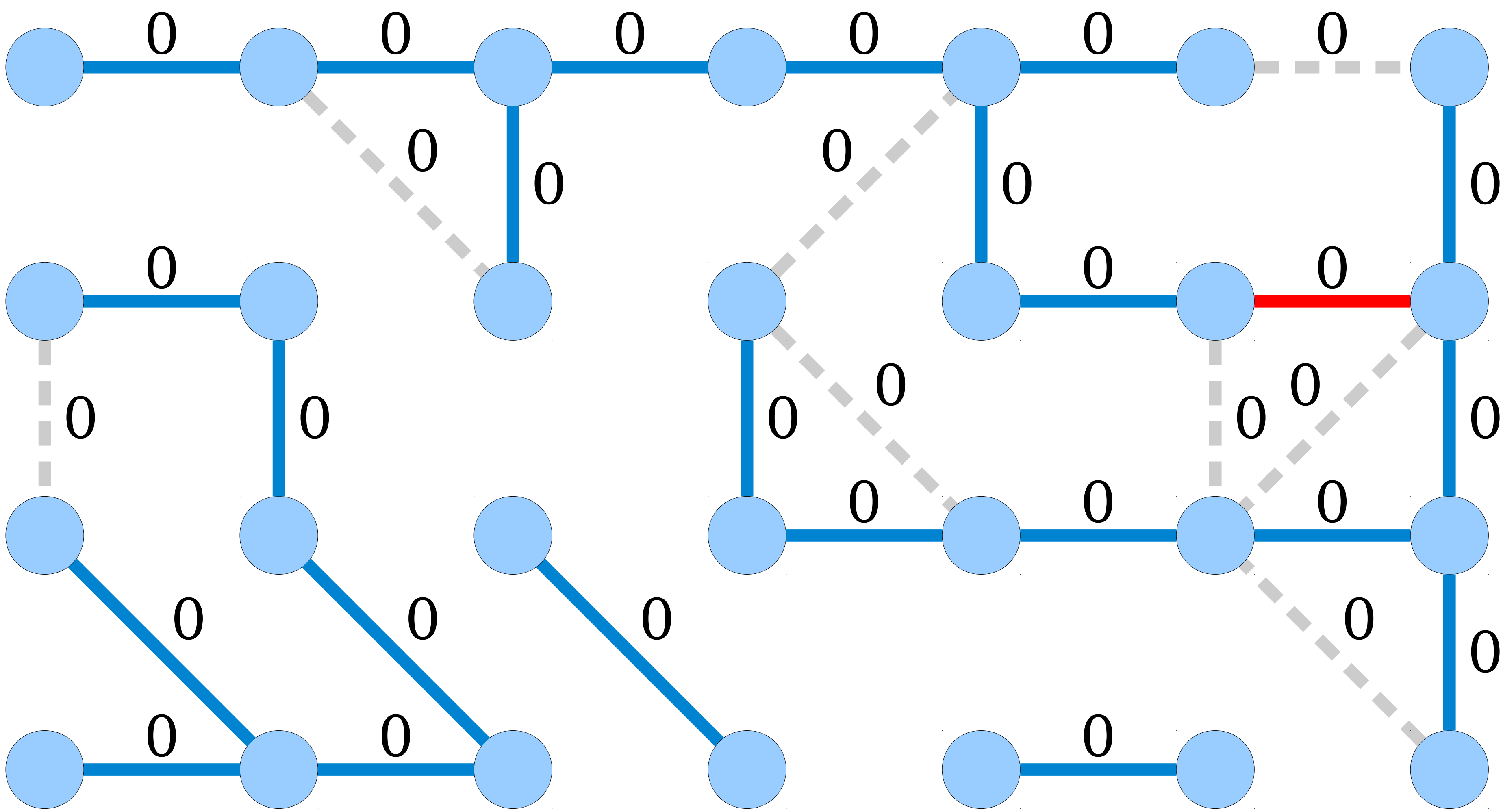
The Challenge

- ***Goal:*** Devise a way of storing edges such that we don't repeatedly rescan the same edges trying to glue trees together.
- ***Idea:*** Associate a "specificity" with each edge, initially 0.
- Edges with higher specificity refer to more restricted regions of the graph.
- Edges with lower specificity refer to more general regions of the graph.
- Adjust the specificity of edges in response to deletions.

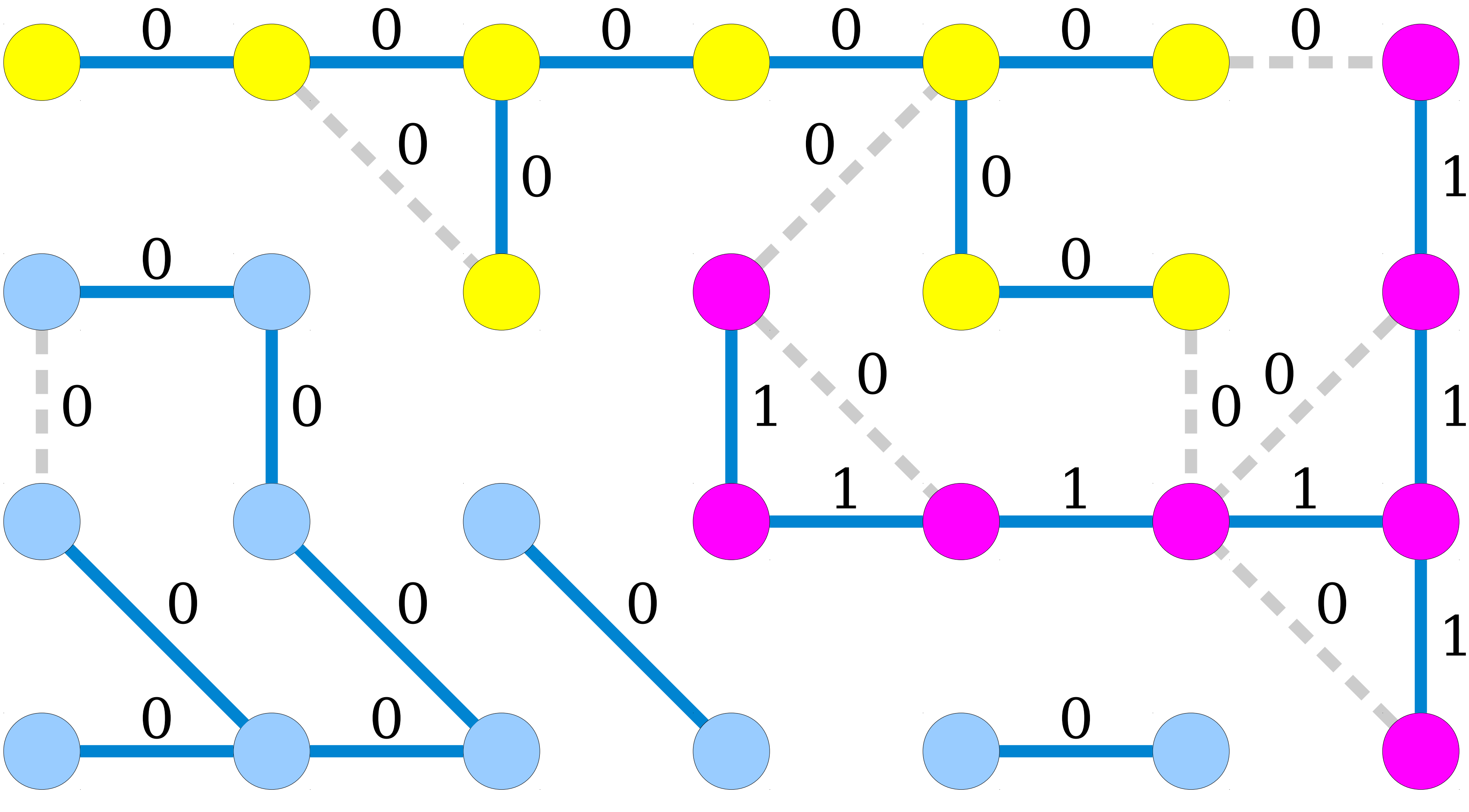
Edge Specificity



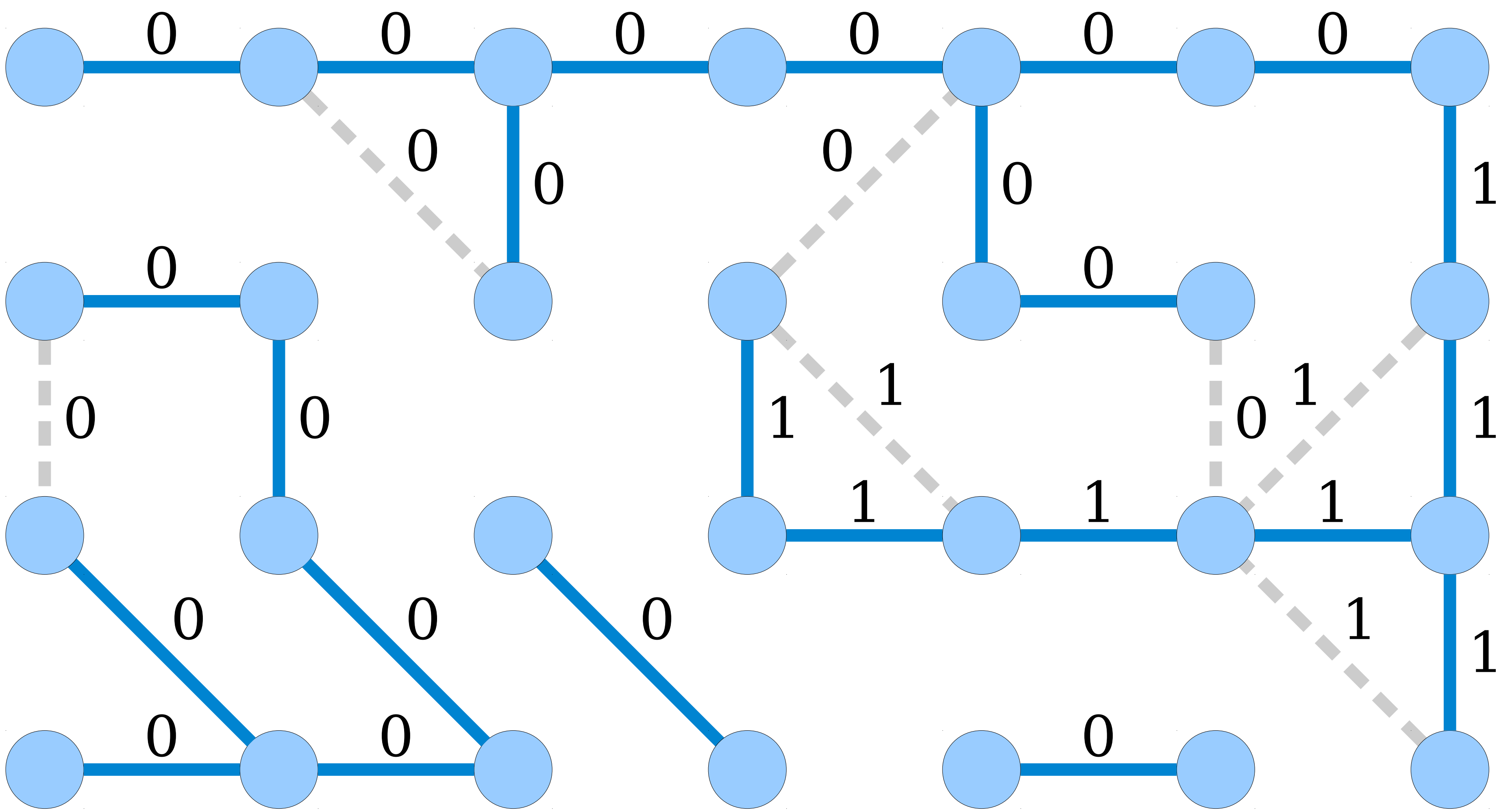
Edge Specificity



Edge Specificity



Edge Specificity



The Approach

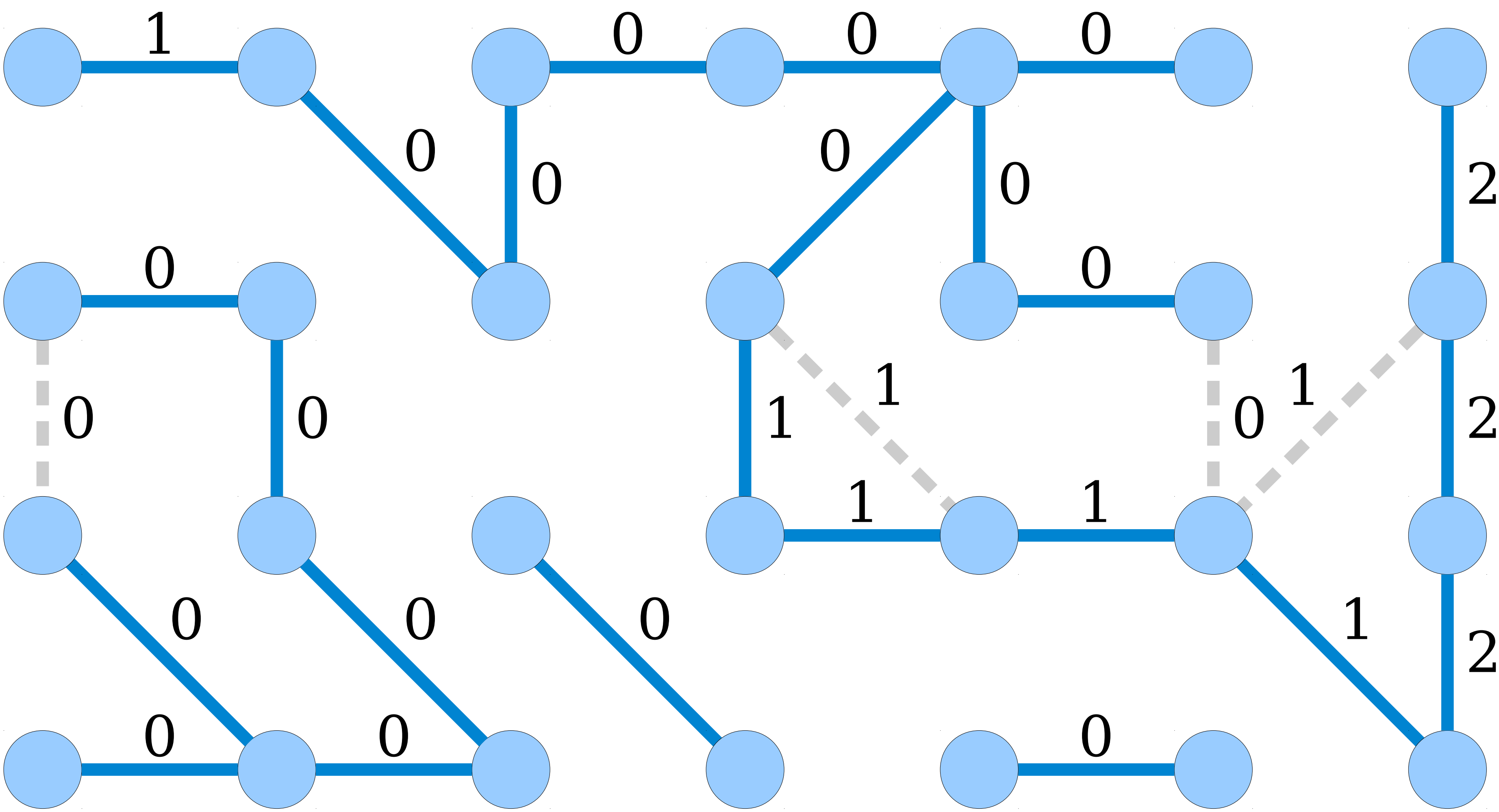
- To delete a tree edge $\{u, v\}$ with specificity k:
  - Let T_u and T_v be the resulting trees.
  - Push all edges of specificity k in T_u up to specificity $k + 1$.
  - For all edges incident to T_u of specificity k:
    - If that edge connects T_u to T_v, add it to $\mathscr{F}$ and stop.
    - Otherwise, it connects T_u to itself, so increase its specificity.
  - If the previous iteration didn't reconnect T_u and T_v, repeat the above loop on specificity $k - 1$.

The Runtime Analysis

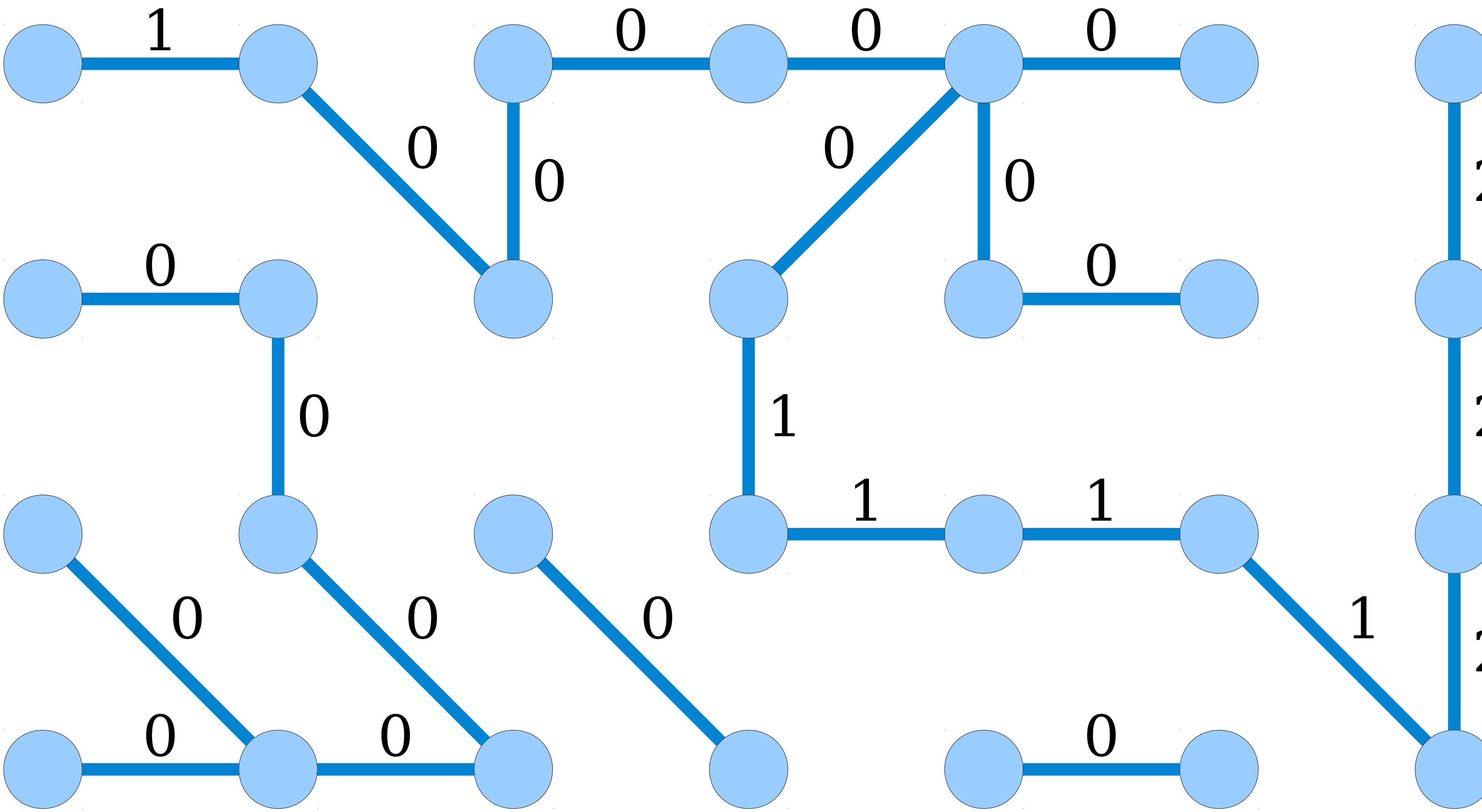
The Representation

- Store a series of forests $\mathscr{F}_0$, $\mathscr{F}_1$, $\mathscr{F}_2$, …, .
- Forest $\mathscr{F}_k$ stores all edges at specificities k or greater.
  - Thus $\mathscr{F}_0 = \mathscr{F}$ and $\mathscr{F}_0 \supseteq \mathscr{F}_1 \supseteq \ldots$
- This enables us to query whether two nodes are connected by edges of level k or greater.

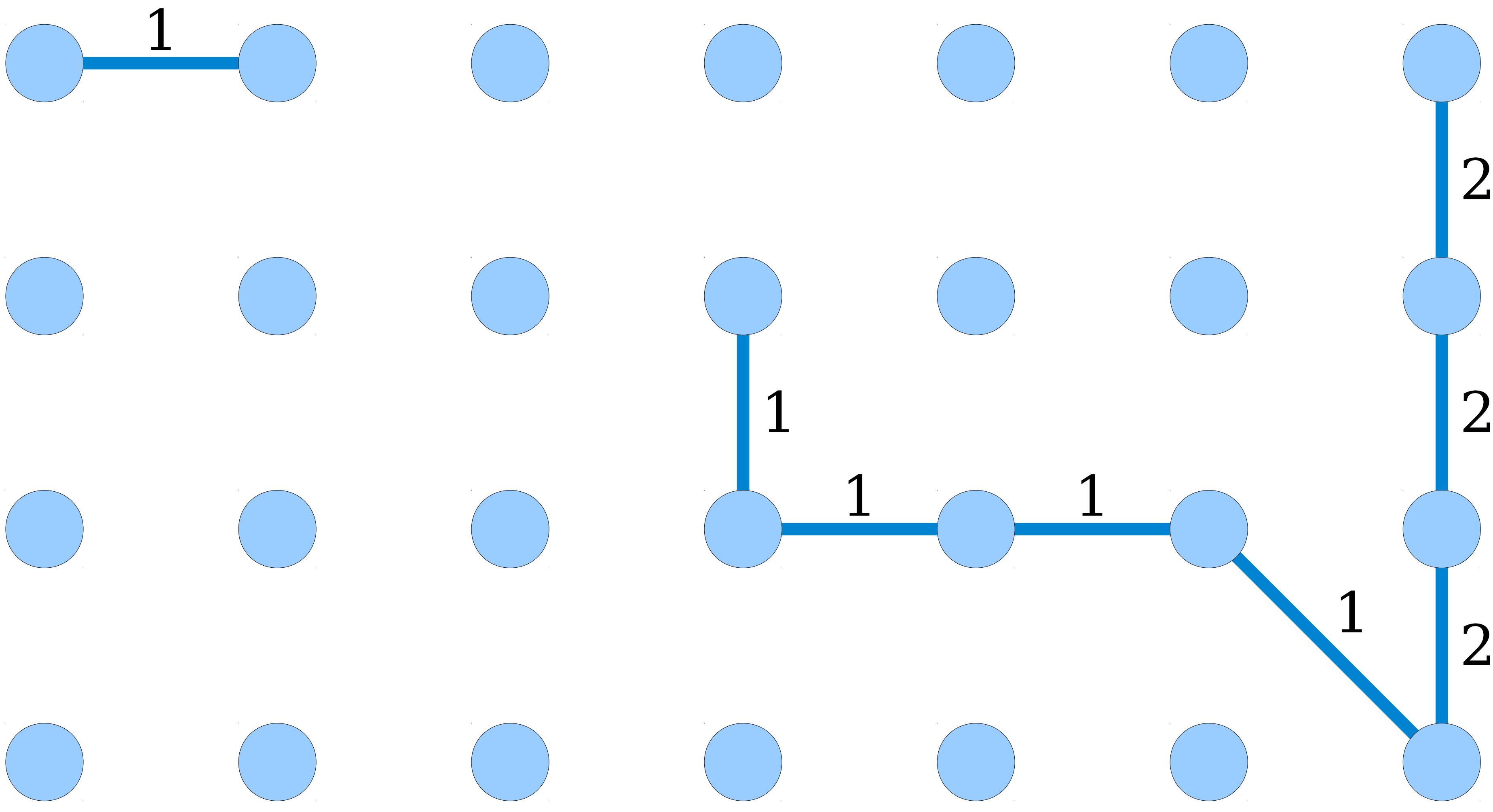
Interpreting $\mathscr{F}_k$



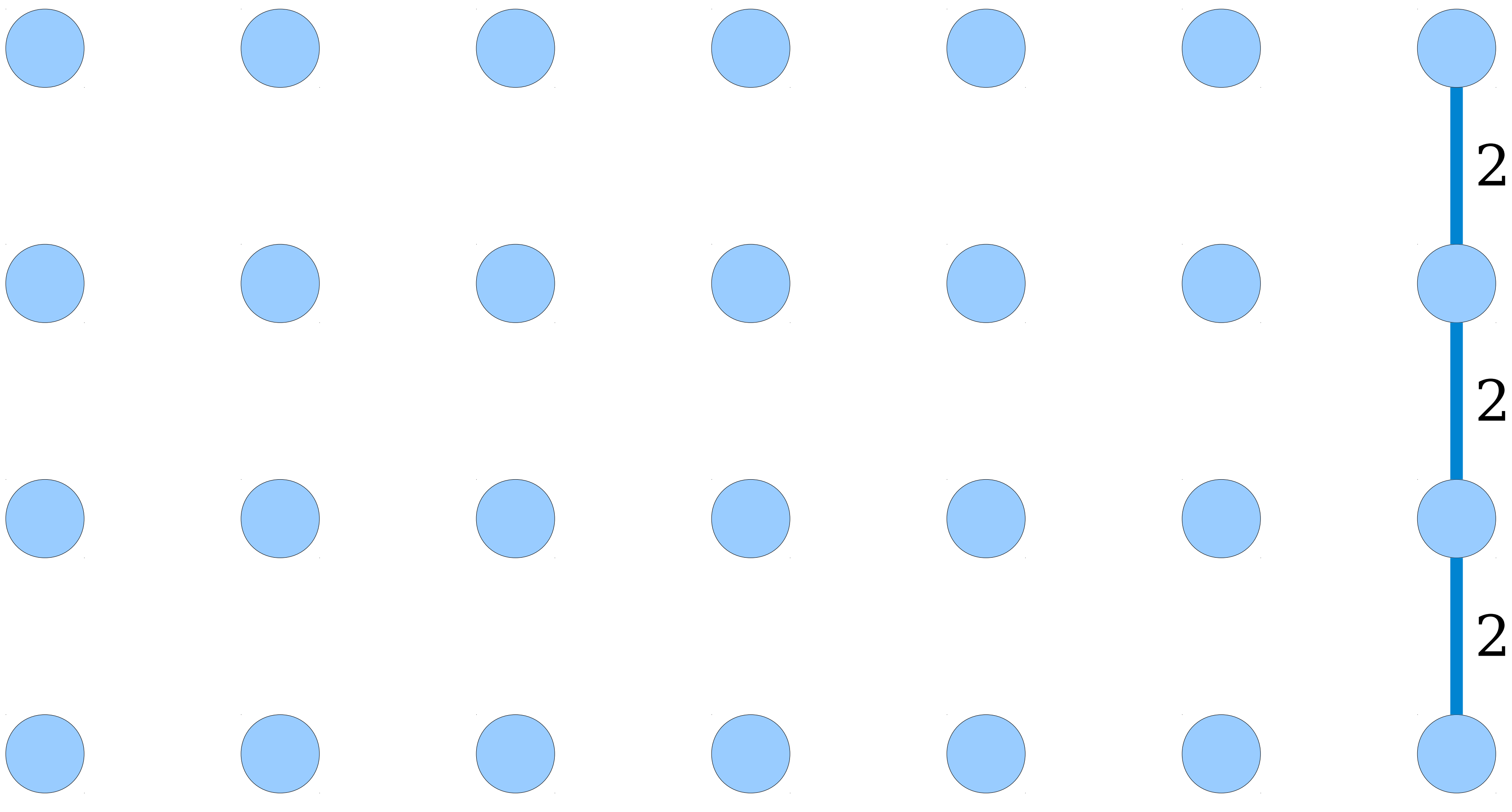
Interpreting $\mathscr{F}_k$



Interpreting $\mathscr{F}_k$



Interpreting $\mathscr{F}_k$



The Representation

- We can now think about our operations in terms of the hierarchical $\mathscr{F}_k$ forests.
- Inserting a new tree edge can be done in time O(log n) by ***link***ing the endpoints in the overall tree $\mathscr{F}_0$.
- Pushing a tree edge e of specificity k up to level $k + 1$ can be done in time O(log n) ***link***ing the endpoints of e in $\mathscr{F}_{k+1}$.
  - No need to ***cut*** them in $\mathscr{F}_k$; the forests are structured so that $\mathscr{F}_{k+1} \subseteq \mathscr{F}_k$.

Details We'll Ignore

- I'm going to gloss over some details, but you can trust me on these:
  - Auxiliary edges are stored in auxiliary data structures. Insertion or deletion takes time O(log n) each.
  - It's possible to iterate across all edges of level k incident to a given tree “efficiently.”
- Check the original paper for details; it's not really worth focusing on right now.

Runtime Analysis

- Connectivity queries can be answered in time O(log n) by querying $\mathscr{F}_0$.
- Inserting an edge takes time O(log n).
- Deleting an auxiliary edge takes time O(log n) (due to bookkeeping overhead.)

Analyzing Deletions

- Deleting a tree edge requires the following:
  - Deleting that tree edge from each of the forests in total time $O(r \log n)$, where r is the number of forests.
  - Possibly push up k edges from one layer to the next in total time $O(k \log n)$.
  - Possibly insert a new edge at some level l, which requires it to be inserted at levels $0, 1, 2, \ldots, l$ for a cost of $O(r \log n)$ if there are r total levels.
- Total cost: $O(r \log n + k \log n)$.
- This can be pretty large – we don't have a bound on r, and k can be $\Theta(m)$.
- ***Can we do better?***

Analyzing Deletions

Deleting a tree edge requires the following:

- Deleting that tree edge from each of the forests in total time $O(r \log n)$, where r is the number of forests.
- Possibly push up k edges from one layer to the next in total time $O(k \log n)$.
- Possibly insert a new edge a… requires it to be inserted at… t of $O(r \log n)$ if there are r to…

Ideally, we'd like to minimize the number of edges we push up from one layer to the next.

Total cost: $O(r \log n + k \log n)$.

This can be pretty large – we don't have a bound on r, and k can be $\Theta(m)$.

Can we do better?

Analyzing Deletions

Deleting a tree edge requires the following:

- Deleting that tree edge from each of the forests in total time $O(r \log n)$, where r
- Possibly push up k edges in total time $O(k \log n)$.
- Possibly insert a new edge at some level l, which requires it to be inserted at levels 0, 1, 2, …, l for a cost of $O(r \log n)$ if there are r total levels.

We'd also like to keep the number of layers as low as possible.

Total cost: $O(r \log n + k \log n)$.

This can be pretty large – we don't have a bound on r, and k can be $\Theta(m)$.

Can we do better?

Fixing the Problem

- When we cut a tree edge at some level k, we'll split some tree T at level k into two trees T_0 and T_1.
- ***Idea:*** Push the edges of the smaller of T_0 and T_1 to the next level.
- ***Claim 1:*** Every edge will be pushed up at most O(log n) times.
  - Think back to PS4's analysis of building weight-balanced trees: always charging the work to the smaller component is often a good idea.
- ***Claim 2:*** The maximum number of levels is now O(log n).
  - This follows from claim 1.

The Final Analysis

- Deleting a tree edge requires the following:
  - Deleting that tree edge from each of the lg n forests in total time $O(\log^2 n)$.
  - Possibly push up k edges from one layer to the next in total time $O(k \log n)$.
  - Possibly insert an edge into lg n forests in total time $O(\log^2 n)$.
- Total cost: $O(\log^2 n + k \log n)$
- ***Claim:*** We can amortize the $k \log n$ term away.
  - Each edge gets pushed up at most $O(\log n)$ times. What if we pay for those insertions up front?
  - ***Idea:*** When we add an edge, place $O(\log n)$ credits on it. Each credit can pay for pushing the edge up one layer.
- Amortized cost of a delete: $\mathbf{O(\log^2 n)}$.

The Final Analysis

- Insertions have a base cost of $O(\log n)$.
- As part of our amortization scheme, we place $O(\log n)$ credits on each inserted edge.
- Each credit pays for the $O(\log n)$ work of inserting an edge at a higher level.
- Amortized cost: $\mathbf{O(\log^2 n)}$.

The Final Analysis

- Deletions have cost $O(\log^2 n + k \log n)$, where k is the number of edges promoted.
- Can spend one credit from each edge as it's promoted; won't run out of credits.
- Amortized cost: $\mathbf{O(\log^2 \boldsymbol{n})}$.

The Final Analysis

- The dynamic graph data structure supports the following operations in the indicated amortized runtimes:
  - ***is-connected***: O(log n)
  - ***insert***: O(log² n)
  - ***delete***: O(log² n)
- This is significantly better than the naïve solution!
- Can we do better?

One Quick Speedup

- ***Recall:*** Each Euler tour tree is represented by a balanced BST.
- Lookup times in Euler tour trees is proportional to the tree height.
  - Walk from each node up to the root and compare whether the roots are the same.
- If we represent $\mathscr{F}_0$ (and just $\mathscr{F}_0$) using a B-tree of order $\Theta(\log n)$, queries can be answered in time

  $$\mathrm{O}(\log_{\log n} n) = \mathbf{O(\log \boldsymbol{n} / \log \log \boldsymbol{n})}$$

  while insertions take time

  $$\mathbf{O(\log^2 \boldsymbol{n} / \log \log \boldsymbol{n})},$$

  which doesn't affect the overall runtime.

The Final Analysis

- The dynamic graph data structure, with the B-tree modification, supports the following operations in the indicated amortized runtimes:
  - ***is-connected***: O(log n / log log n)
  - ***insert***: O($\log^2 n$)
  - ***delete***: O($\log^2 n$)

Going Forward

- Since this data structure was developed in 1999, there have been some new developments.
- If randomization is allowed, we can get these bounds:
  - ***is-connected***: O(log n / log log log n)
  - ***insert***: O(log n · (log log n)^3) amortized
  - ***delete***: O(log n · (log log n)^3) expected amortized
- A lower-bound of Ω(log n) per insert or deletion is known to exist, and there's still a gap!

More Dynamic Problems

- Many other dynamic graph problems exist:
  - Maintaining an MST; can do in $O(\log^4 n)$ time per insertion or deletion.
  - Maintaining single-source or all-pairs shortest paths.
  - Maintaining reachability in a *directed* graph.
- All of these problems were solved in the static case 40+ years ago.
- We have somewhat decent solutions to the dynamic cases.
- ***This is an active area of research!***

Next Time

- ***The Big Picture***
  - Wow, we covered a lot! What exactly did we see in this class?
- ***Your Questions***
  - What didn't we cover that you wanted to learn in this class?
- ***Where to Go From Here***
  - Next steps in theory (and in life?)