Euler Tour Trees

Outline for Today

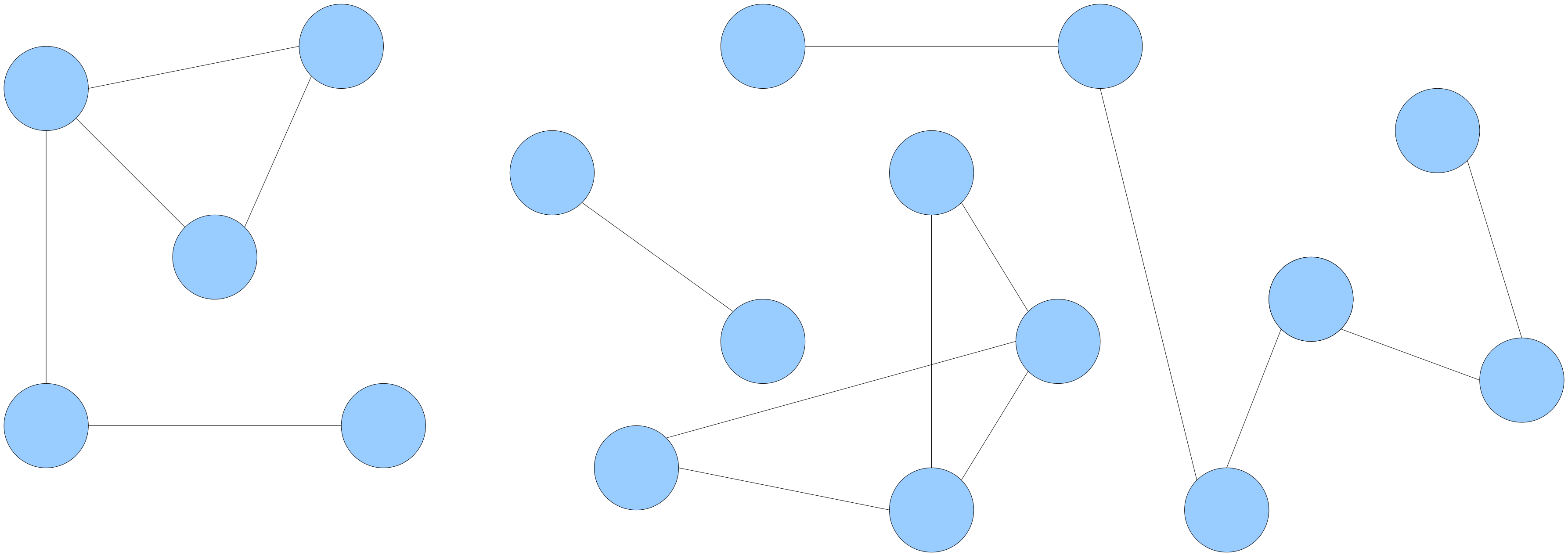
- ***Dynamic Connectivity***
  - Figuring out what's connected in a graph as the edges change.
- ***Euler Tour Representations***
  - An inspired and clever way to represent trees.
- ***Euler Tour Trees***
  - Encoding Euler tours in a creative way.
- ***Extending ETTs***
  - Extending our basic structure.

The Dynamic Connectivity Problem

The Connectivity Problem

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

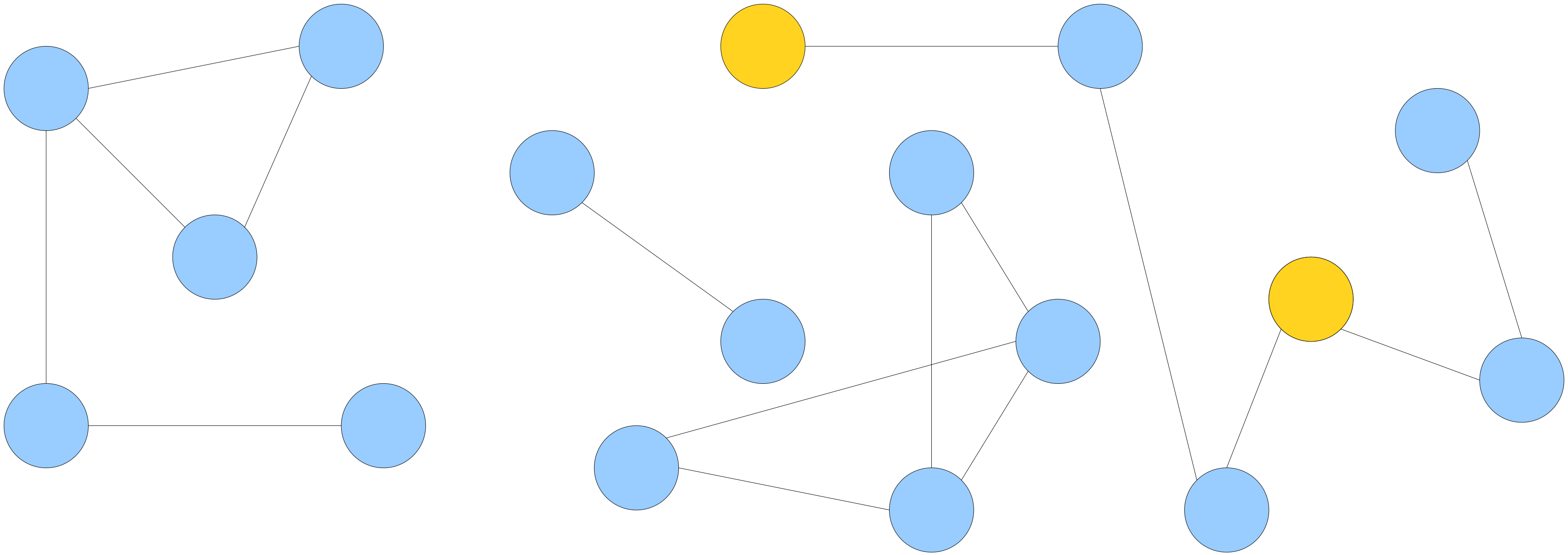
  Given an undirected graph *G*, preprocess the graph so we can answer queries of the form "are nodes *u* and *v* connected?"



The Connectivity Problem

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

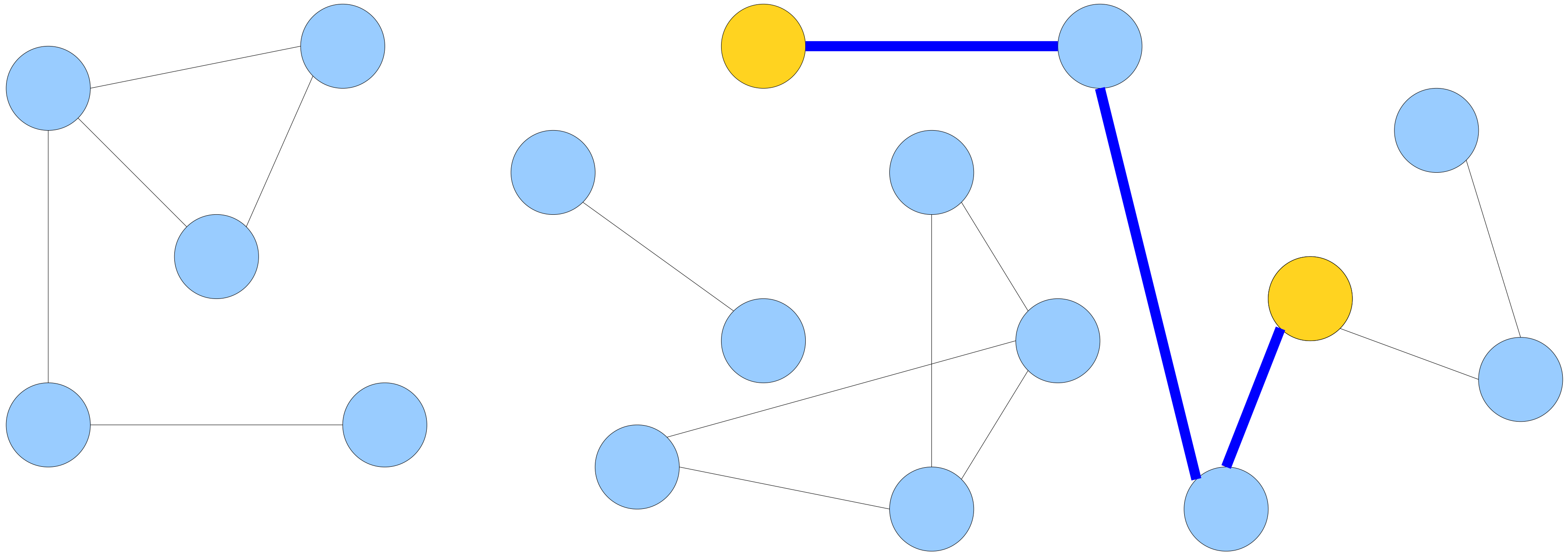
Given an undirected graph *G*, preprocess the graph so we can answer queries of the form "are nodes *u* and *v* connected?"



The Connectivity Problem

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

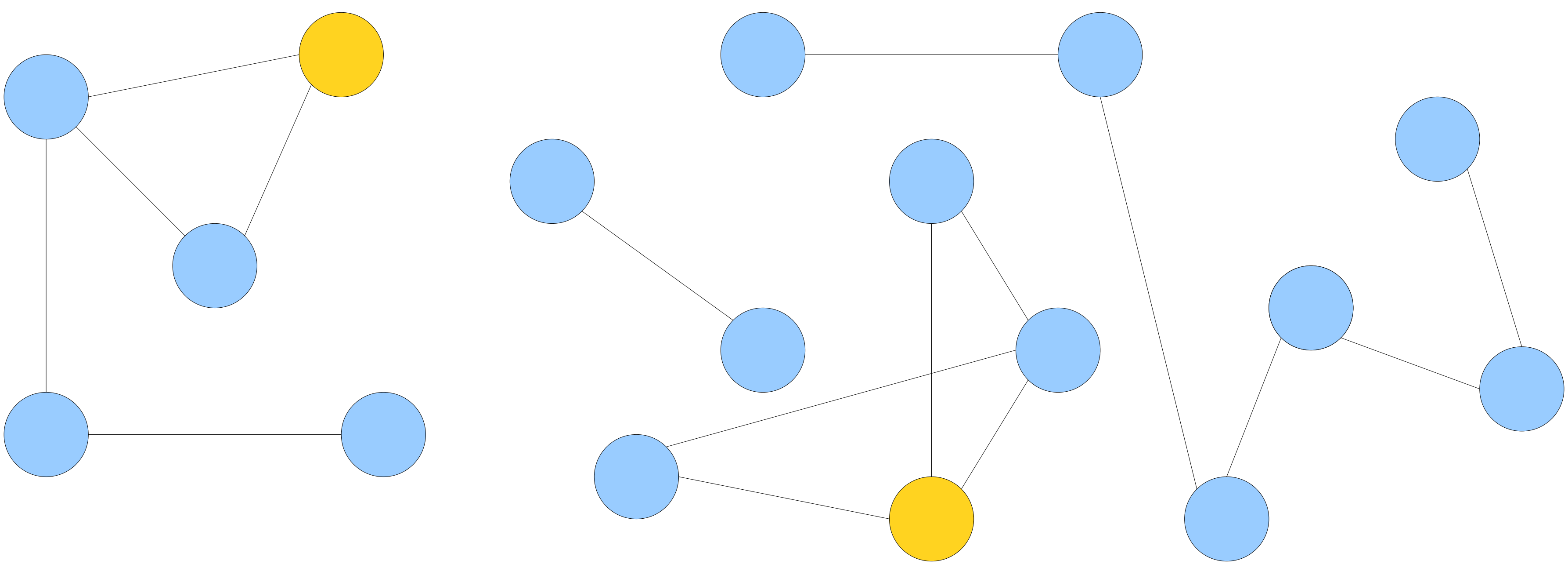
  Given an undirected graph *G*, preprocess the graph so we can answer queries of the form “are nodes *u* and *v* connected?”



The Connectivity Problem

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

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

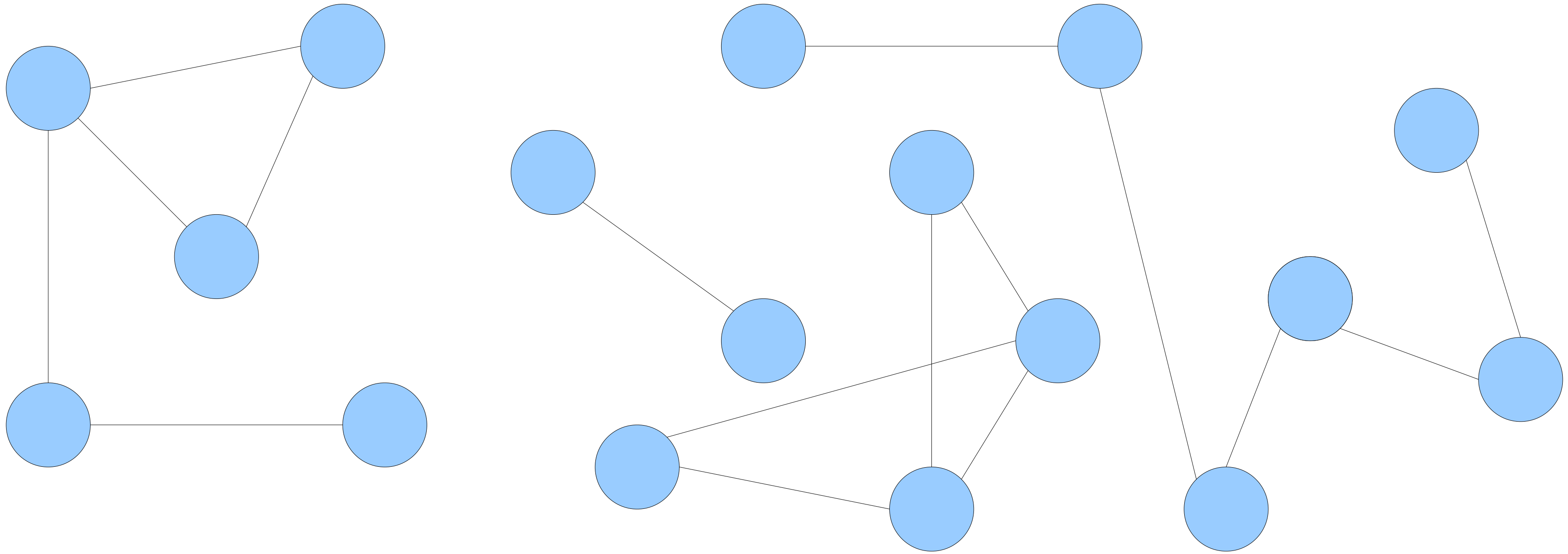


The Connectivity Problem

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

  Given an undirected graph *G*, preprocess the graph so we can answer 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).

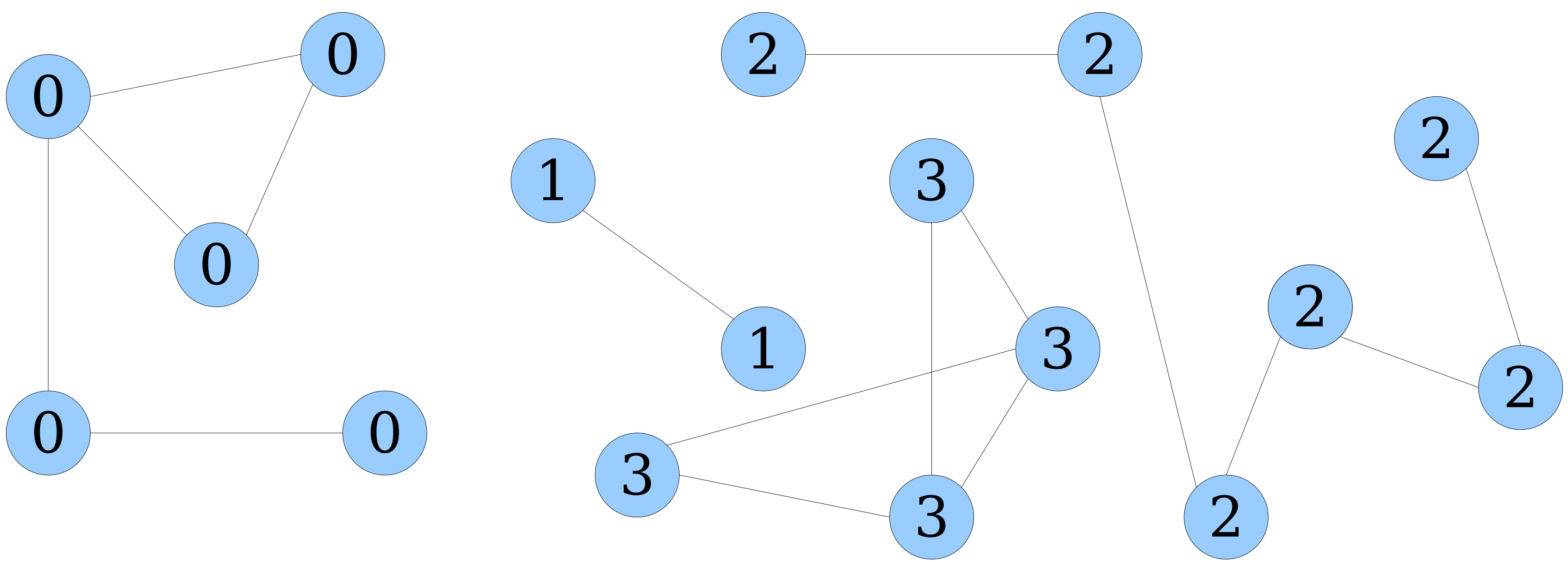


The Connectivity Problem

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

  Given an undirected graph *G*, preprocess the graph so we can answer 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).

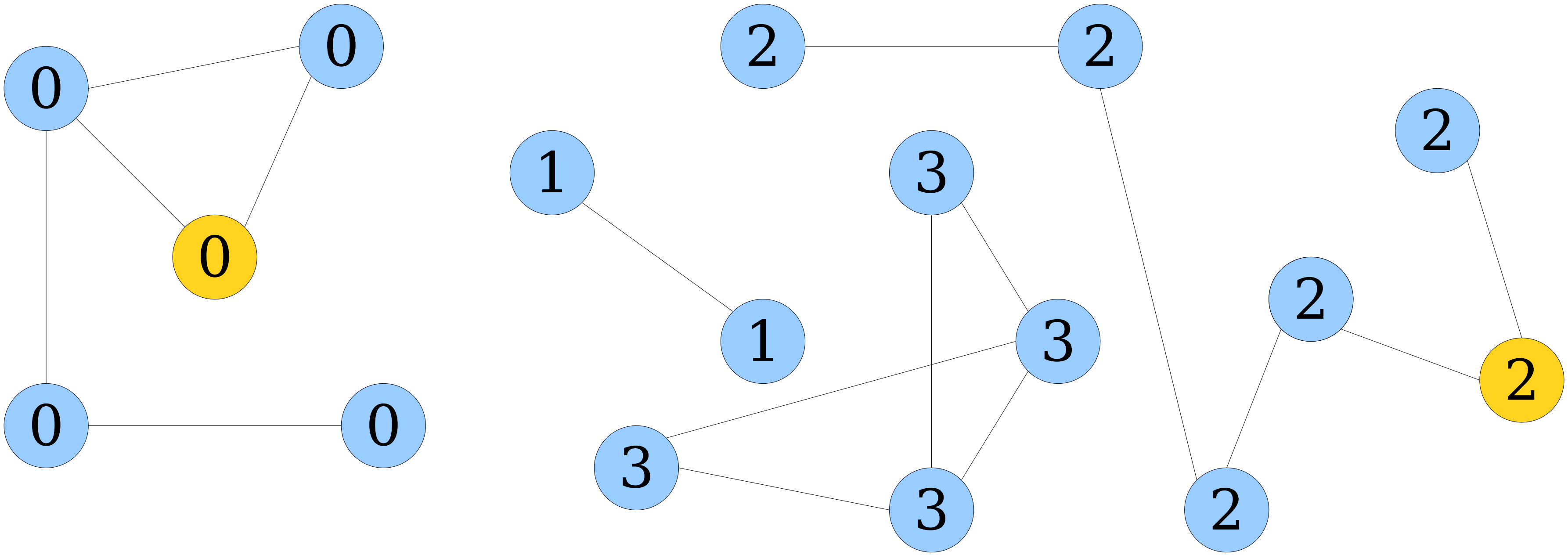


The Connectivity Problem

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

  Given an undirected graph G, preprocess the graph so we can answer 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).

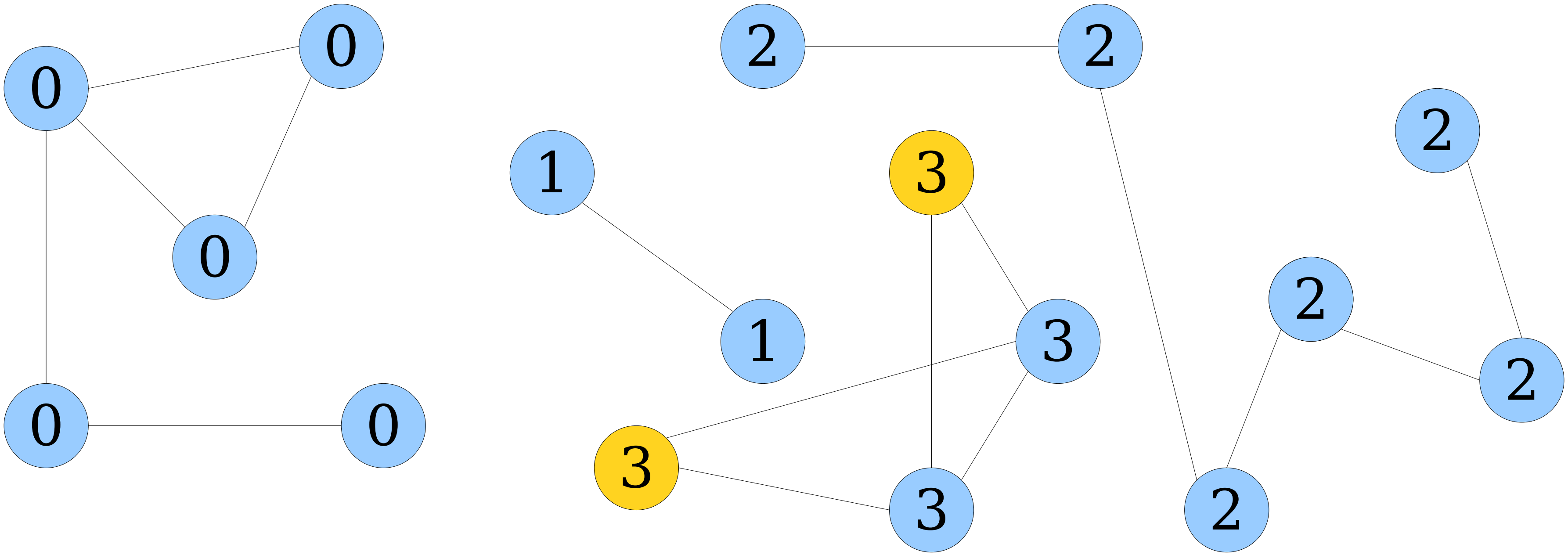


The Connectivity Problem

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

  Given an undirected graph G, preprocess the graph so we can answer 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).

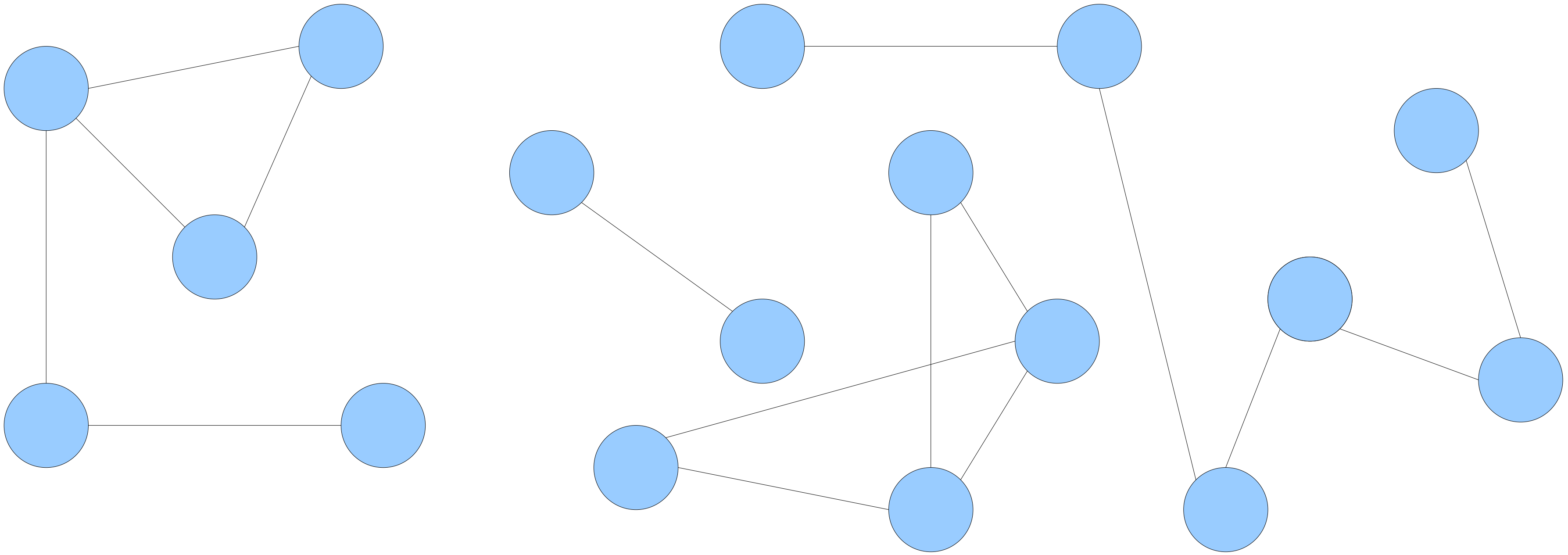


Dynamic Connectivity

- 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!

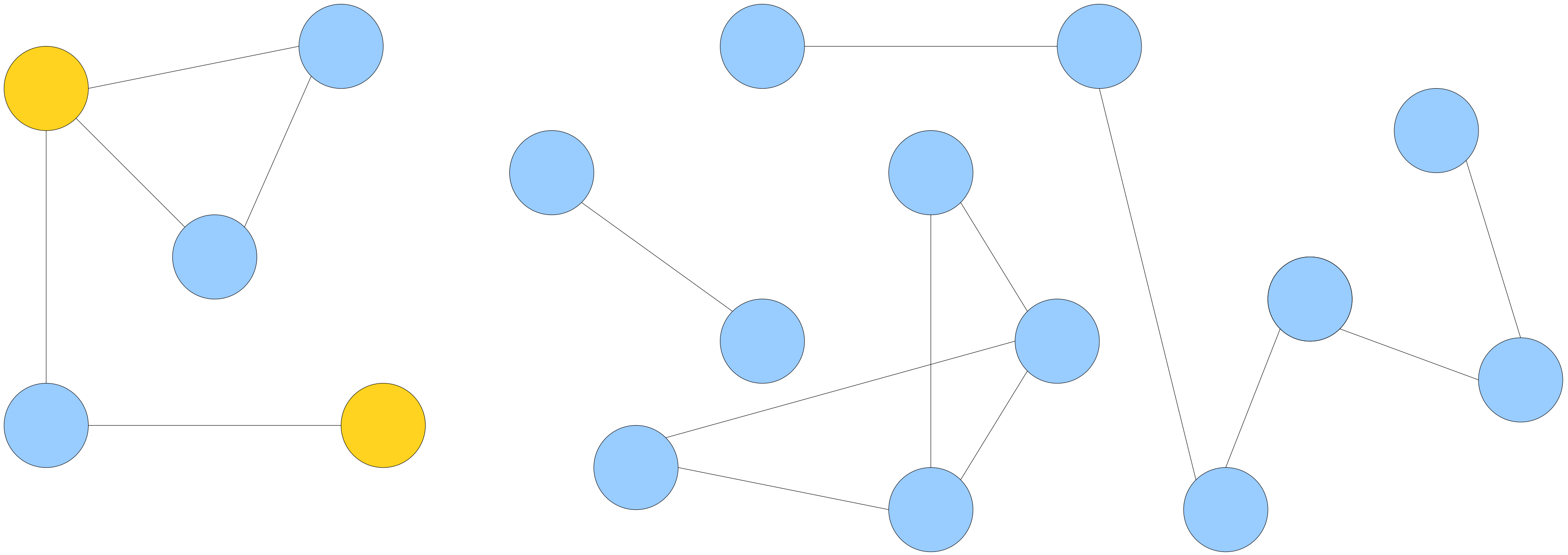


Dynamic Connectivity

- 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!

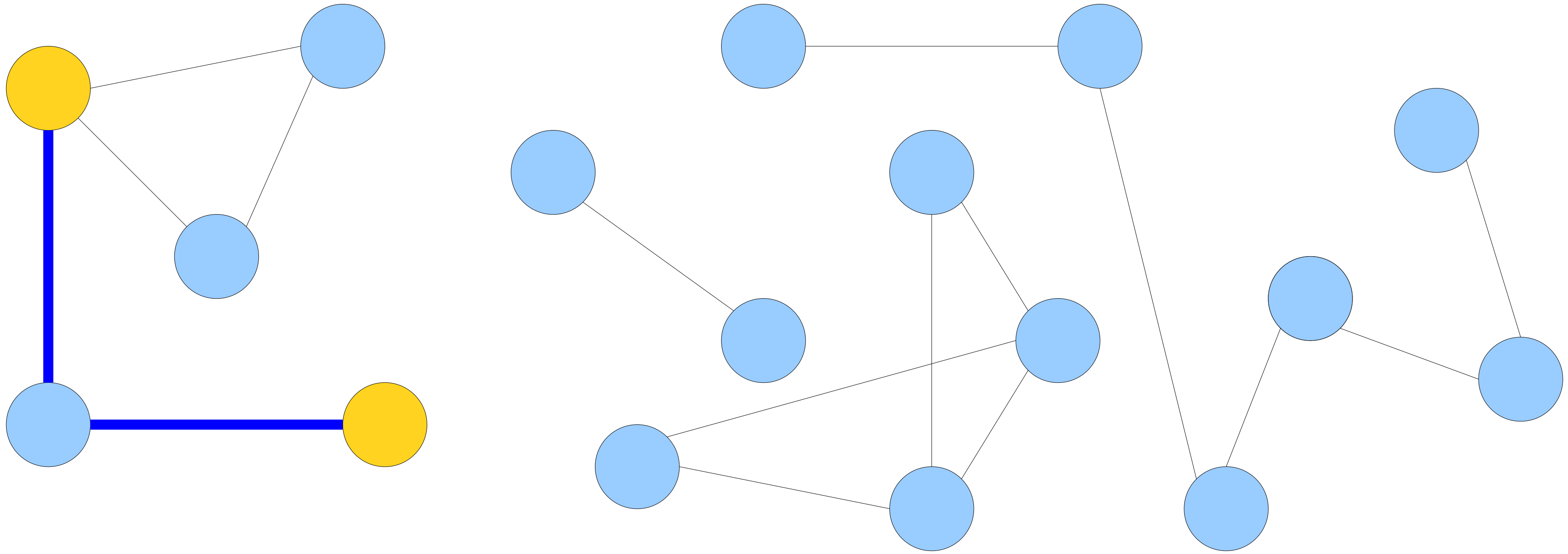


Dynamic Connectivity

- 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!

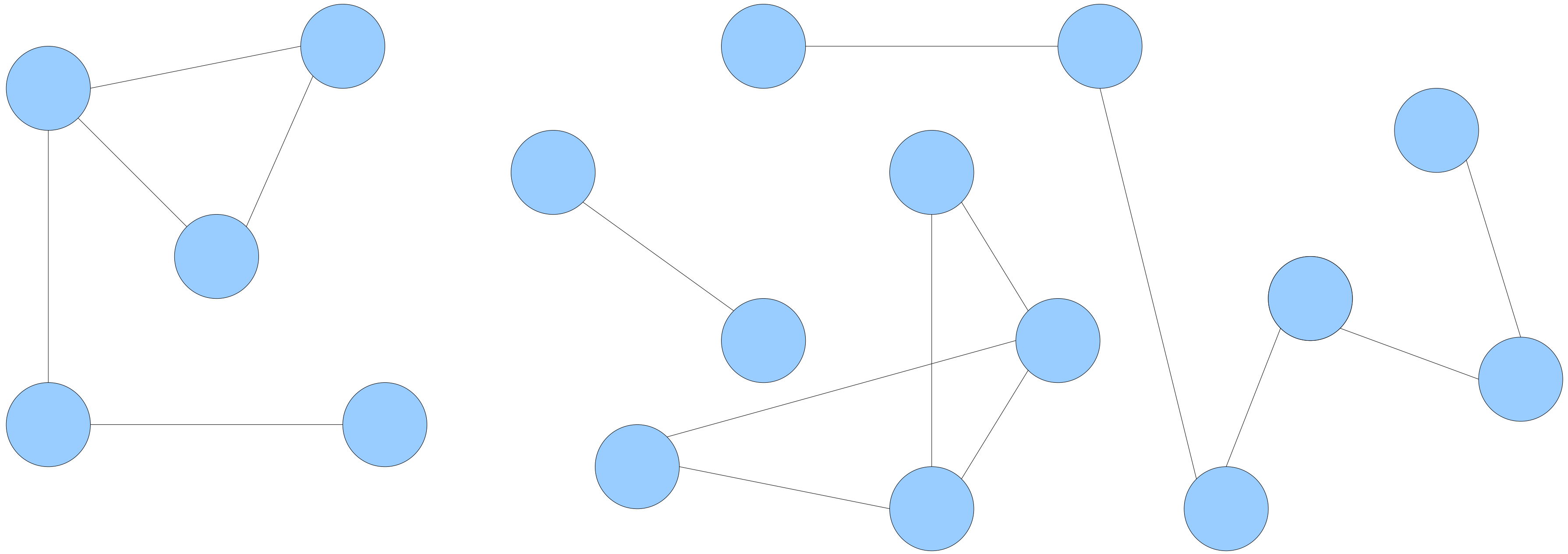


Dynamic Connectivity

- 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!

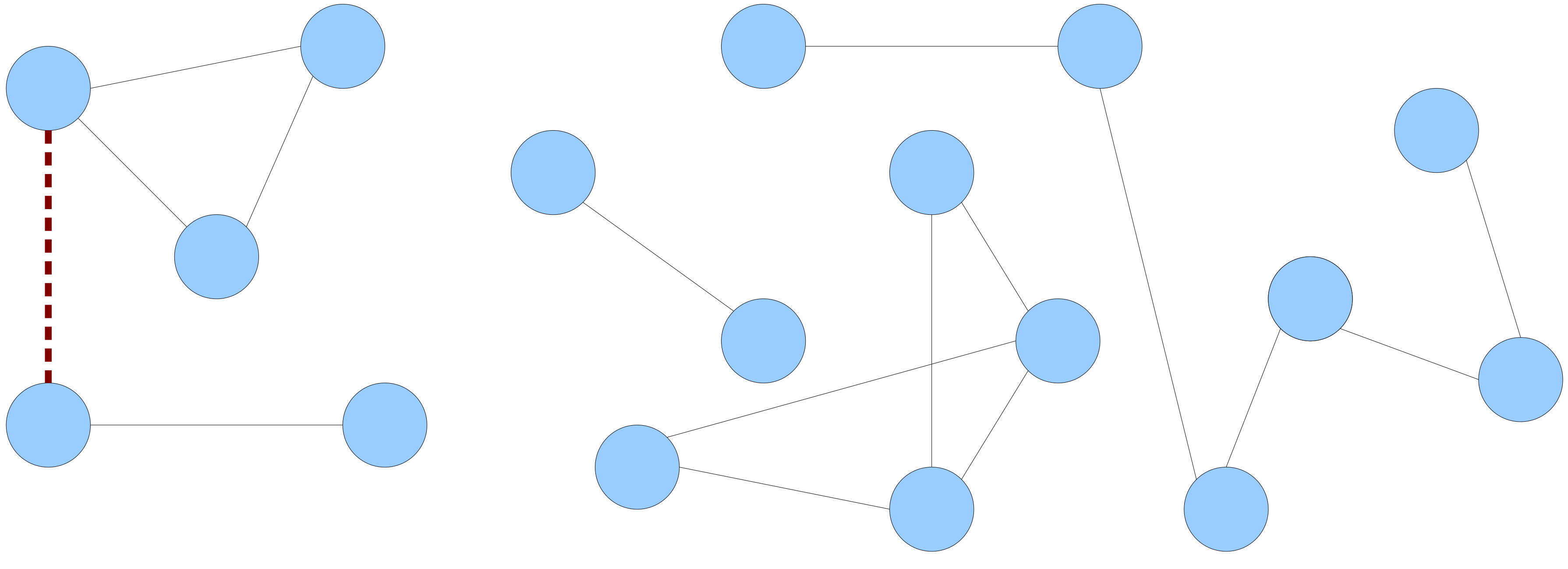


Dynamic Connectivity

- 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!

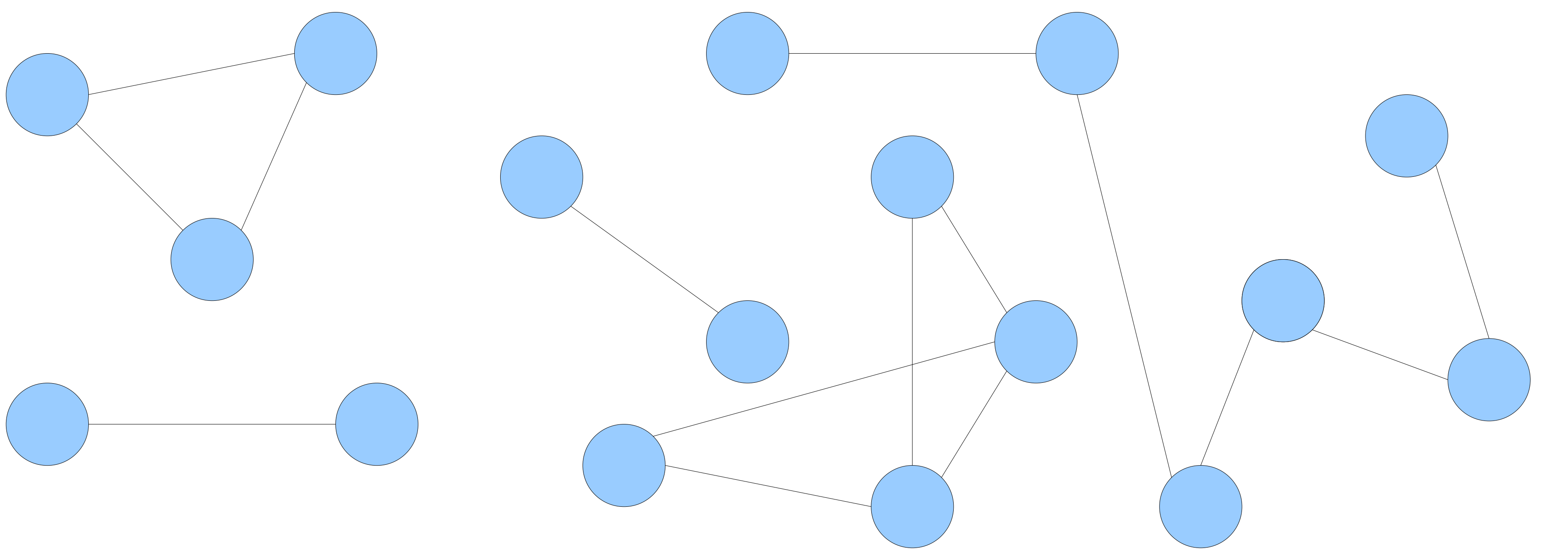


Dynamic Connectivity

- 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!

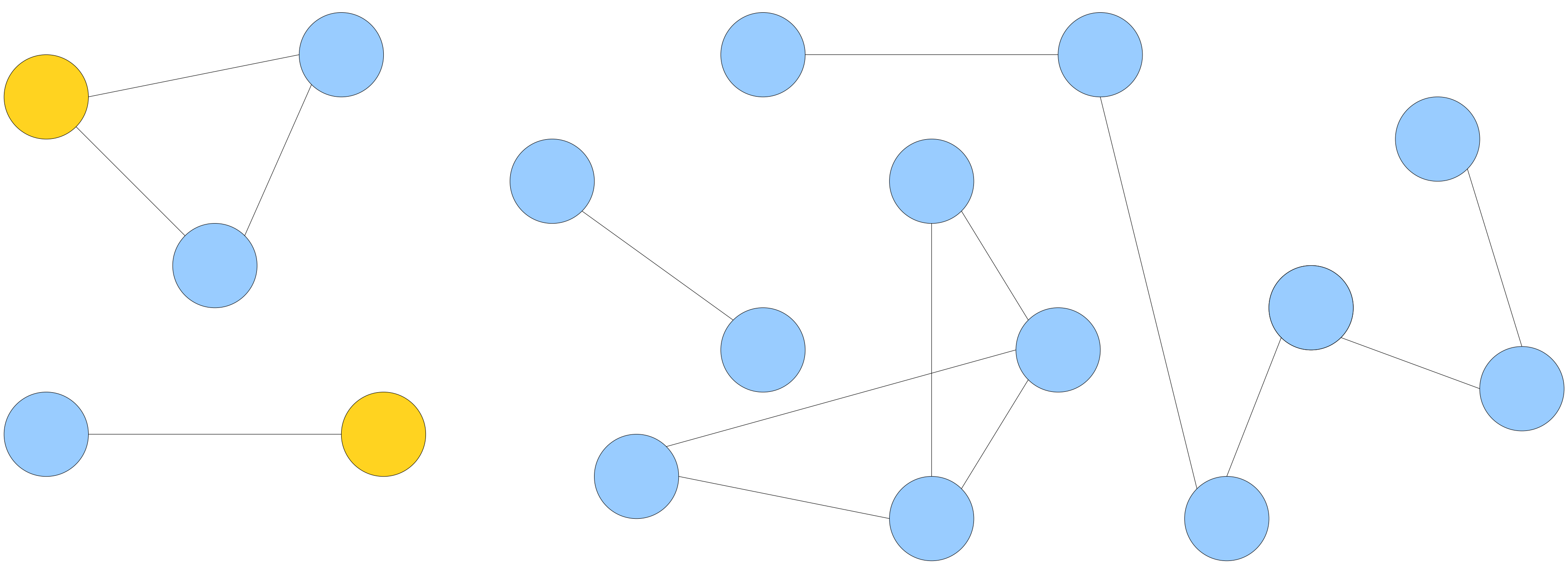


Dynamic Connectivity

- 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!

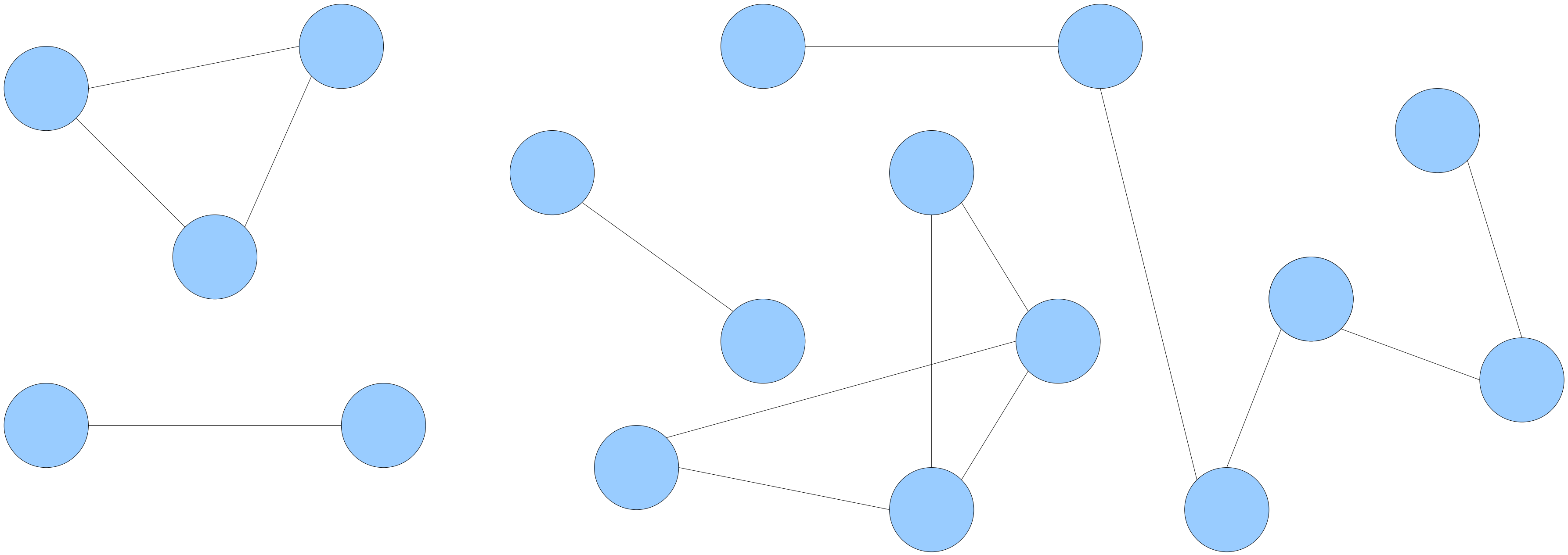


Dynamic Connectivity

- 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!

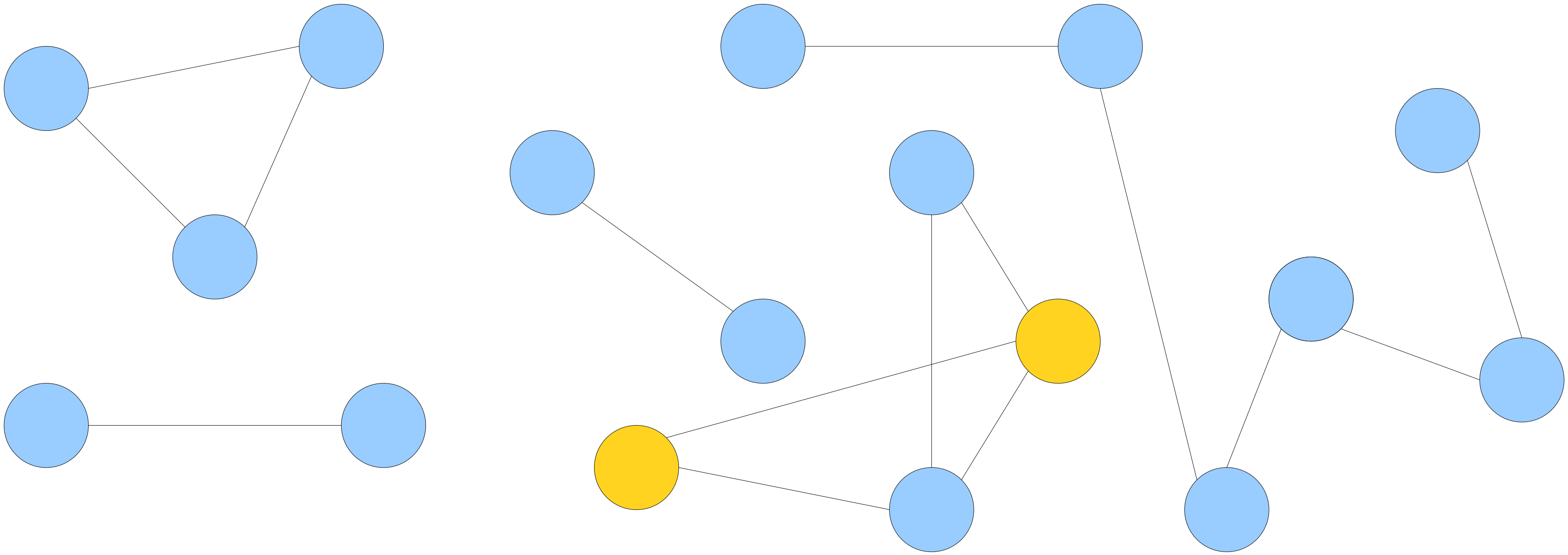


Dynamic Connectivity

- 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!

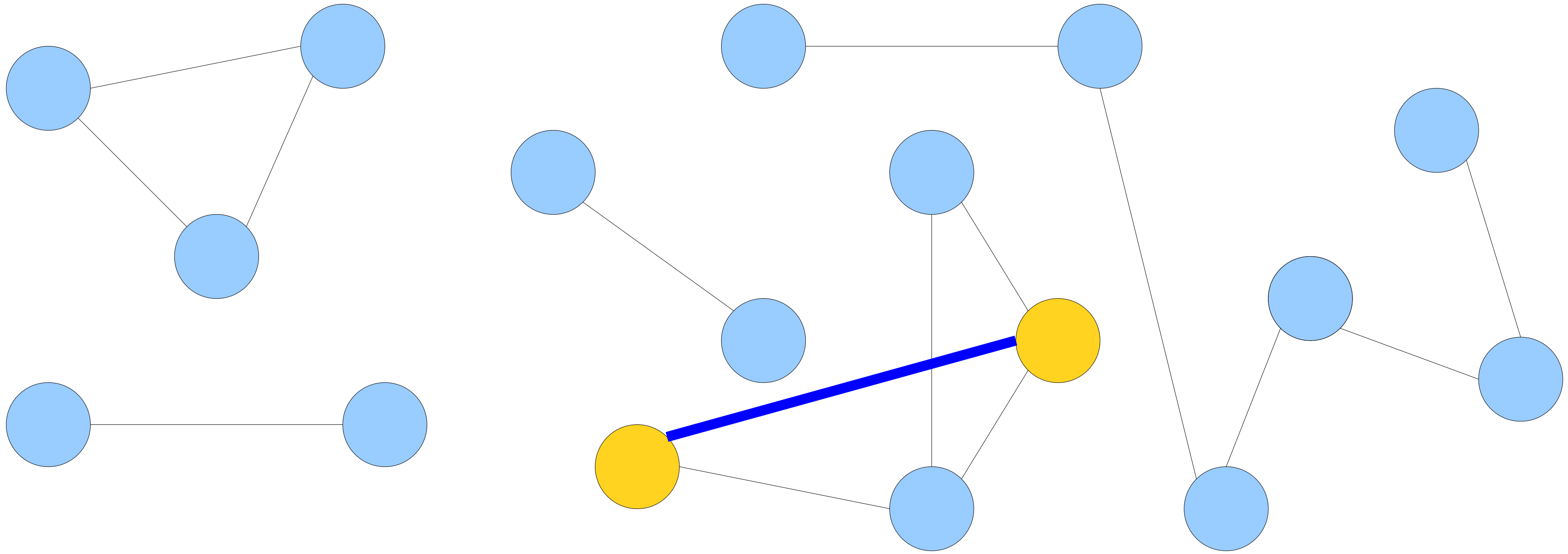


Dynamic Connectivity

- 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!

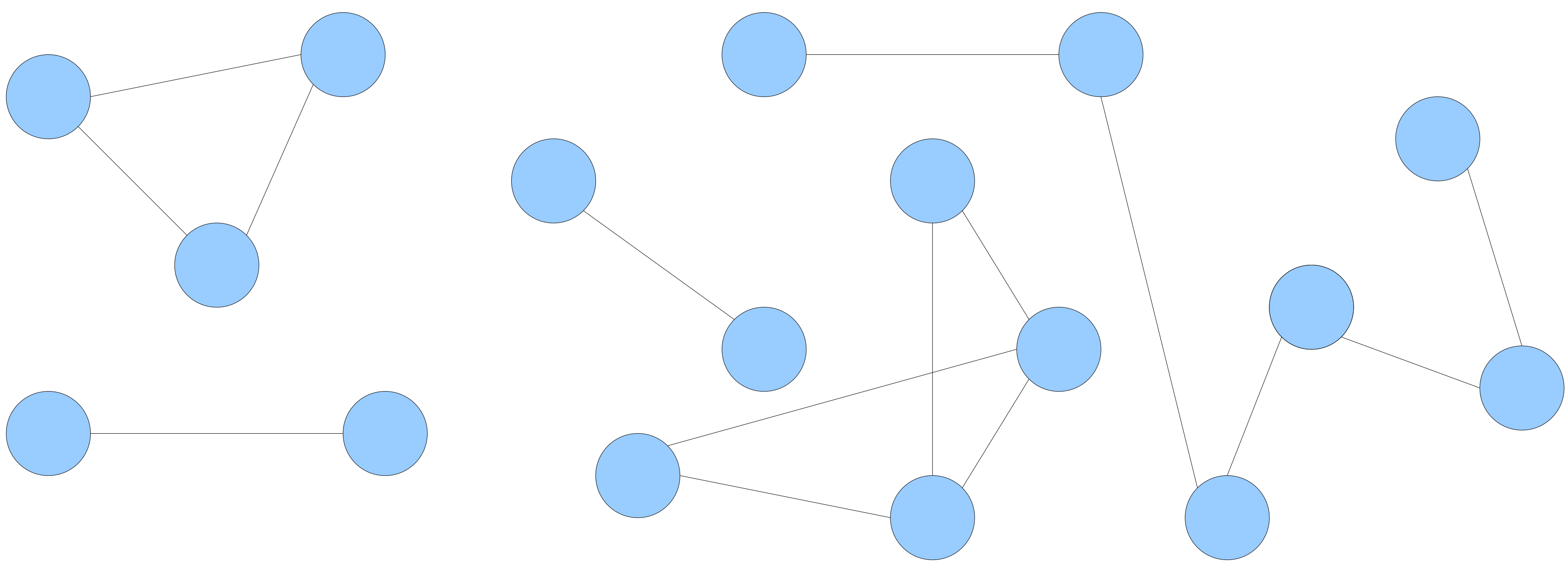


Dynamic Connectivity

- 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!

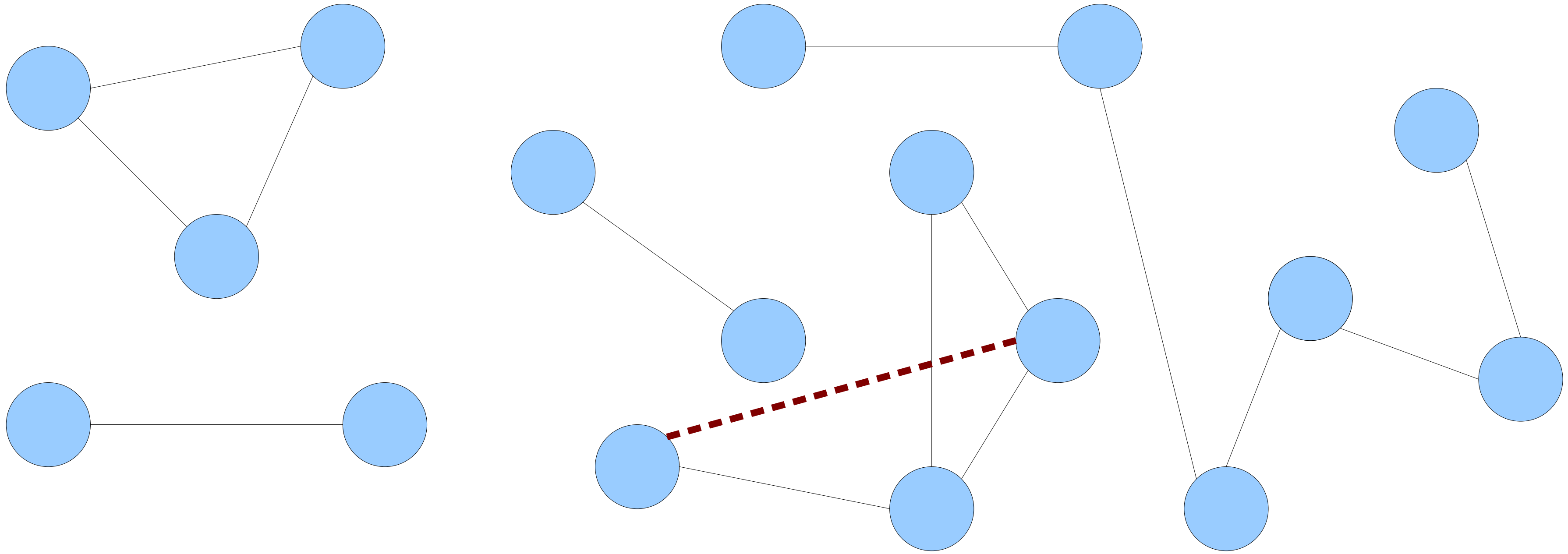


Dynamic Connectivity

- 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!

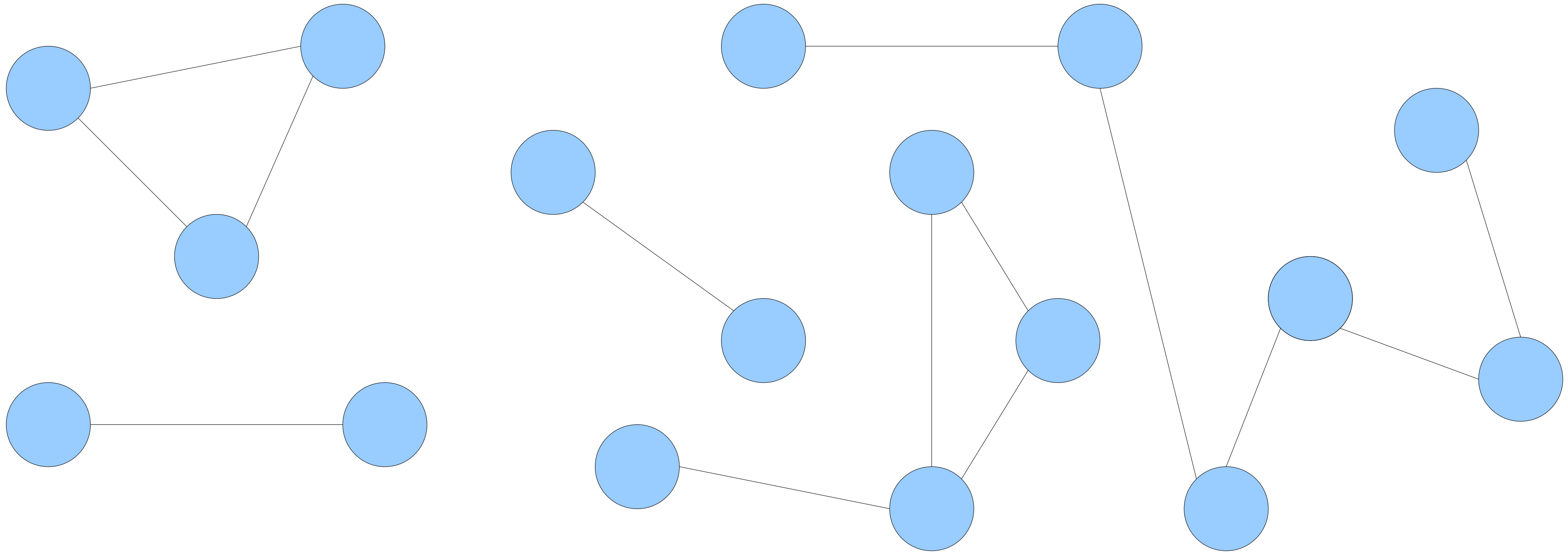


Dynamic Connectivity

- 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!

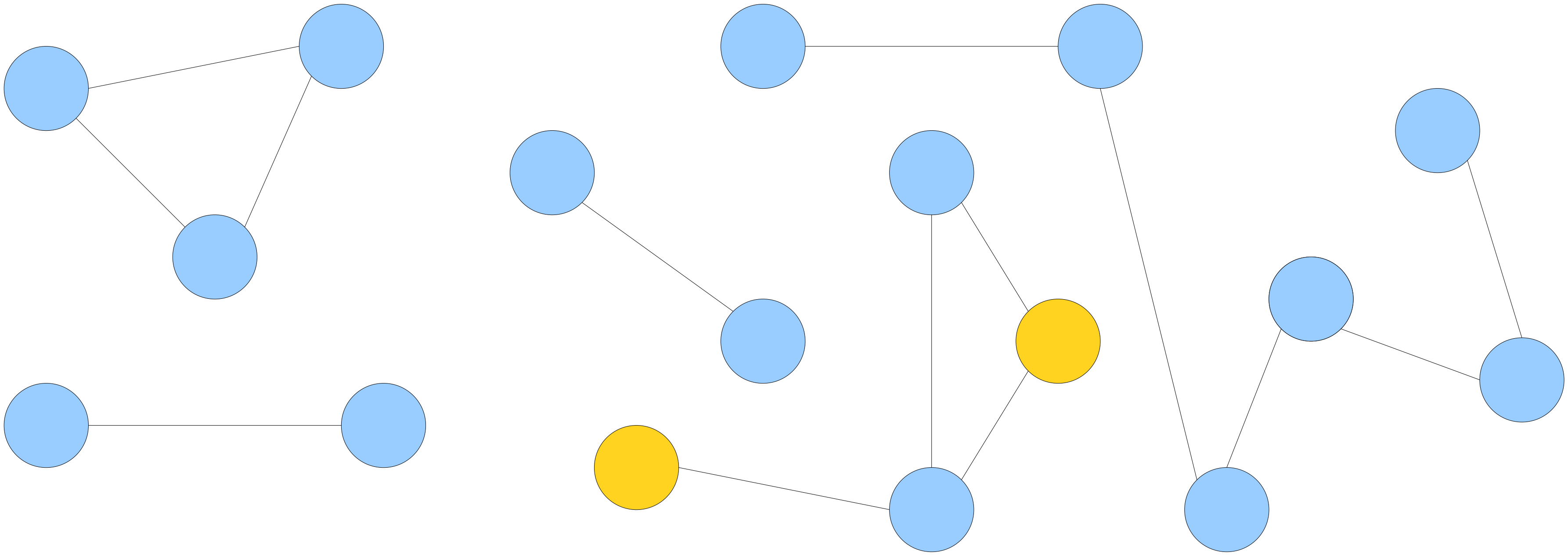


Dynamic Connectivity

- 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!

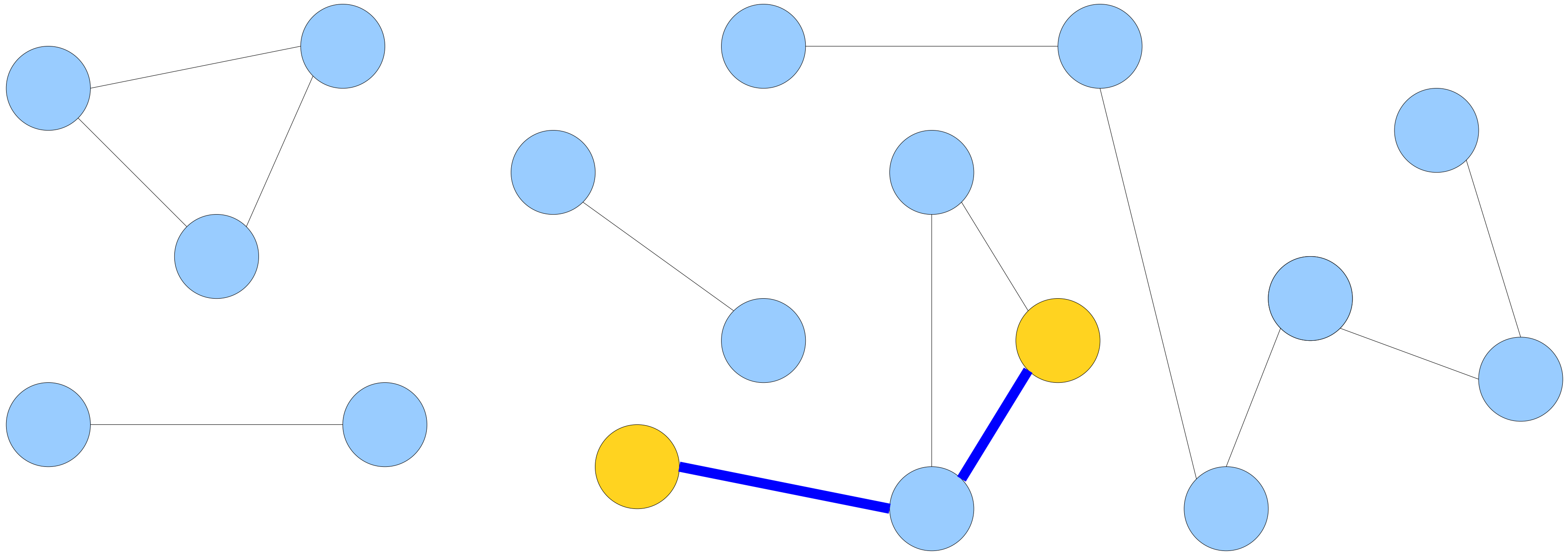


Dynamic Connectivity

- 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!

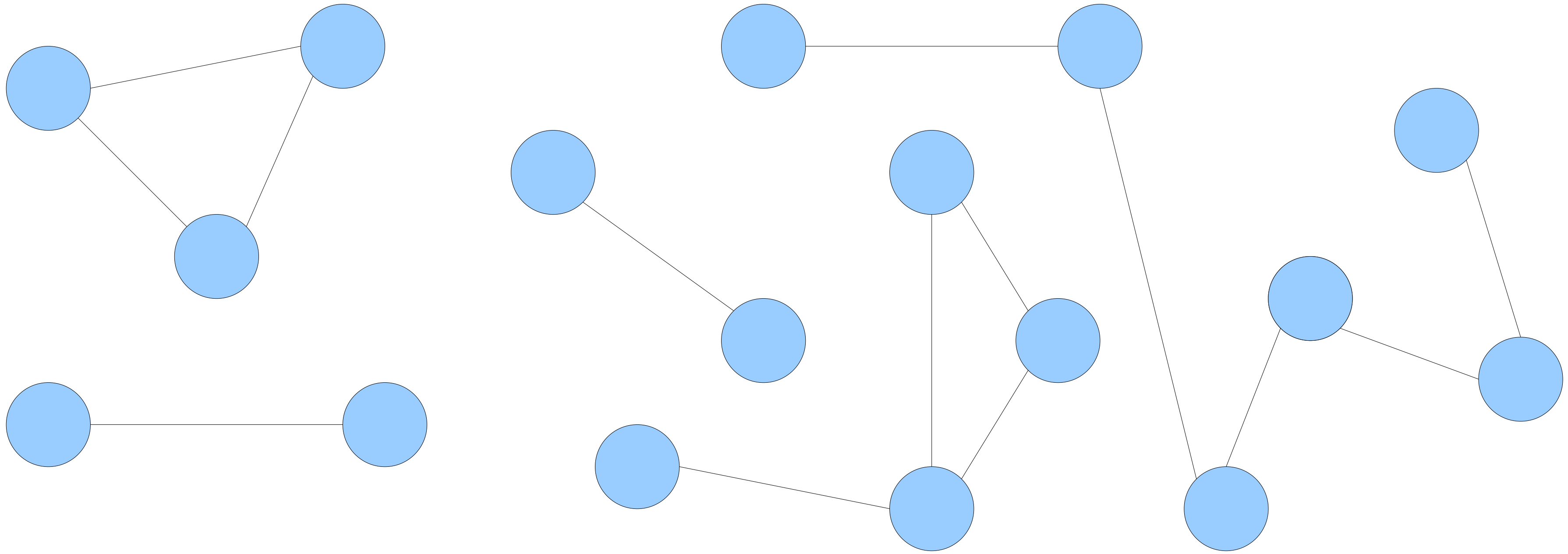


Dynamic Connectivity

- 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!

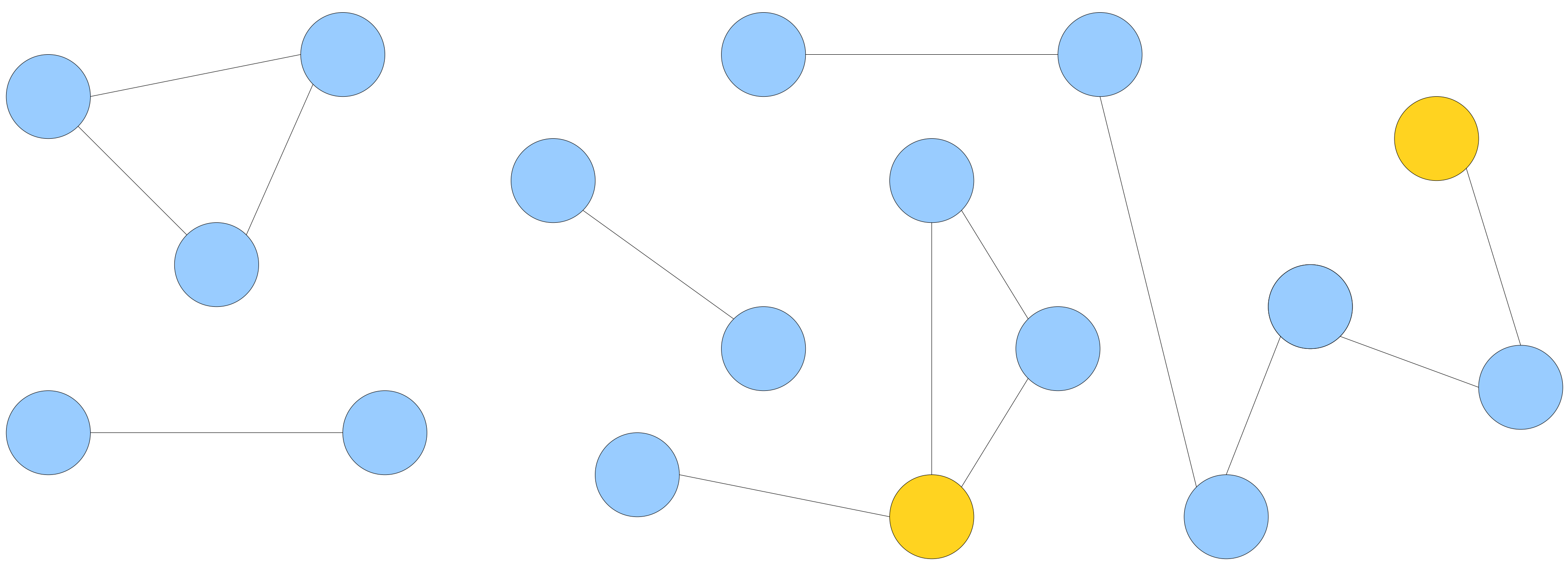


Dynamic Connectivity

- 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!

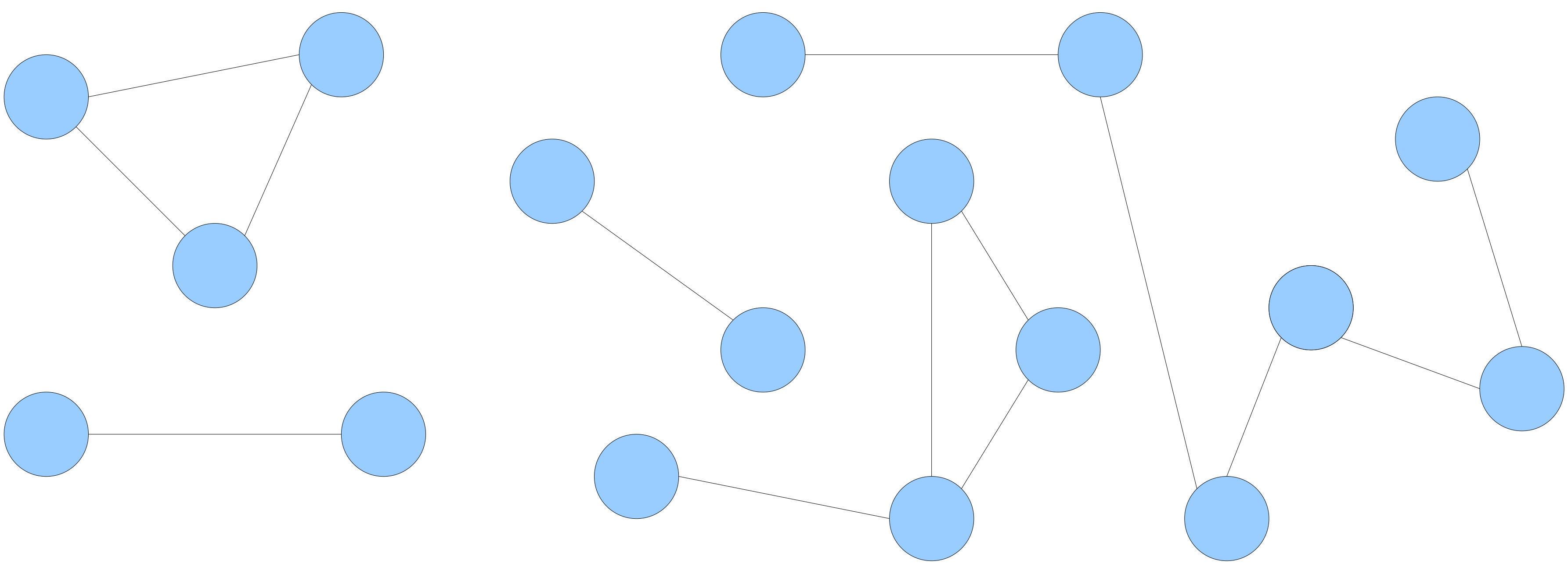


Dynamic Connectivity

- 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!

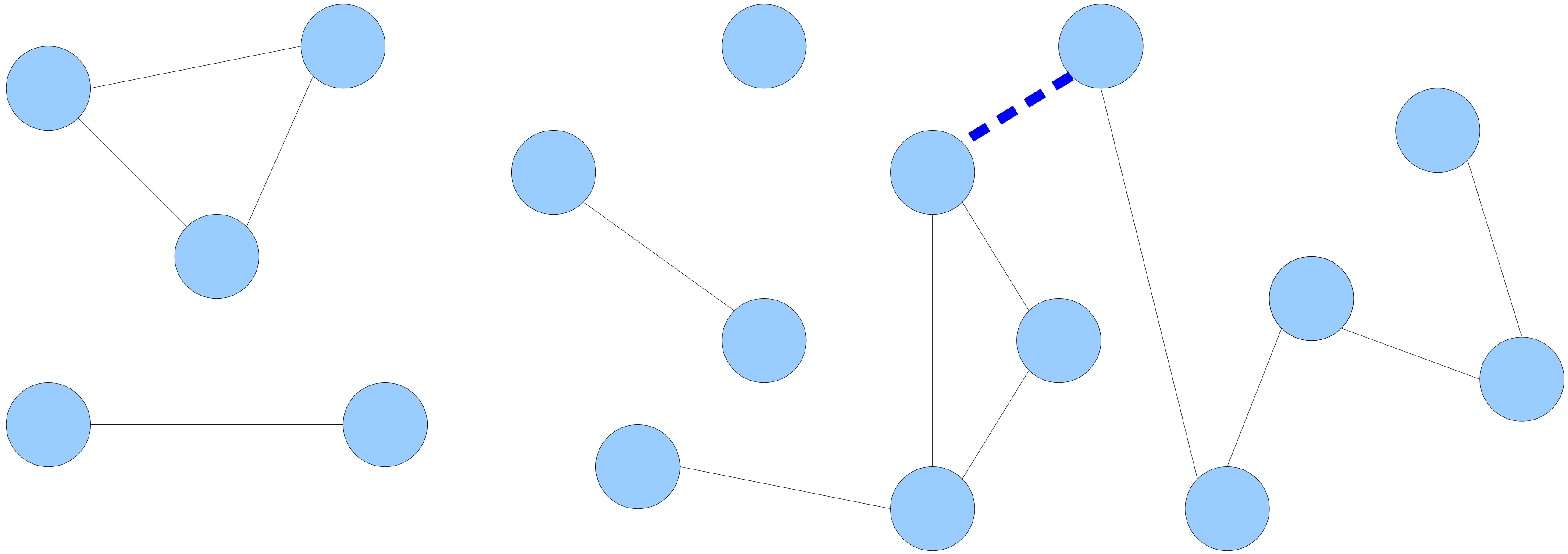


Dynamic Connectivity

- 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!

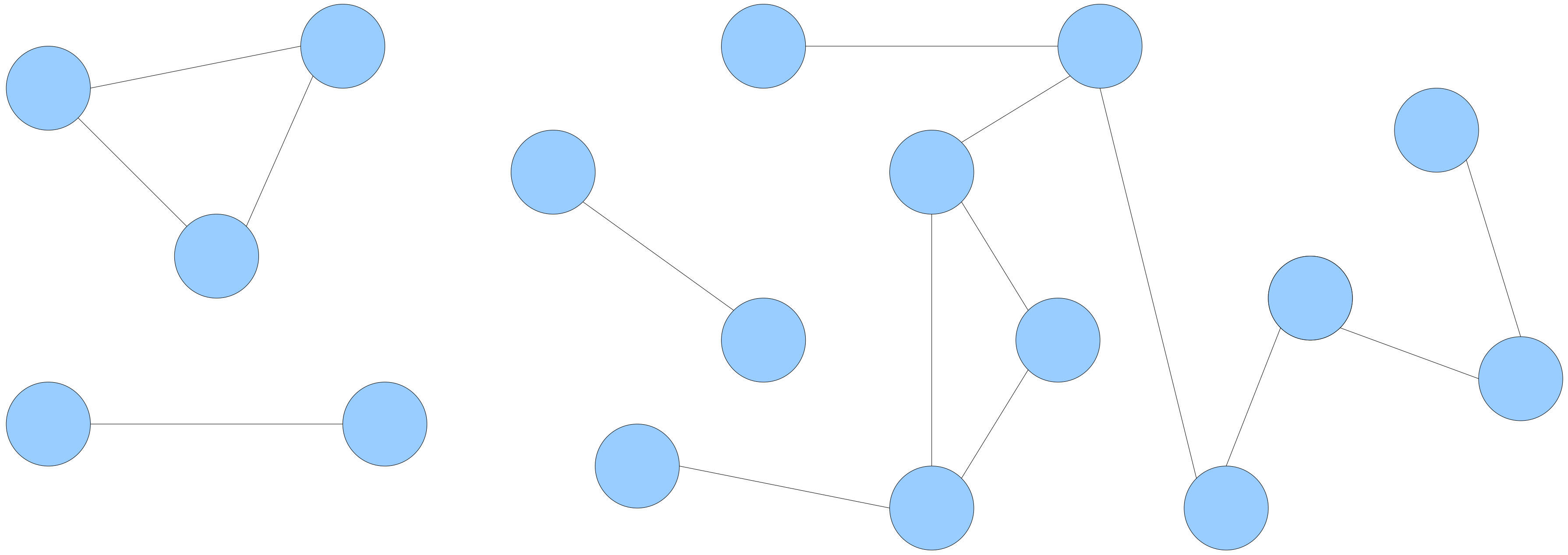


Dynamic Connectivity

- 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!

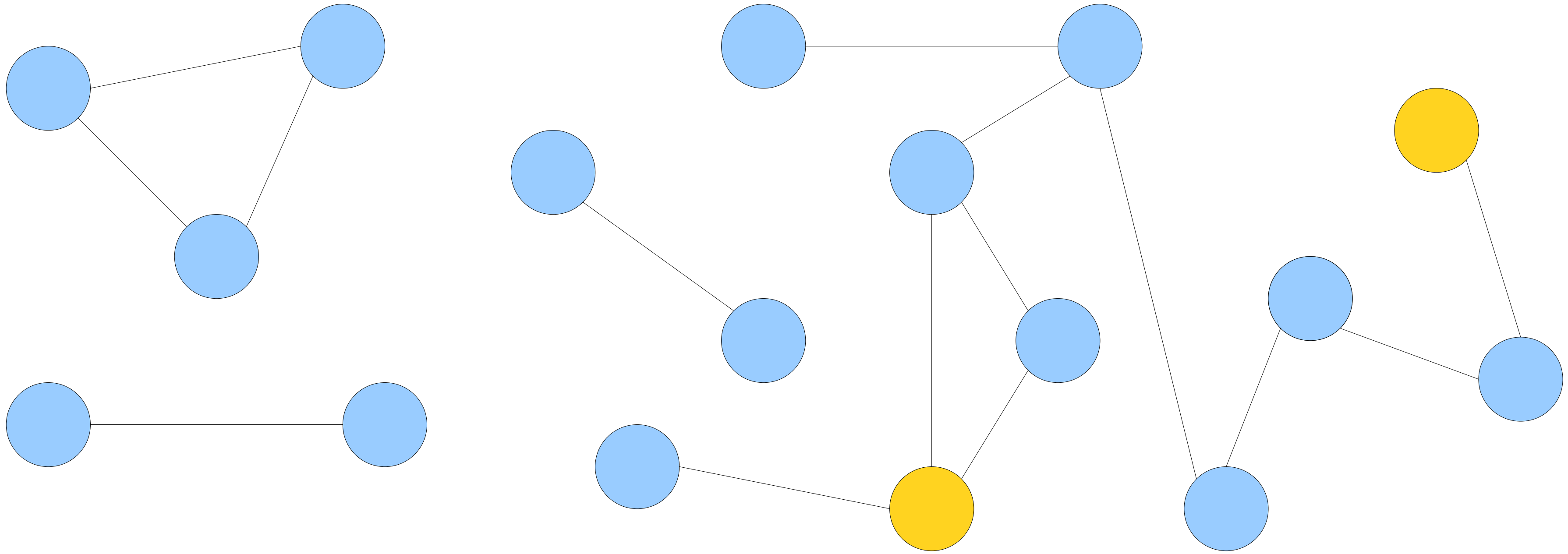


Dynamic Connectivity

- 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!

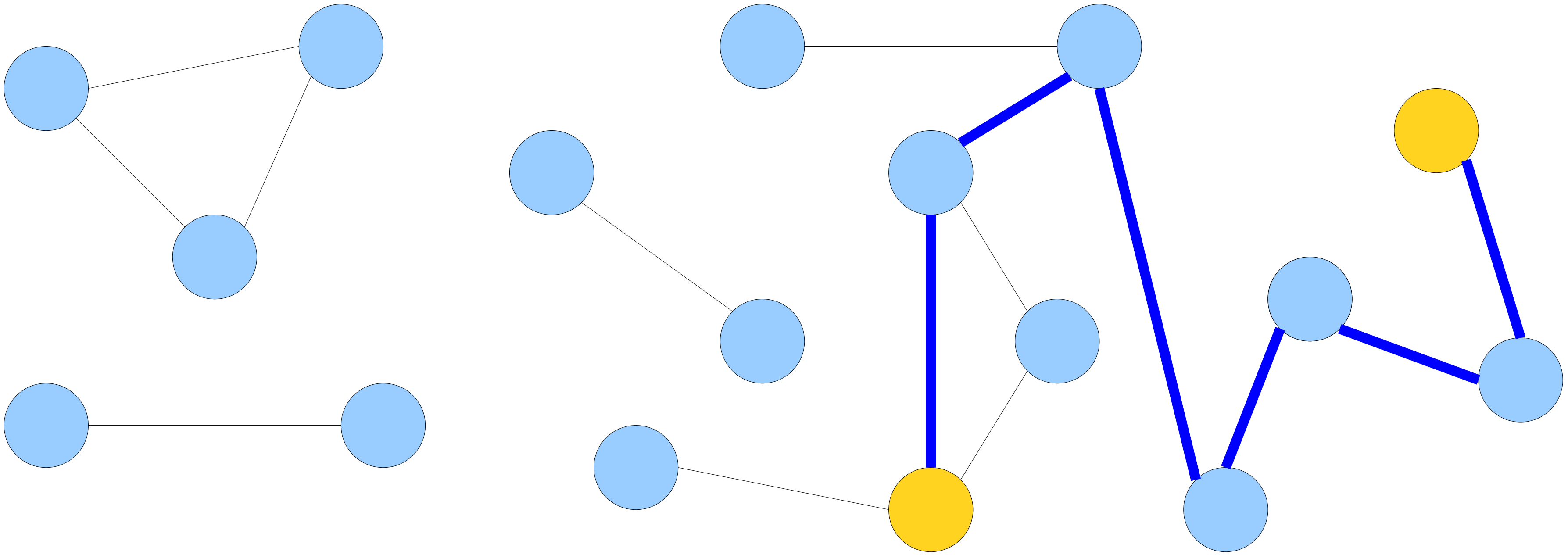


Dynamic Connectivity

- 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!



Special Cases

- Last time, we covered the ***incremental connectivity problem*** in which edges can only be added and not removed.
- Today, we'll cover ***dynamic connectivity in forests***, a special case in which the graph is known to be a forest.
- Next time, we'll cover ***fully-dynamic connectivity***, in which there are no restrictions on which edges can be added and removed.

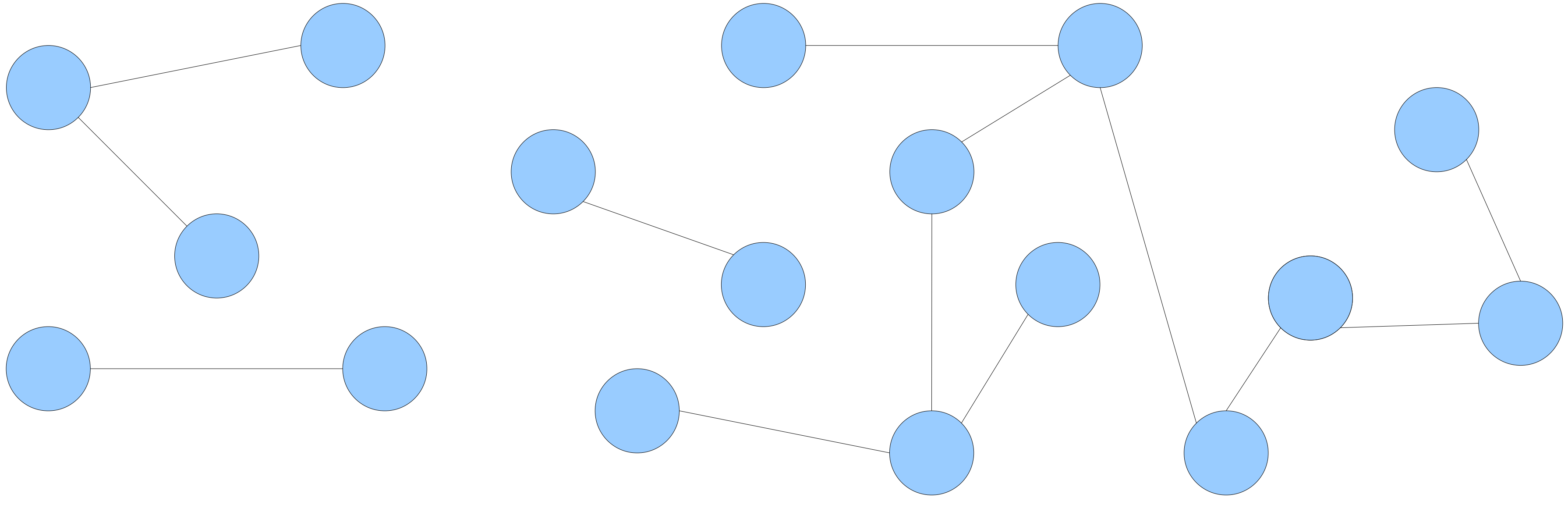
Dynamic Connectivity in Forests

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.

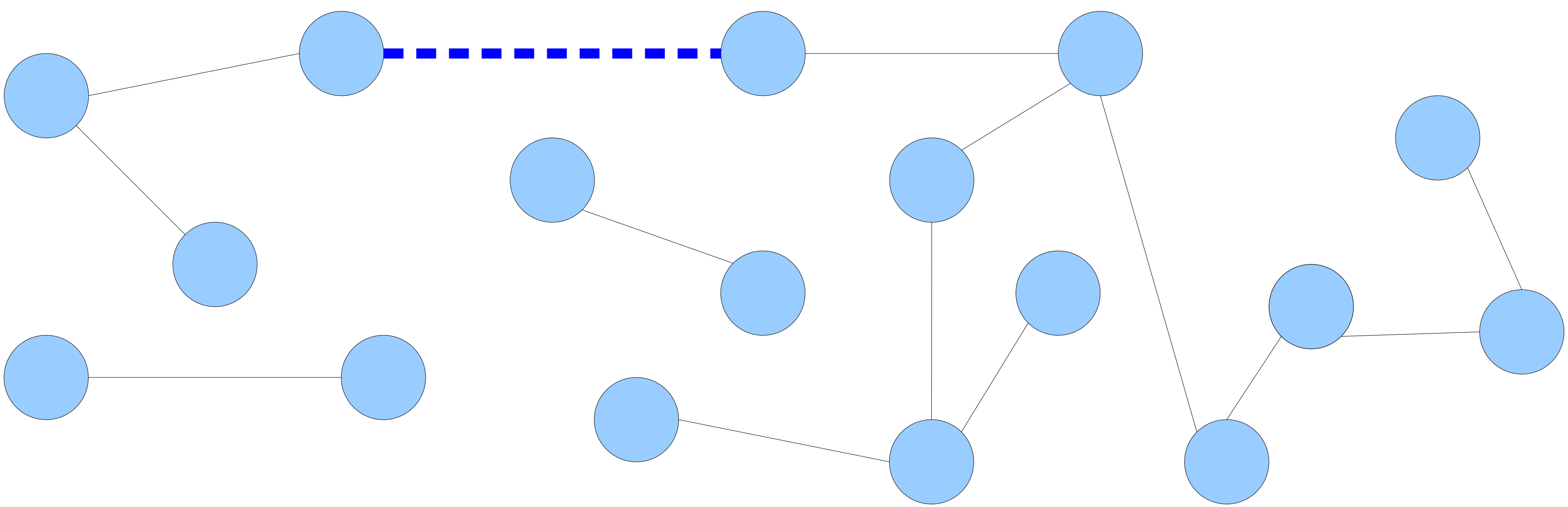


Dynamic Connectivity in Forests

- 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.

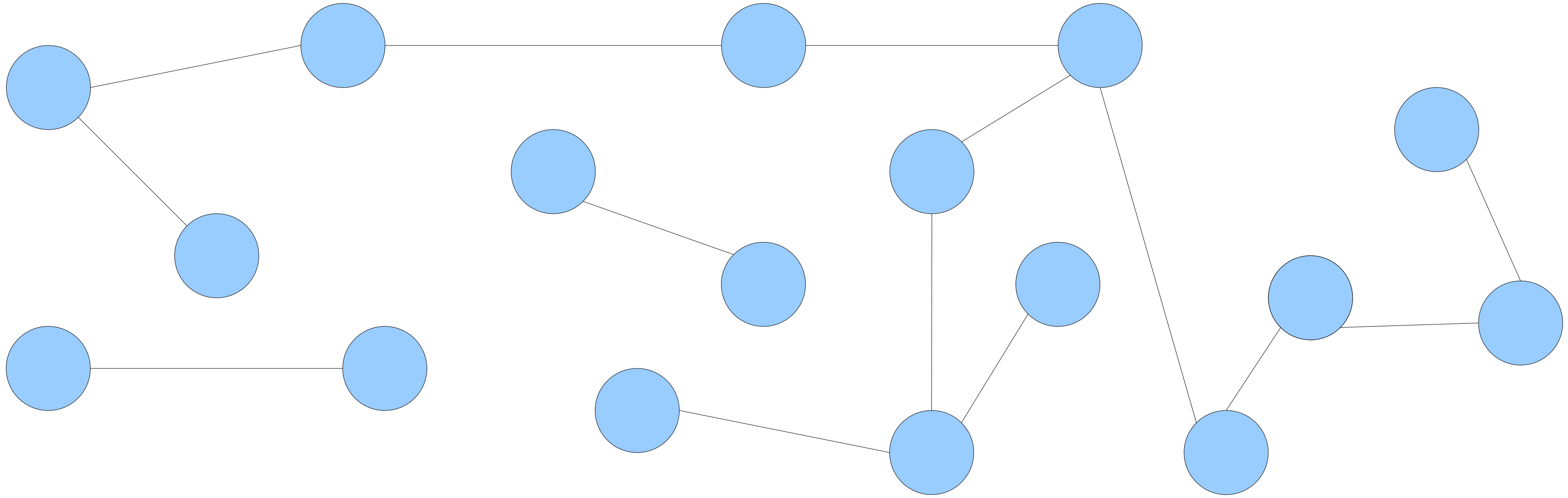


Dynamic Connectivity in Forests

- 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.

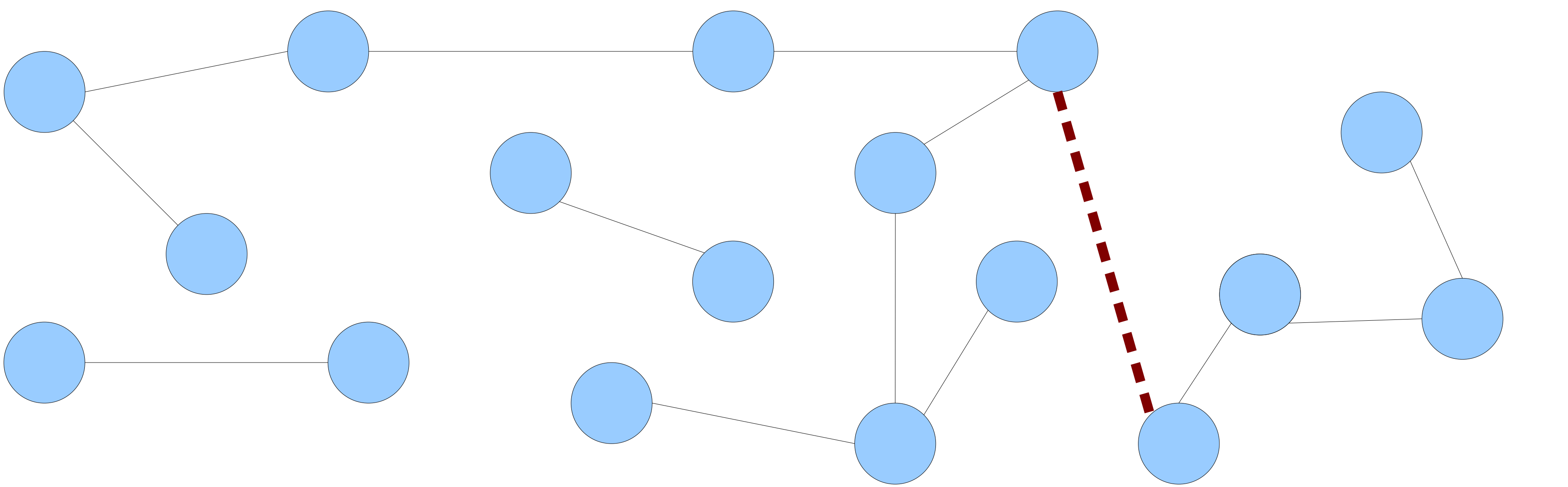


Dynamic Connectivity in Forests

- 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.

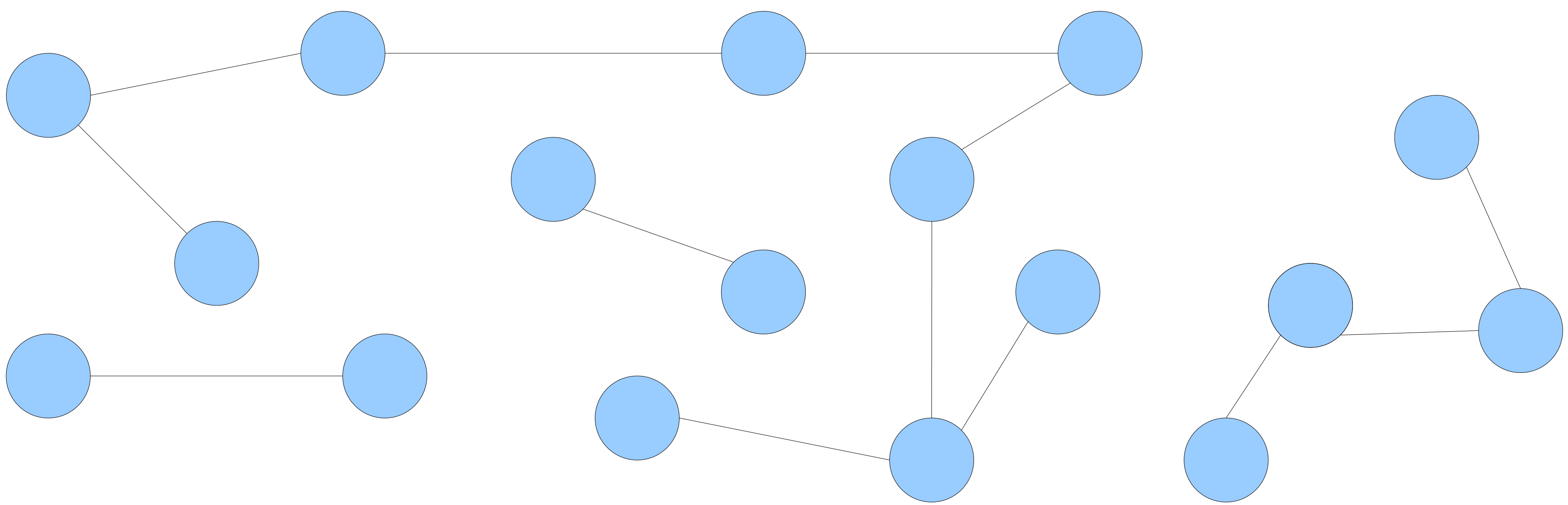


Dynamic Connectivity in Forests

- 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.



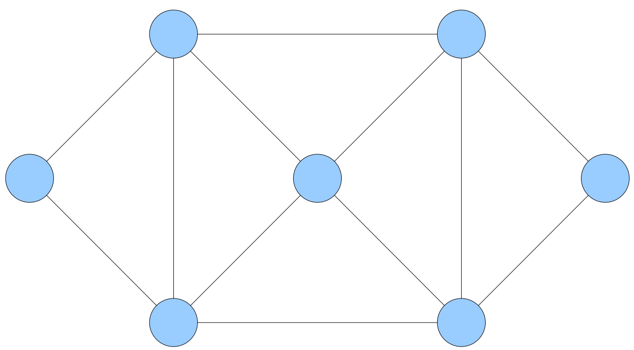
Dynamic Connectivity in Forests

- ***Goal***: Support these three operations:
  - ***link***(u, v): Add in edge uv. The assumption is that u and v are in separate trees.
  - ***cut***(u, v): Cut the edge uv. The assumption is that the edge exists in the forest.
  - ***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

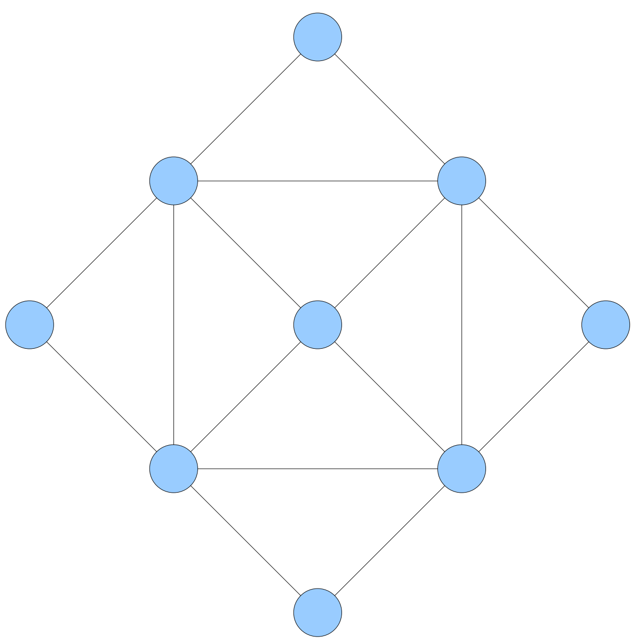
Euler Tours

- An ***Euler tour*** is a path through a graph G that visits every edge exactly once.
- It mathematically formalizes the “trace this figure without picking up your pencil or redrawing any lines” puzzles.



Euler Tours

- An ***Euler tour*** is a path through a graph G that visits every edge exactly once.
- It mathematically formalizes the "trace this figure without picking up your pencil or redrawing any lines" puzzles.

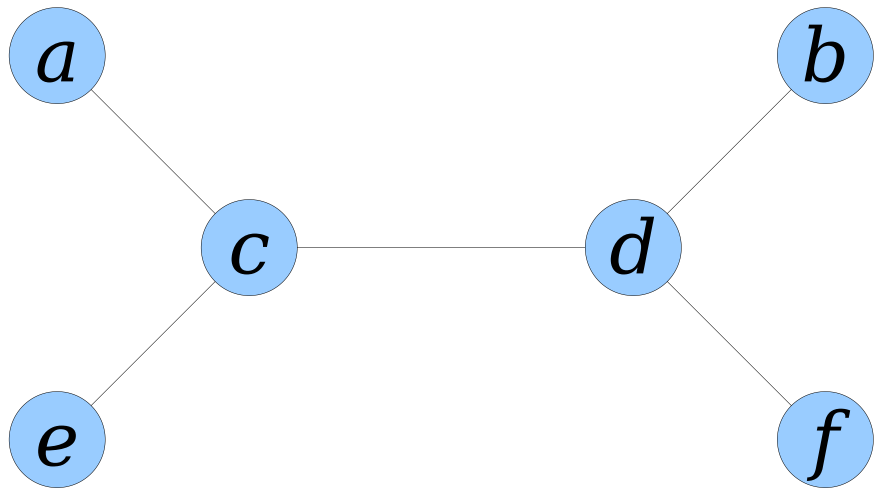


Euler Tours

- An ***Euler tour*** is a path through a graph G that visits every edge exactly once.
- It mathematically formalizes the "trace this figure without picking up your pencil or redrawing any lines" puzzles.
- ***Classic Theorem 1:*** A graph G has a closed Euler tour if and only if G is connected and every node in G has even degree.
- ***Classic Theorem 2:*** A directed graph G has a closed Euler tour if and only if G is strongly connected and every node's indegree equals its outdegree.

Euler Tours on Trees

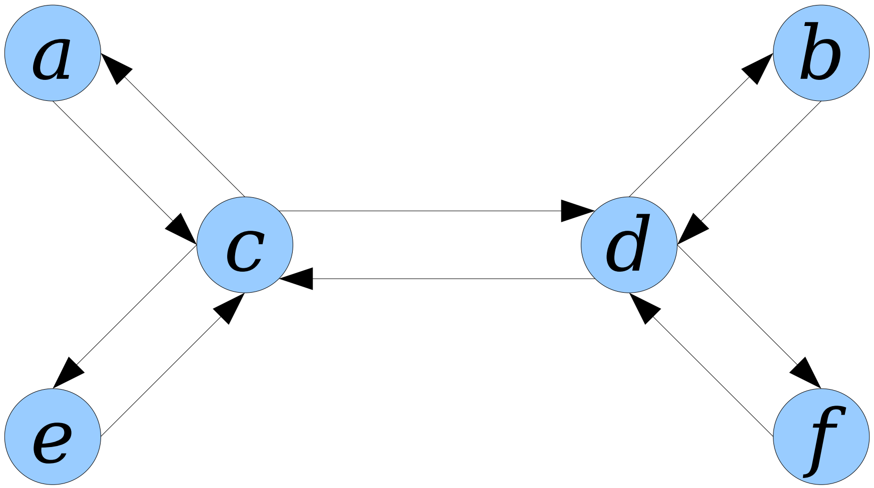
- Trees do not have Euler tours.



- ***Technique:*** replace each undirected edge uv with two directed edges uv and vu.

Euler Tours on Trees

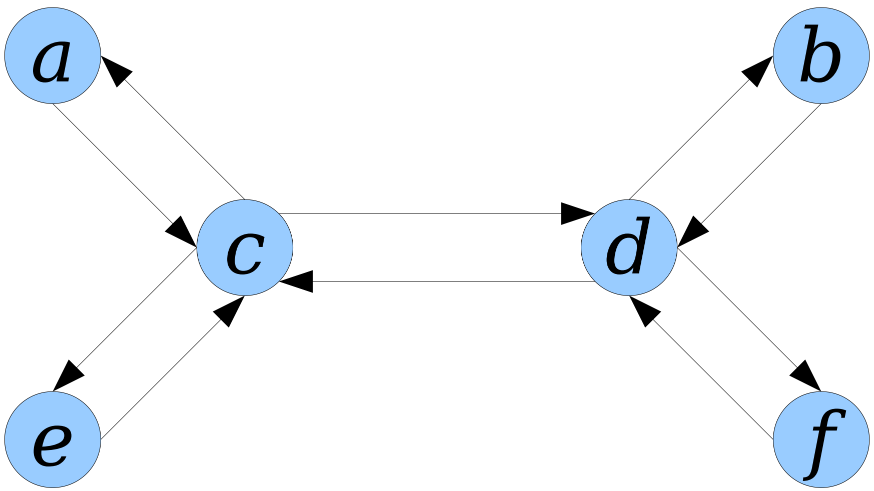
- Trees do not have Euler tours.



- ***Technique:*** replace each undirected edge uv with two directed edges uv and vu.
- The resulting graph then has an Euler tour.

Euler Tours on Trees

- Trees do not have Euler tours.

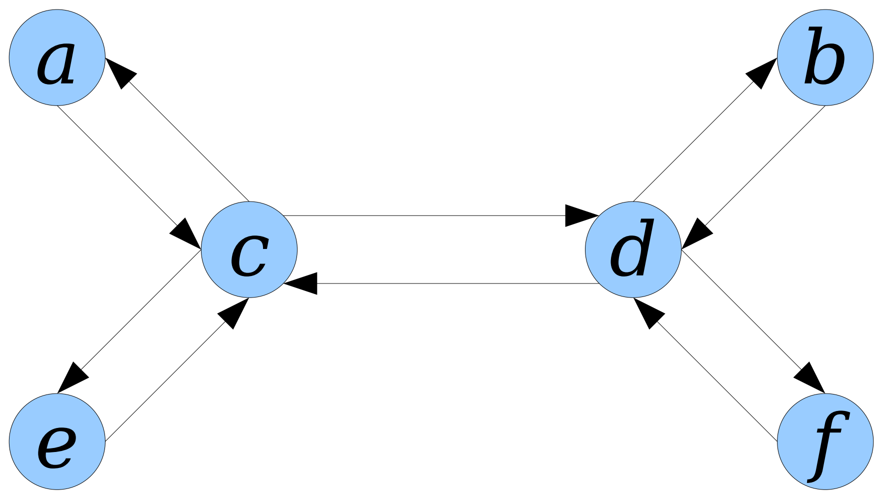


ac cd db bd df fd dc ce ec ca

- ***Technique:*** replace each undirected edge *uv* with two directed edges *uv* and *vu*.
- The resulting graph then has an Euler tour.

Euler Tours on Trees

- Trees do not have Euler tours.

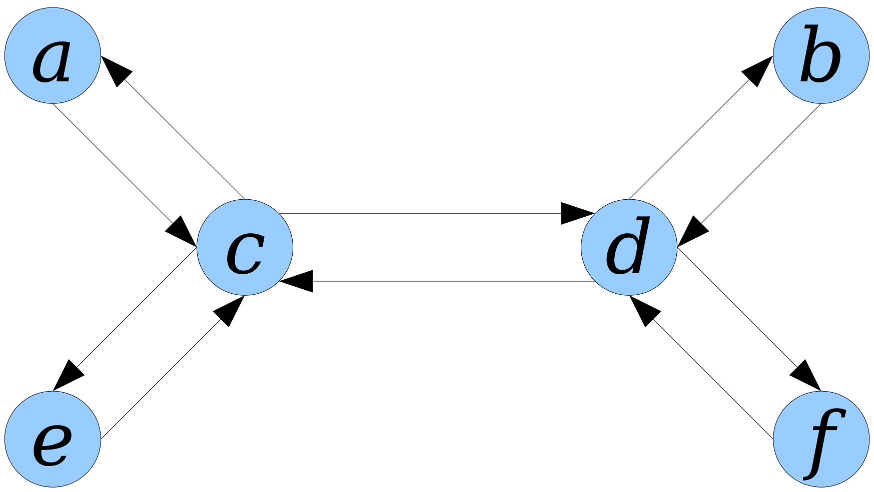


ce ec cd db bd df fd dc ca ac

- ***Technique:*** replace each undirected edge *uv* with two directed edges *uv* and *vu*.
- The resulting graph then has an Euler tour.

Euler Tours on Trees

- Trees do not have Euler tours.

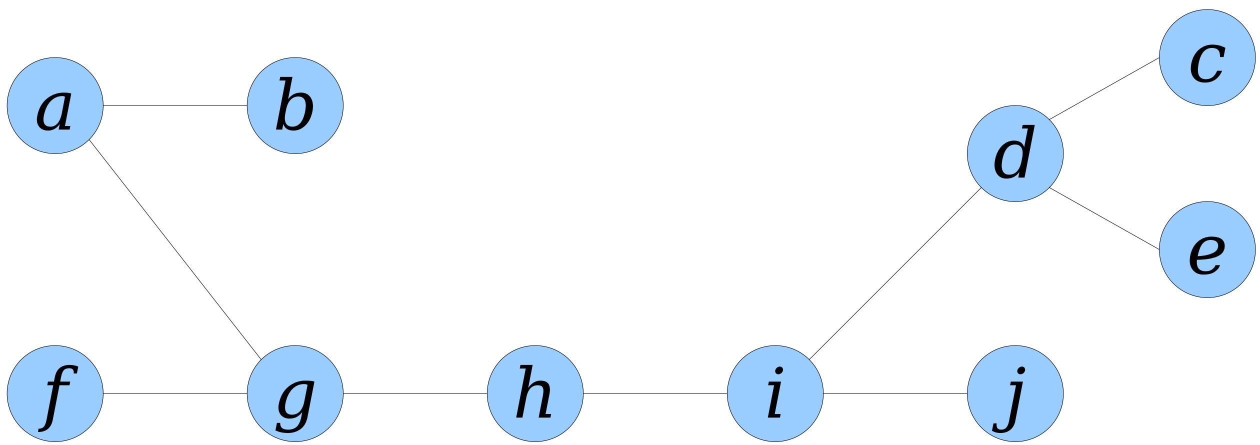


fd db bd dc ca ac ce ec cd df

- ***Technique:*** replace each undirected edge *uv* with two directed edges *uv* and *vu*.
- The resulting graph then has an Euler tour.

Properties of Euler Tours

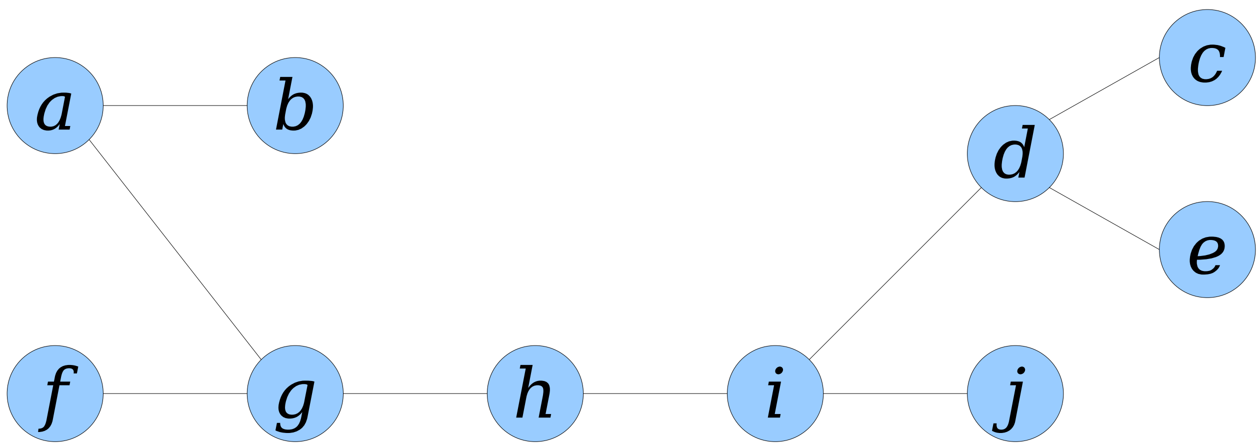
- ***Fact:*** Any cyclic shift of an Euler tour of a tree is also an Euler tour.



ab ba ag gh hi id dc cd de ed di ij ji ih hg gf fg ga

Properties of Euler Tours

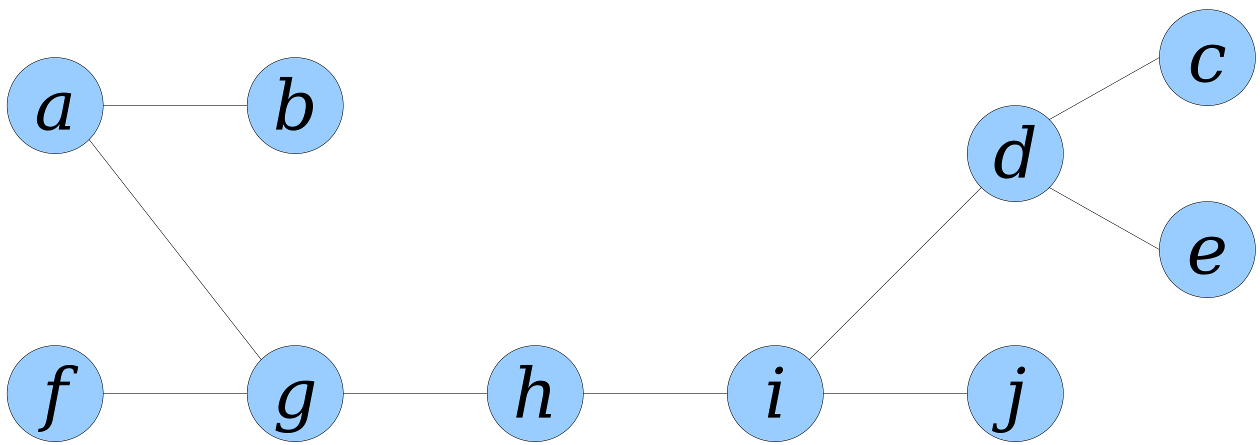
- ***Fact:*** Any cyclic shift of an Euler tour of a tree is also an Euler tour.



ab ba ag gh hi id dc cd de ed di ij ji ih hg gf fg ga

Properties of Euler Tours

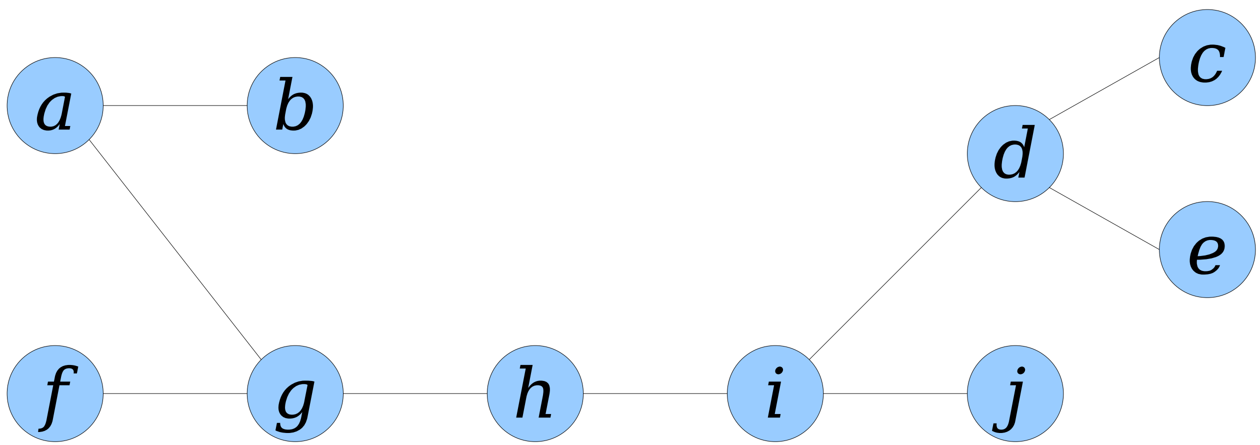
- ***Fact:*** Any cyclic shift of an Euler tour of a tree is also an Euler tour.



cd de ed di ij ji ih hg gf fg ga ab ba ag gh hi id dc

Properties of Euler Tours

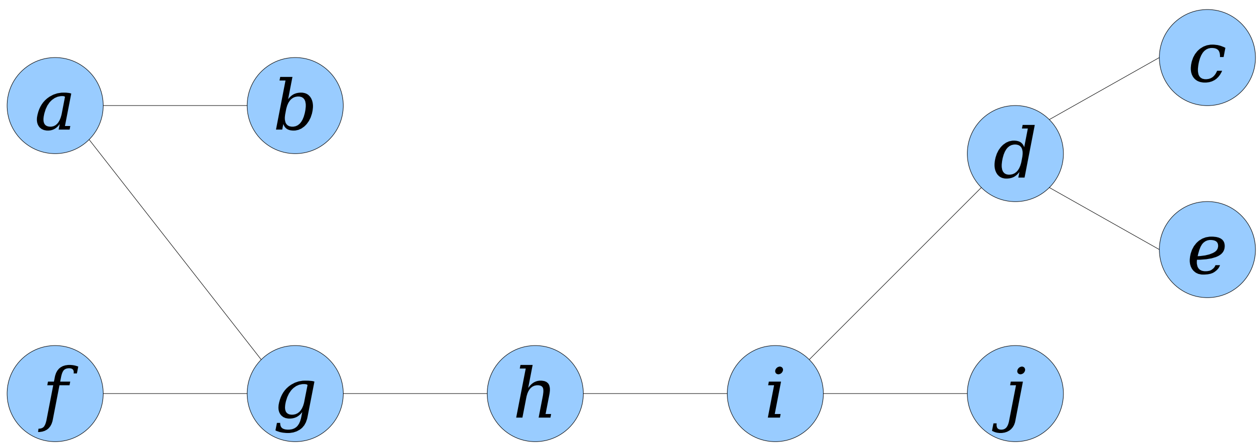
- ***Fact:*** Any cyclic shift of an Euler tour of a tree is also an Euler tour.



cd de ed di ij ji ih hg gf fg ga ab ba ag gh hi id dc

Properties of Euler Tours

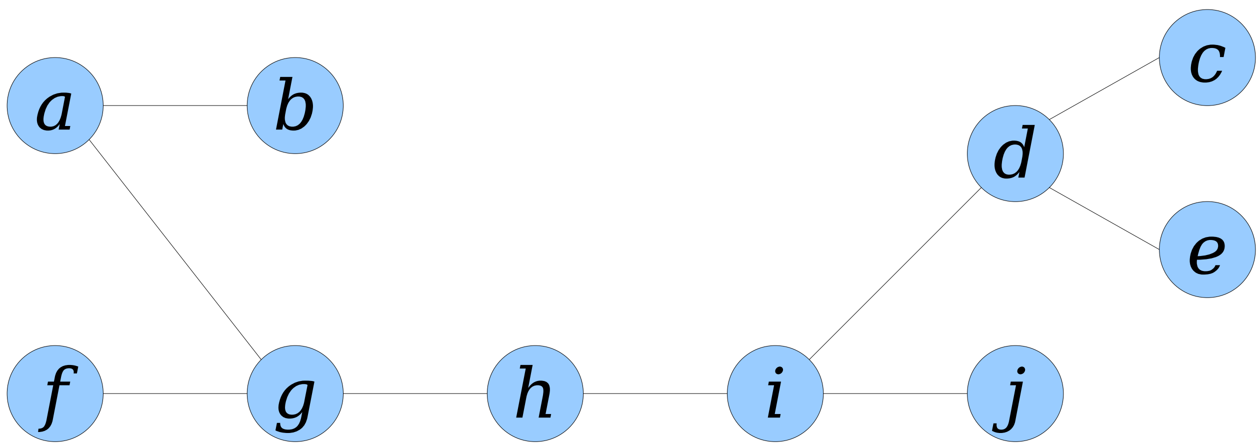
- ***Fact:*** Any cyclic shift of an Euler tour of a tree is also an Euler tour.



cd de ed di ij ji ih hg gf fg ga ab ba ag gh hi id dc

Properties of Euler Tours

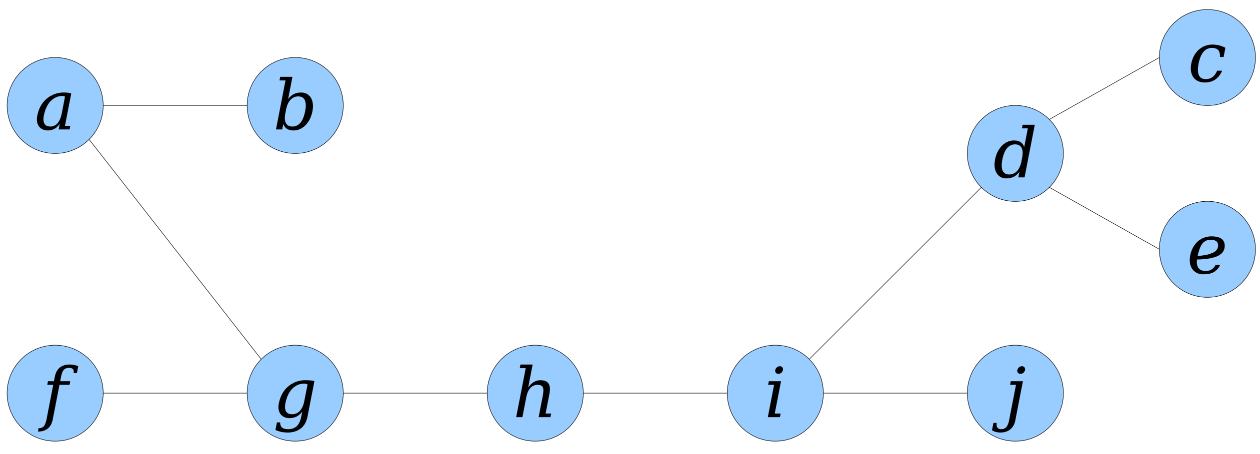
- ***Fact:*** Any cyclic shift of an Euler tour of a tree is also an Euler tour.



hg gf fg ga ab ba ag gh hi id dc cd de ed di ij ji ih

Properties of Euler Tours

- ***Fact:*** Any cyclic shift of an Euler tour of a tree is also an Euler tour.



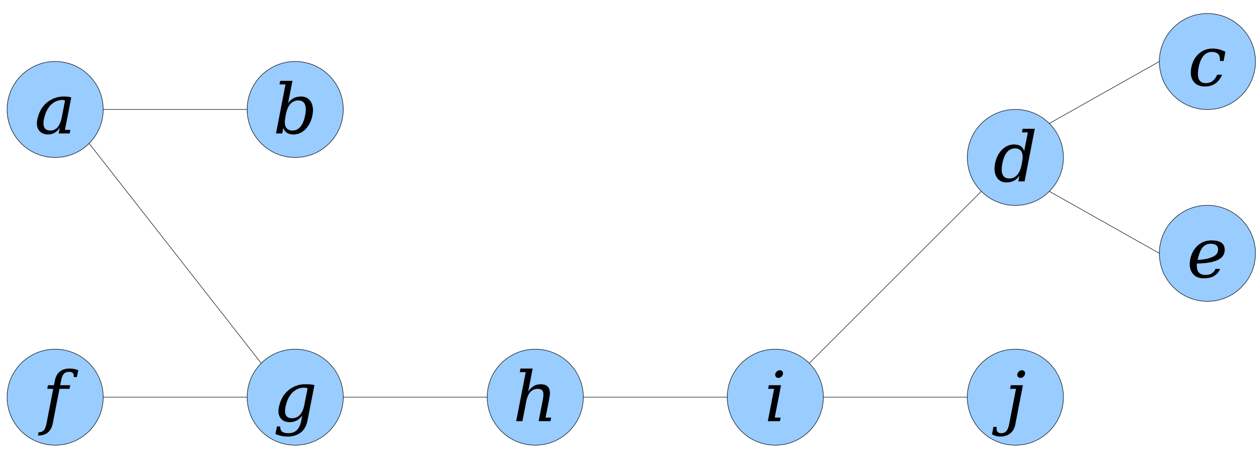
hg gf fg ga ab ba ag gh hi id dc cd de ed di ij ji ih

Rerooting a Tour

- In some cases, we will need to cyclicly shift a tour to put an edge leaving a particular node x at front.
- We will call this operation ***reroot***(x).

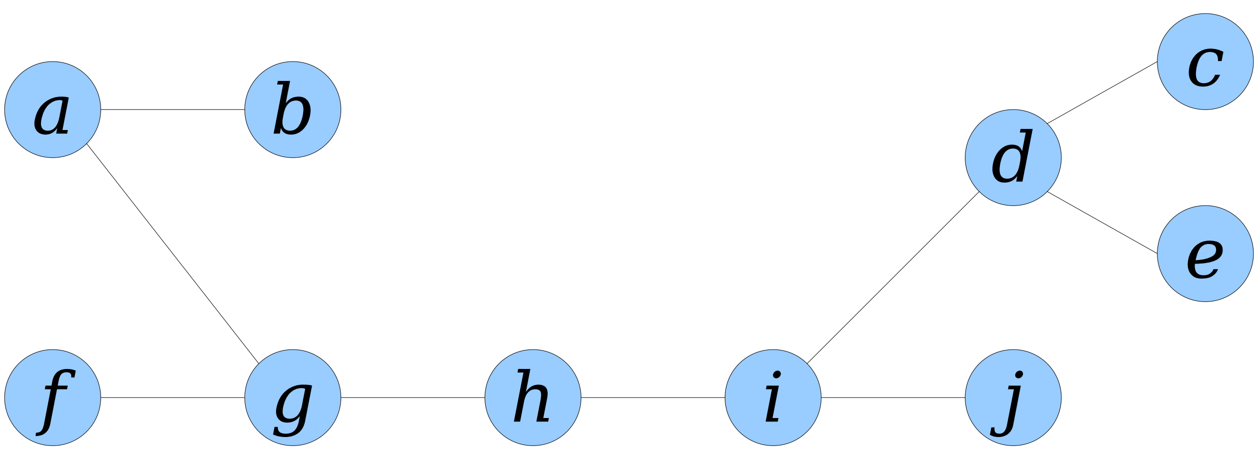
Rerooting a Tour

- In some cases, we will need to cyclicly shift a tour to put an edge leaving a particular node x at front.
- We will call this operation ***reroot***(x).



Rerooting a Tour

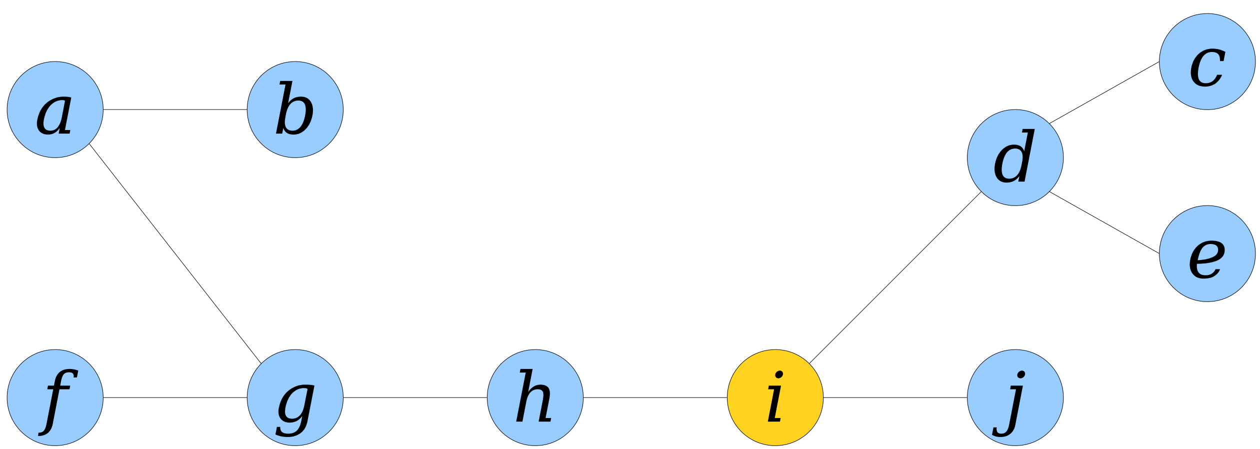
- In some cases, we will need to cyclicly shift a tour to put an edge leaving a particular node x at front.
- We will call this operation ***reroot***(x).



ab ba ag gh hi id dc cd de ed di ij ji ih hg gf fg ga

Rerooting a Tour

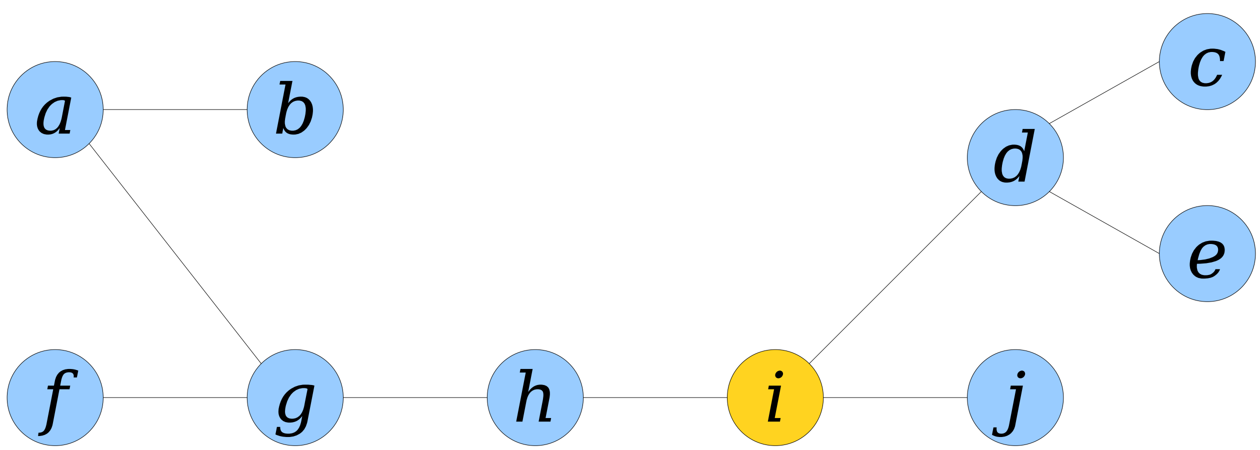
- In some cases, we will need to cyclicly shift a tour to put an edge leaving a particular node *x* at front.
- We will call this operation ***reroot***(*x*).



ab ba ag gh hi id dc cd de ed di ij ji ih hg gf fg ga

Rerooting a Tour

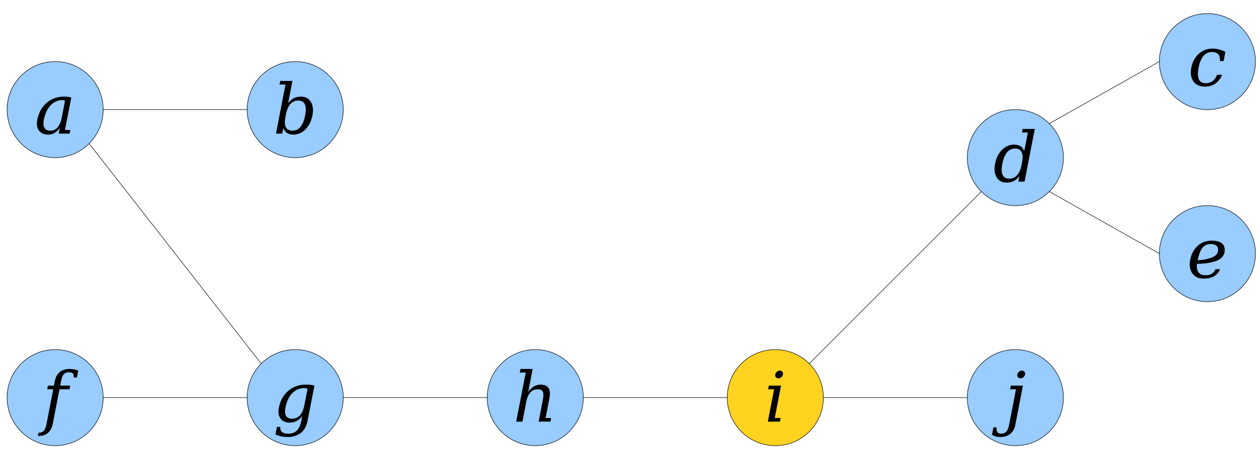
- In some cases, we will need to cyclicly shift a tour to put an edge leaving a particular node x at front.
- We will call this operation ***reroot***(x).



ab ba ag gh hi id dc cd de ed di ij ji ih hg gf fg ga

Rerooting a Tour

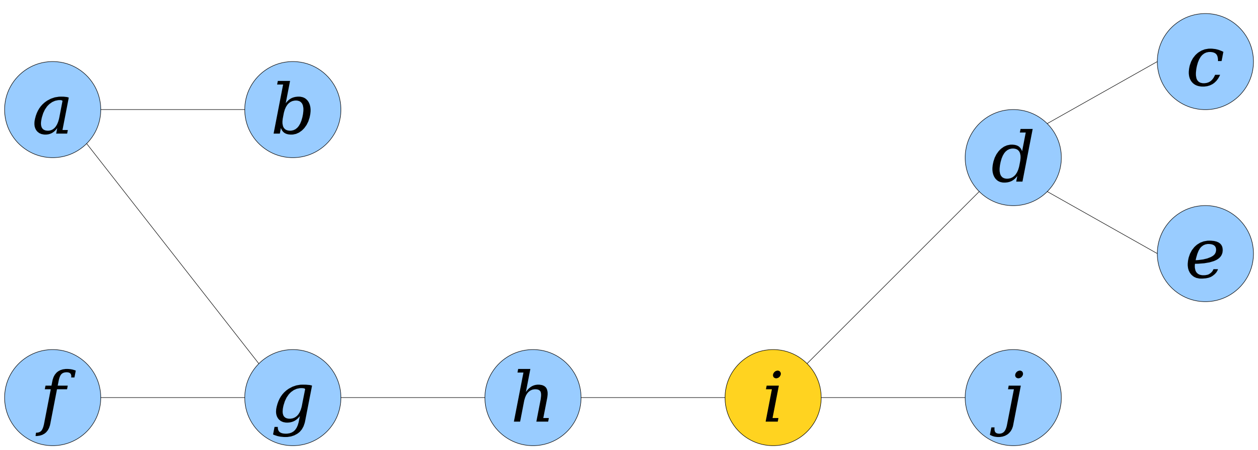
- In some cases, we will need to cyclicly shift a tour to put an edge leaving a particular node *x* at front.
- We will call this operation ***reroot***(*x*).



ab ba ag gh hi id dc cd de ed di ij ji ih hg gf fg ga

Rerooting a Tour

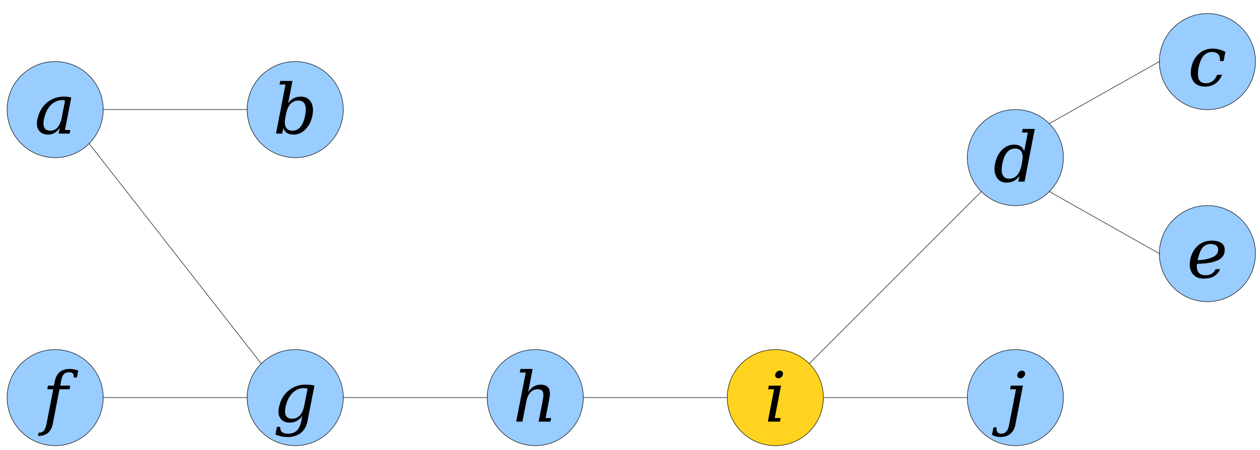
- In some cases, we will need to cyclicly shift a tour to put an edge leaving a particular node x at front.
- We will call this operation ***reroot***(x).



ij ji ih hg gf fg ga ab ba ag gh hi id dc cd de ed di

Rerooting a Tour

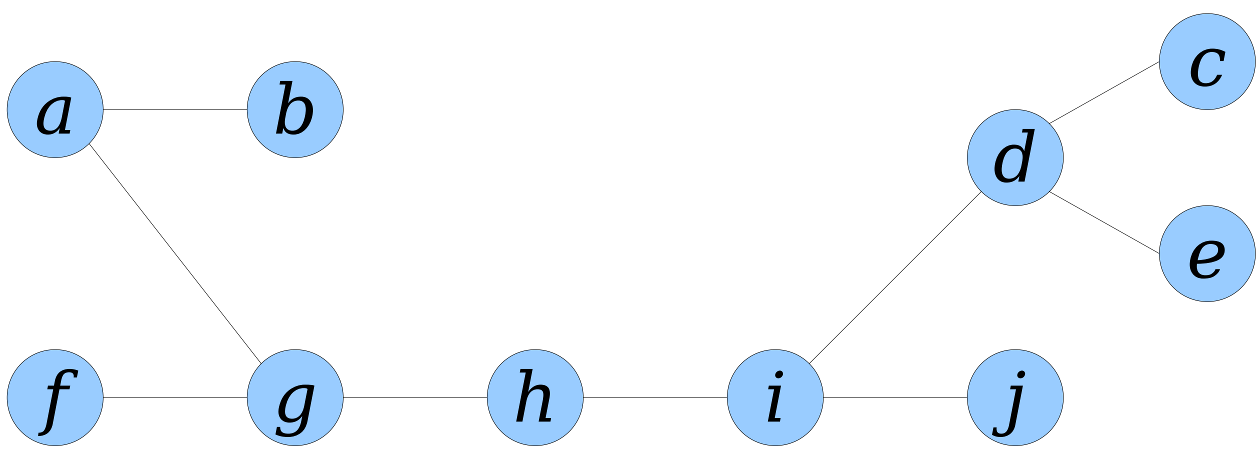
- In some cases, we will need to cyclicly shift a tour to put an edge leaving a particular node *x* at front.
- We will call this operation ***reroot***(*x*).



ij ji ih hg gf fg ga ab ba ag gh hi id dc cd de ed di

Rerooting a Tour

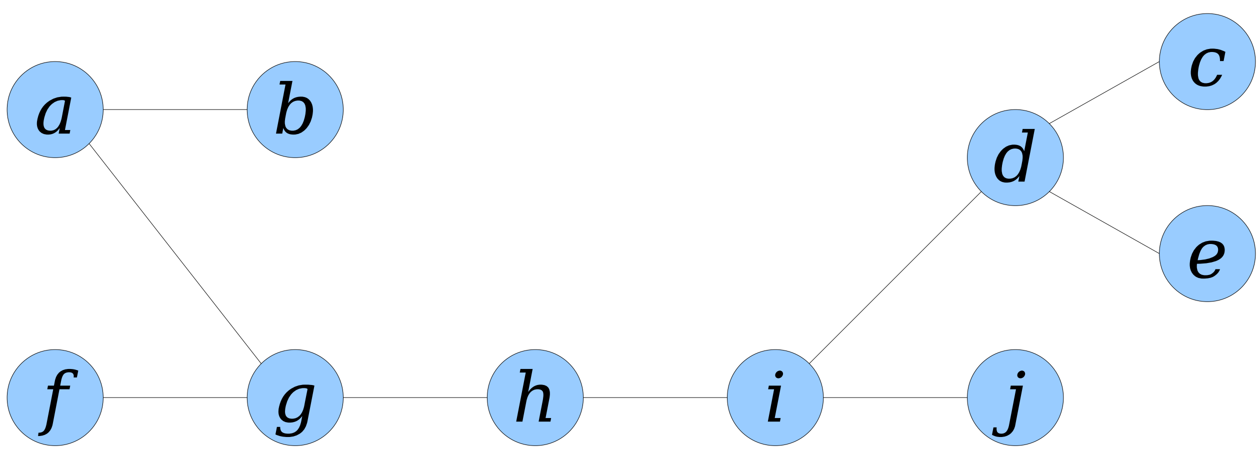
- In some cases, we will need to cyclicly shift a tour to put an edge leaving a particular node x at front.
- We will call this operation ***reroot***(x).



ij ji ih hg gf fg ga ab ba ag gh hi id dc cd de ed di

Rerooting a Tour

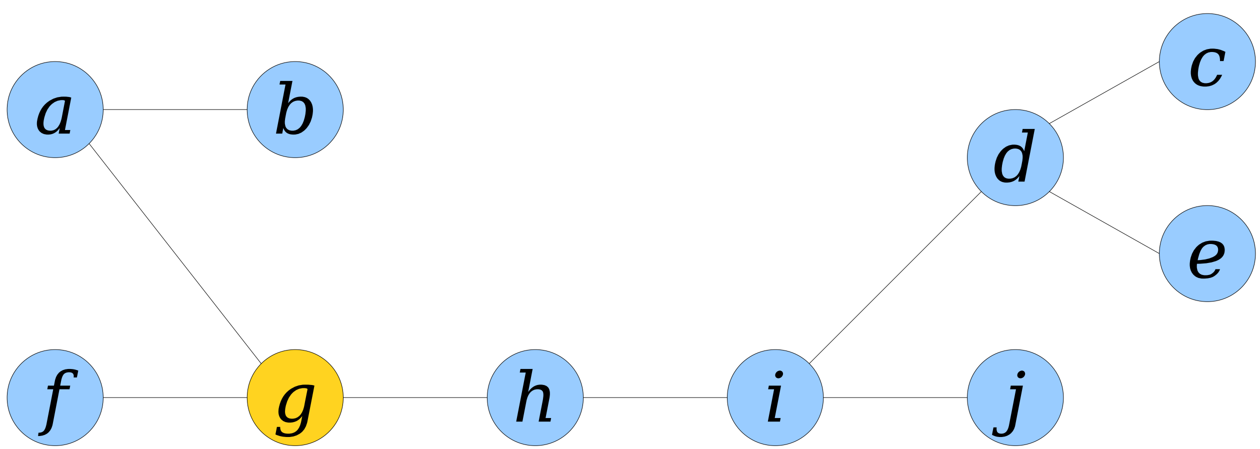
- To perform ***reroot***(x):
  - Pick any edge rx leaving our new start node r.
  - Split the tour into A and B, where A consists of everything up to but not including rx and B consists of everything from rx forward.
  - Concatenate $B\ A$.



ij ji ih hg gf fg ga ab ba ag gh hi id dc cd de ed di

Rerooting a Tour

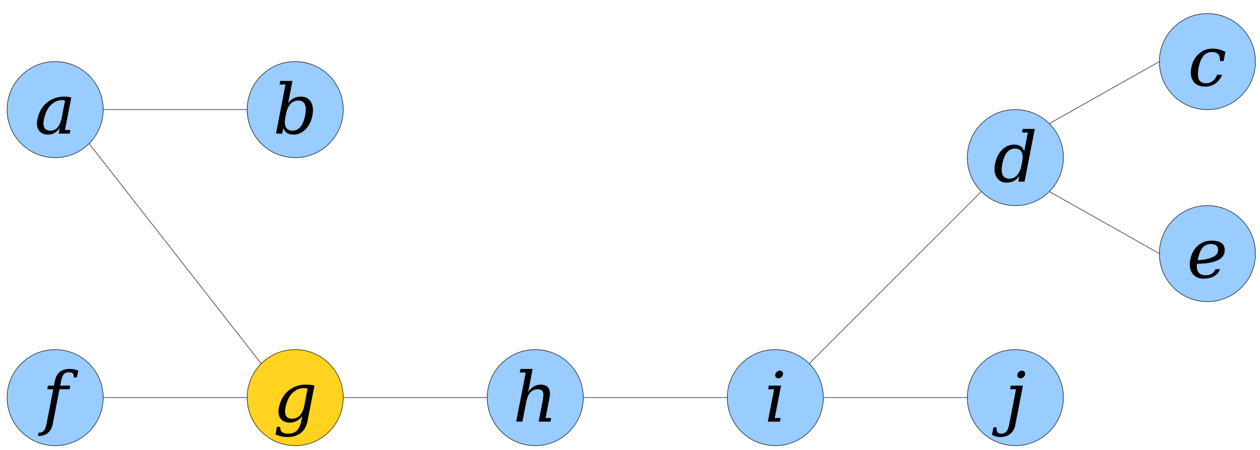
- To perform ***reroot***(x):
  - Pick any edge rx leaving our new start node r.
  - Split the tour into A and B, where A consists of everything up to but not including rx and B consists of everything from rx forward.
  - Concatenate B A.



ij ji ih hg gf fg ga ab ba ag gh hi id dc cd de ed di

Rerooting a Tour

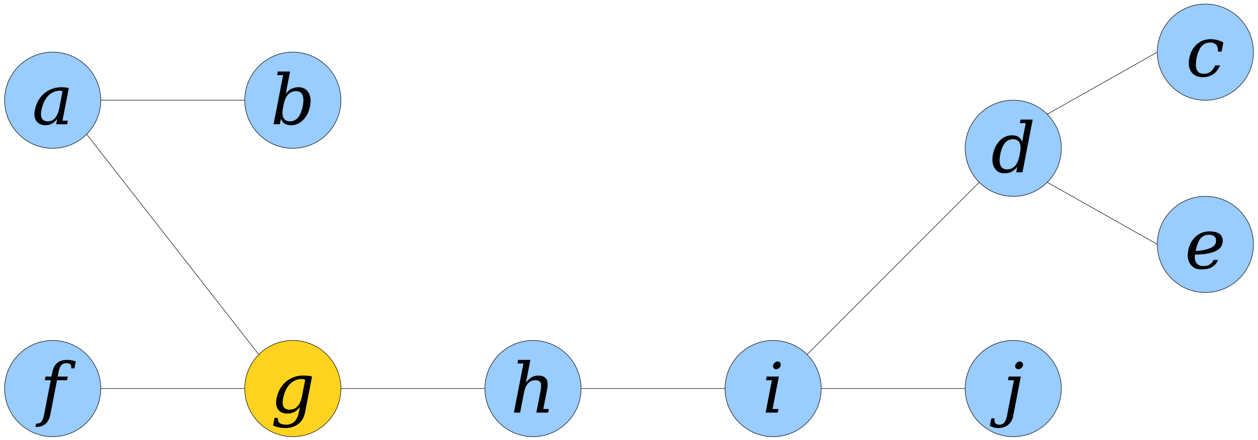
- To perform ***reroot***(x):
  - Pick any edge rx leaving our new start node r.
  - Split the tour into A and B, where A consists of everything up to but not including rx and B consists of everything from rx forward.
  - Concatenate B A.



ij ji ih hg gf fg ga ab ba ag gh hi id dc cd de ed di

Rerooting a Tour

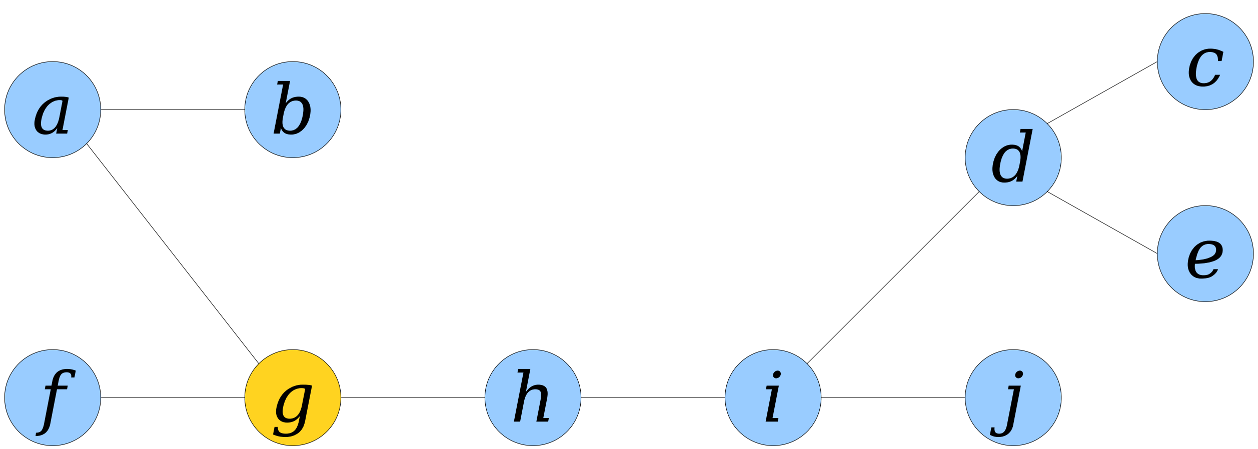
- To perform ***reroot***(x):
  - Pick any edge rx leaving our new start node r.
  - Split the tour into A and B, where A consists of everything up to but not including rx and B consists of everything from rx forward.
  - Concatenate $B\ A$.



ij ji ih hg gf fg ga ab ba ag gh hi id dc cd de ed di

Rerooting a Tour

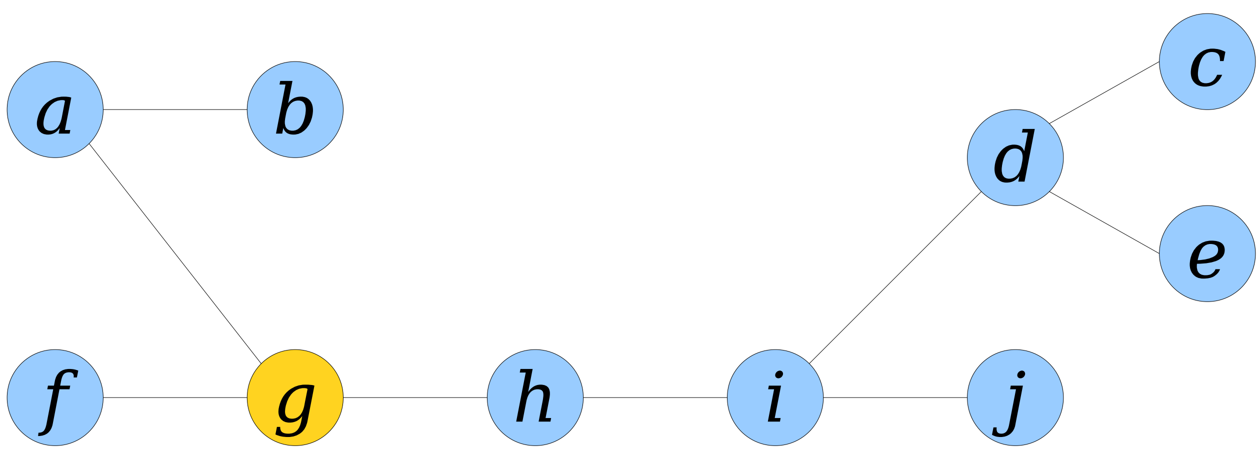
- To perform ***reroot***(x):
  - Pick any edge rx leaving our new start node r.
  - Split the tour into A and B, where A consists of everything up to but not including rx and B consists of everything from rx forward.
  - Concatenate B A.



gf fg ga ab ba ag gh hi id dc cd de ed di ij ji ih hg

Rerooting a Tour

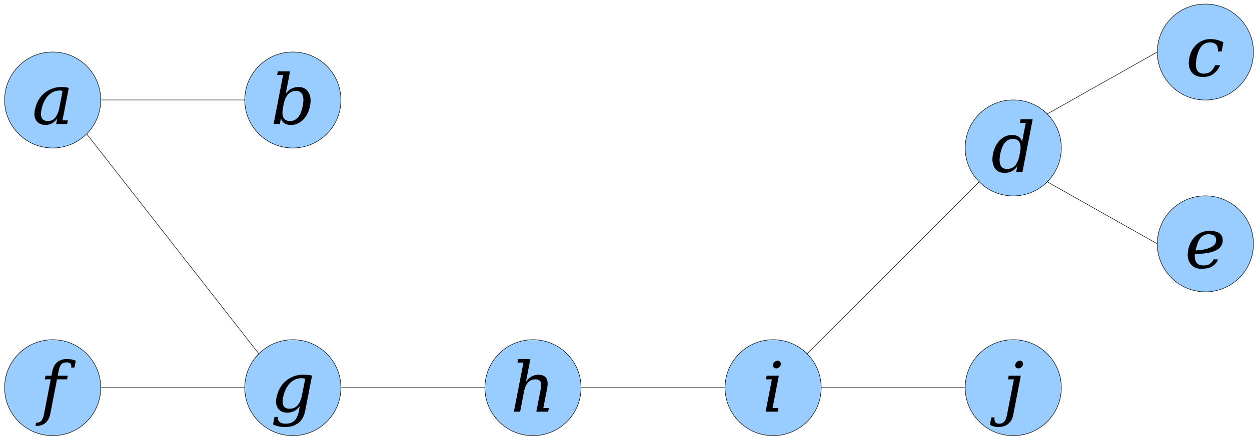
- To perform ***reroot***(x):
  - Pick any edge rx leaving our new start node r.
  - Split the tour into A and B, where A consists of everything up to but not including rx and B consists of everything from rx forward.
  - Concatenate B A.



gf fg ga ab ba ag gh hi id dc cd de ed di ij ji ih hg

Rerooting a Tour

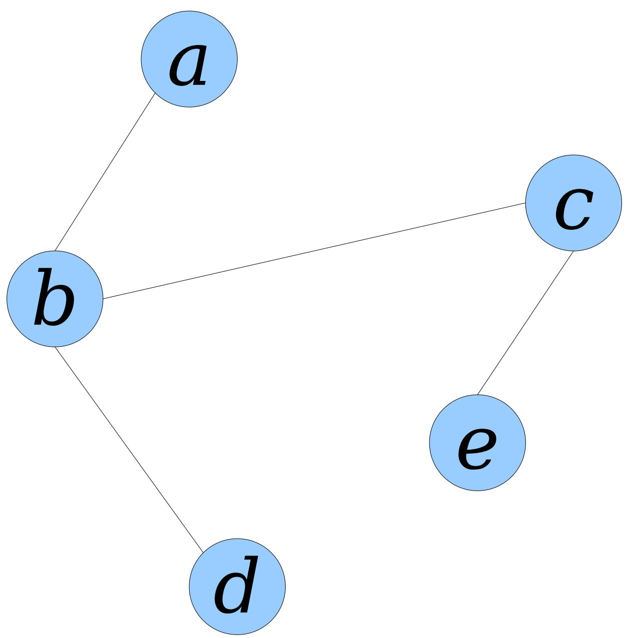
- To perform ***reroot***(x):
  - Pick any edge rx leaving our new start node r.
  - Split the tour into A and B, where A consists of everything up to but not including rx and B consists of everything from rx forward.
  - Concatenate B A.

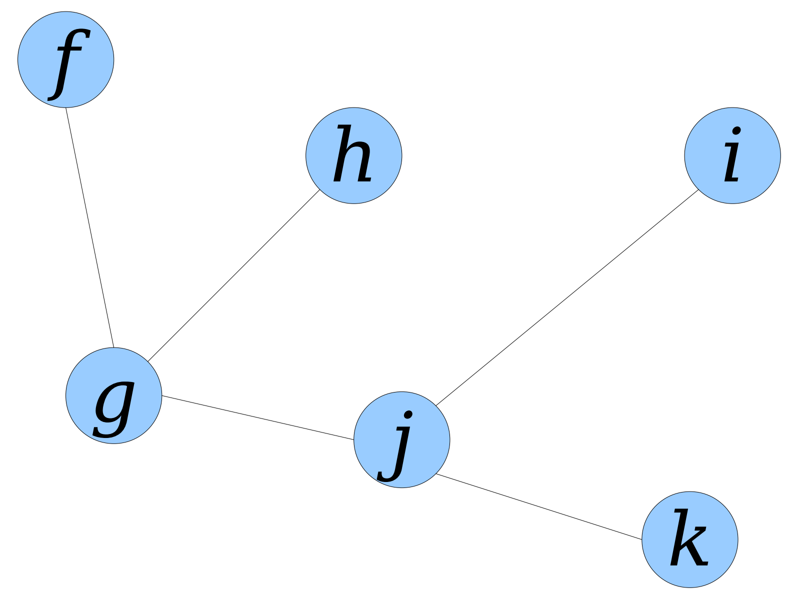


gf fg ga ab ba ag gh hi id dc cd de ed di ij ji ih hg

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 uv.
- Watch what happens to the Euler tours:



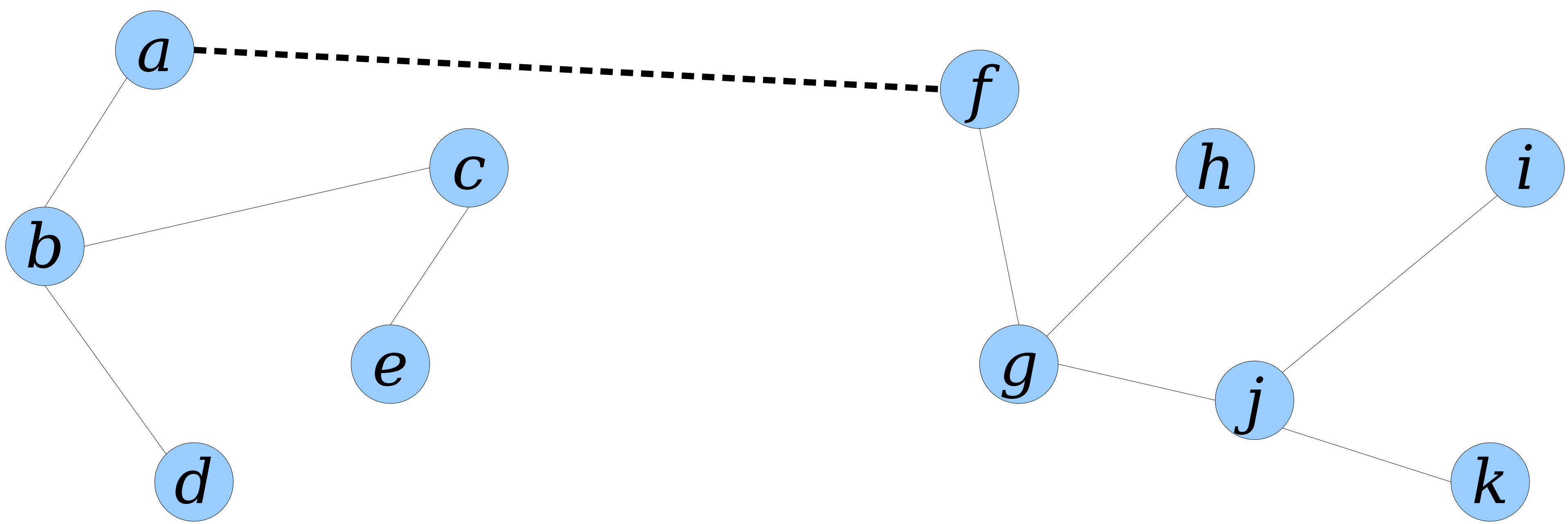


ab bd db bc ce ec cb ba

fg gj jk kj ji ij jg gh hg gf

Euler Tours and Dynamic Trees

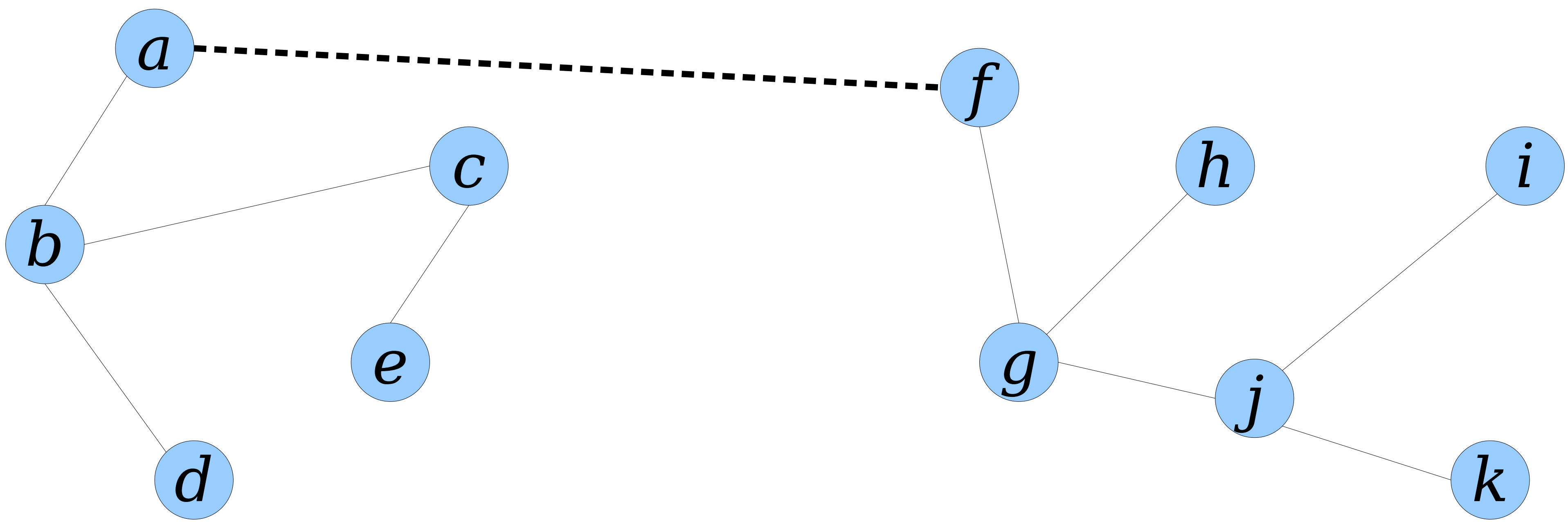
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ab bd db bc ce ec cb ba ***fg gj jk kj ji ij jg gh hg gf***

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 uv.
- Watch what happens to the Euler tours:

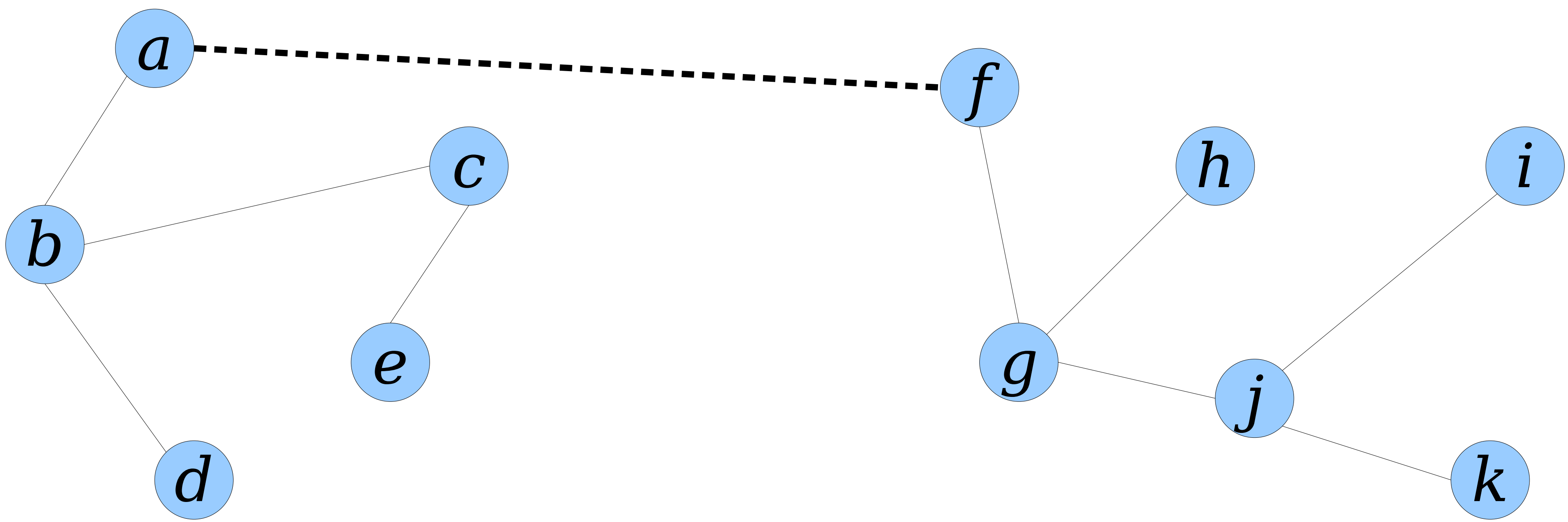


ab bd db bc ce ec cb ba

fg gj jk kj ji ij jg gh hg gf

Euler Tours and Dynamic Trees

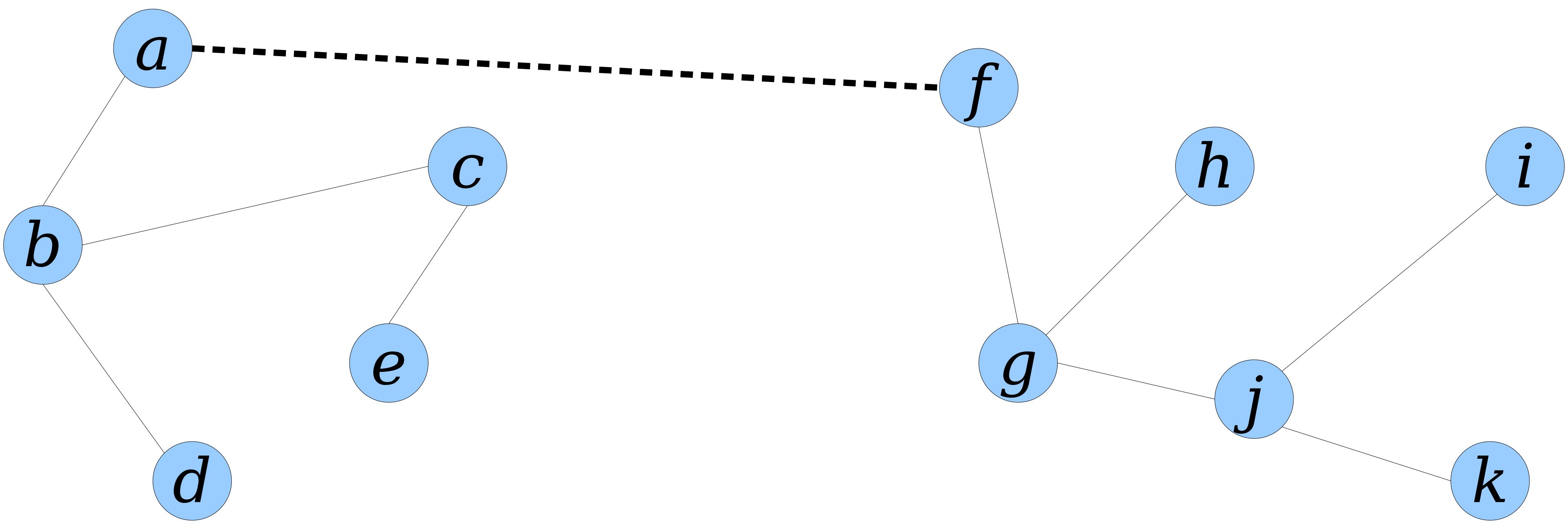
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ab bd db bc ce ec cb ba ***fg gj jk kj ji ij jg gh hg gf***

Euler Tours and Dynamic Trees

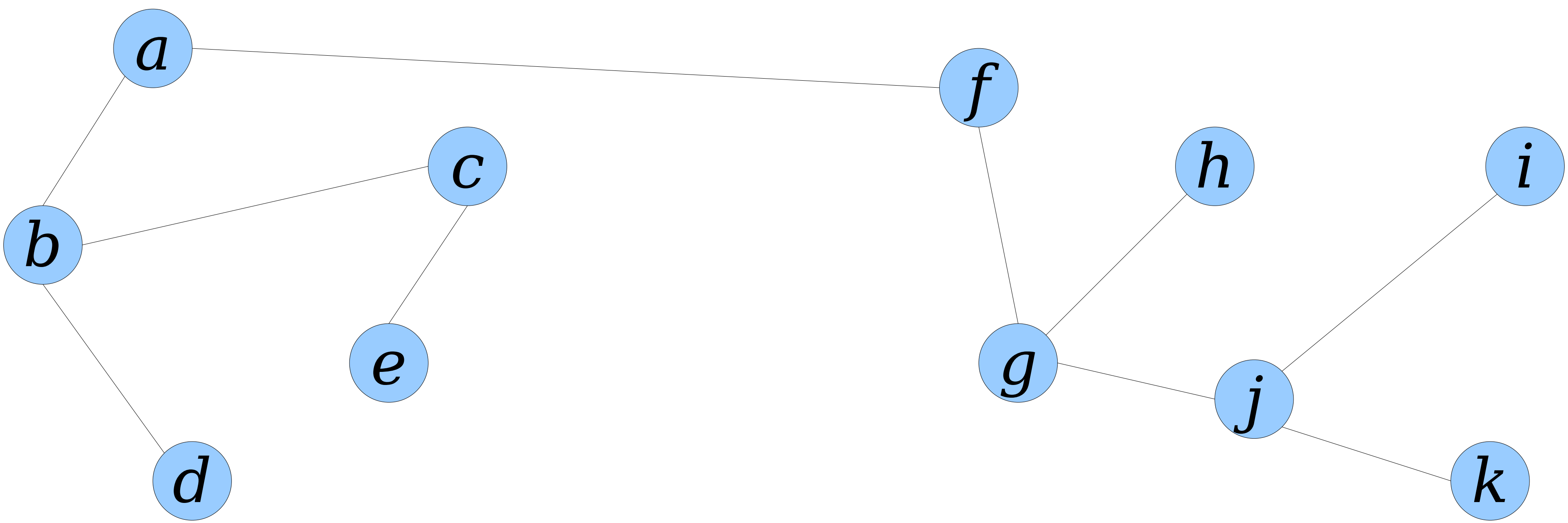
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ab bd db bc ce ec cb ba af fg gj jk kj ji ij jg gh hg gf fa

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 uv.
- Watch what happens to the Euler tours:



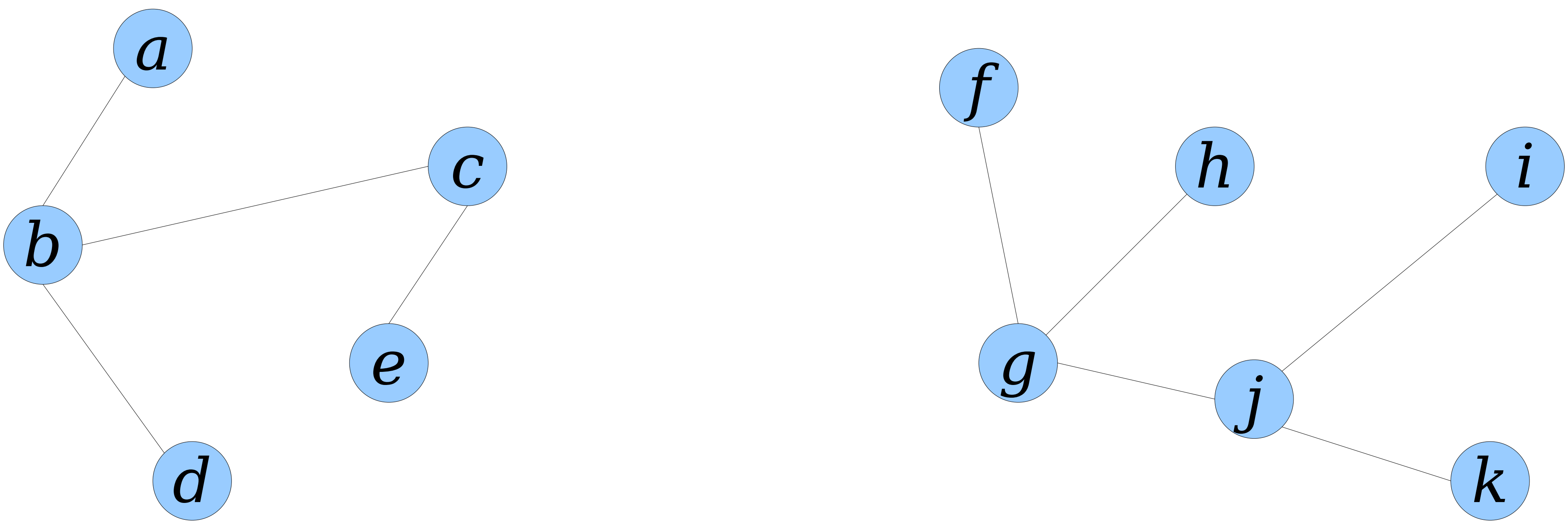
ab bd db bc ce ec cb ba af fg gj jk kj ji ij jg gh hg gf fa

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 uv.
- Watch what happens to the Euler tours:

Euler Tours and Dynamic Trees

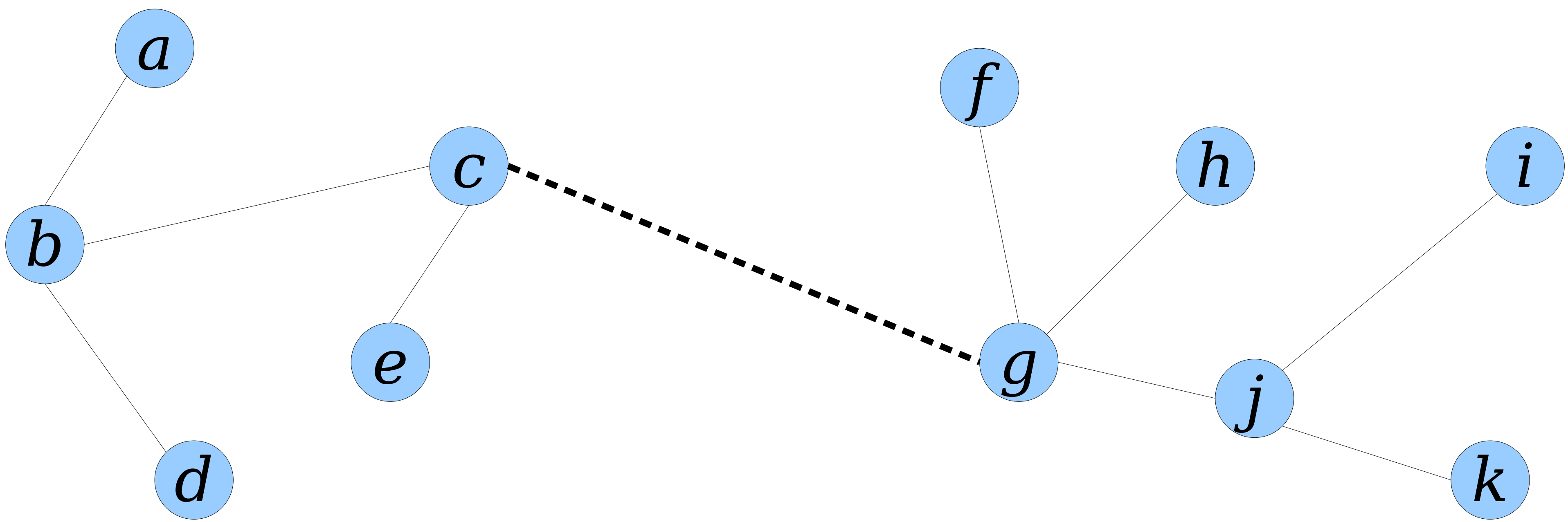
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ab bd db bc ce ec cb ba ***fg gj jk kj ji ij jg gh hg gf***

Euler Tours and Dynamic Trees

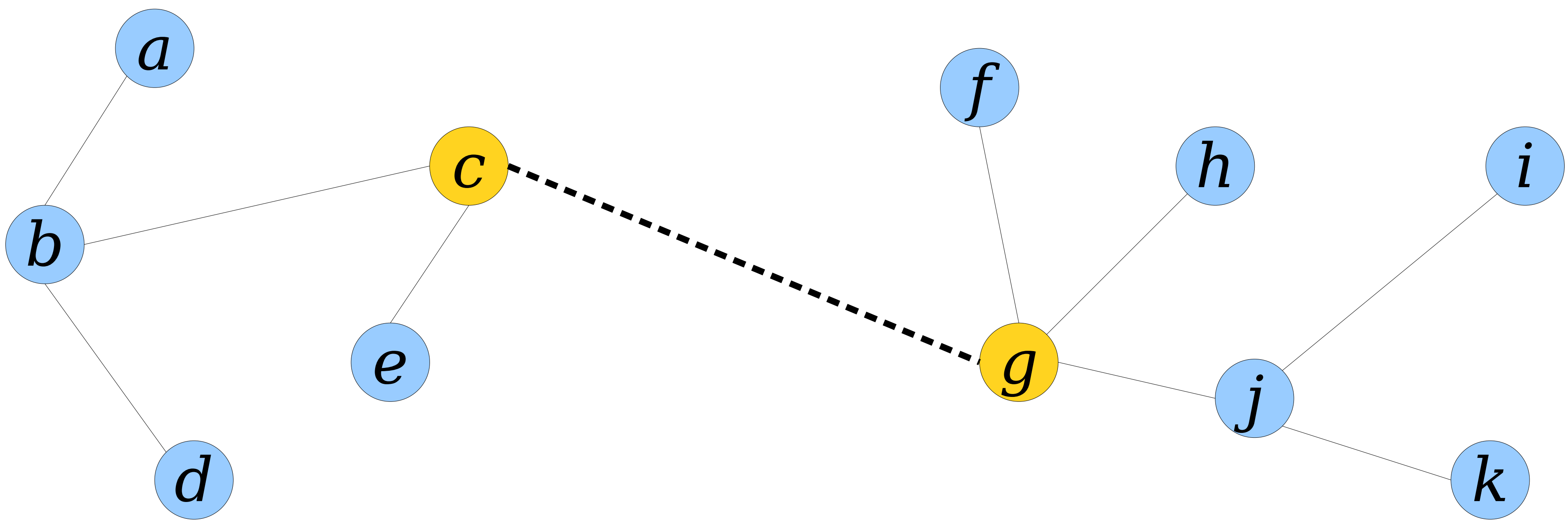
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ab bd db bc ce ec cb ba ***fg gj jk kj ji ij jg gh hg gf***

Euler Tours and Dynamic Trees

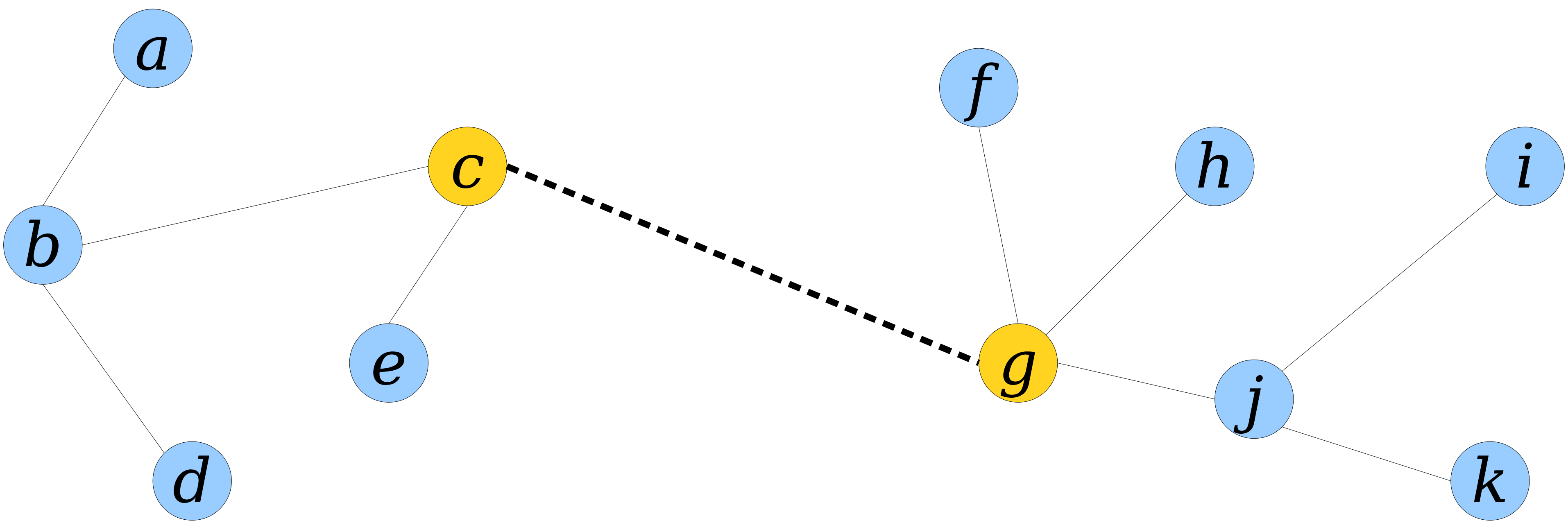
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ab bd db bc ce ec cb ba ***fg gj jk kj ji ij jg gh hg gf***

Euler Tours and Dynamic Trees

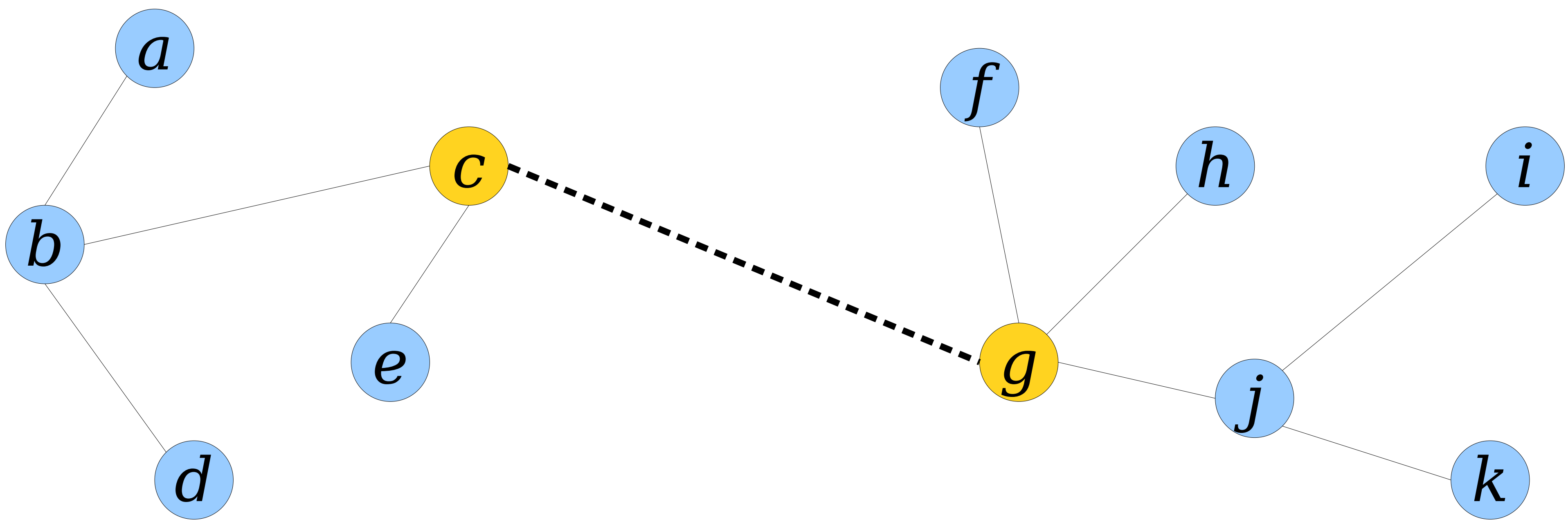
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ab bd db bc ce ec cb ba   ***fg gj jk kj ji ij jg gh hg gf***

Euler Tours and Dynamic Trees

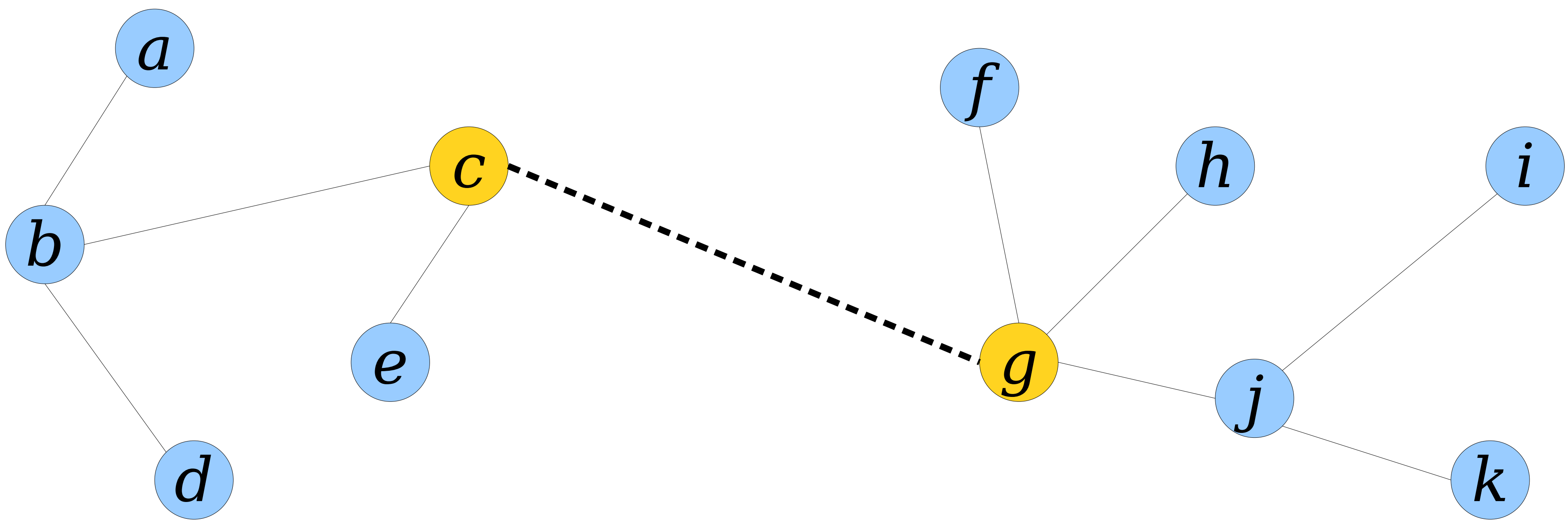
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ce ec cb ba ab bd db bc ***gh hg gf fg gj jk kj ji ij jg***

Euler Tours and Dynamic Trees

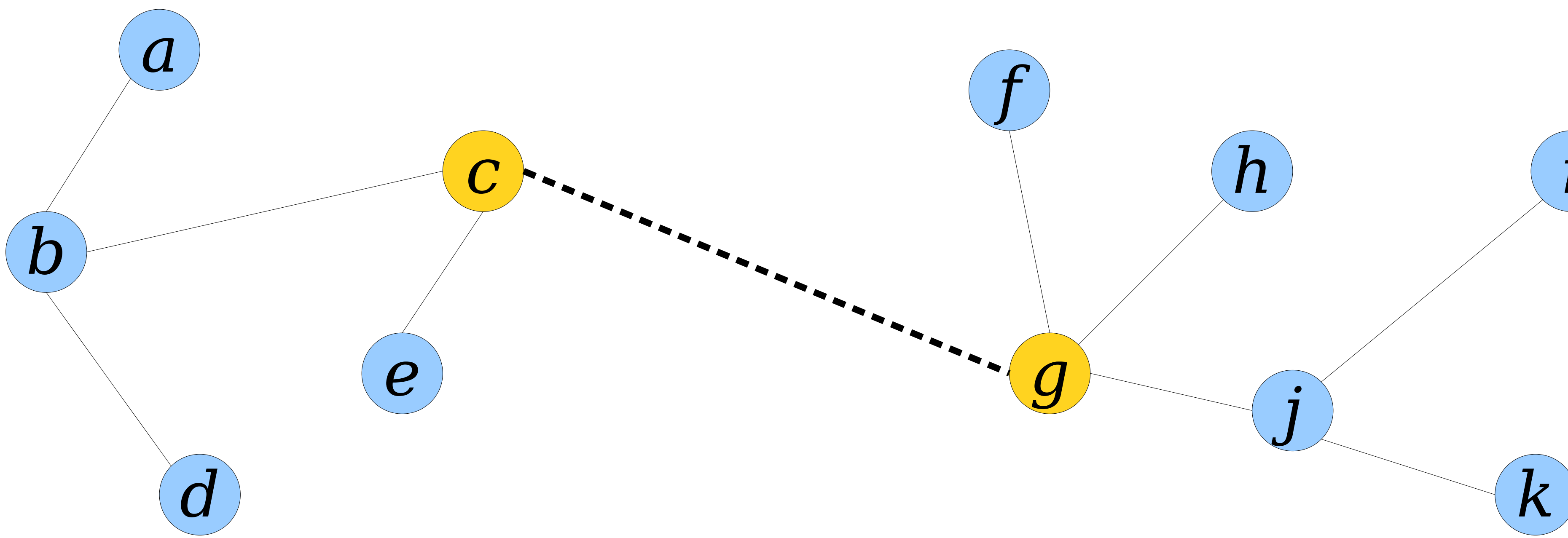
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ce ec cb ba ab bd db bc ***gh hg gf fg gj jk kj ji ij jg***

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 uv.
- Watch what happens to the Euler tours:

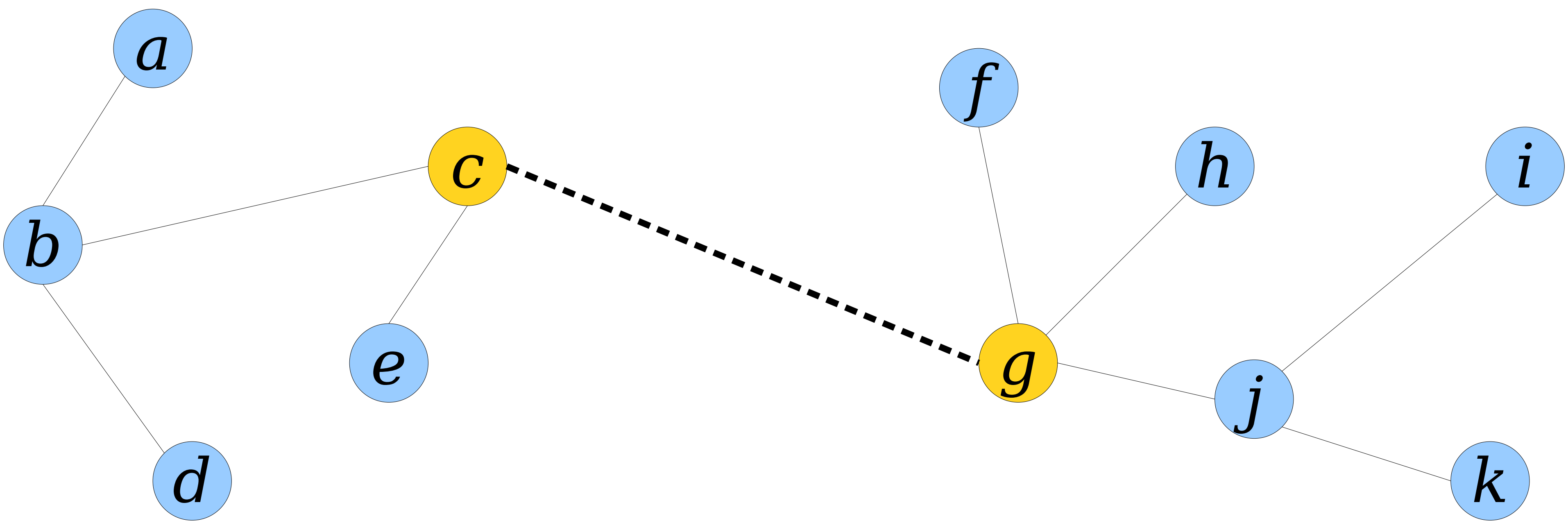


ce ec cb ba ab bd db bc

gh hg gf fg gj jk kj ji ij jg

Euler Tours and Dynamic Trees

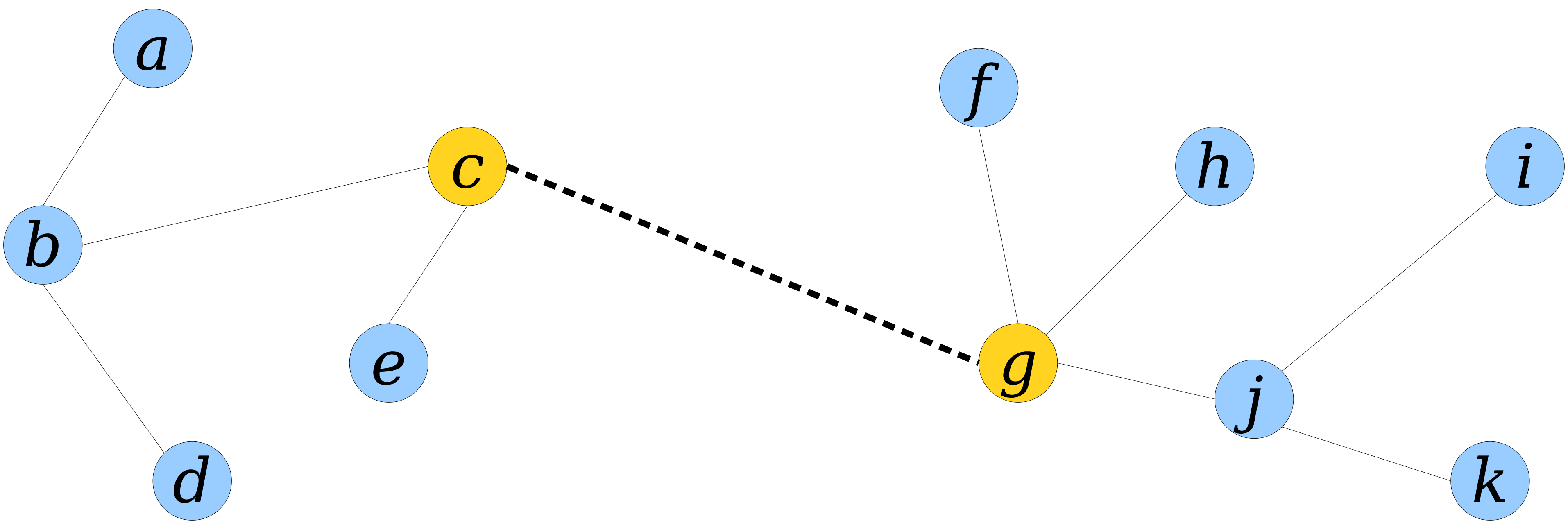
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ce ec cb ba ab bd db bc ***gh hg gf fg gj jk kj ji ij jg***

Euler Tours and Dynamic Trees

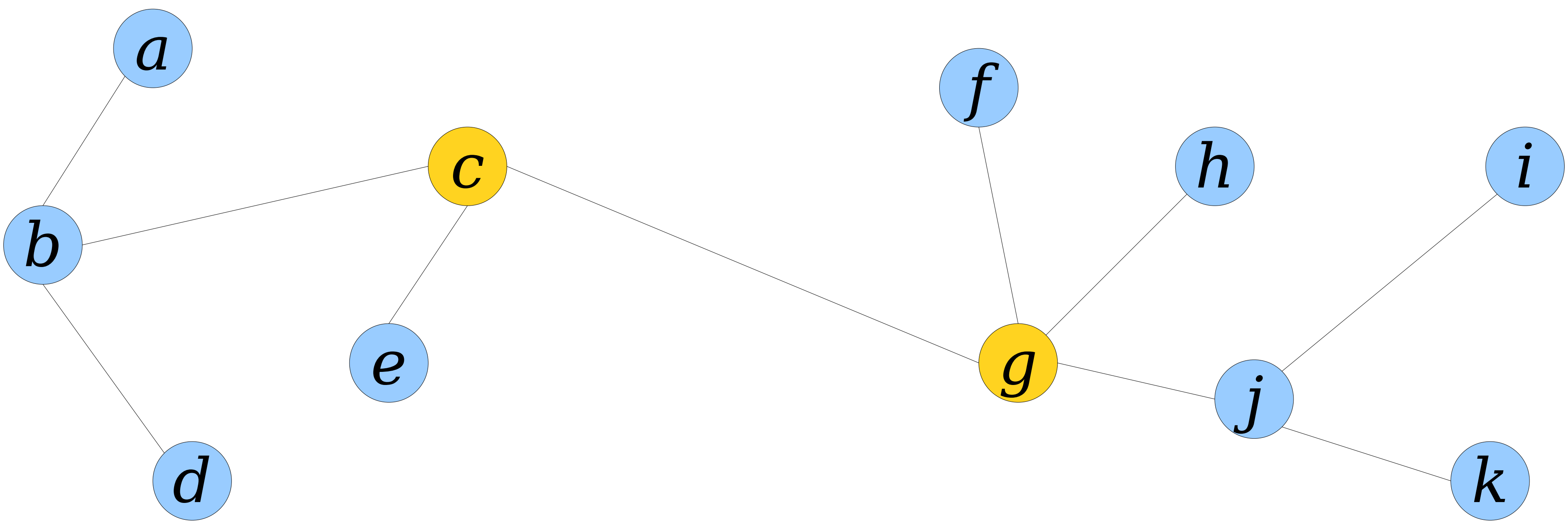
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ce ec cb ba ab bd db bc cg gh hg gf fg gj jk kj ji ij jg gc

Euler Tours and Dynamic Trees

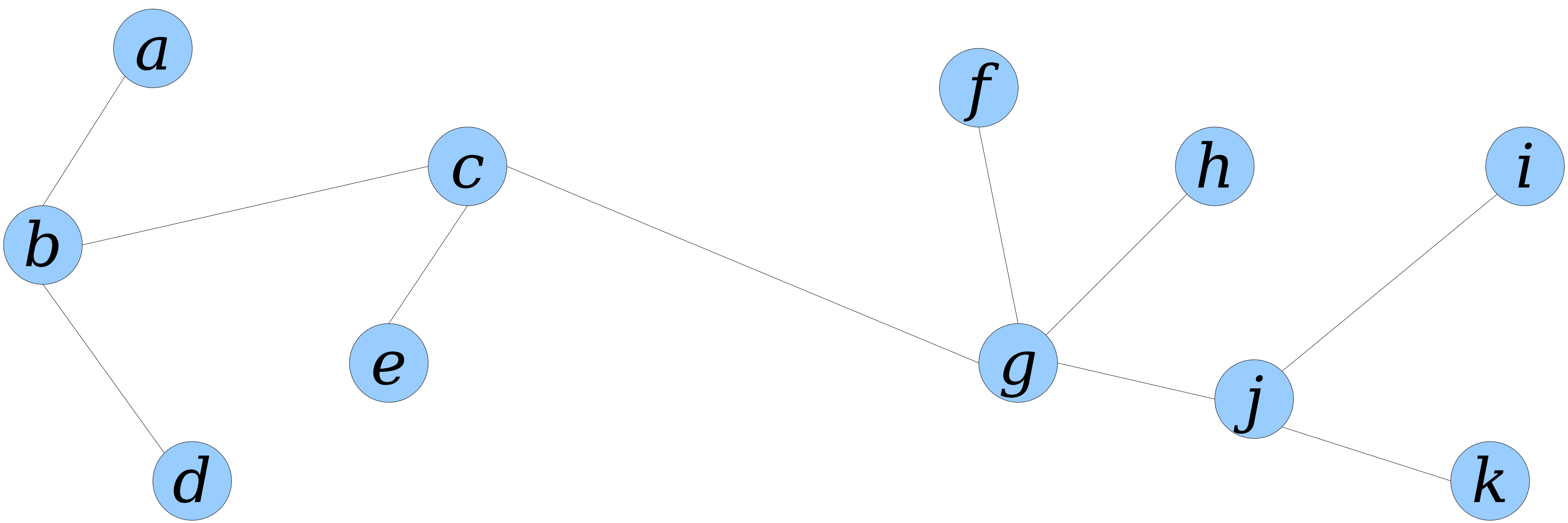
- Given two trees T_1 and T_2, where $u \in T_1$ and $v \in T_2$, executing ***link***(u, v) links the trees together by adding edge uv.
- Watch what happens to the Euler tours:



ce ec cb ba ab bd db bc cg gh hg gf fg gj jk kj ji ij jg gc

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 uv.
- Watch what happens to the Euler tours:



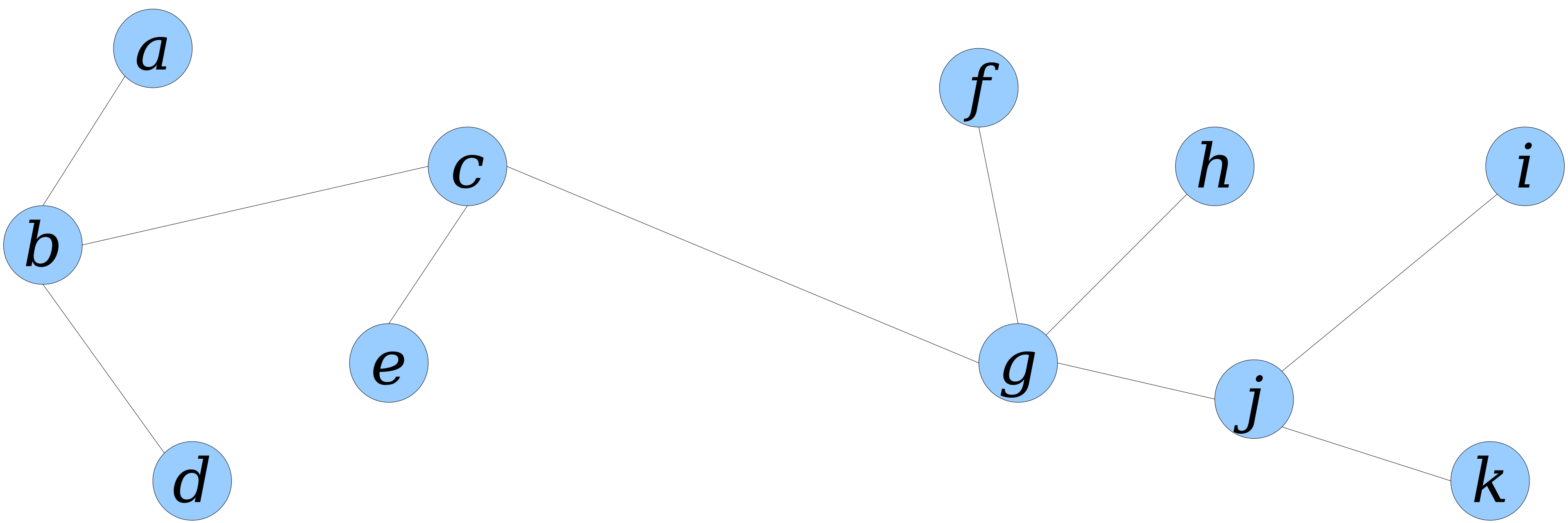
ce ec cb ba ab bd db bc cg gh hg gf fg gj jk kj ji ij jg gc

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 uv.
- To ***link***(u, v):
  - Let E_1 and E_2 be Euler tours of T_1 and T_2, respectively.
  - ***reroot***(u).
  - ***reroot***(v).
  - Concatenate E_1 uv E_2 vu.

Euler Tours and Dynamic Trees

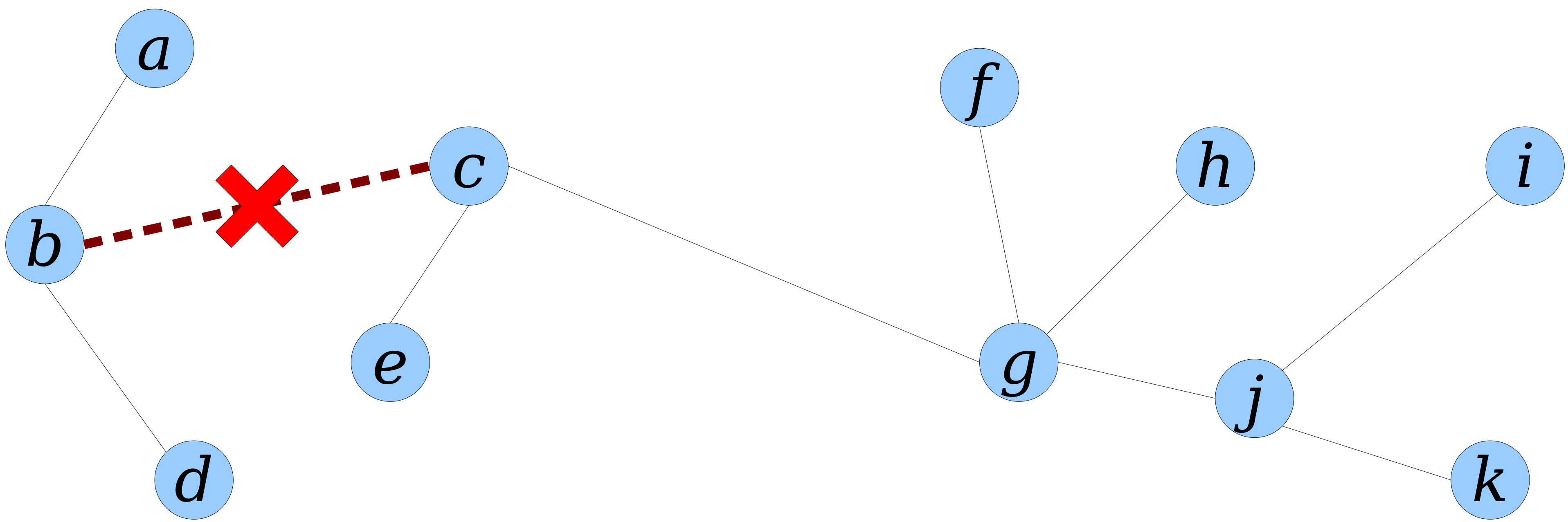
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ce ec cb ba ab bd db bc cg gh hg gf fg gj jk kj ji ij jg gc

Euler Tours and Dynamic Trees

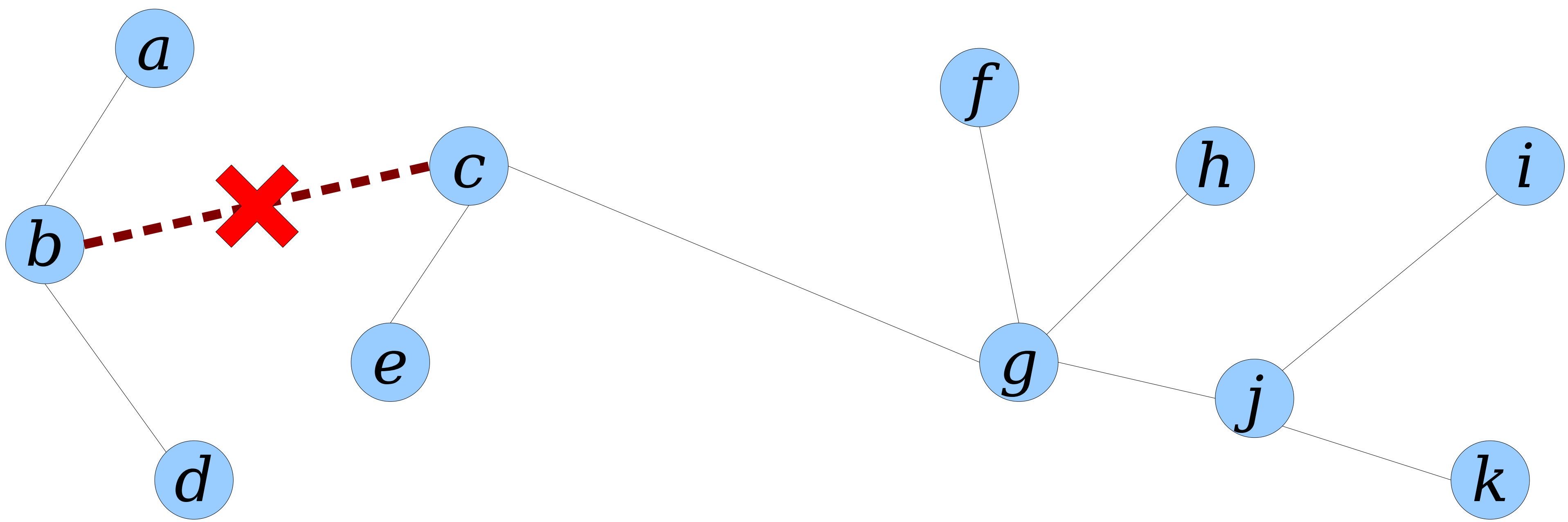
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ce ec cb ba ab bd db bc cg gh hg gf fg gj jk kj ji ij jg gc

Euler Tours and Dynamic Trees

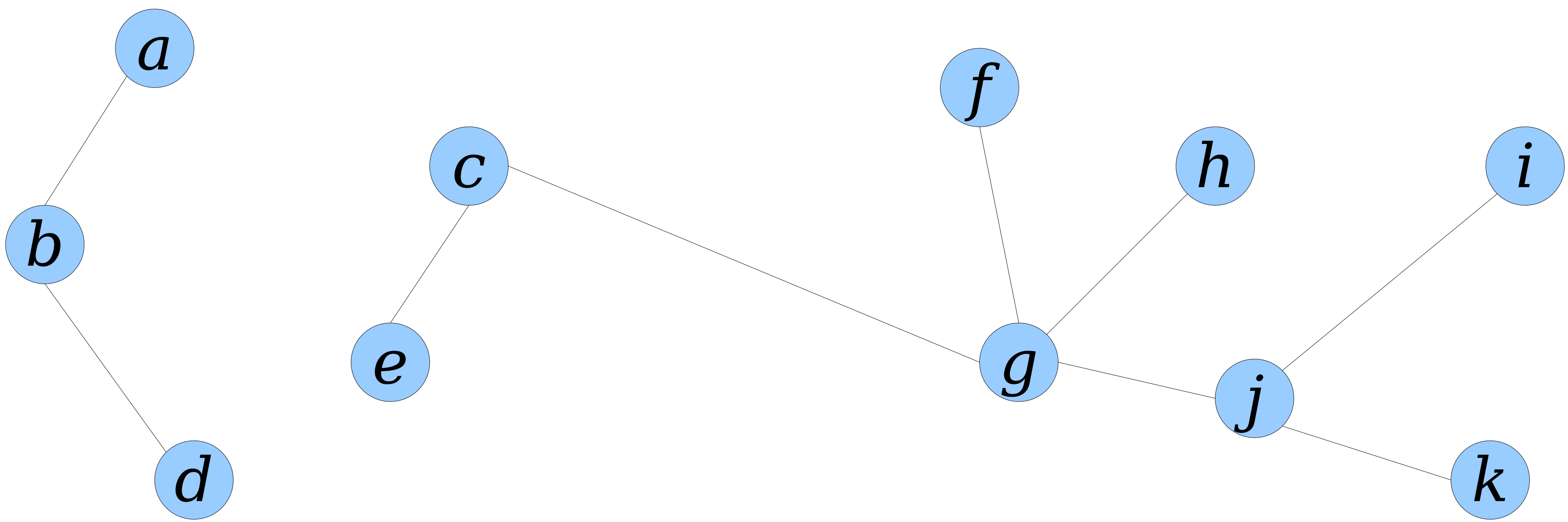
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ce ec cb ba ab bd db bc cg gh hg gf fg gj jk kj ji ij jg gc

Euler Tours and Dynamic Trees

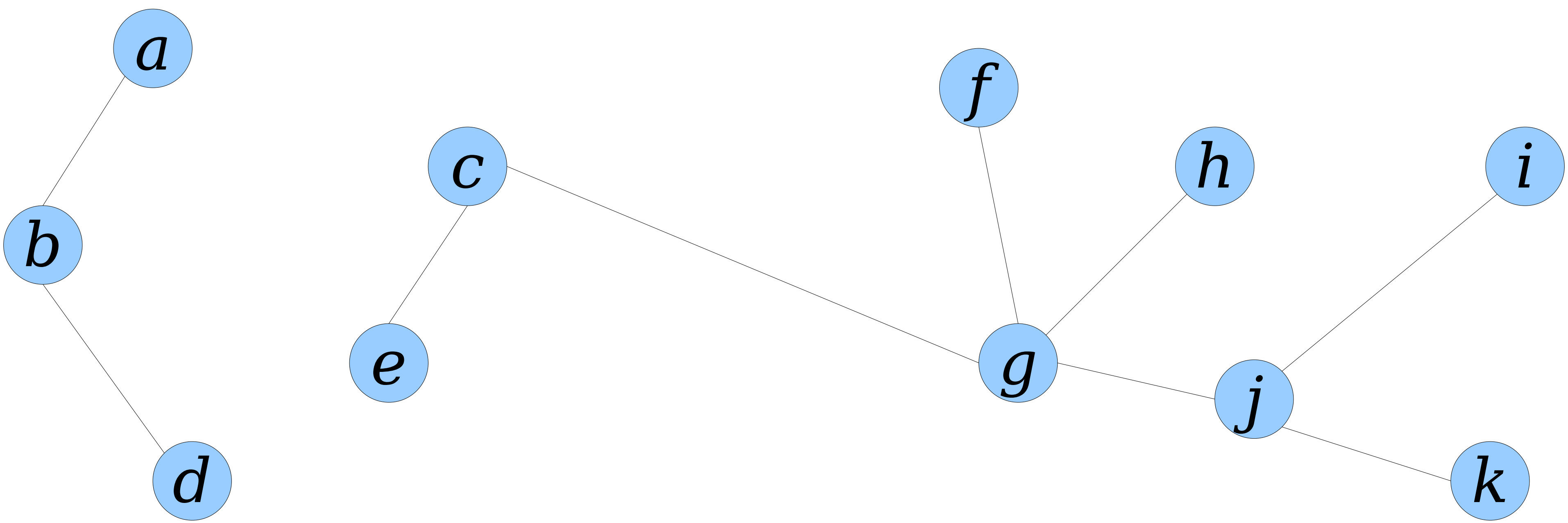
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ce ec cb ba ab bd db bc cg gh hg gf fg gj jk kj ji ij jg gc

Euler Tours and Dynamic Trees

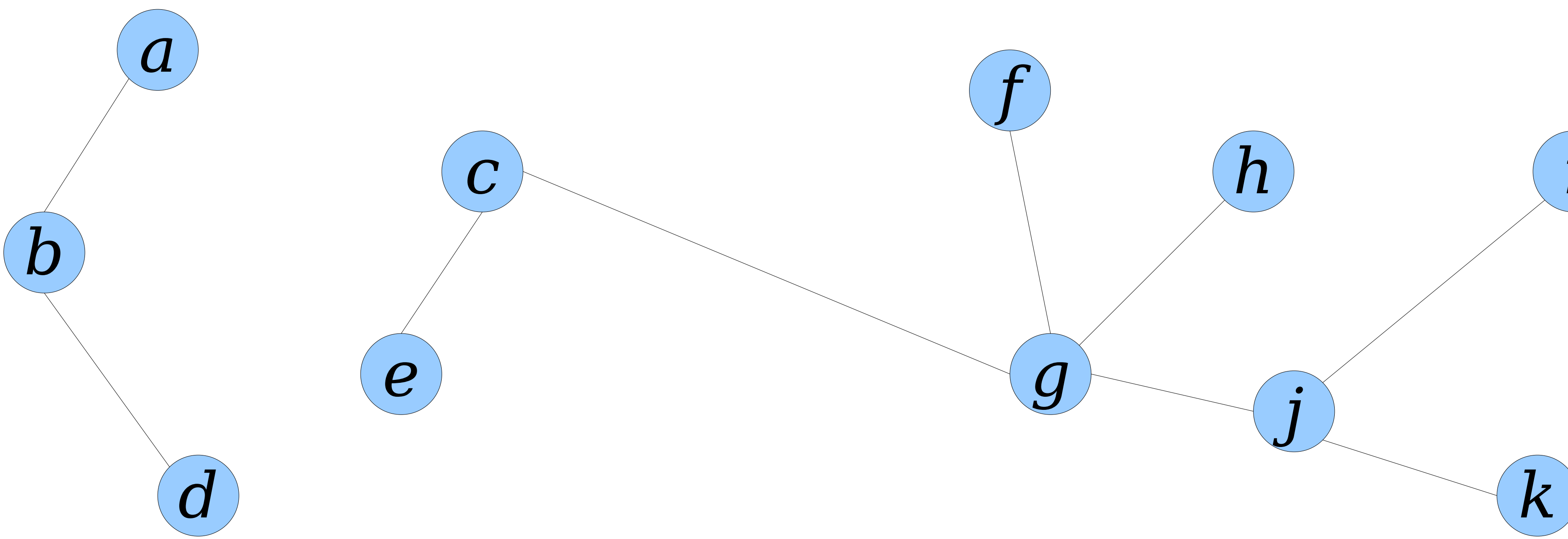
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ce ec   ***ba ab bd db***   ***cg gh hg gf fg gj jk kj ji ij jg gc***

Euler Tours and Dynamic Trees

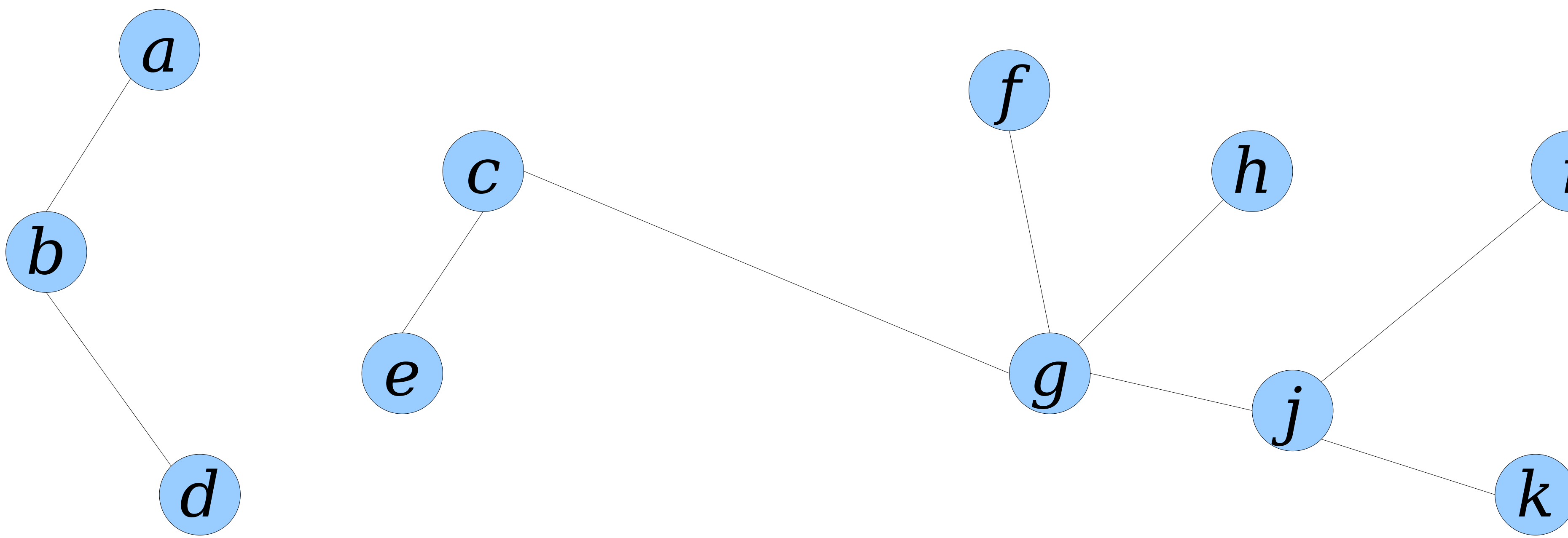
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ce ec ***ba ab bd db*** ***cg gh hg gf fg gj jk kj ji ij jg gc***

Euler Tours and Dynamic Trees

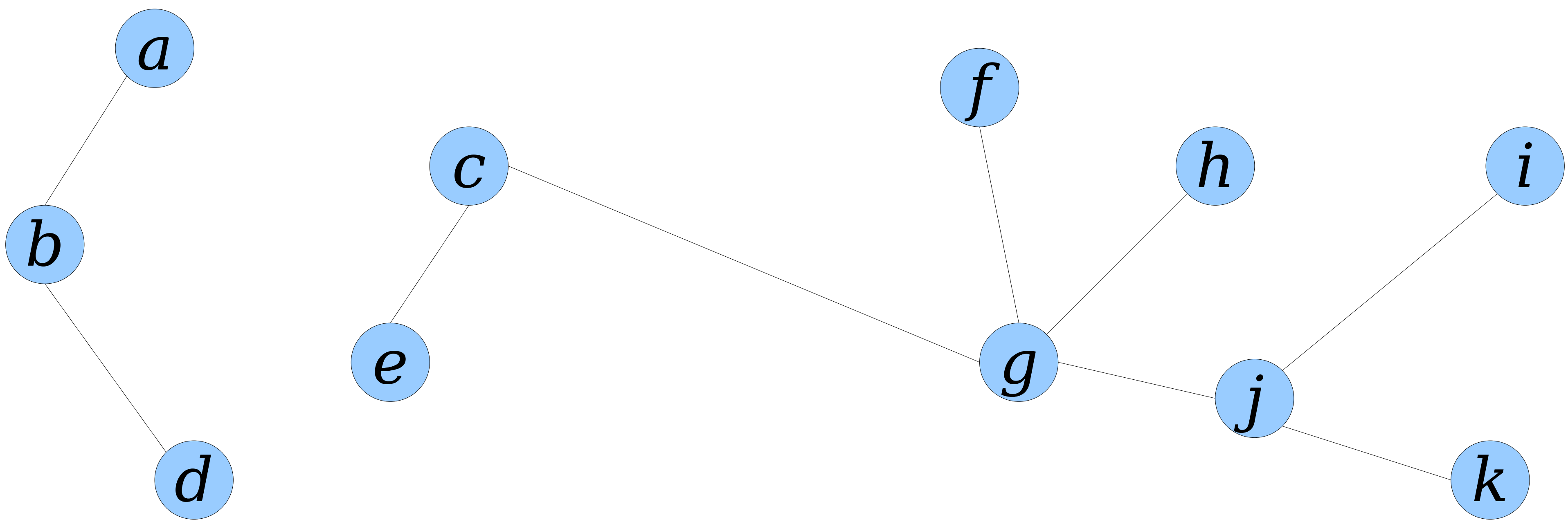
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ba ab bd db ***ce ec cg gh hg gf fg gj jk kj ji ij jg gc***

Euler Tours and Dynamic Trees

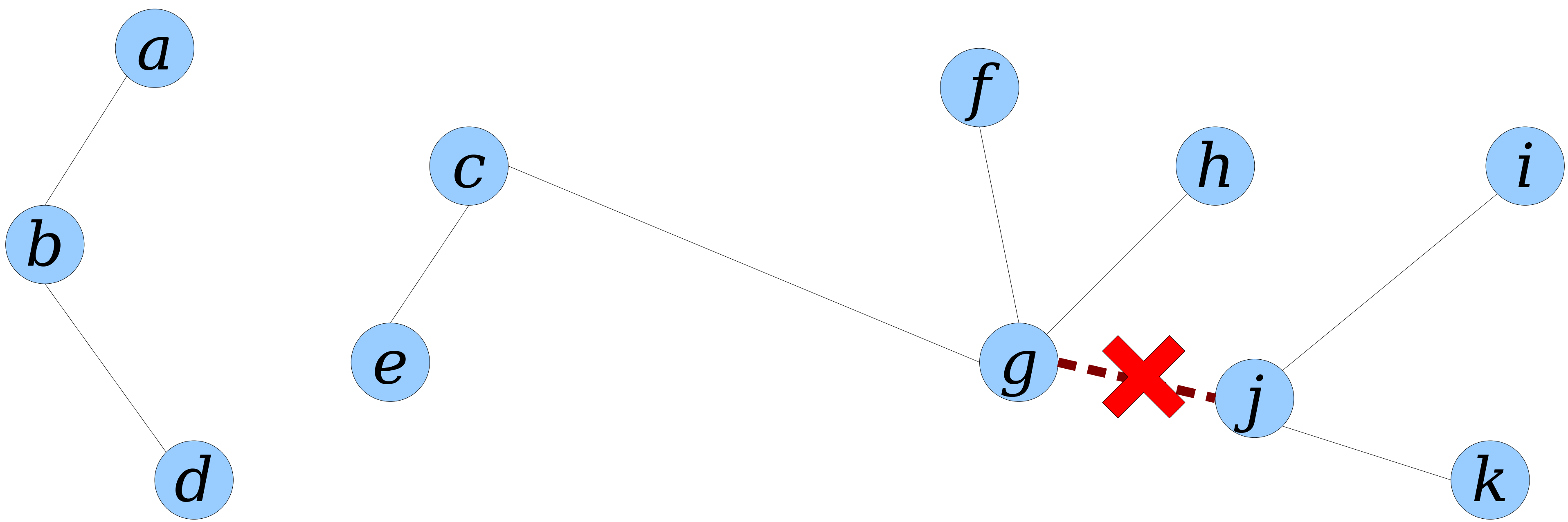
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ba ab bd db ***ce ec cg gh hg gf fg gj jk kj ji ij jg gc***

Euler Tours and Dynamic Trees

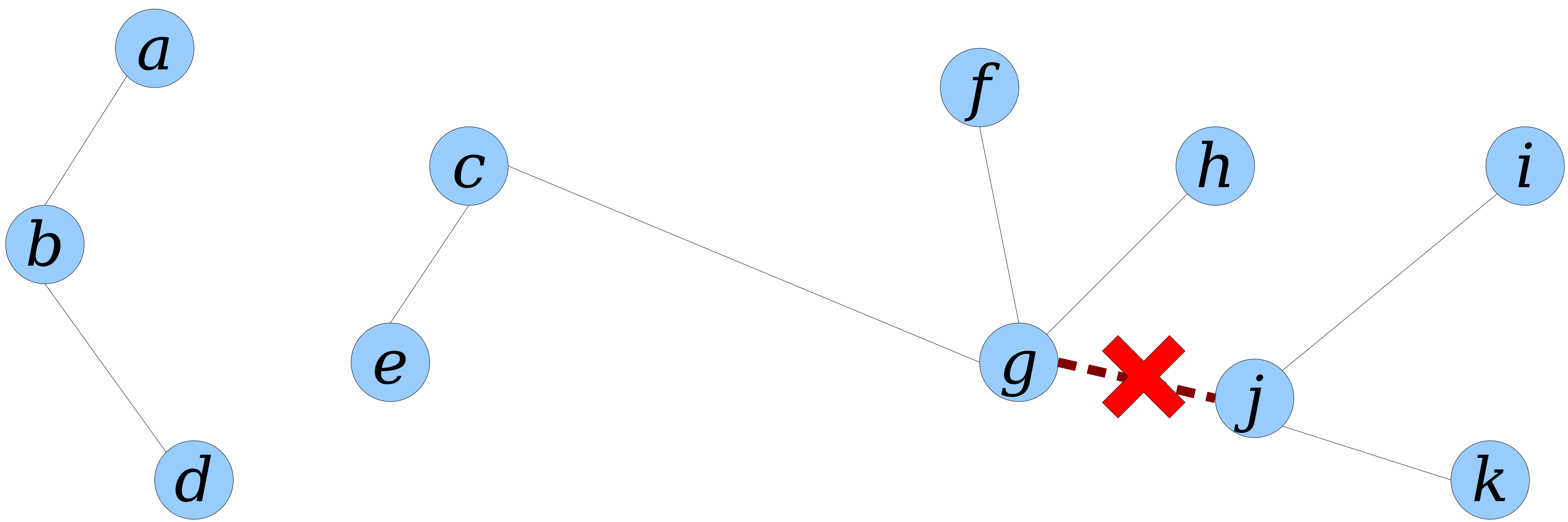
- Given a tree *T*, executing ***cut***(*u*, *v*) cuts the edge *uv* from the tree (assuming it exists).
- Watch what happens to the Euler tour of *T*:



ba ab bd db ***ce ec cg gh hg gf fg gj jk kj ji ij jg gc***

Euler Tours and Dynamic Trees

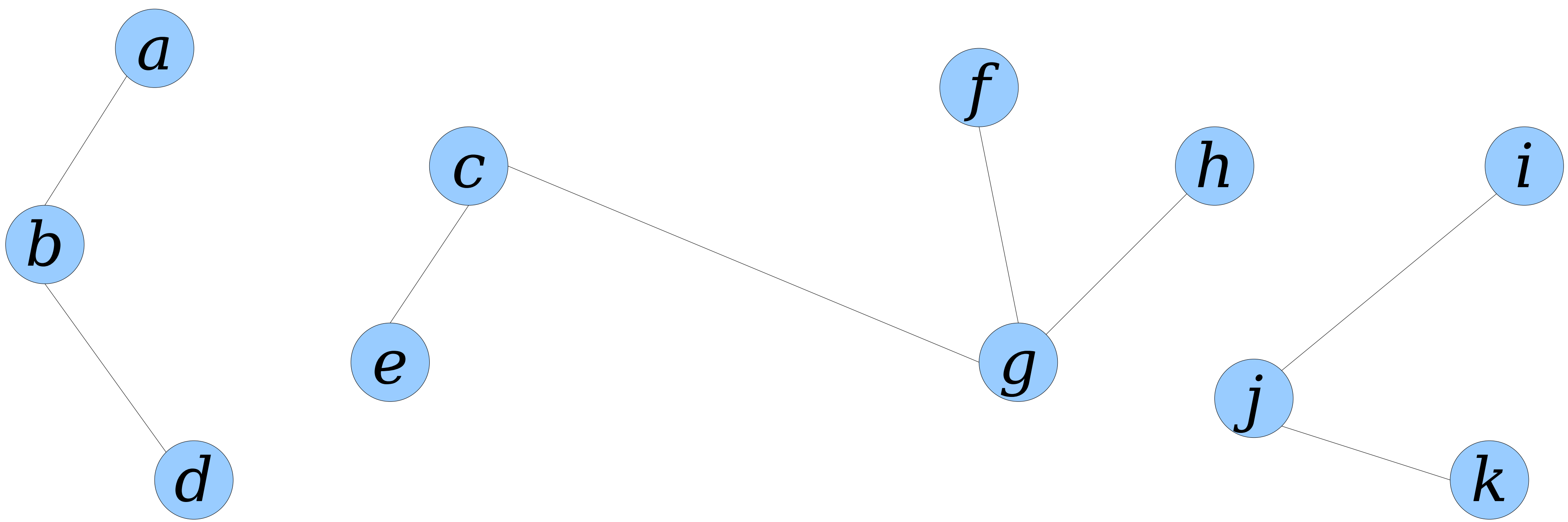
- Given a tree *T*, executing ***cut***(*u*, *v*) cuts the edge *uv* from the tree (assuming it exists).
- Watch what happens to the Euler tour of *T*:



ba ab bd db ***ce ec cg gh hg gf fg gj jk kj ji ij jg gc***

Euler Tours and Dynamic Trees

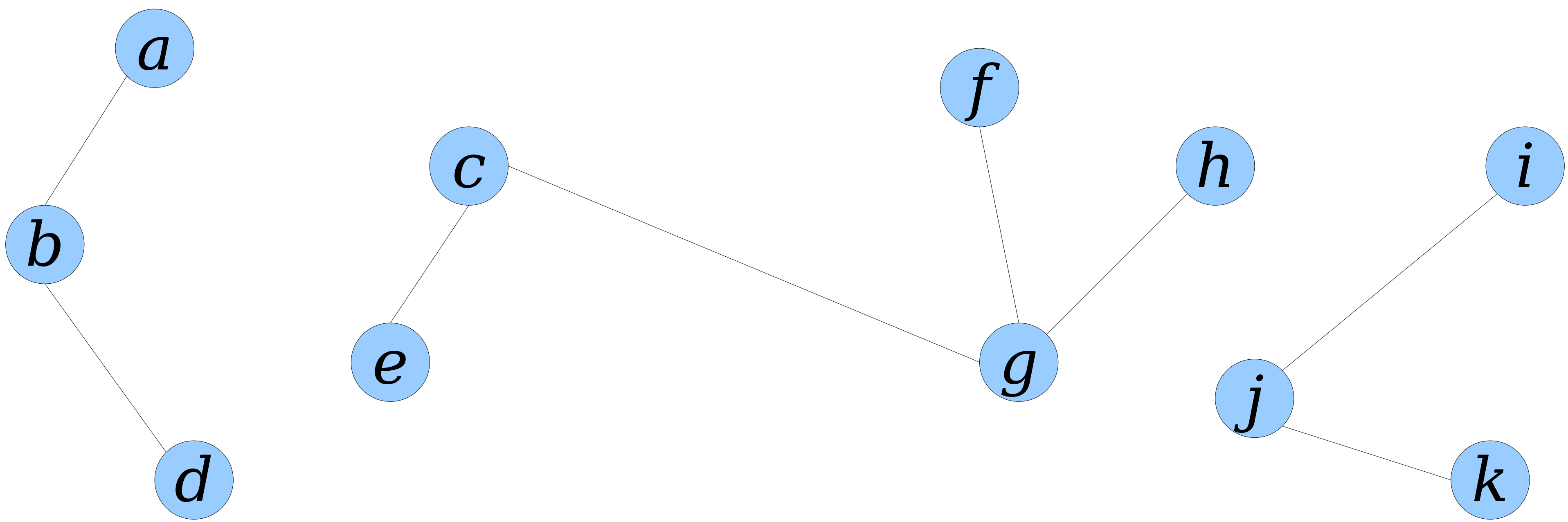
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ba ab bd db ***ce ec cg gh hg gf fg*** ***jk kj ji ij*** ***gc***

Euler Tours and Dynamic Trees

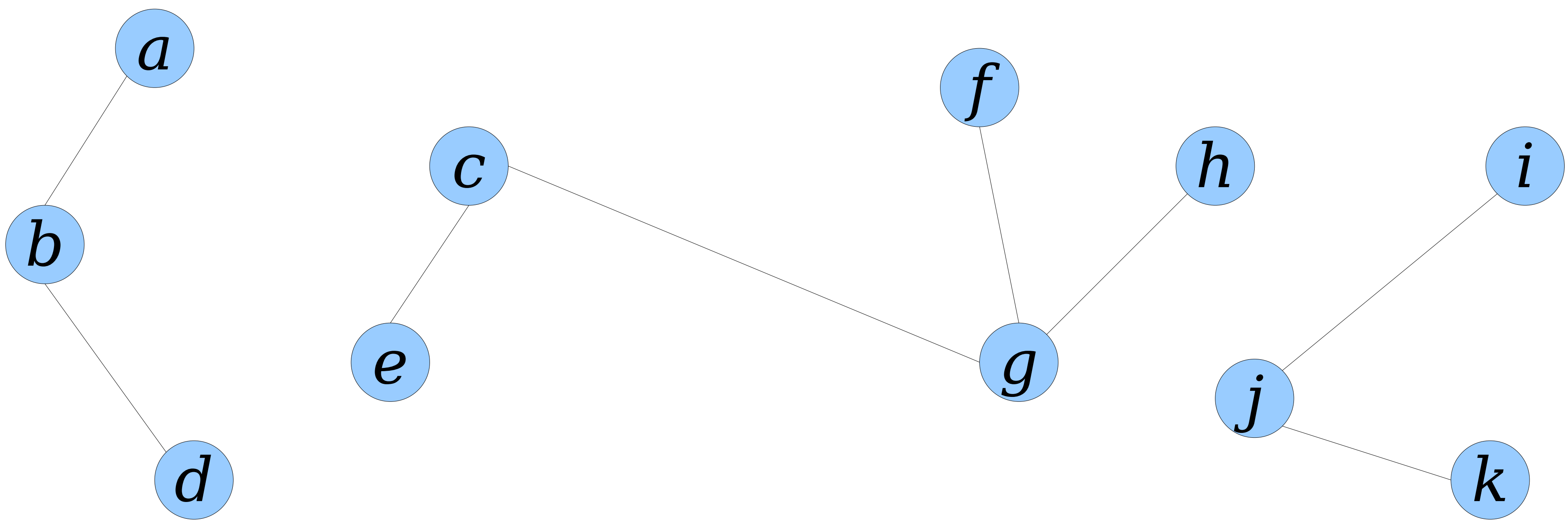
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ba ab bd db ***ce ec cg gh hg gf fg*** ***jk kj ji ij*** ***gc***

Euler Tours and Dynamic Trees

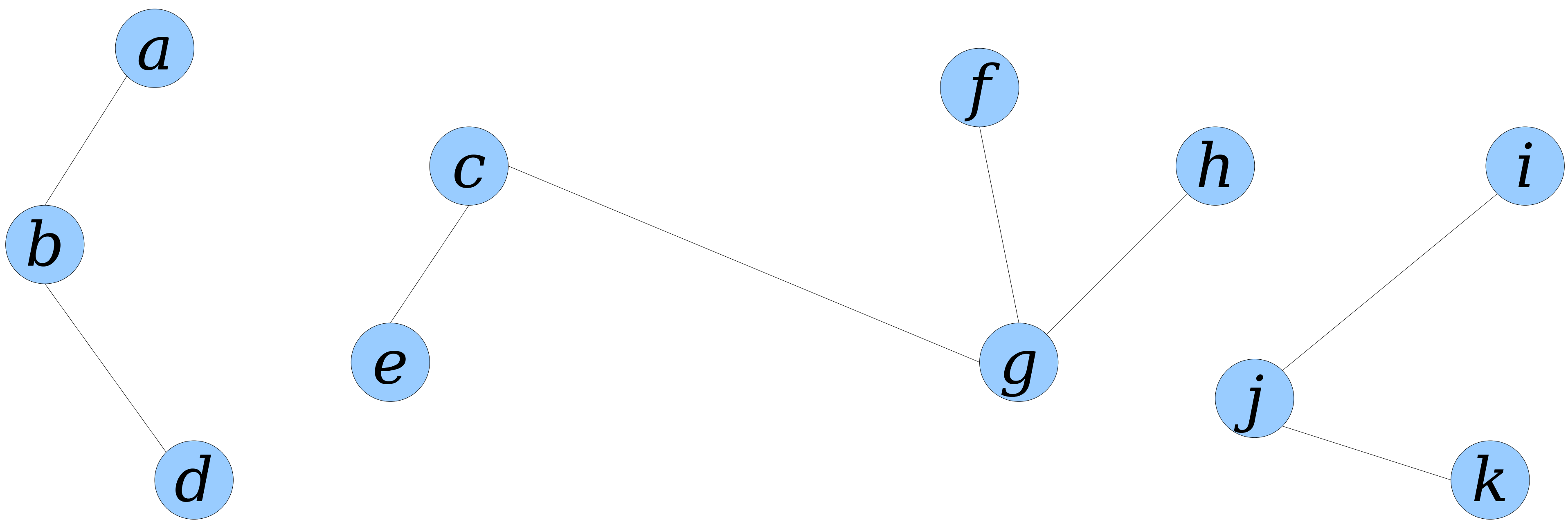
- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- Watch what happens to the Euler tour of T:



ba ab bd db ***ce ec cg gh hg gf fg gc*** ***jk kj ji ij***

Euler Tours and Dynamic Trees

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



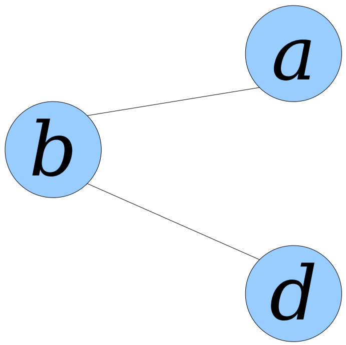
ba ab bd db ***ce ec cg gh hg gf fg gc*** ***jk kj ji ij***

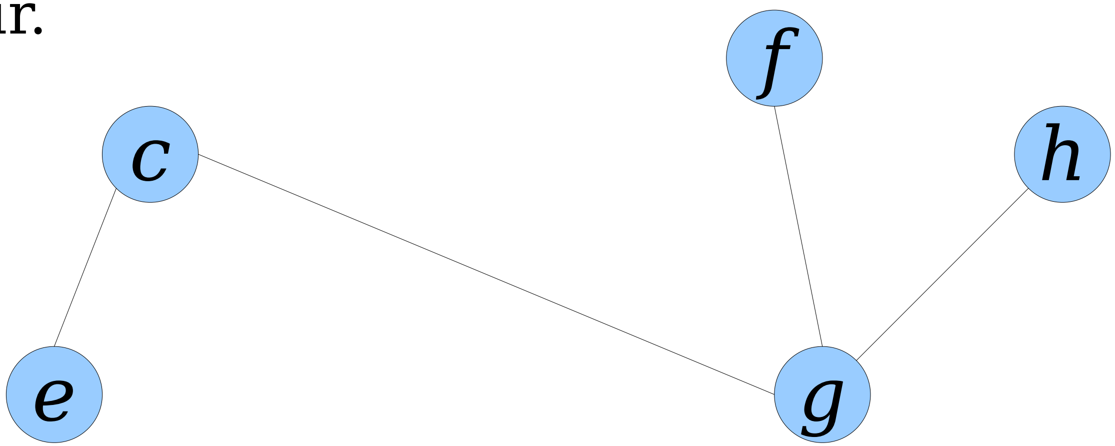
Euler Tours and Dynamic Trees

- Given a tree T, executing ***cut***(u, v) cuts the edge uv from the tree (assuming it exists).
- To perform ***cut***(u, v):
  - Let E be the Euler tour containing uv and vu.
  - Remove uv and vu from E to form E_1, E_2, and E_3.
  - Then E_1E_3 and E_2 are Euler tours of the two new trees.

Checking Connectivity

- We also need a way to answer queries of the form ***are-connected***(*u*, *v*).
- This query focuses on *nodes*, but our Euler tours store *edges*.
- ***Cute Trick:*** Introduce a self-loop on each node that represents the node itself. Add that to each tour as a proxy for the node itself.
- Now, we can answer ***are-connected***(*x*, *y*) by seeing if *xx* and *yy* are part of the same tour.



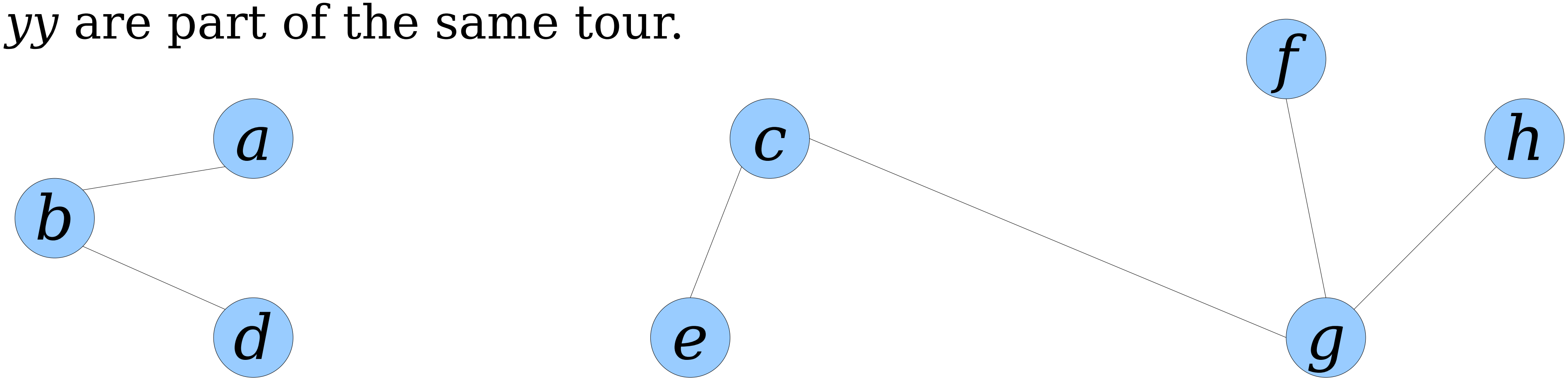


ba ab bd db

ce ec cg gh hg gf fg gc

Checking Connectivity

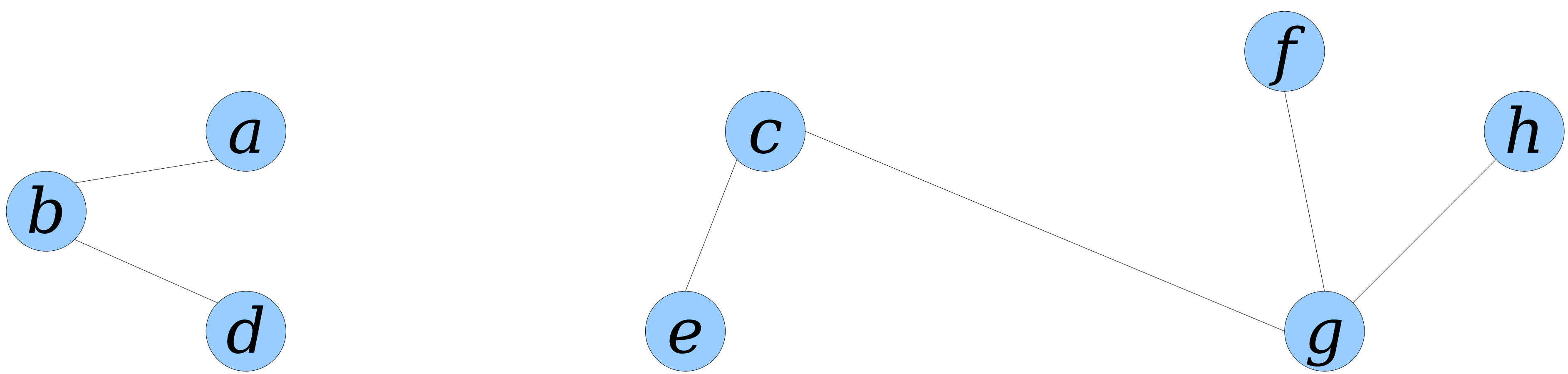
- We also need a way to answer queries of the form ***are-connected***(*u*, *v*).
- This query focuses on *nodes*, but our Euler tours store *edges*.
- ***Cute Trick:*** Introduce a self-loop on each node that represents the node itself. Add that to each tour as a proxy for the node itself.
- Now, we can answer ***are-connected***(*x*, *y*) by seeing if *xx* and *yy* are part of the same tour.



ba aa ab bb bd dd db ***ce ee ec cg gg gh hh hg gf ff fg gc cc***

Checking Connectivity

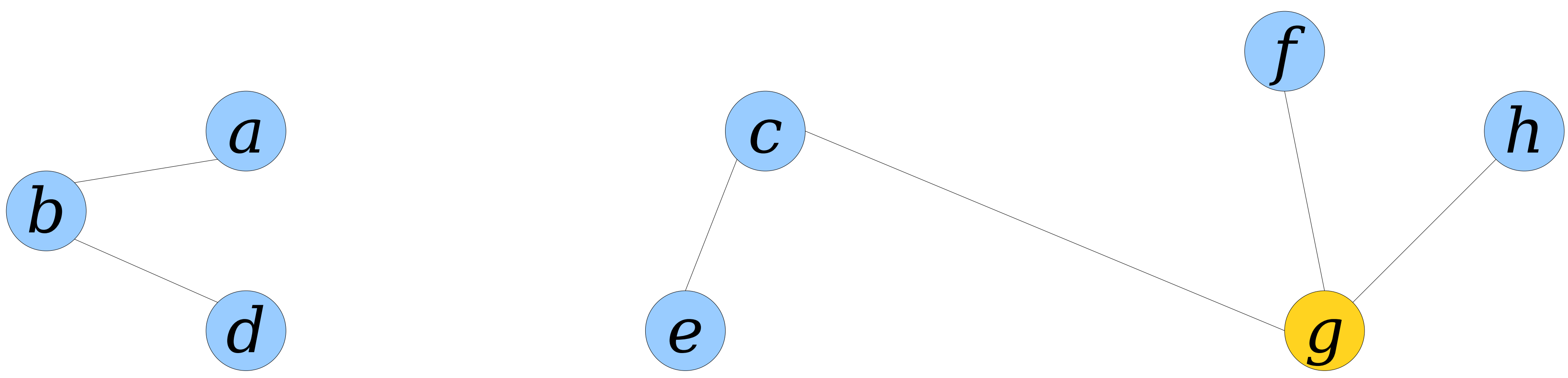
- This also makes it a lot easier to reroot a tour at a node *x*.
- We simply find *xx*, then rotate that edge to the front of the tour.



ba aa ab bb bd dd db ***ce ee ec cg gg gh hh hg gf ff fg gc cc***

Checking Connectivity

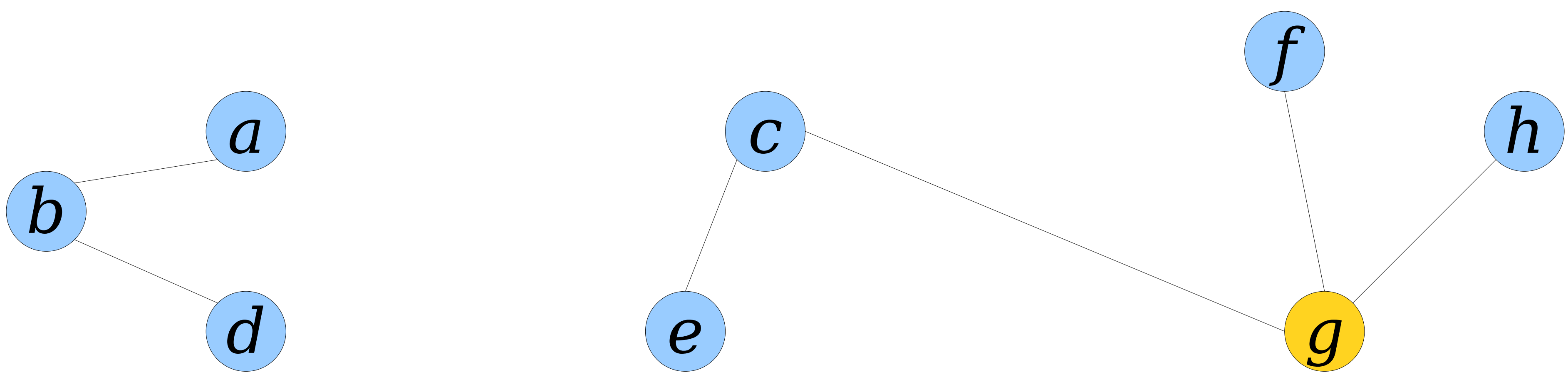
- This also makes it a lot easier to reroot a tour at a node *x*.
- We simply find *xx*, then rotate that edge to the front of the tour.



ba aa ab bb bd dd db ***ce ee ec cg gg gh hh hg gf ff fg gc cc***

Checking Connectivity

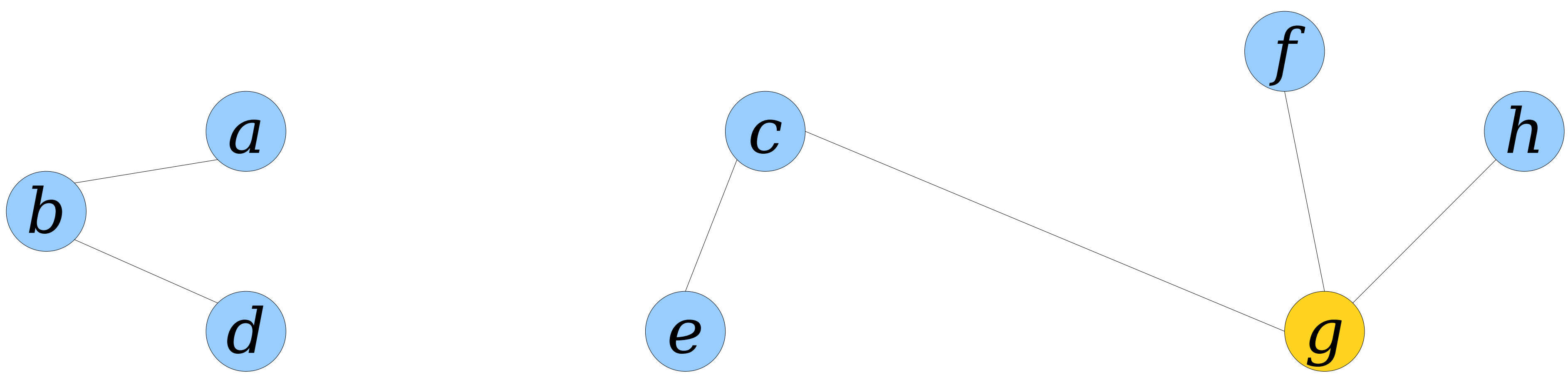
- This also makes it a lot easier to reroot a tour at a node *x*.
- We simply find *xx*, then rotate that edge to the front of the tour.



ba aa ab bb bd dd db ***ce ee ec cg gg gh hh hg gf ff fg gc cc***

Checking Connectivity

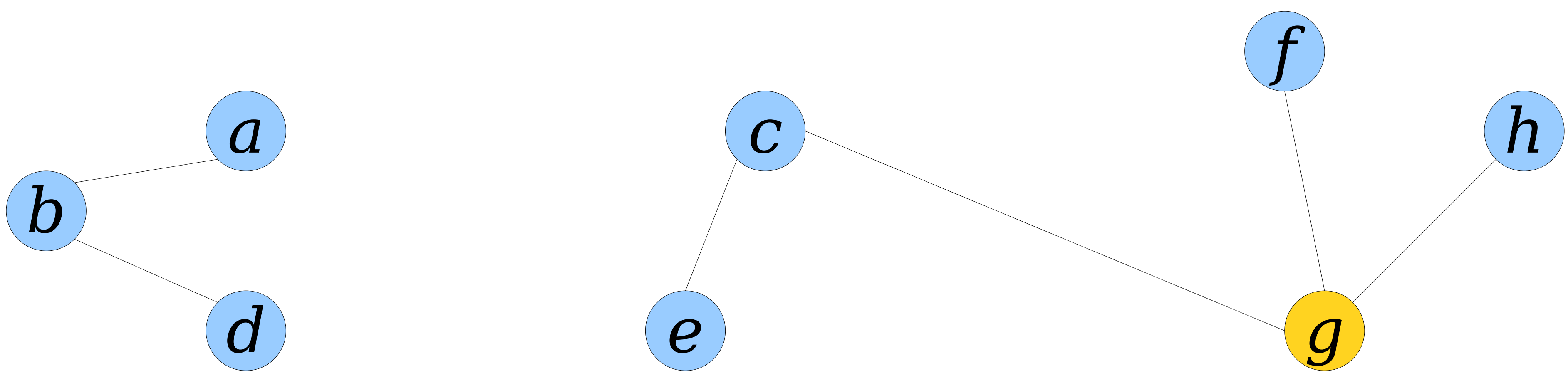
- This also makes it a lot easier to reroot a tour at a node *x*.
- We simply find *xx*, then rotate that edge to the front of the tour.



ba aa ab bb bd dd db ***gg gh hh hg gf ff fg gc cc ce ee ec cg***

Checking Connectivity

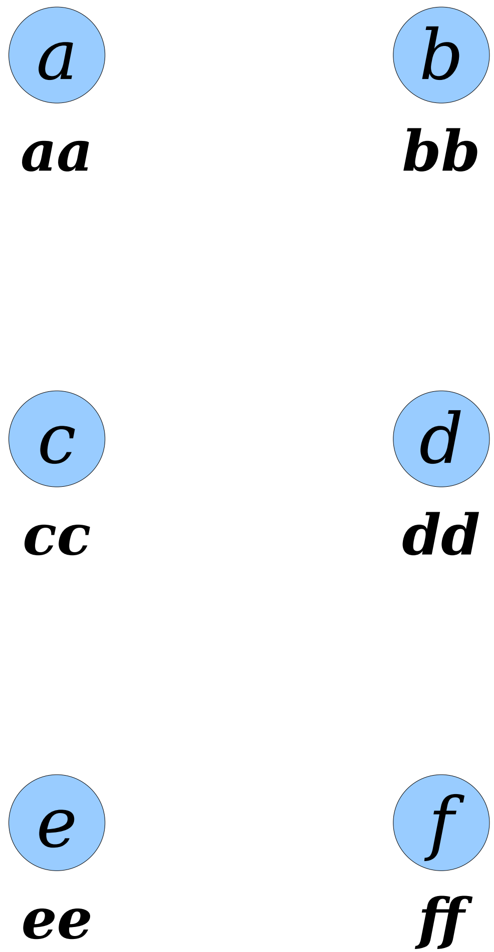
- This also makes it a lot easier to reroot a tour at a node *x*.
- We simply find *xx*, then rotate that edge to the front of the tour.



ba aa ab bb bd dd db ***gg gh hh hg gf ff fg gc cc ce ee ec cg***

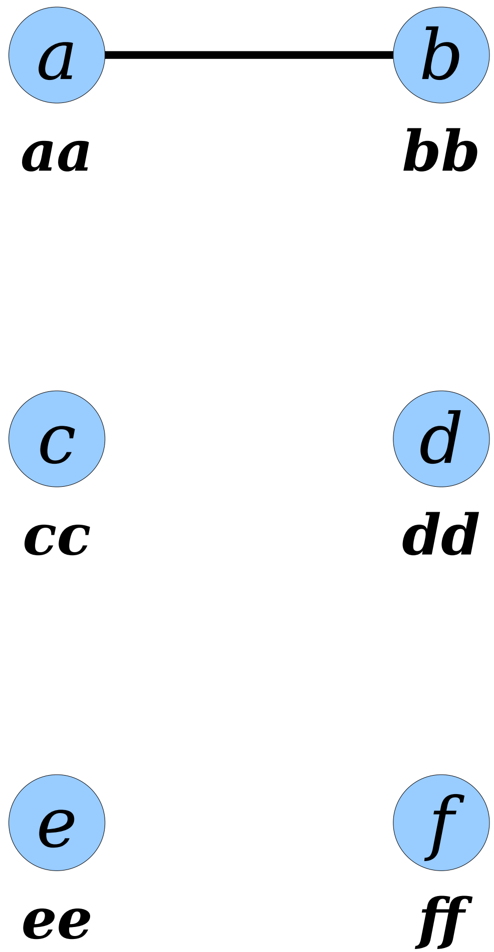
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as T_x xy T_y yx.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as T_1 T_3 and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



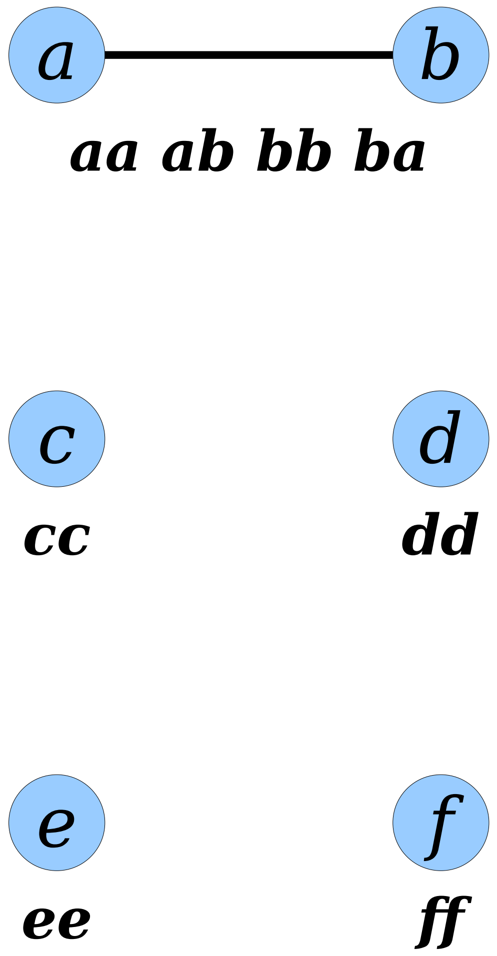
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



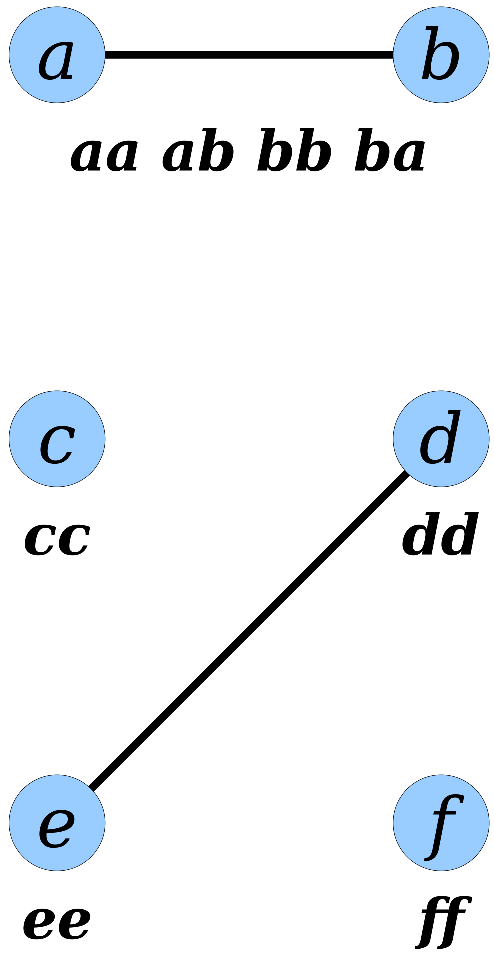
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



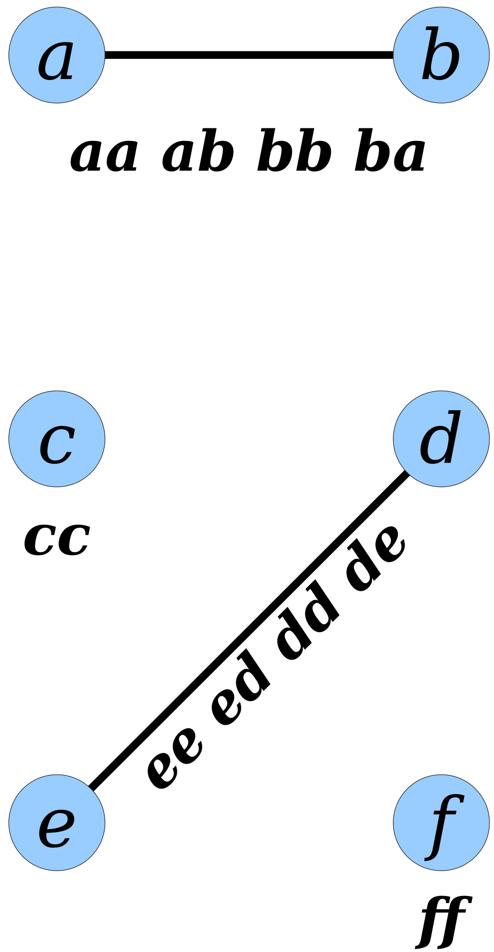
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



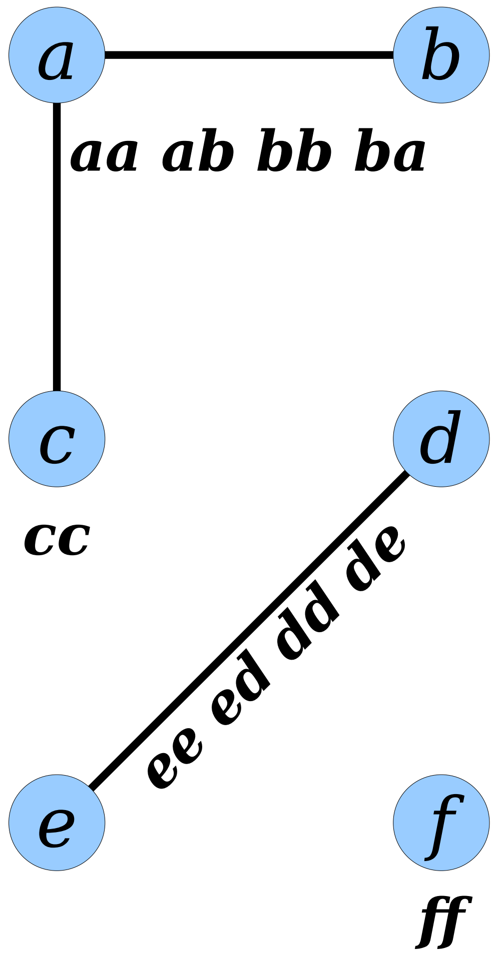
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



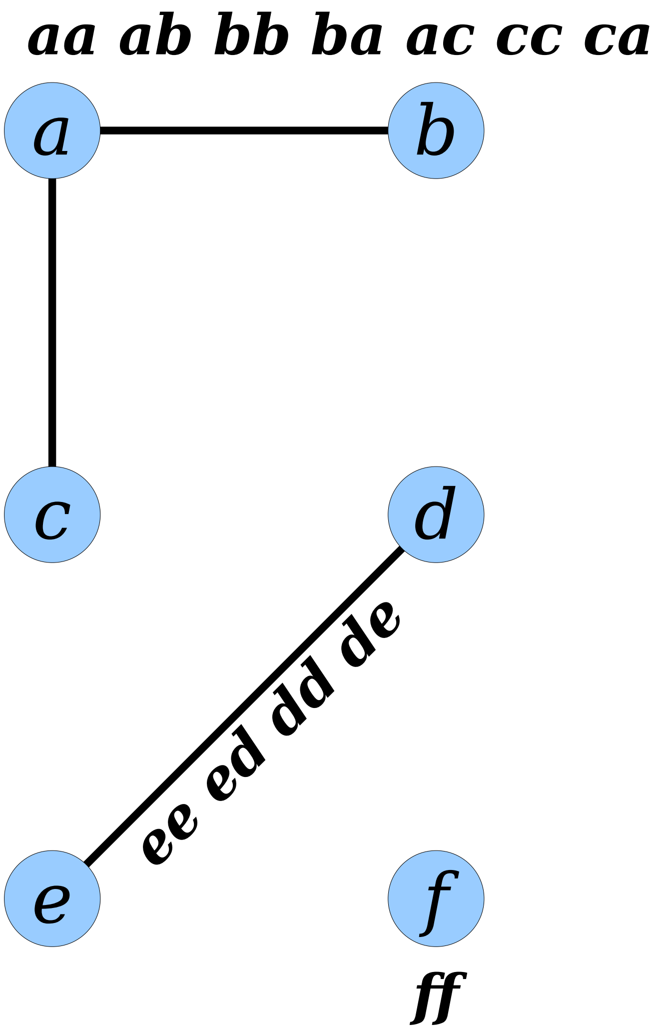
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



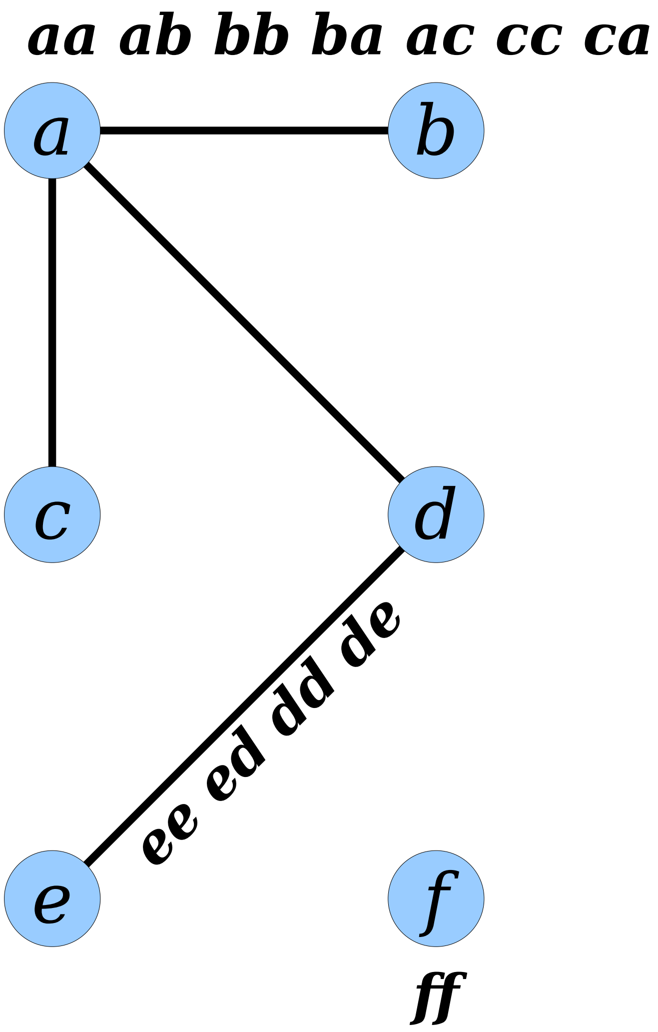
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



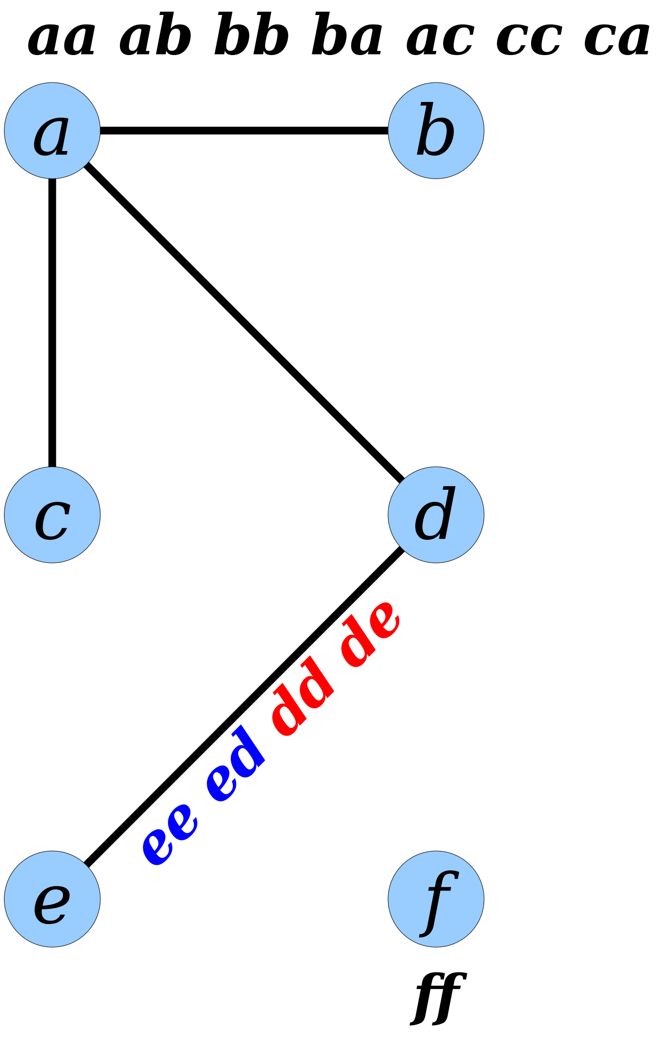
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x \; xy \; T_y \; yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1 \; T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



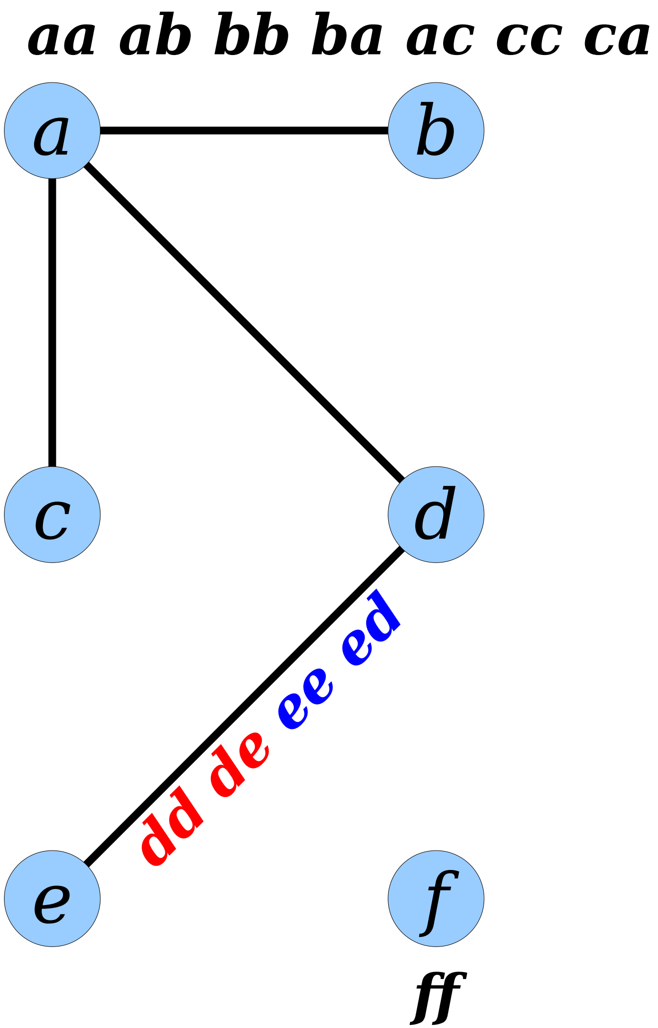
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



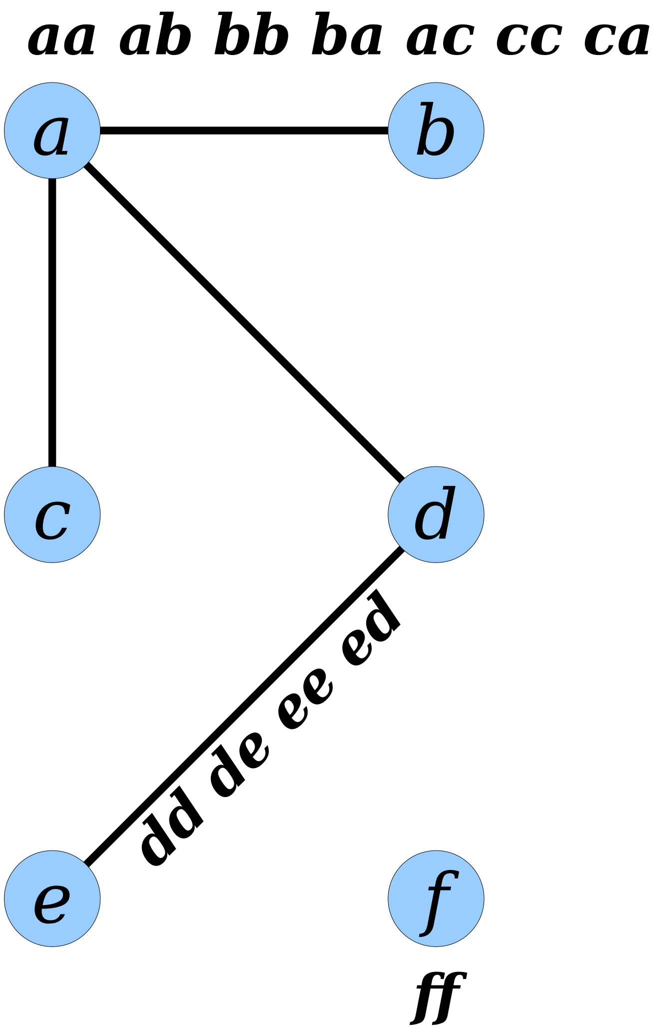
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



Putting It All Together

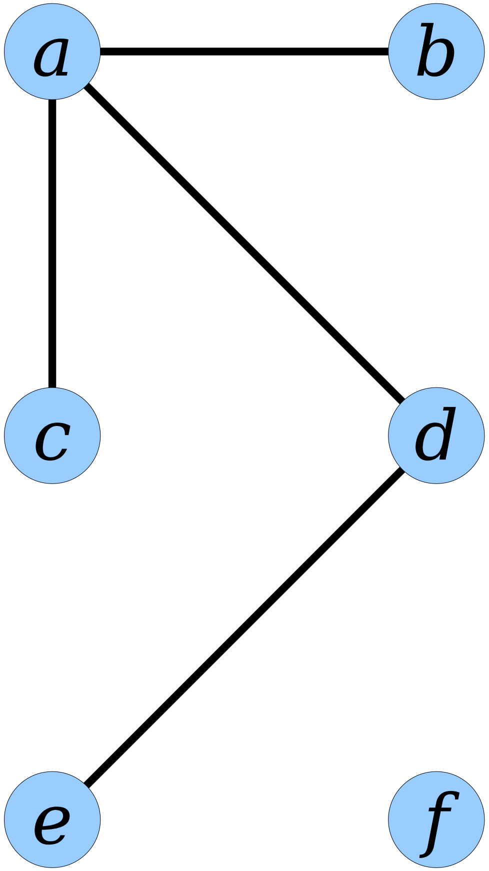
- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.

aa ab bb ba ac cc ca ad dd de ee ed da

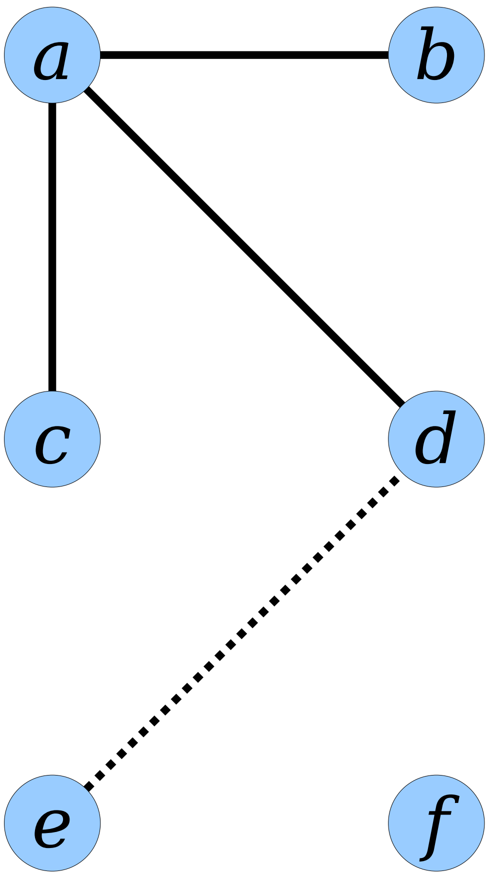


ff

Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.

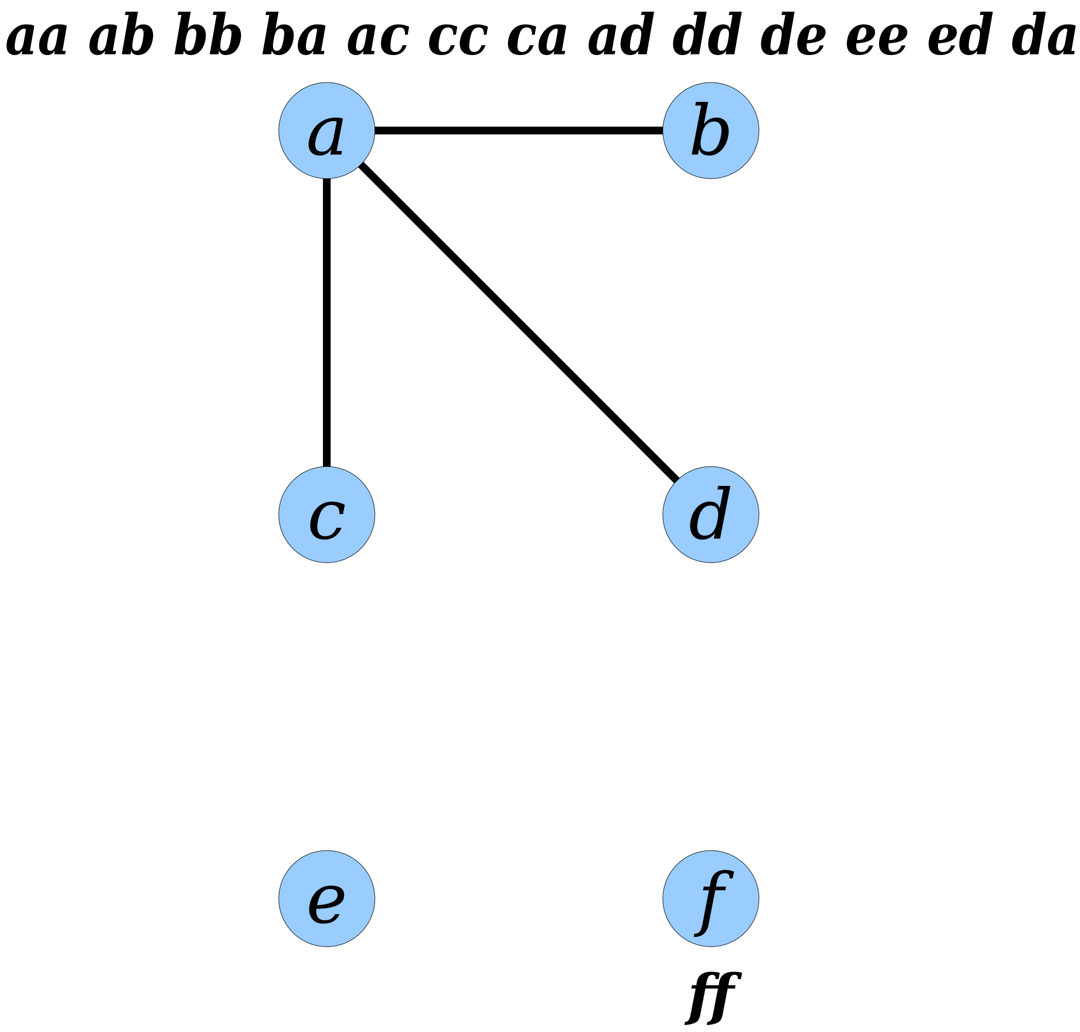
aa ab bb ba ac cc ca ad dd de ee ed da



ff

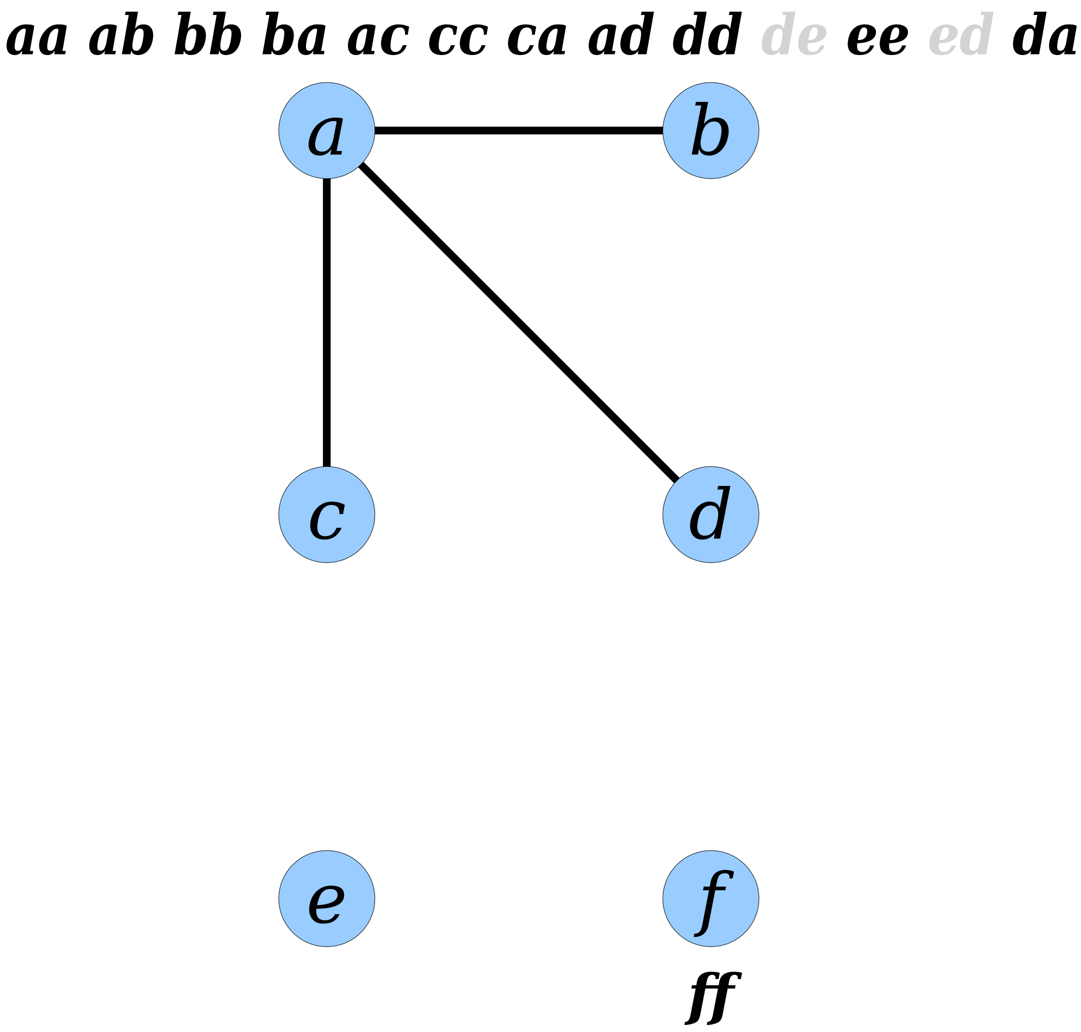
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



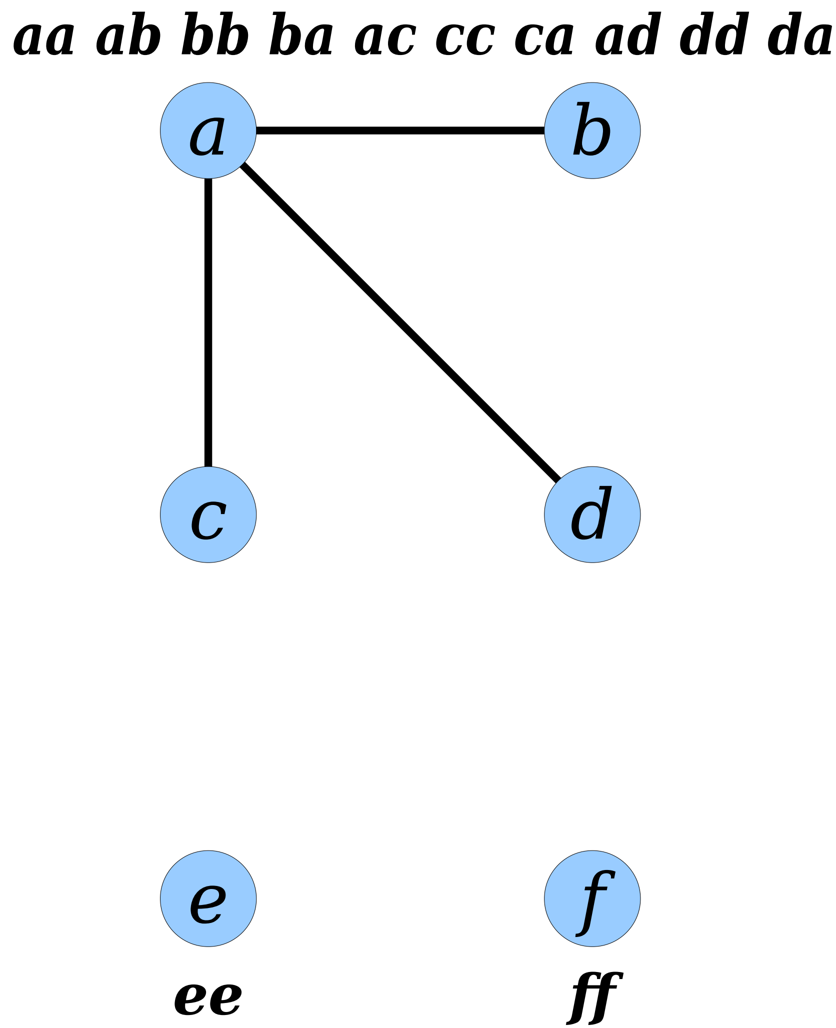
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



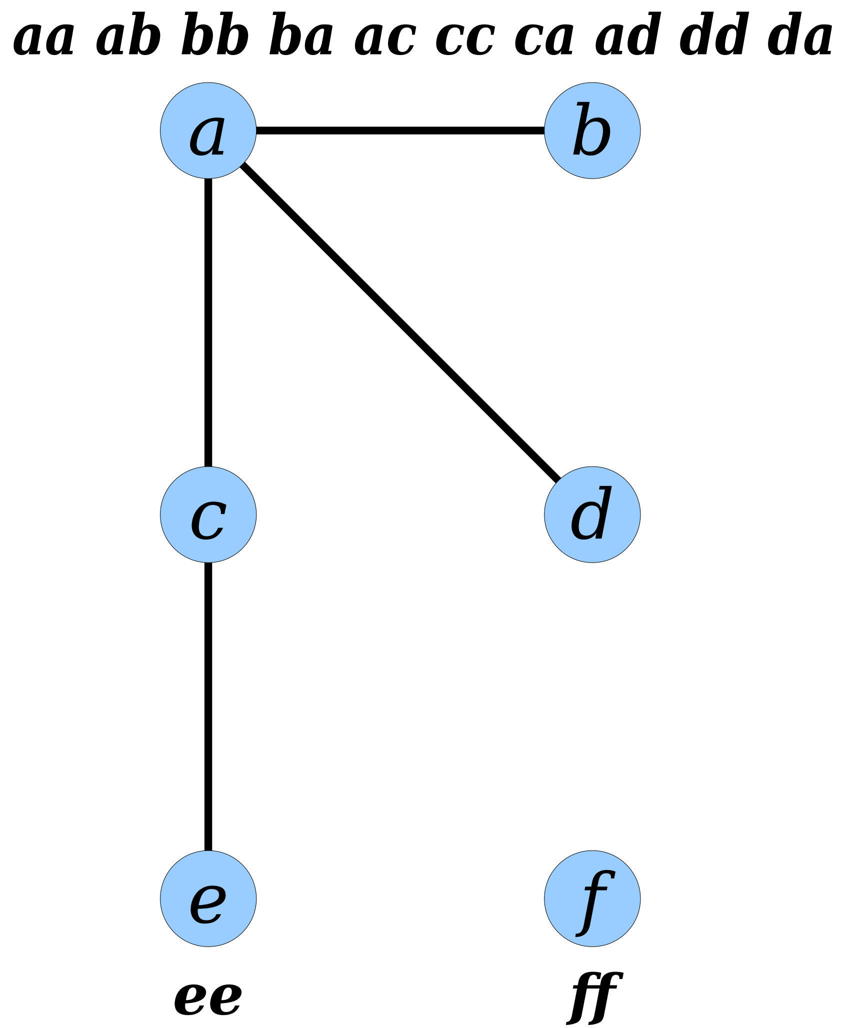
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as T_x xy T_y yx.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as T_1 T_3 and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



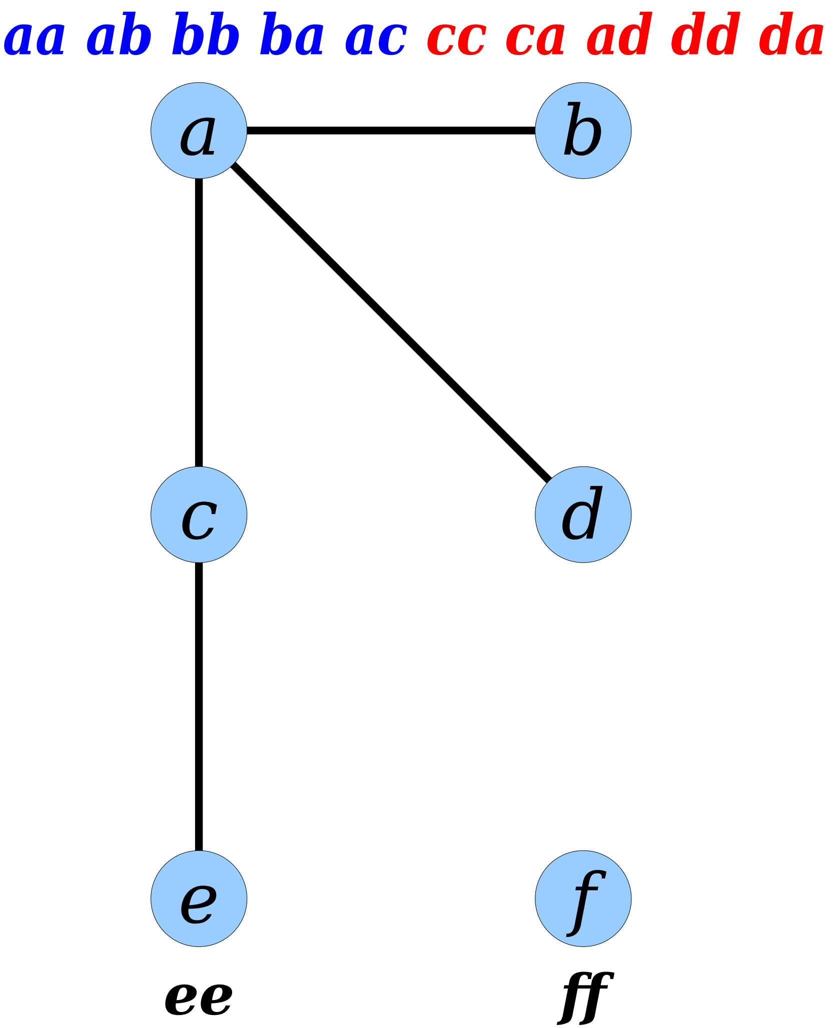
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



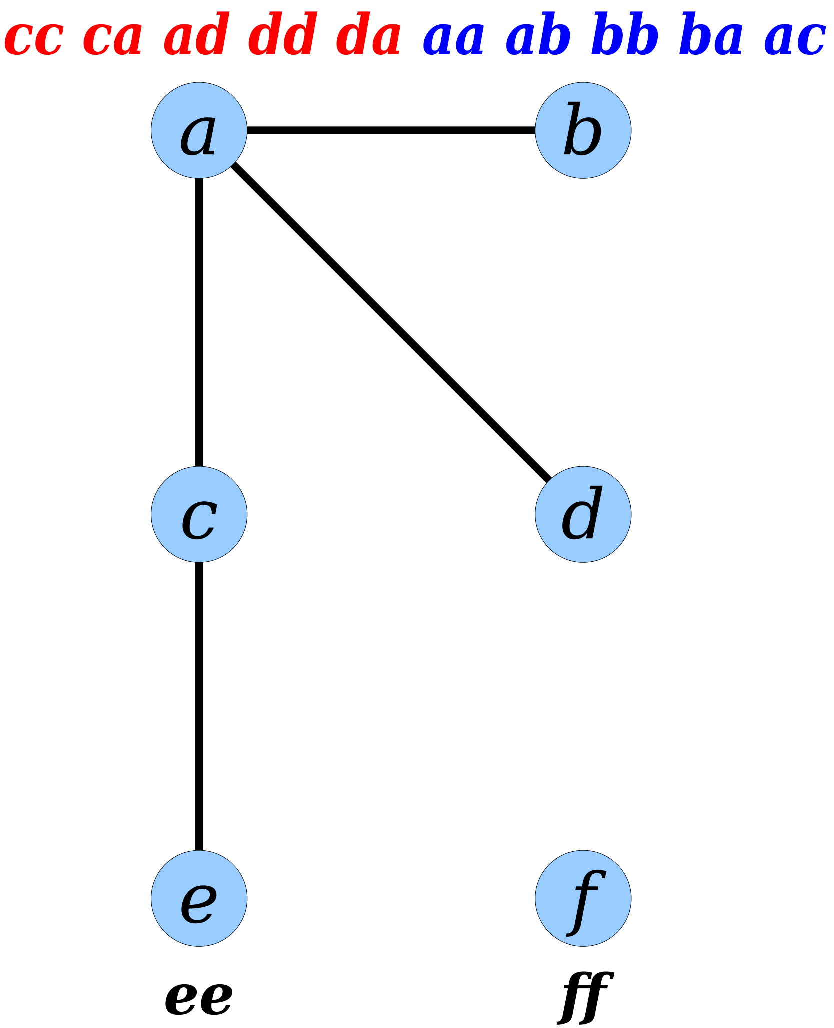
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



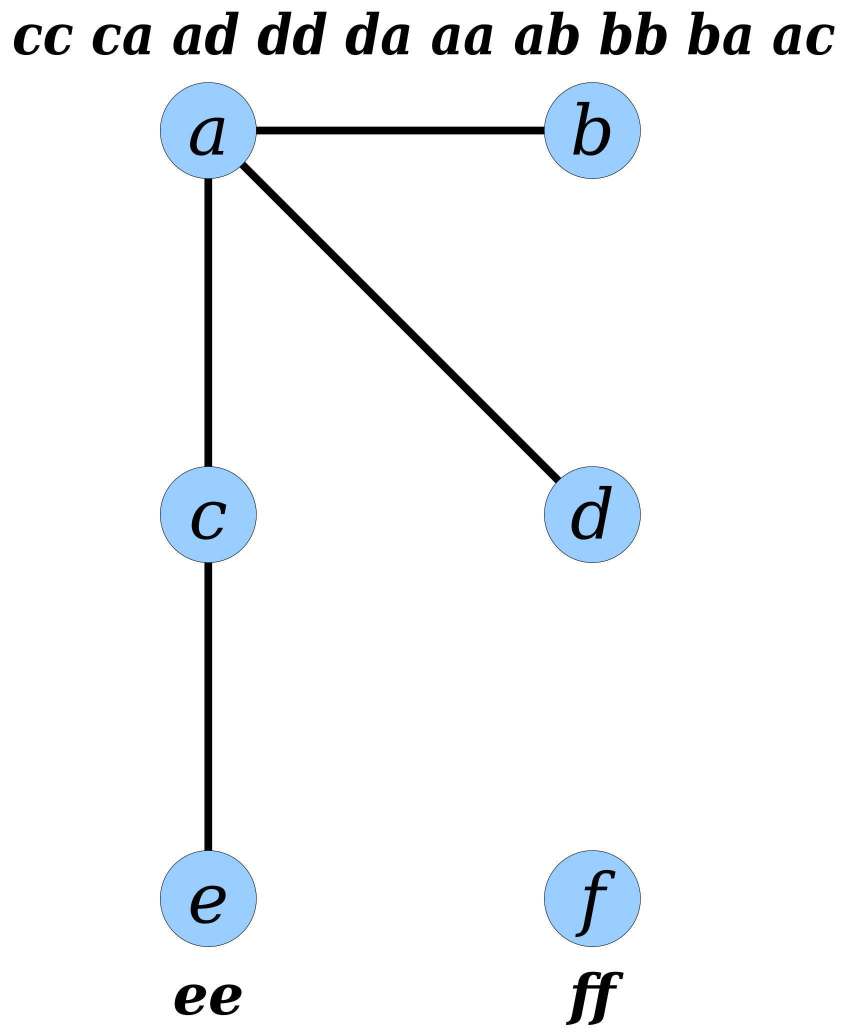
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



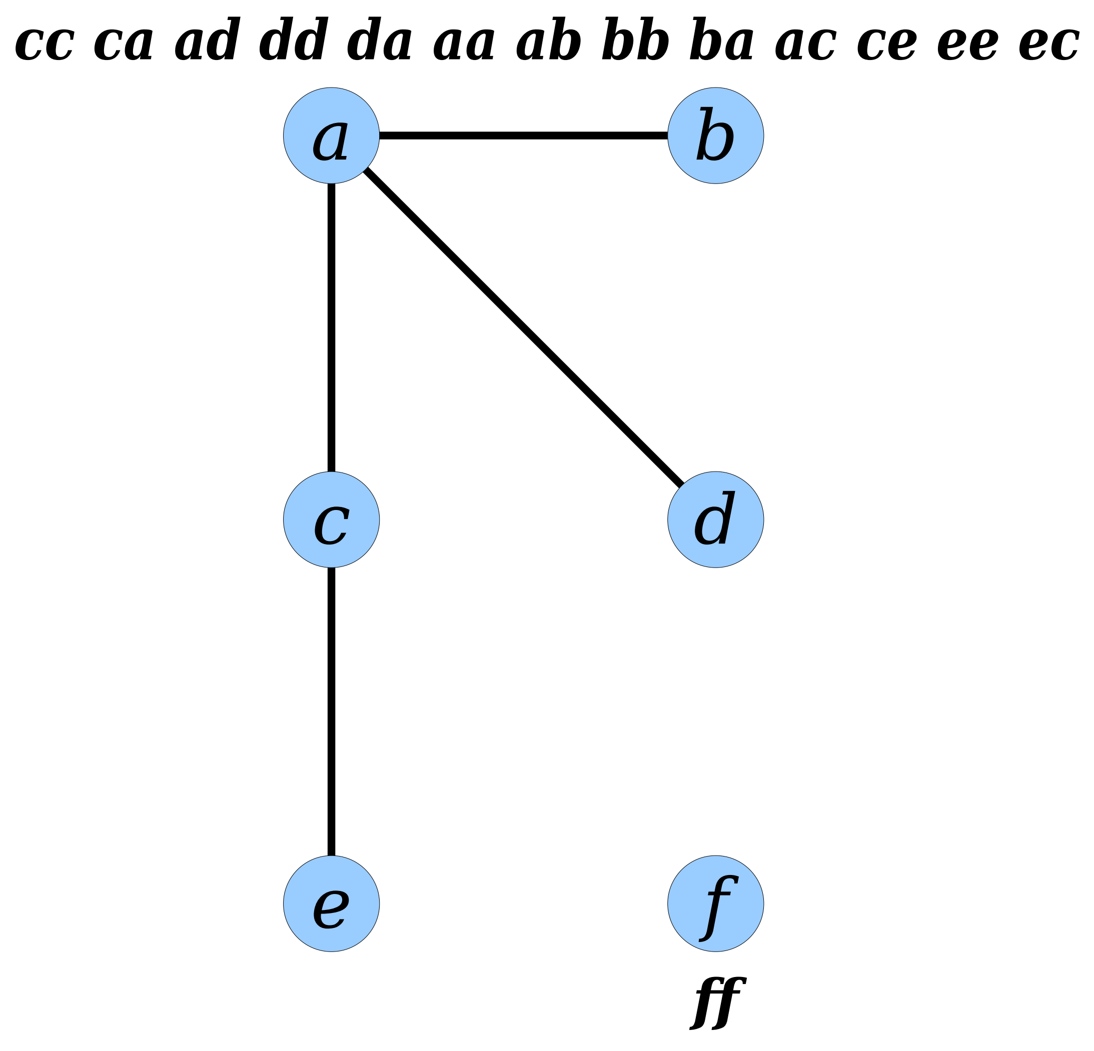
Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



Putting It All Together

- To ***link***(x, y):
  - Rotate xx and yy to the fronts of their tours T_x and T_y.
  - Join the tours together as $T_x\ xy\ T_y\ yx$.
- To ***cut***(x, y):
  - Delete the edges xy and yx from the tour T to form tours T_1, T_2, T_3.
  - Regroup the tours as $T_1\ T_3$ and T_2.
- To answer ***are-connected***(x, y):
  - Determine whether xx and yy are in the same tour.



Implementing This Approach

The Story So Far

- We’ve seen how to implement ***reroot***, ***link***, ***cut***, and ***are-connected*** in terms of operations on Euler tours.
- The efficiency of those operations depend on how we choose to encode our sequences.
- ***Question:*** What data structure should we use to store those sequences?

Representation Issues

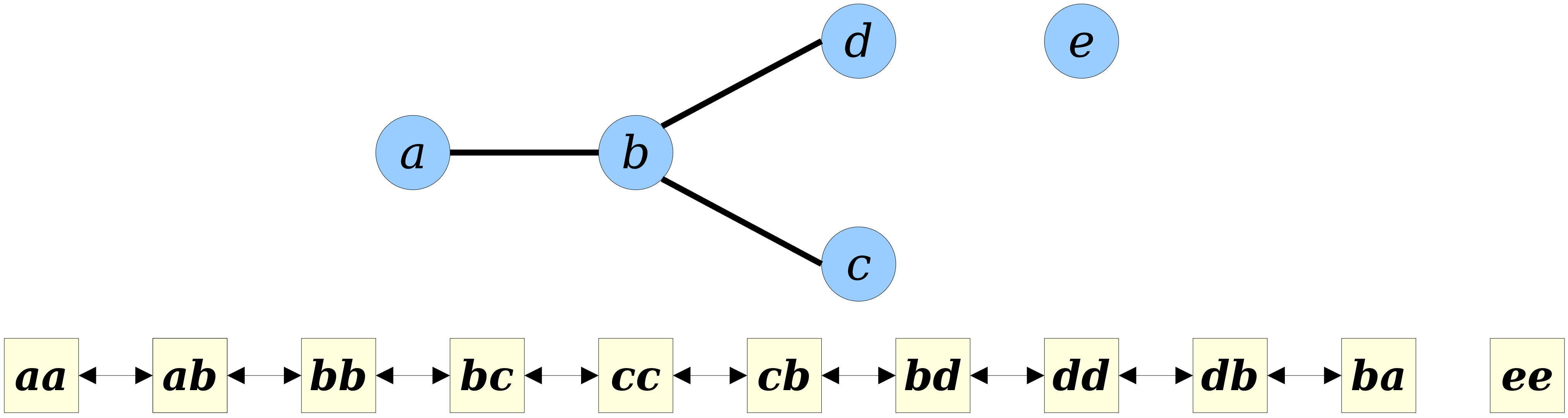
- We need a representation that lets us perform the following operations:
  - Locate specific edges (***reroot***, ***link***, ***cut***, ***are-connected***).
  - Split a sequence at a point (***reroot***, ***cut***).
  - Join two sequences together (***reroot***, ***link***).
  - Remove an edge from a sequence (***cut***).
  - Append an edge to a sequence (***link***).
  - Check if two edges are in the same sequence (***are-connected***).
- What data structures might be appropriate here?

Answer at

https://pollev.com/cs166spr23

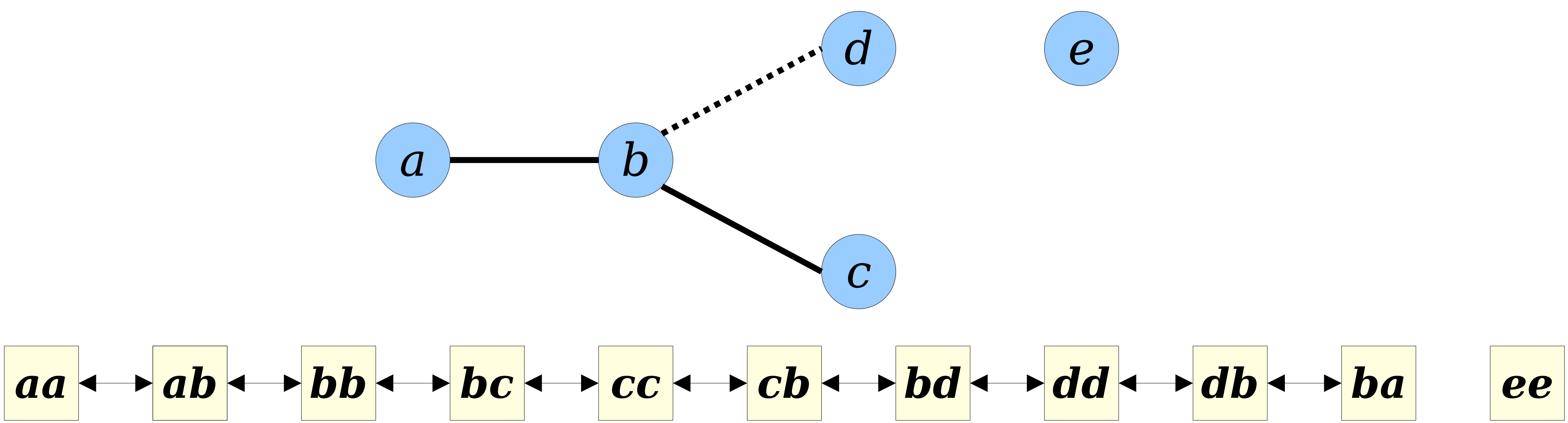
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



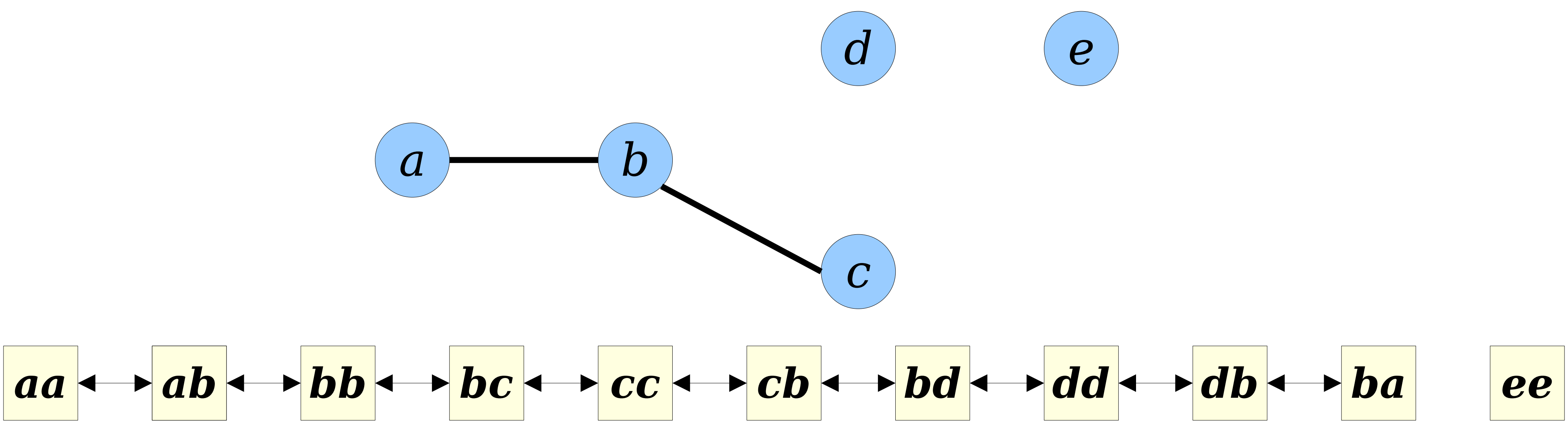
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



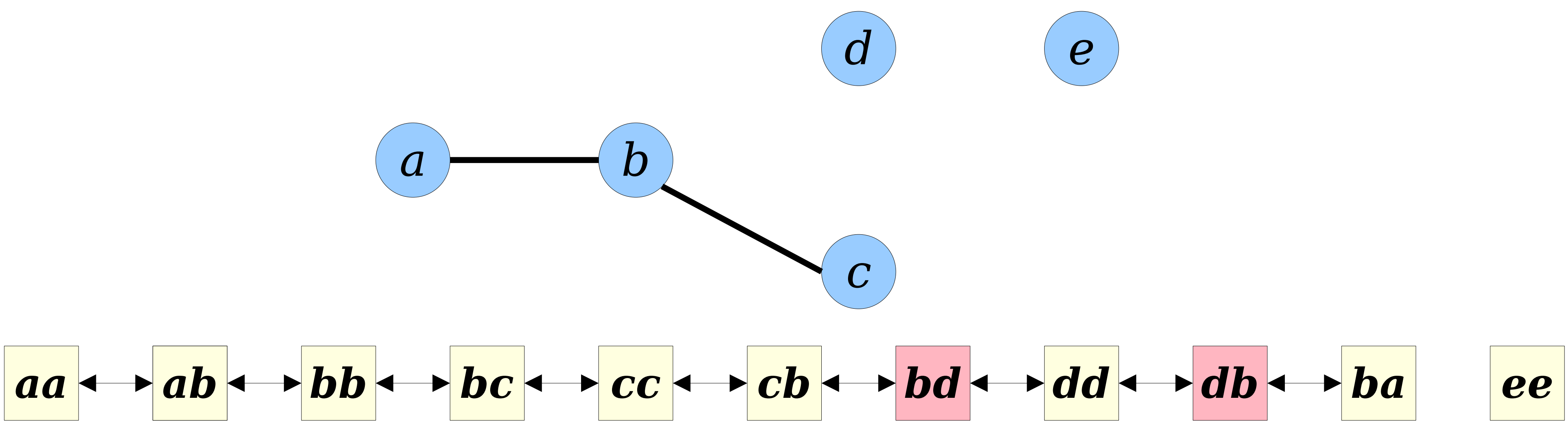
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



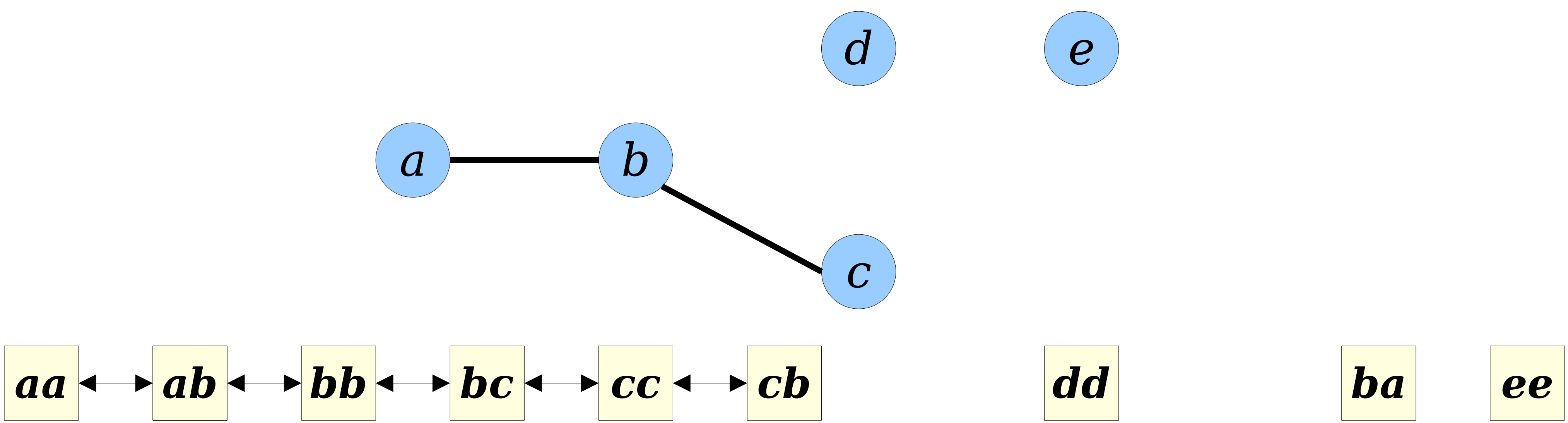
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



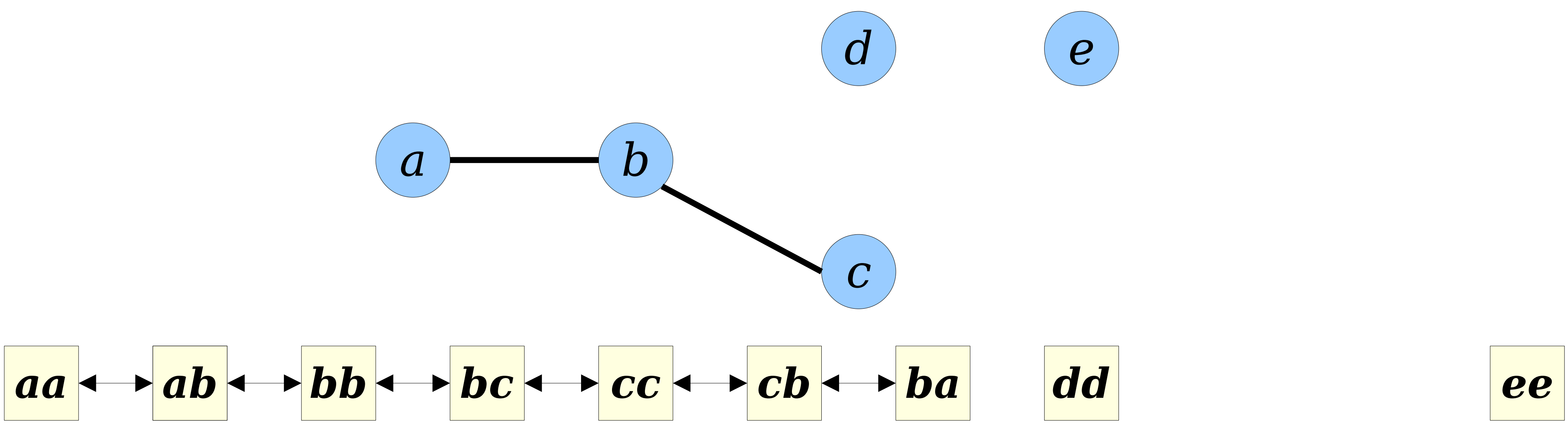
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



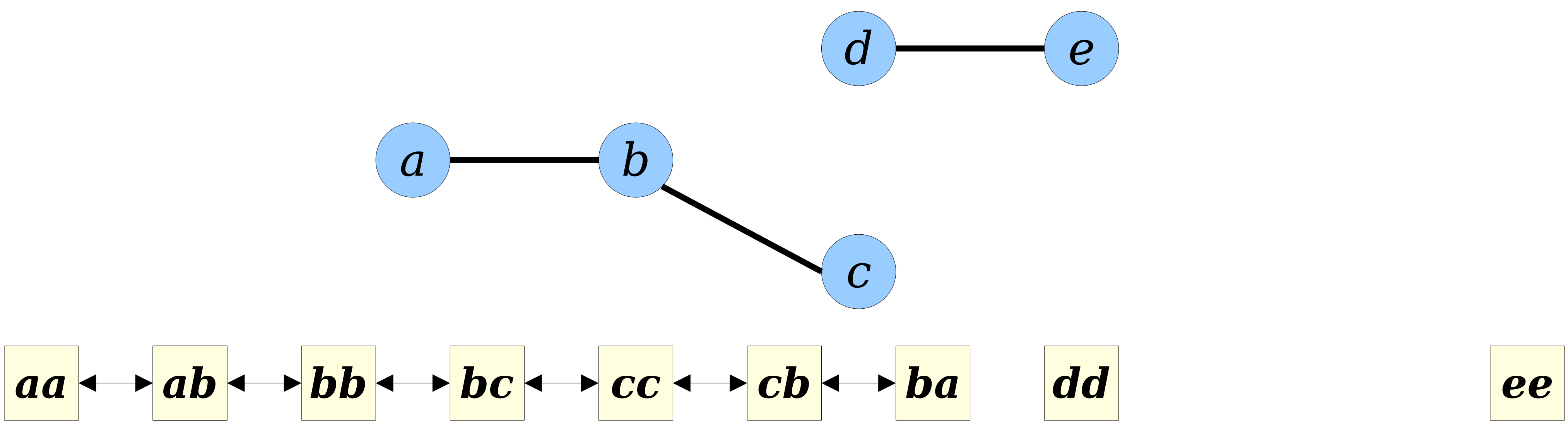
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



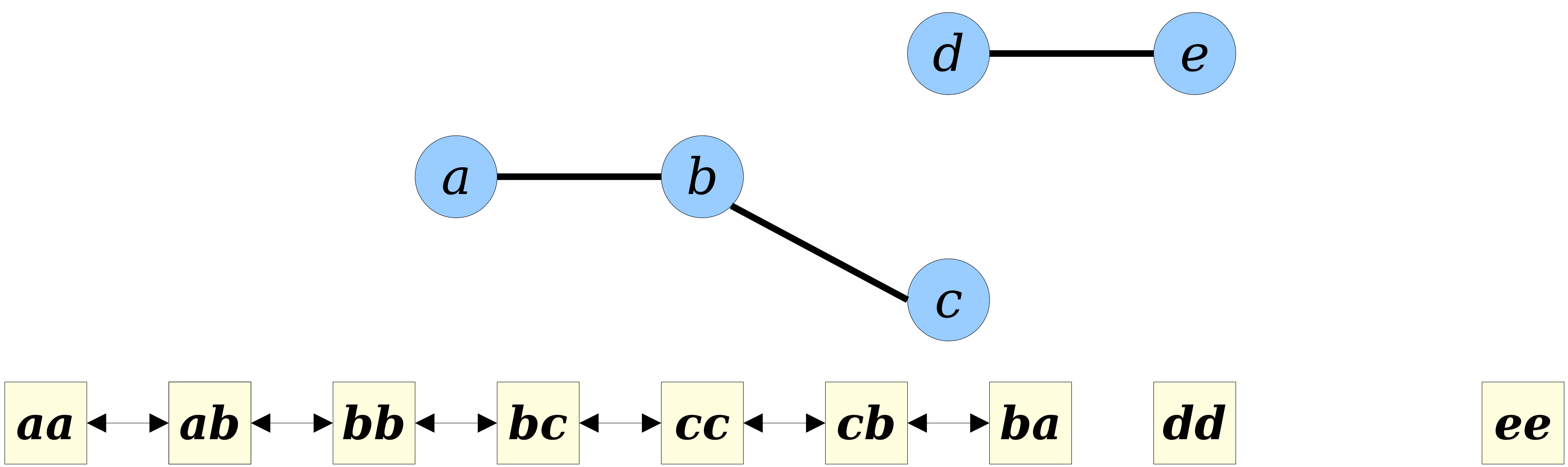
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



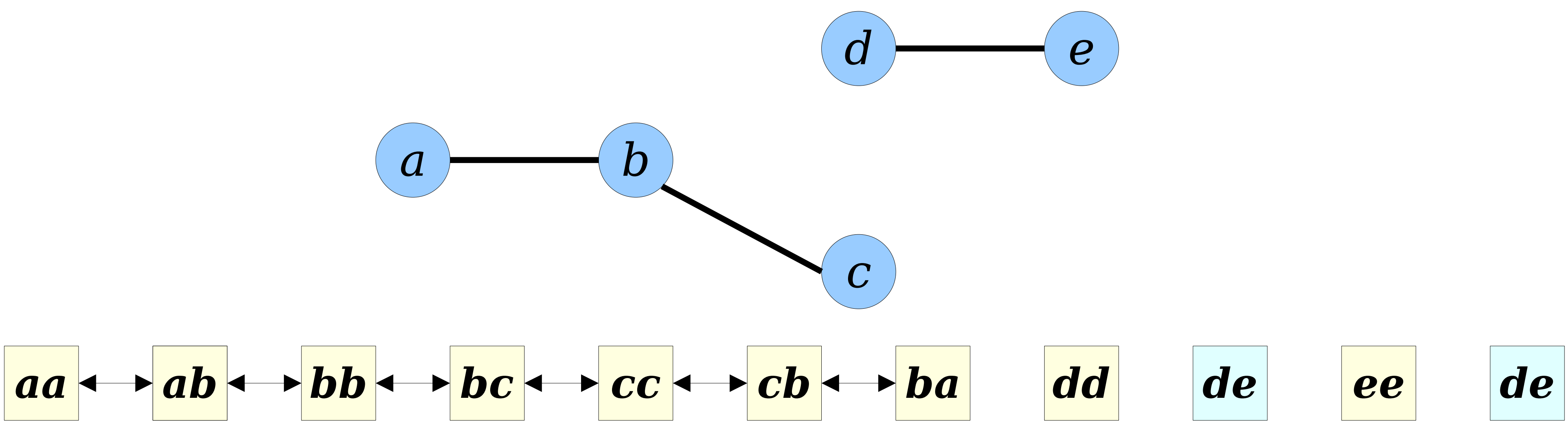
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



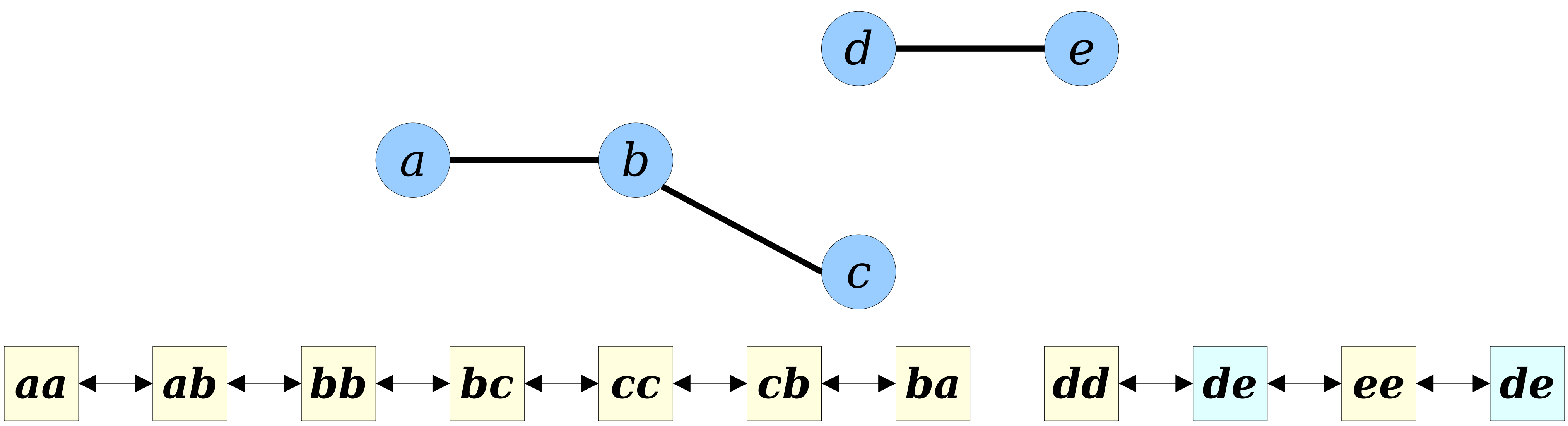
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



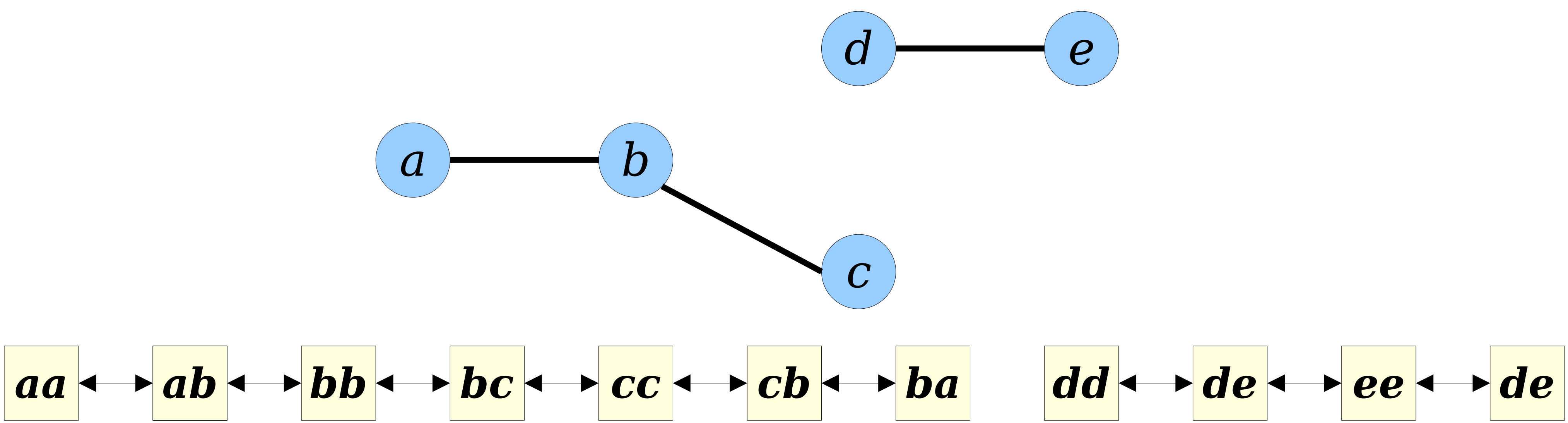
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



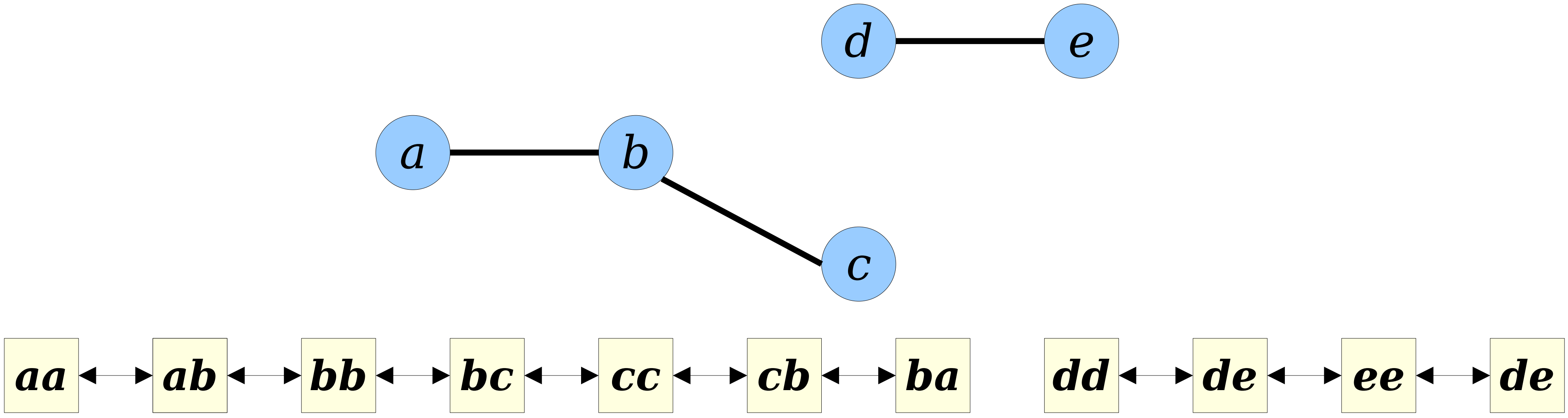
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).



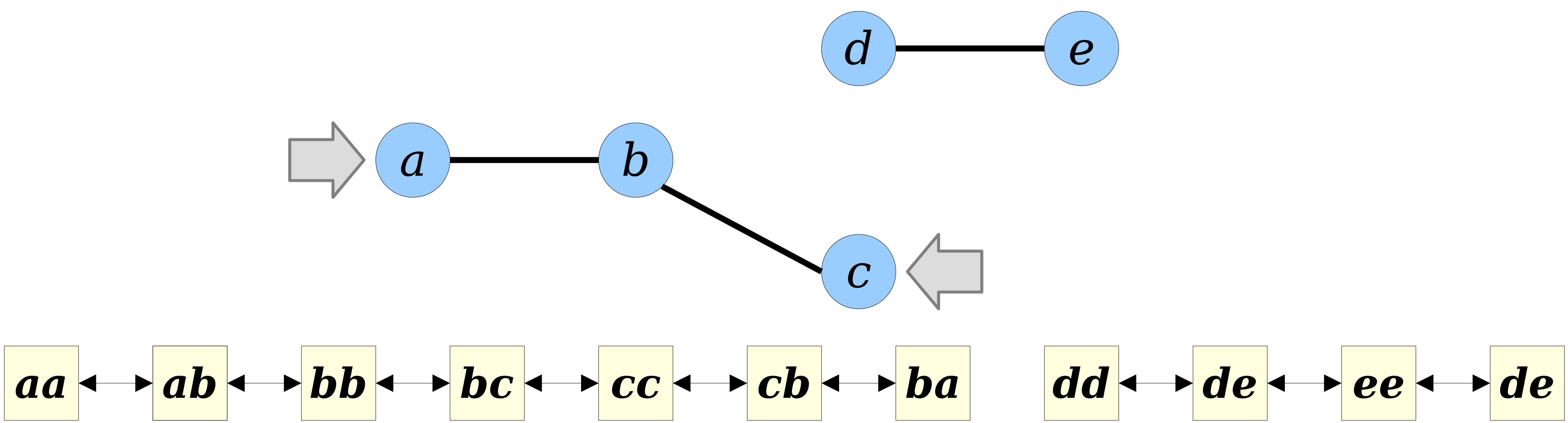
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
    - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).
- ***Problem:*** There isn't an easy way to test whether two nodes are in the same tour. Scanning within the linked list make take time $\Theta(n)$.



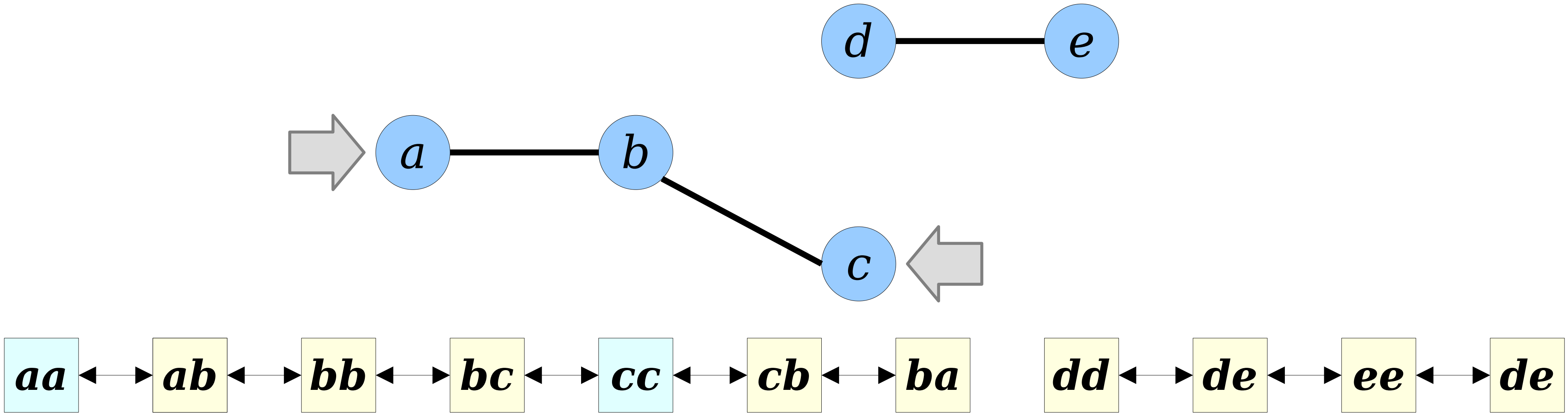
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).
- ***Problem:*** There isn't an easy way to test whether two nodes are in the same tour. Scanning within the linked list make take time $\Theta(n)$.



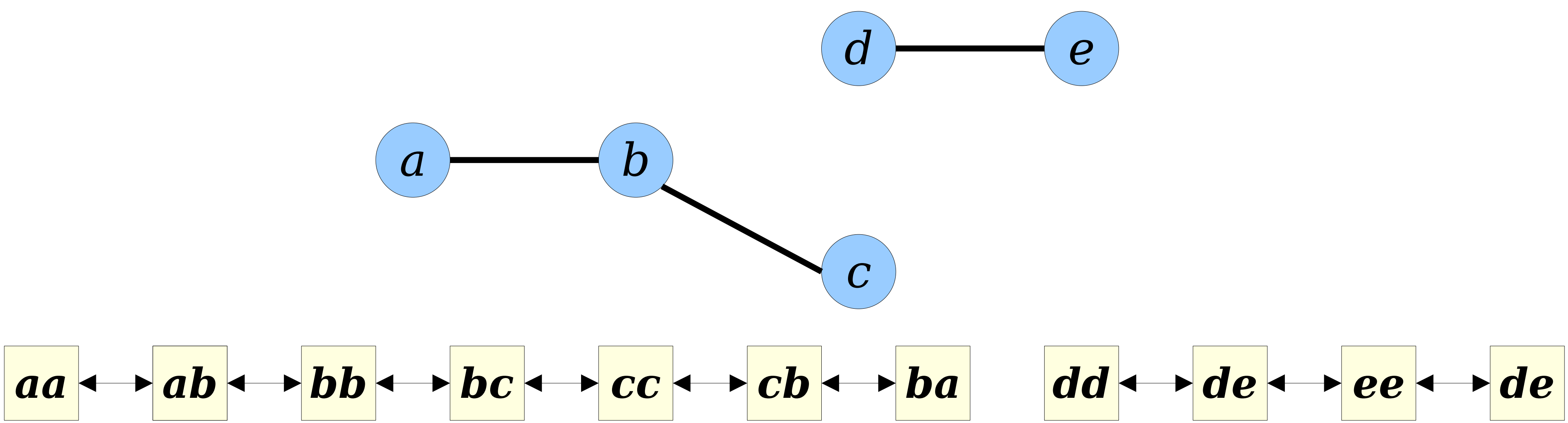
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).
- ***Problem:*** There isn't an easy way to test whether two nodes are in the same tour. Scanning within the linked list make take time $\Theta(n)$.



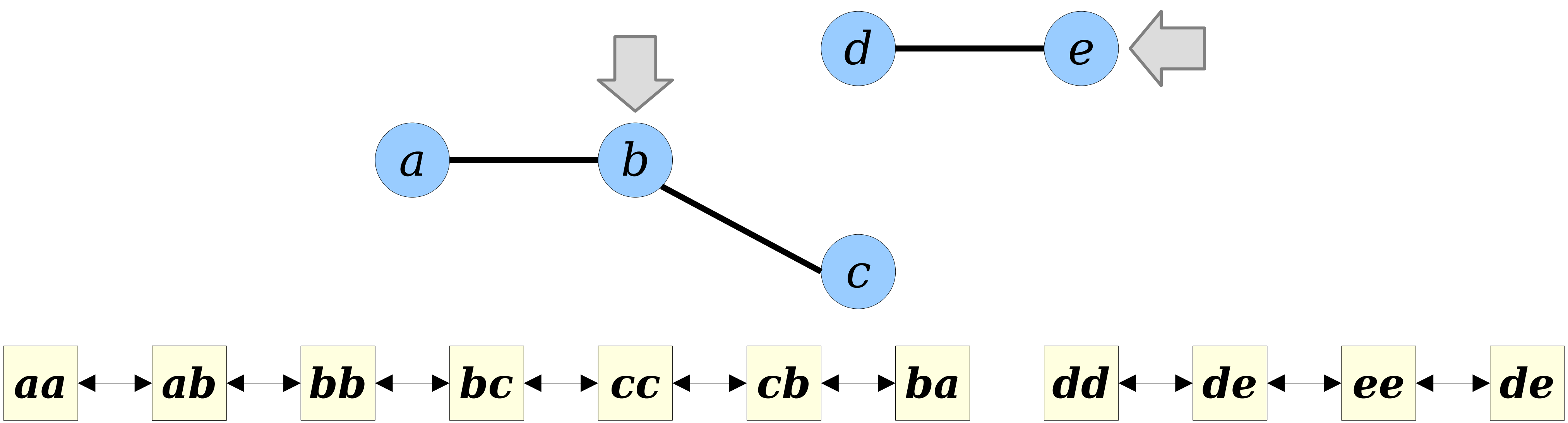
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).
- ***Problem:*** There isn't an easy way to test whether two nodes are in the same tour. Scanning within the linked list make take time $\Theta(n)$.



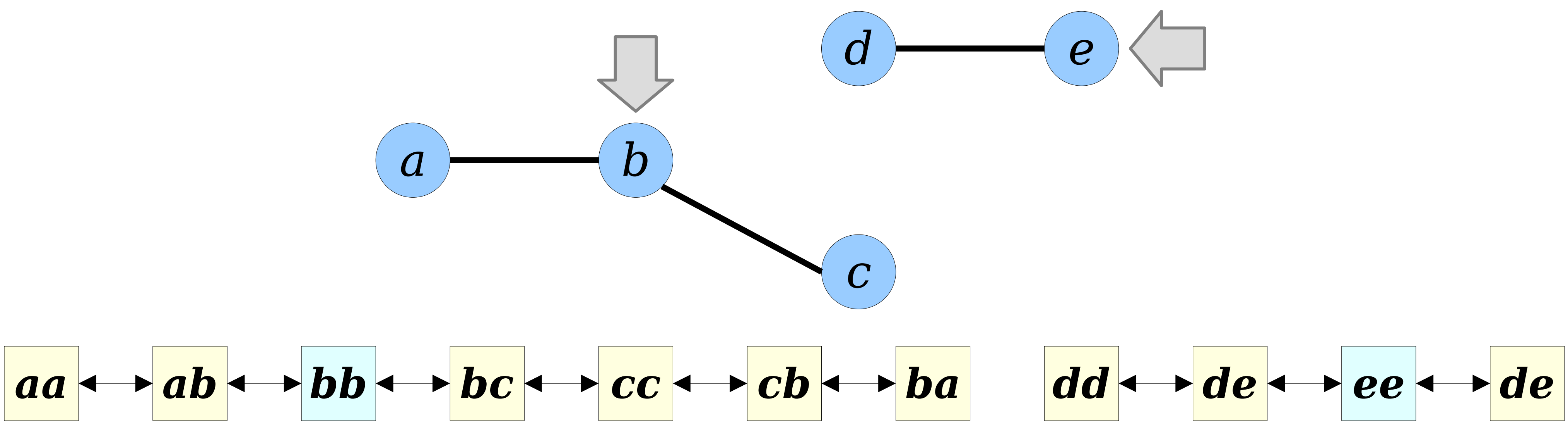
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).
- ***Problem:*** There isn't an easy way to test whether two nodes are in the same tour. Scanning within the linked list make take time Θ(n).



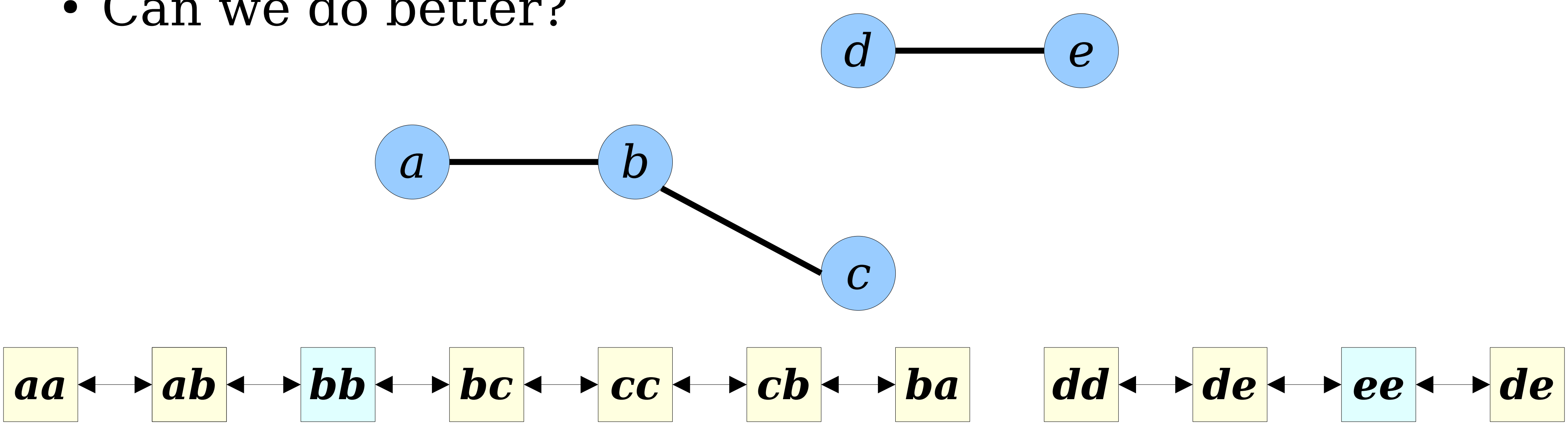
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).
- ***Problem:*** There isn't an easy way to test whether two nodes are in the same tour. Scanning within the linked list make take time $\Theta(n)$.



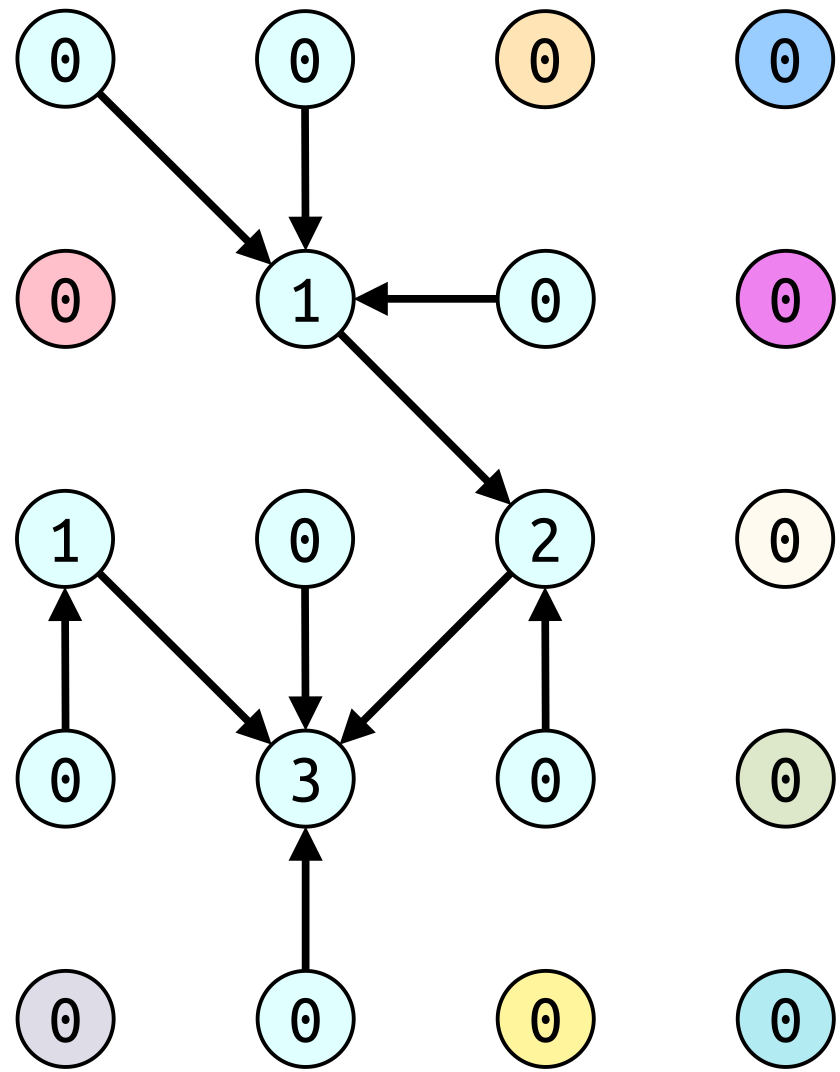
Representation Issues

- ***Idea 1:*** Use doubly-linked lists, plus an auxiliary hash table / BST to locate edges.
  - Assuming we have a hash table telling us where edges are, we can split, join, and rotate tours in time O(1).
- ***Problem:*** There isn't an easy way to test whether two nodes are in the same tour. Scanning within the linked list make take time $\Theta(n)$.
- Can we do better?



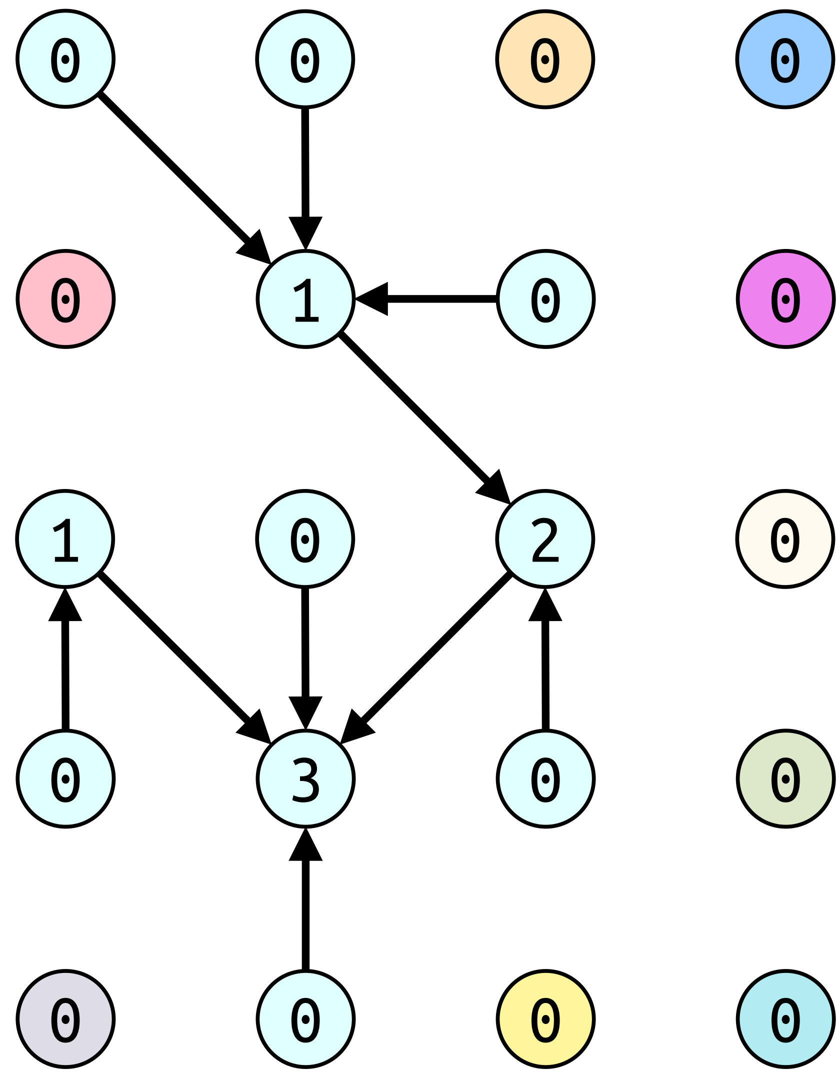
Representation Issues

- In incremental connectivity, we selected a representative for each CC.
- We then had elements store parent pointers that formed a path to the representative.
- Could we do something like that here?



Representation Issues

- The idea of using trees to store representatives is a good one.
  - If the trees are wide and flat, it won't take too long to find the representative.
  - If we don't have to update "too many" pointers when CC's change, our operations can run quickly.
- The trees we used last time won't (immediately) work here.
  - We have to store the elements of the tour in sequential order. There was no such notion of order in disjoint set forests.
  - In disjoint-set forests, linked items can never be cut, allowing for some clever optimizations.
- What's another tree we can use?



Representation Issues

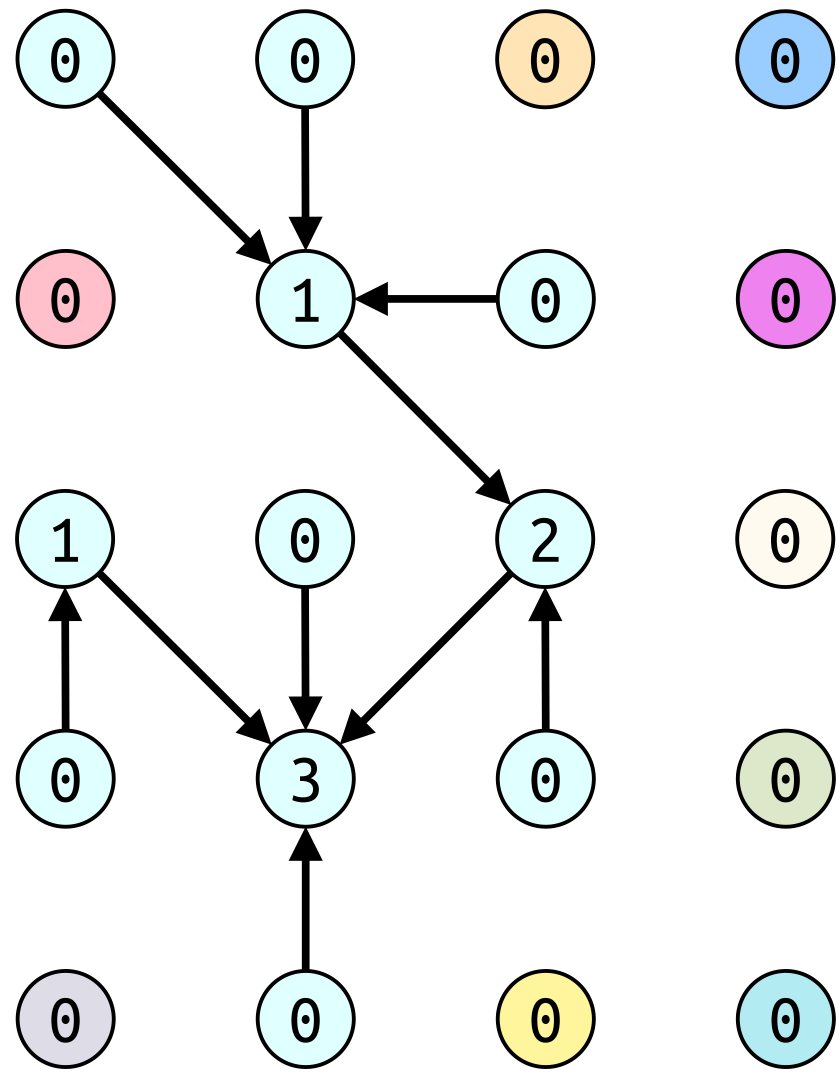
The idea of using trees to store representatives is a good one.

- If the trees are wide and flat, it won't take too long to find the representative.
- If we don't have to update "too many" pointers when CC's change, our operations can run quickly.

The trees we used last time won't (immediately) work here.

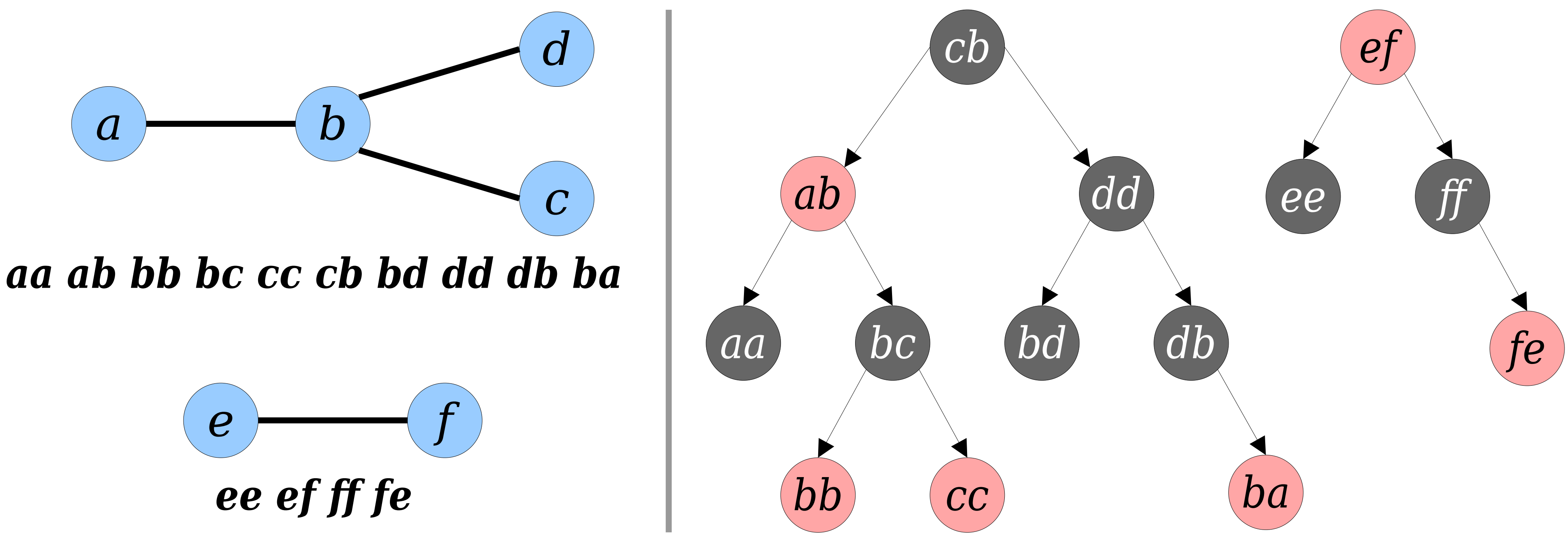
- We have to store the elements of the tour in sequential order. There was no such notion of order in disjoint set forests.
- In disjoint-set forests, linked items can never be cut, allowing for some clever optimizations.

What's another tree we can use?



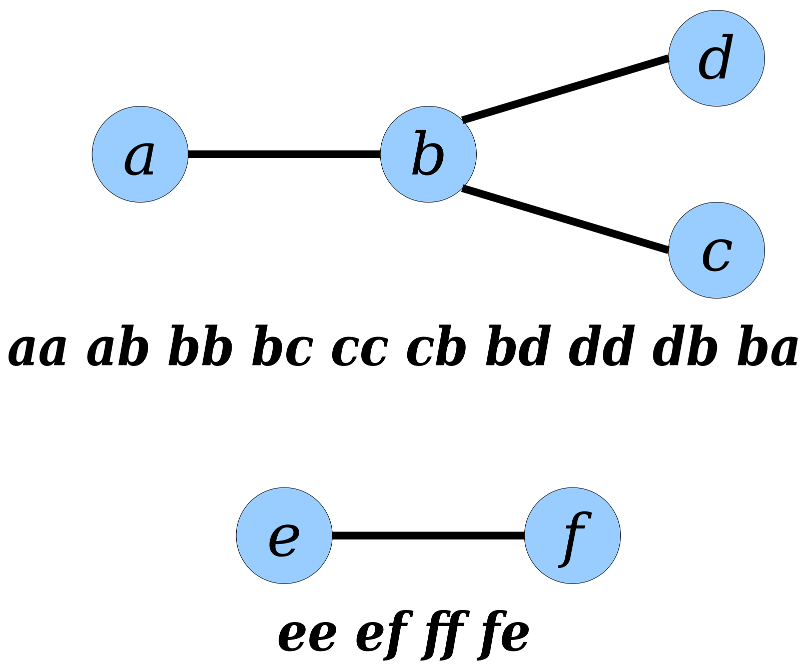
Binary Search(less) Trees

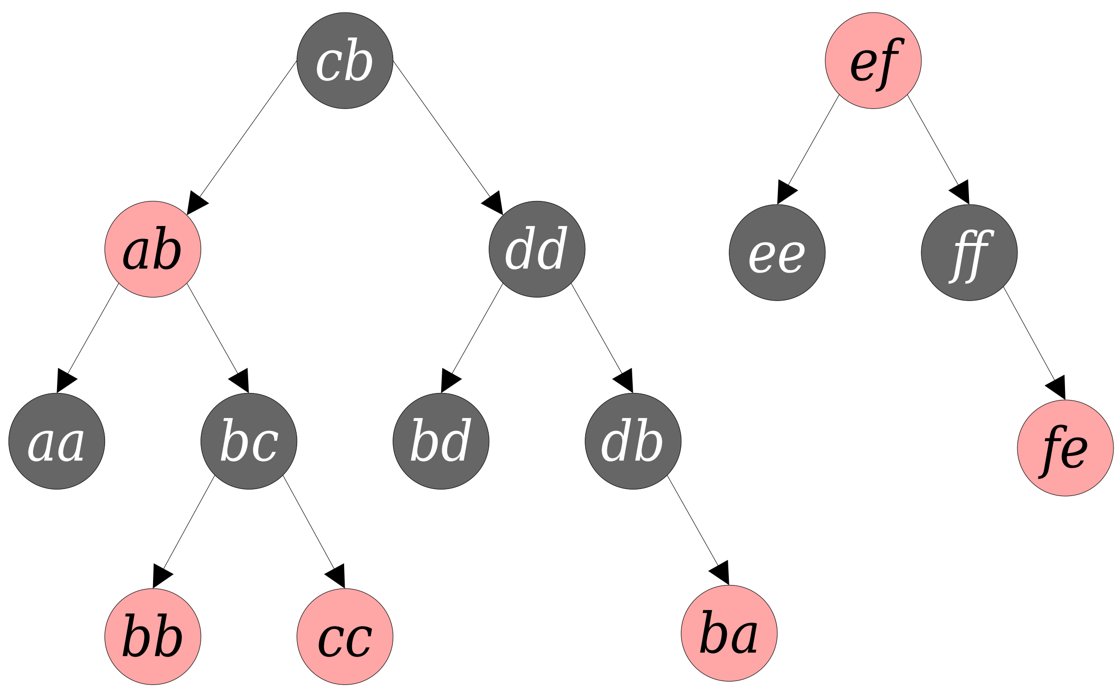
- ***Idea 2:*** Store our sequences in a balanced BST, sorted by their position within the sequence.
- We'll use the *shape* and *algorithm* of a BST, but won't have the ability to conventionally search the tree top-down.
- We'll rely on the fact that we have external pointers that let us jump to items within the BST.



Binary Search(less) Trees

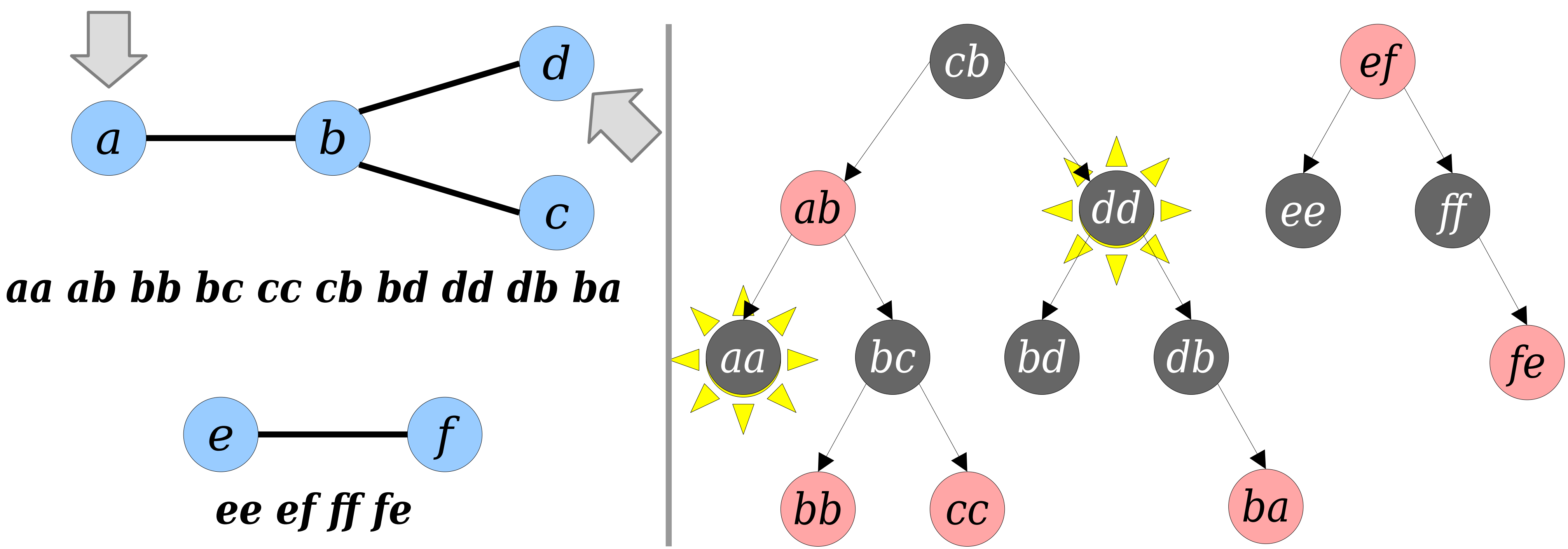
- We can now answer ***are-connected***(x, y) in time O(log n).
  - Find xx and yy using our auxiliary lookup table.
  - Walk up from xx and yy to the roots of their trees.
  - See if they're the same root.





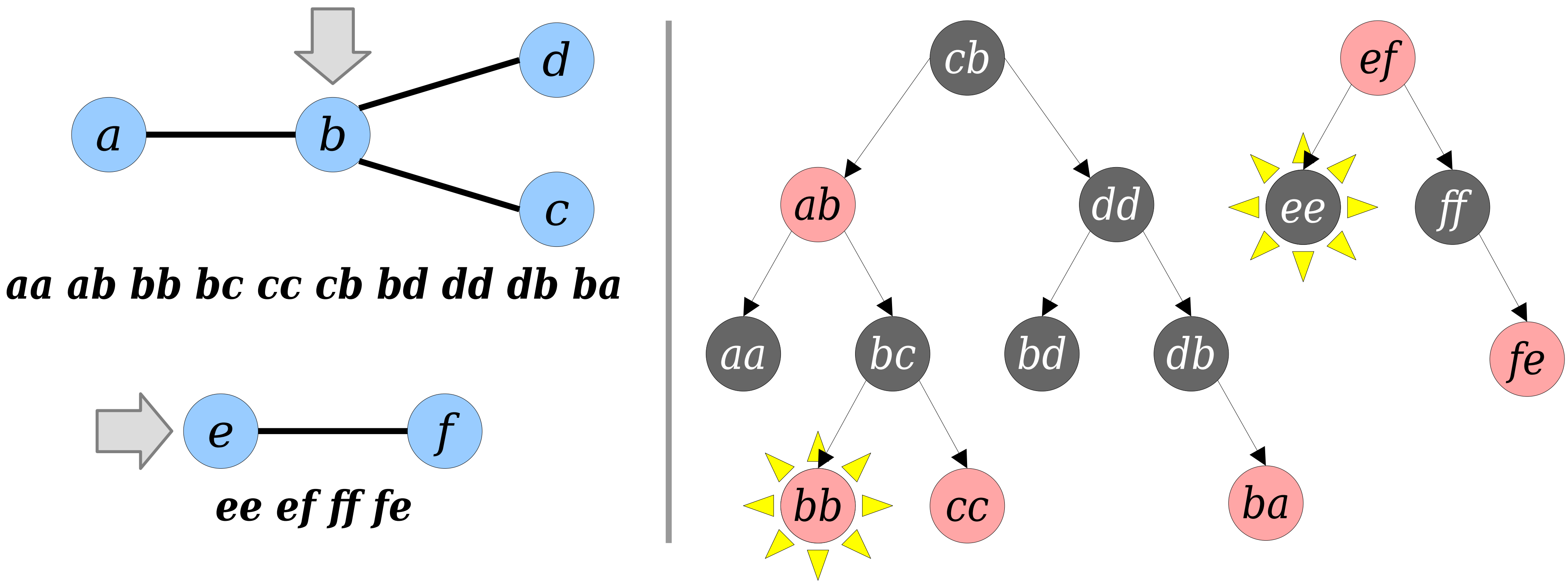
Binary Search(less) Trees

- We can now answer ***are-connected***(x, y) in time O(log n).
  - Find xx and yy using our auxiliary lookup table.
  - Walk up from xx and yy to the roots of their trees.
  - See if they're the same root.



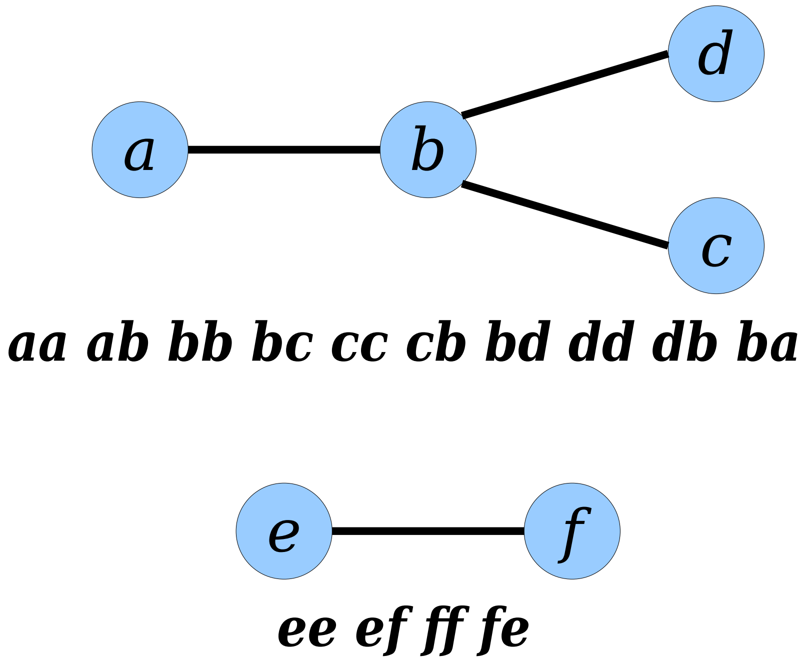
Binary Search(less) Trees

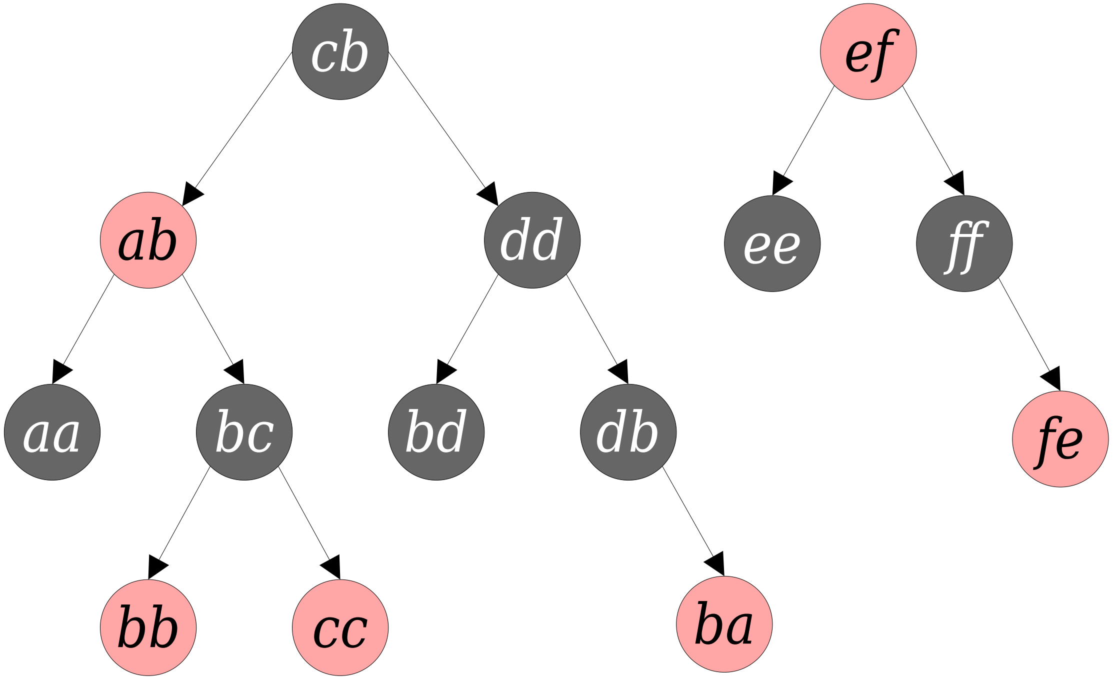
- We can now answer ***are-connected***(x, y) in time O(log n).
  - Find xx and yy using our auxiliary lookup table.
  - Walk up from xx and yy to the roots of their trees.
  - See if they're the same root.



Binary Search(less) Trees

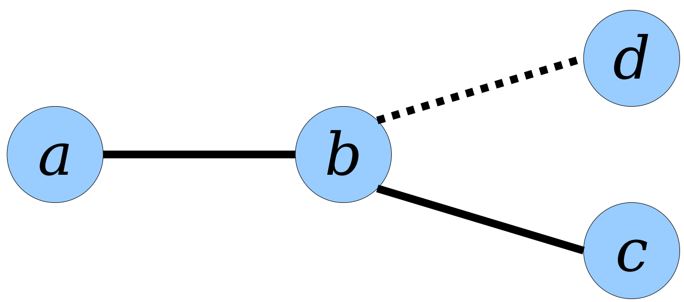
- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.





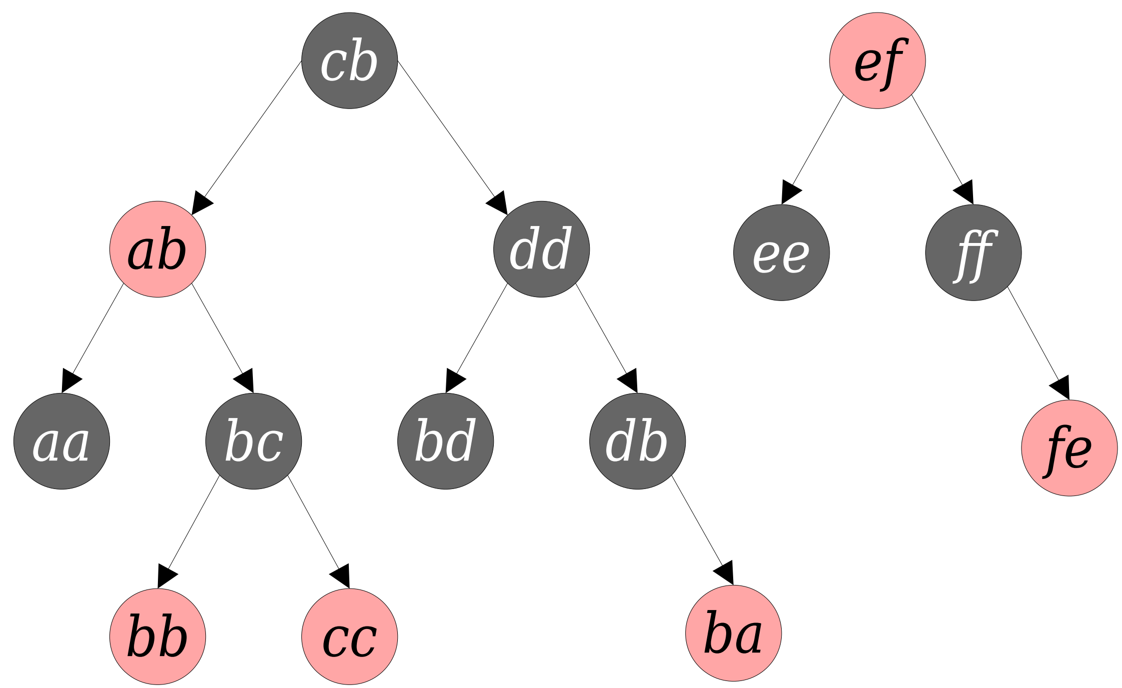
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



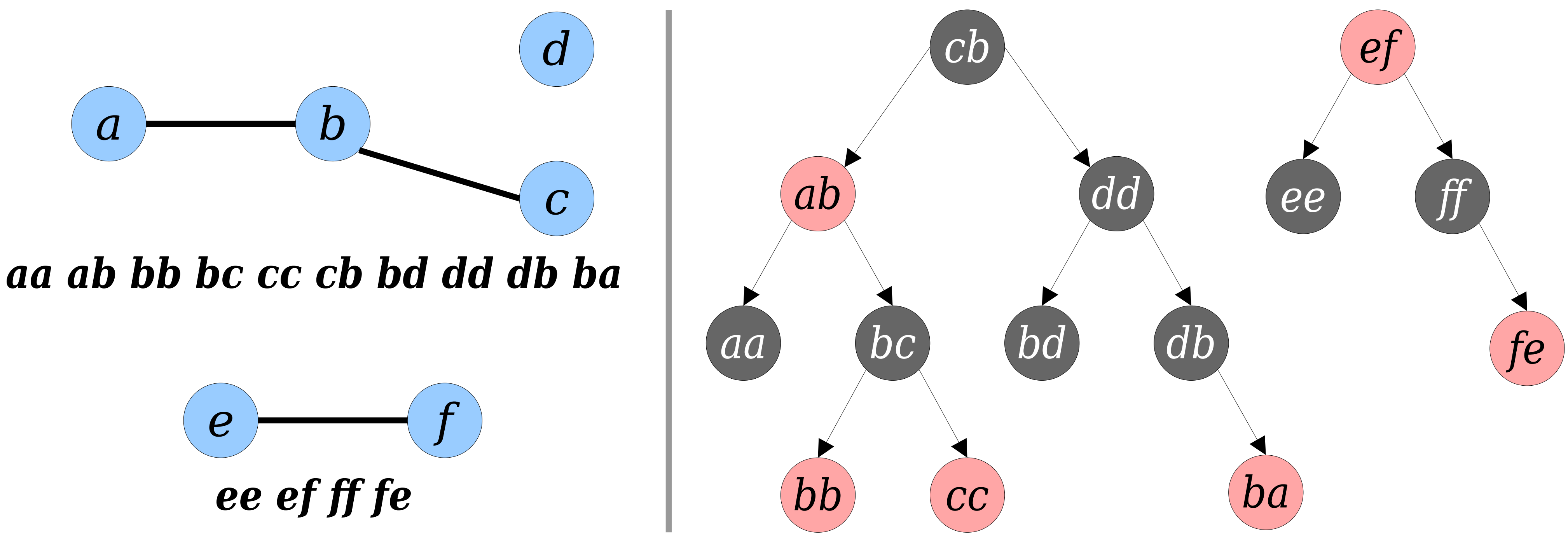
aa ab bb bc cc cb bd dd db ba

ee ef ff fe



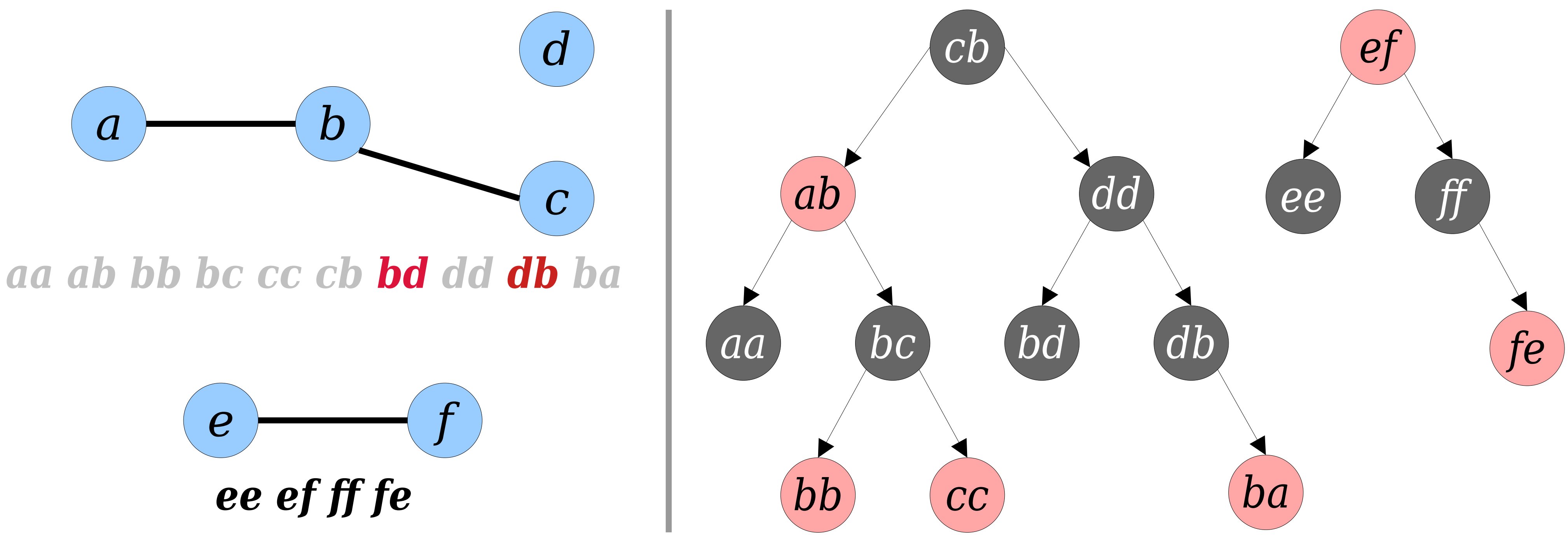
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



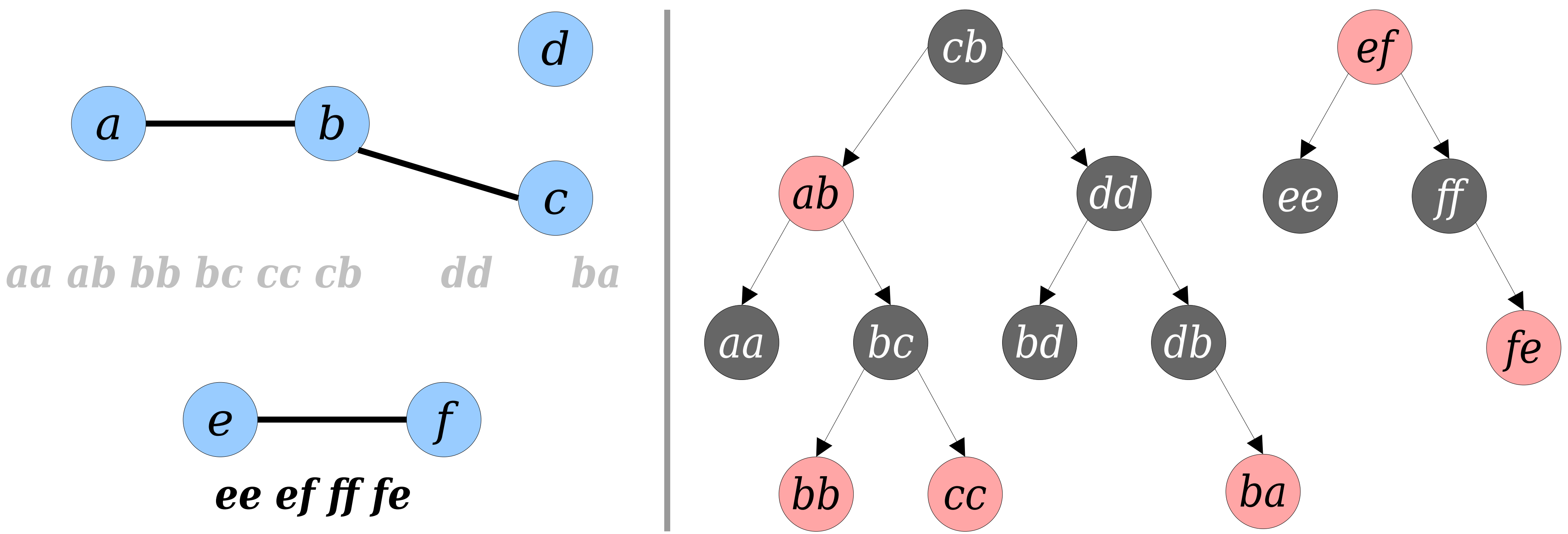
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



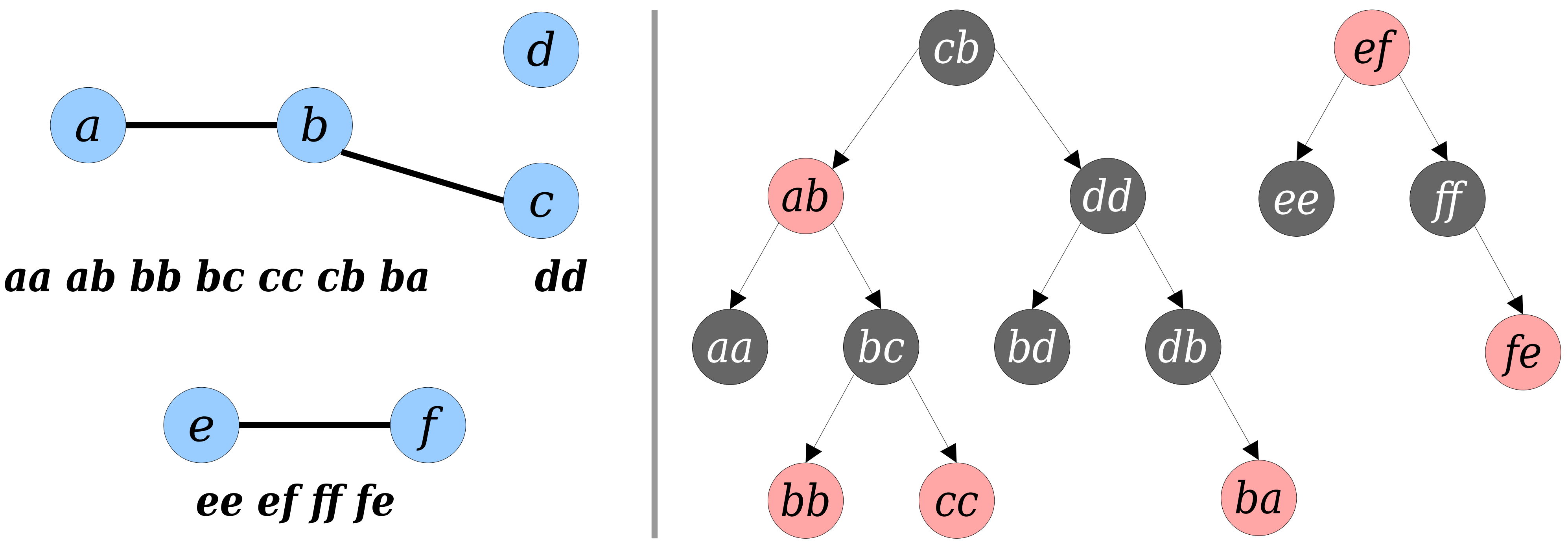
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



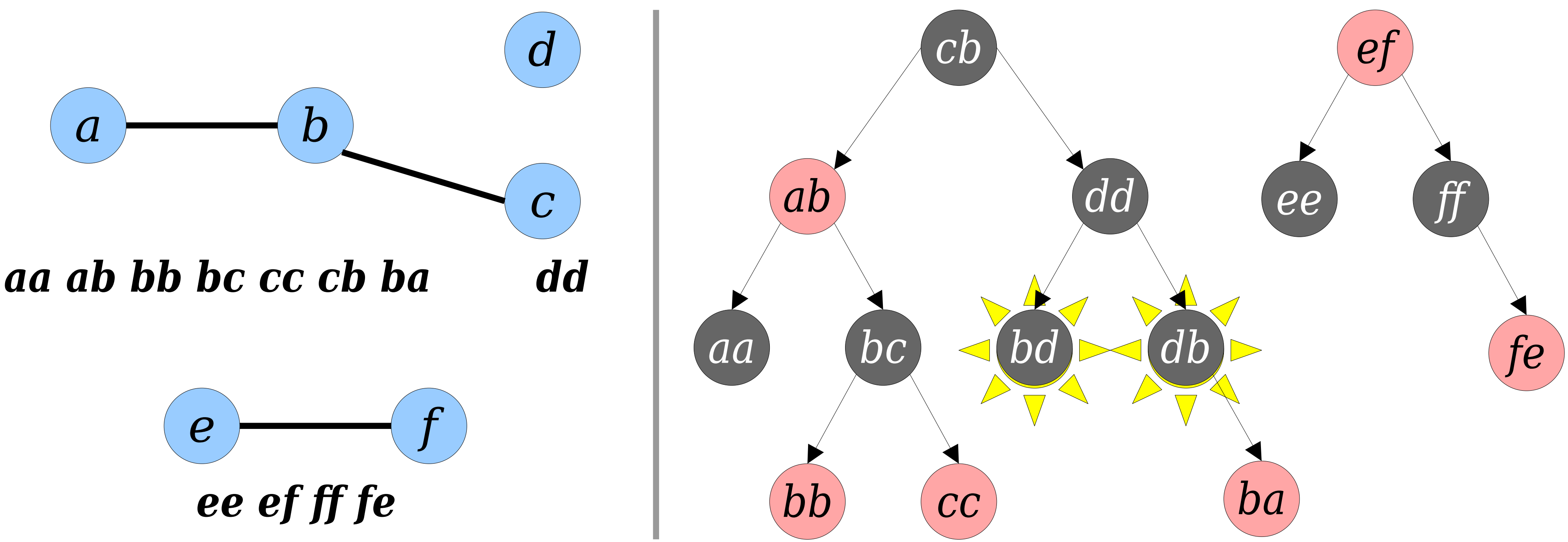
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



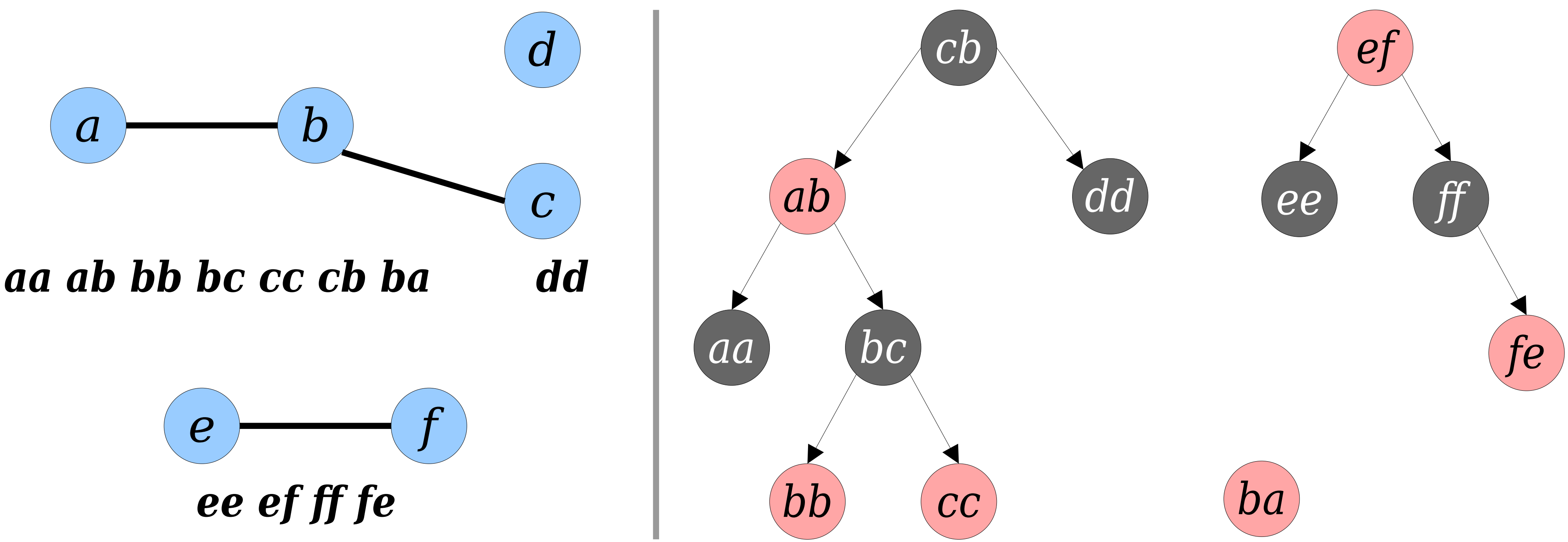
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



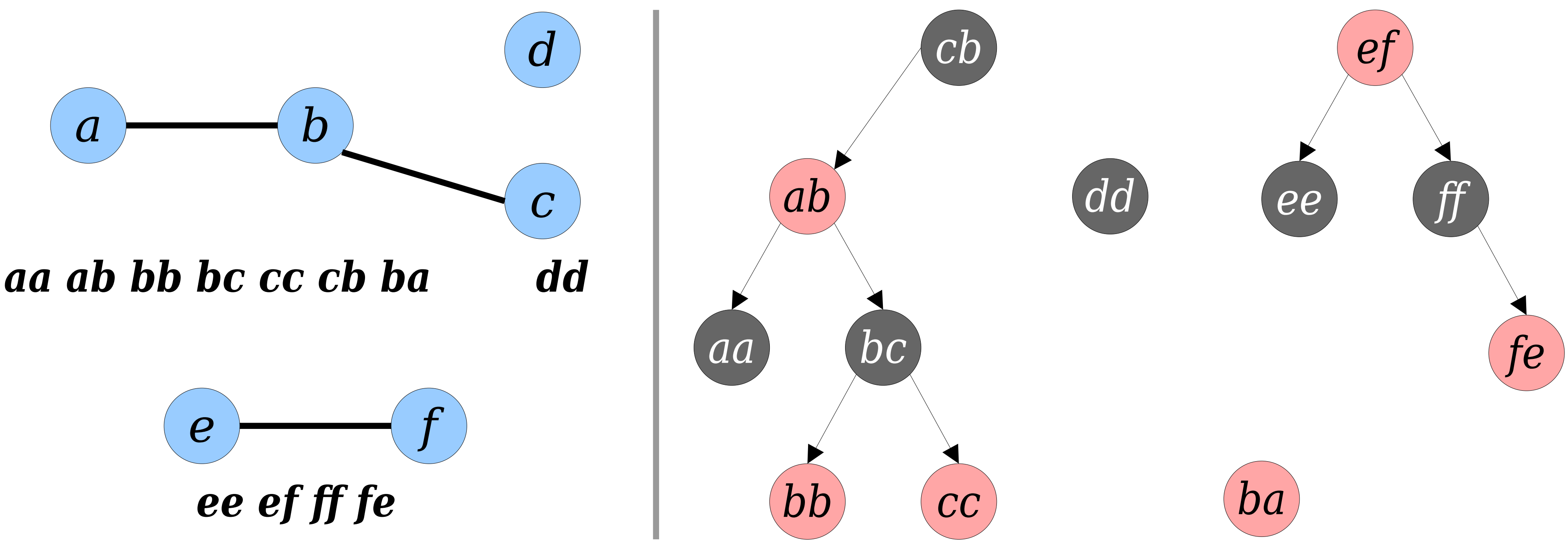
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



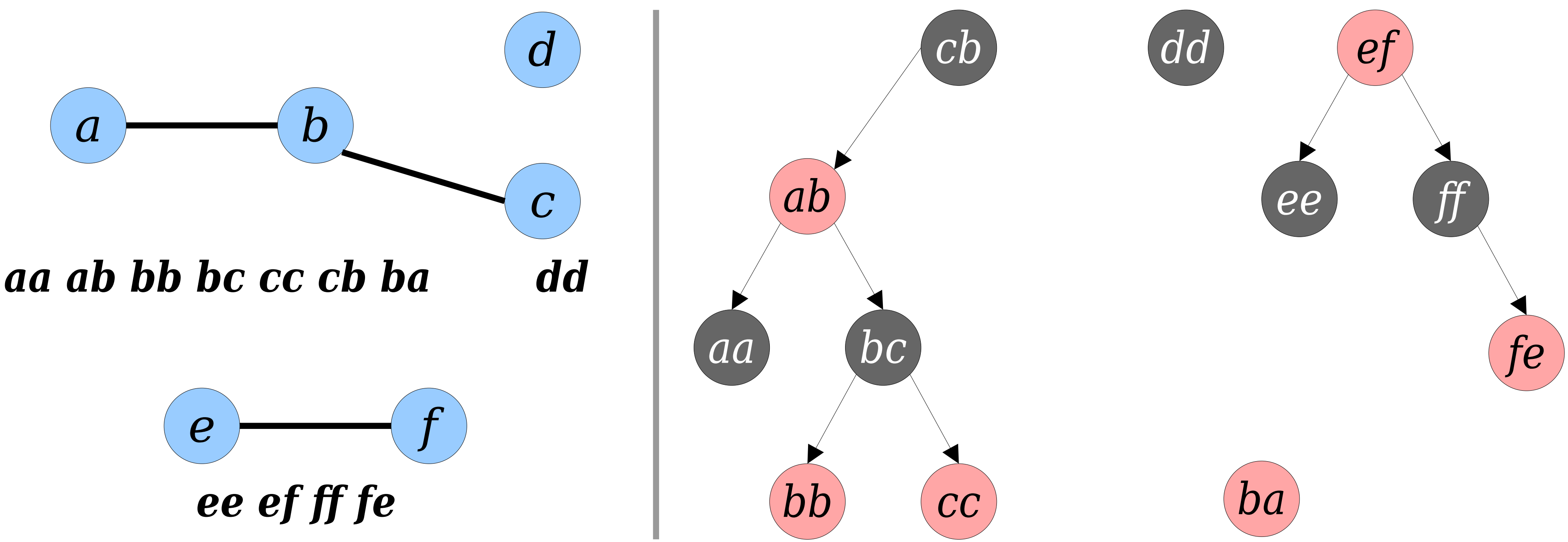
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



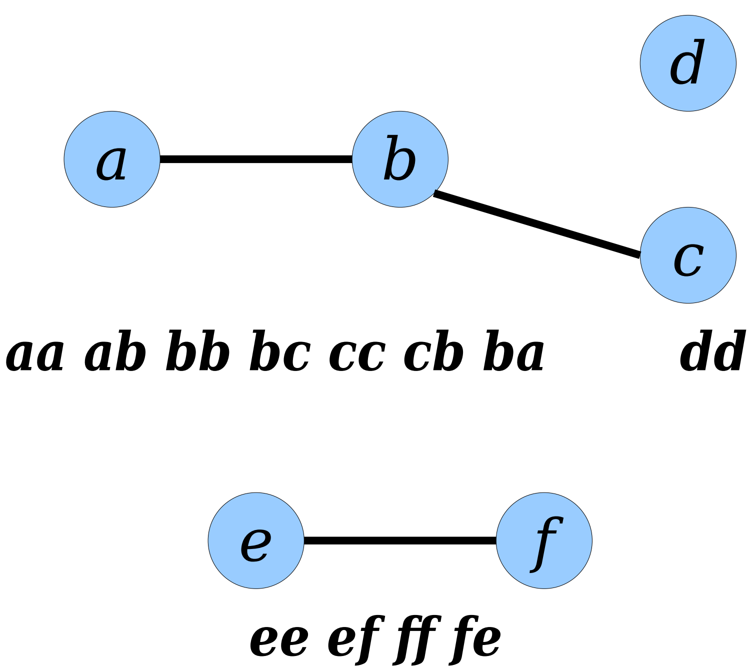
Binary Search(less) Trees

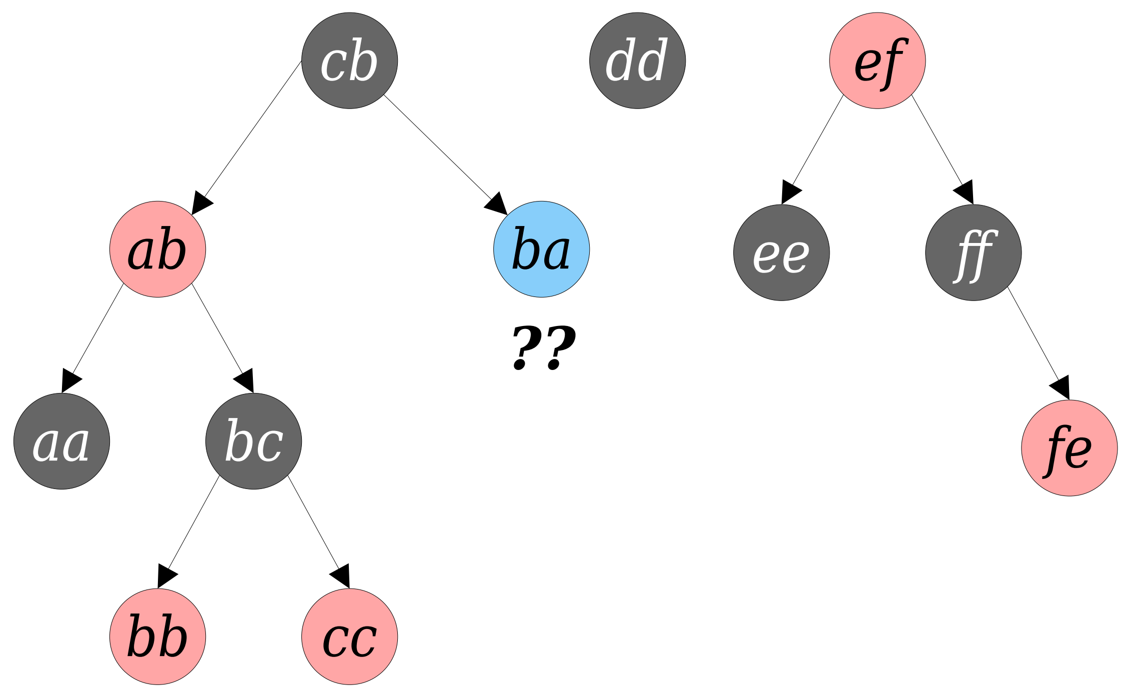
- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



Binary Search(less) Trees

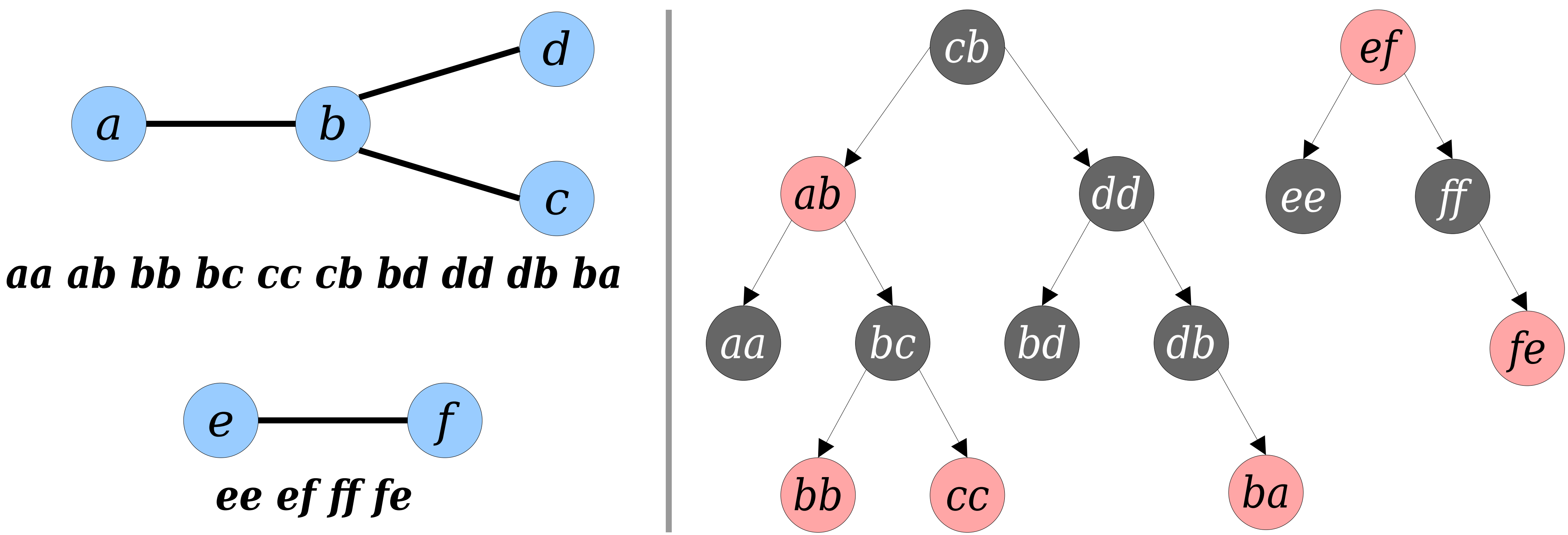
- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.





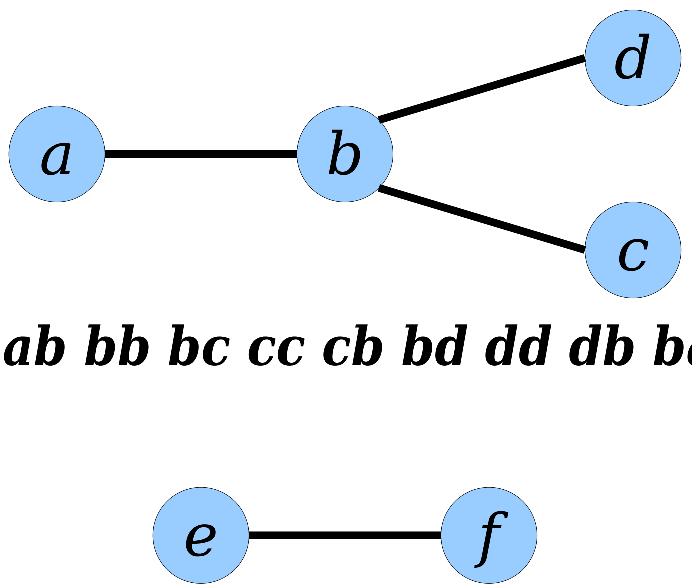
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.



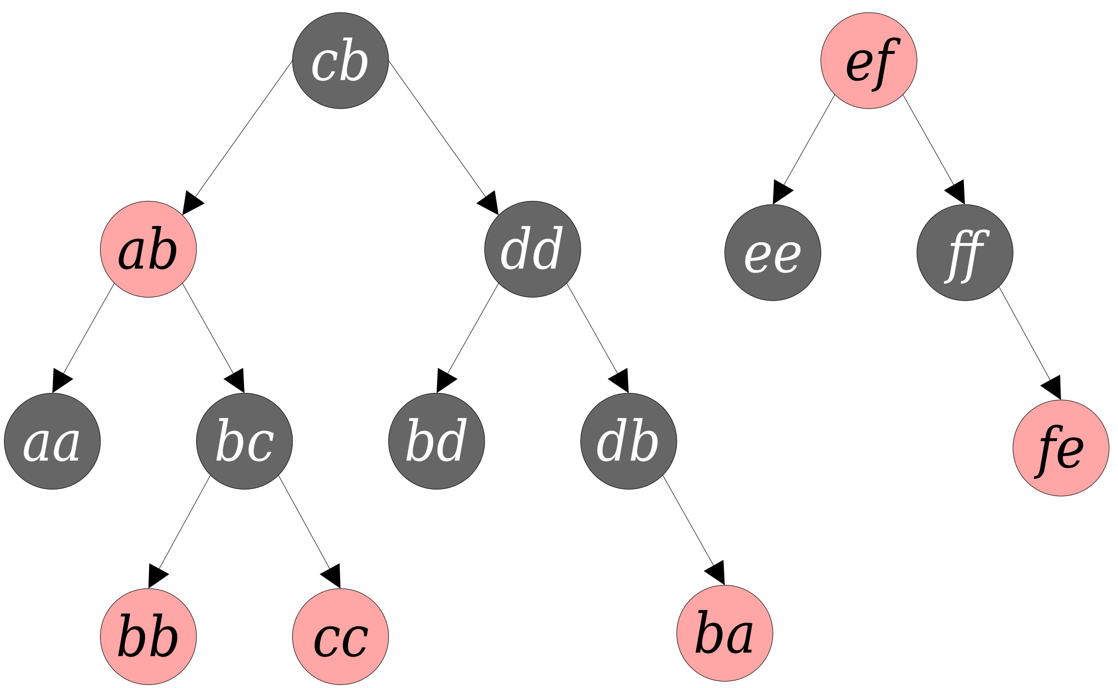
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.
- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).



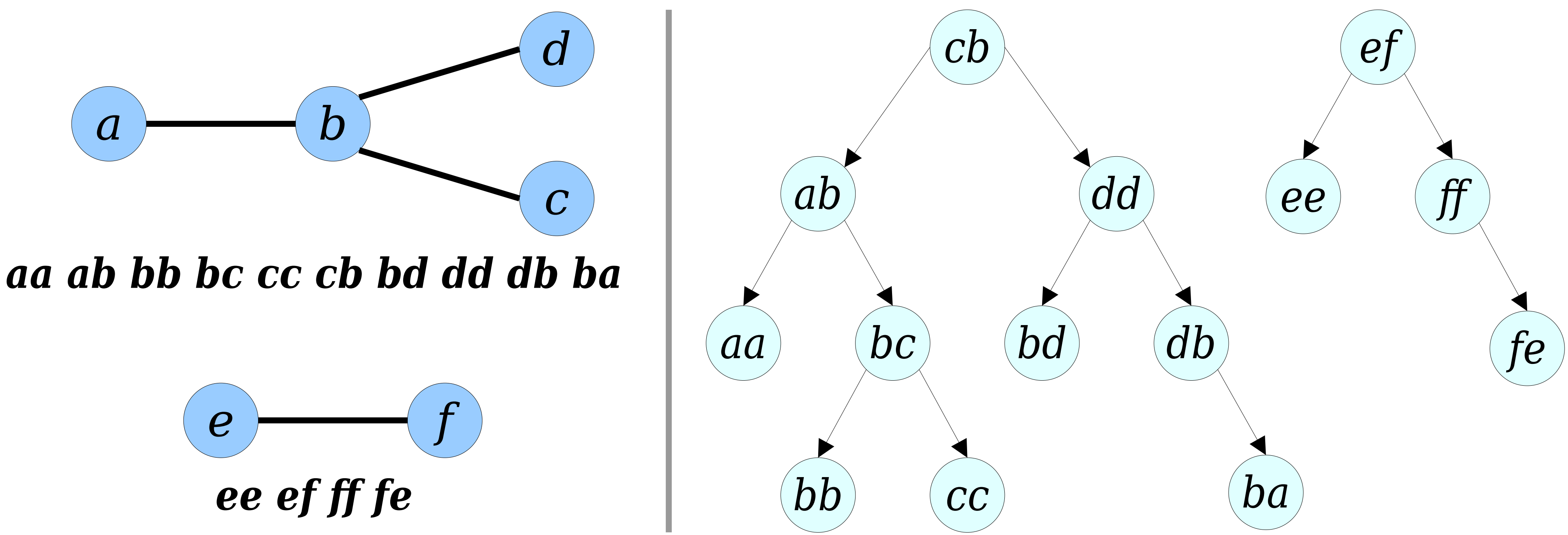
aa ab bb bc cc cb bd dd db ba

ee ef ff fe



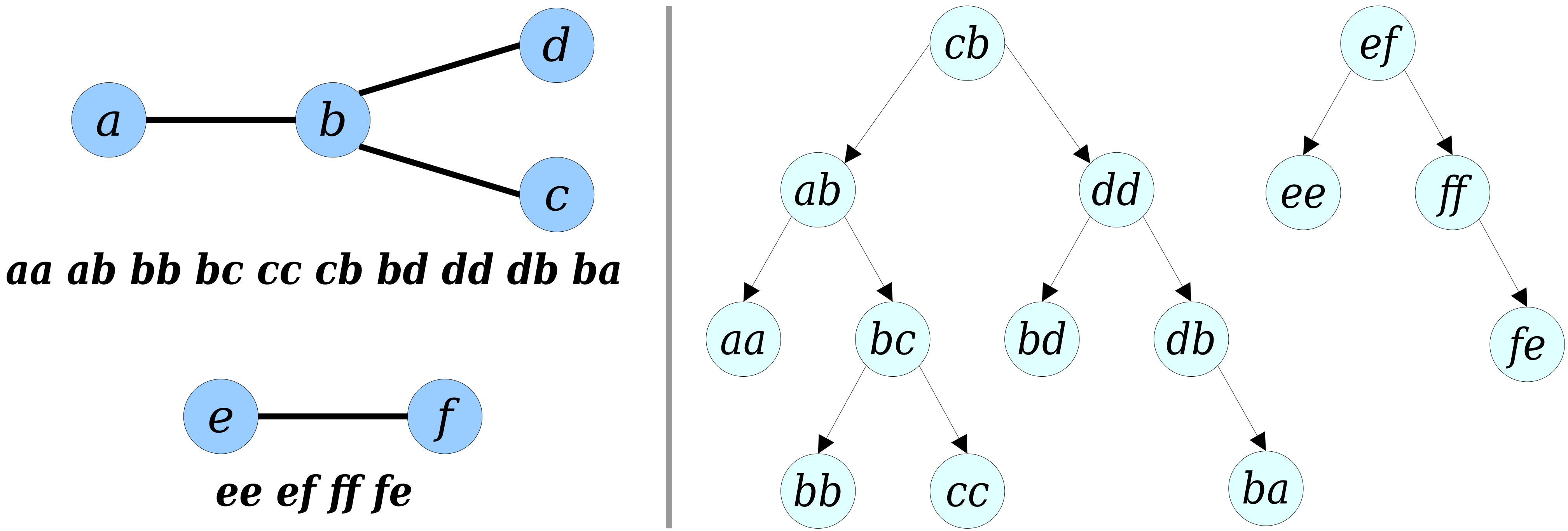
Binary Search(less) Trees

- ***Challenge:*** We need to be able to cut a sequence just before an edge, and we need to be able to join two sequences together efficiently.
- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).



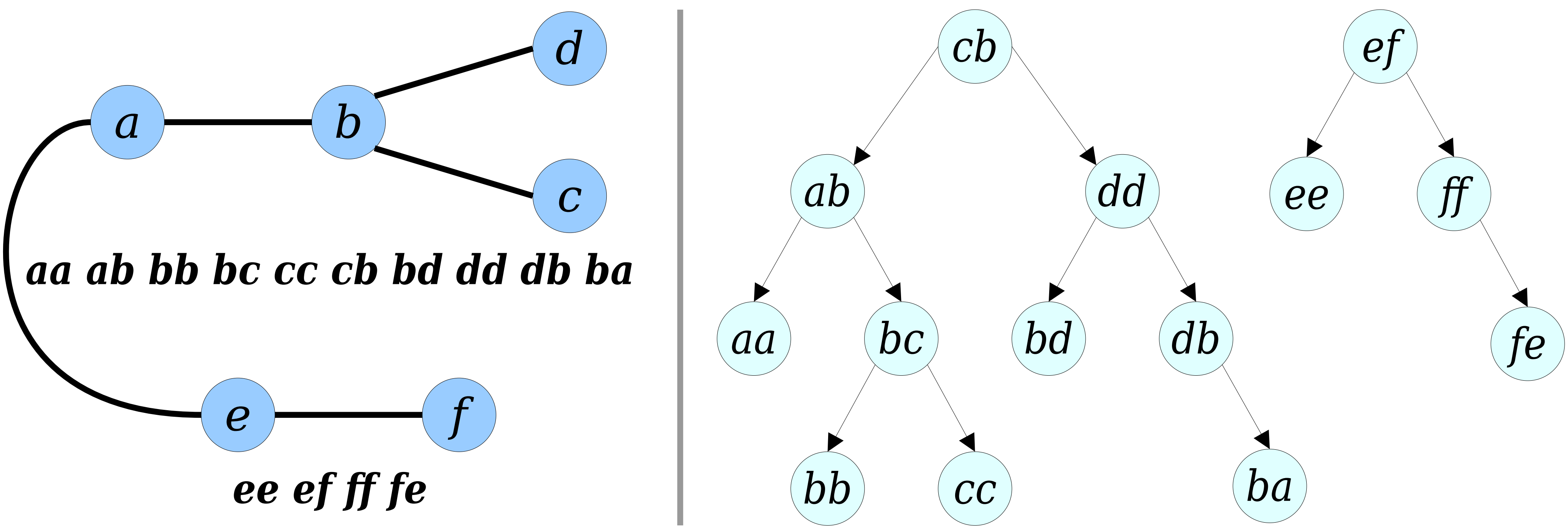
Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).



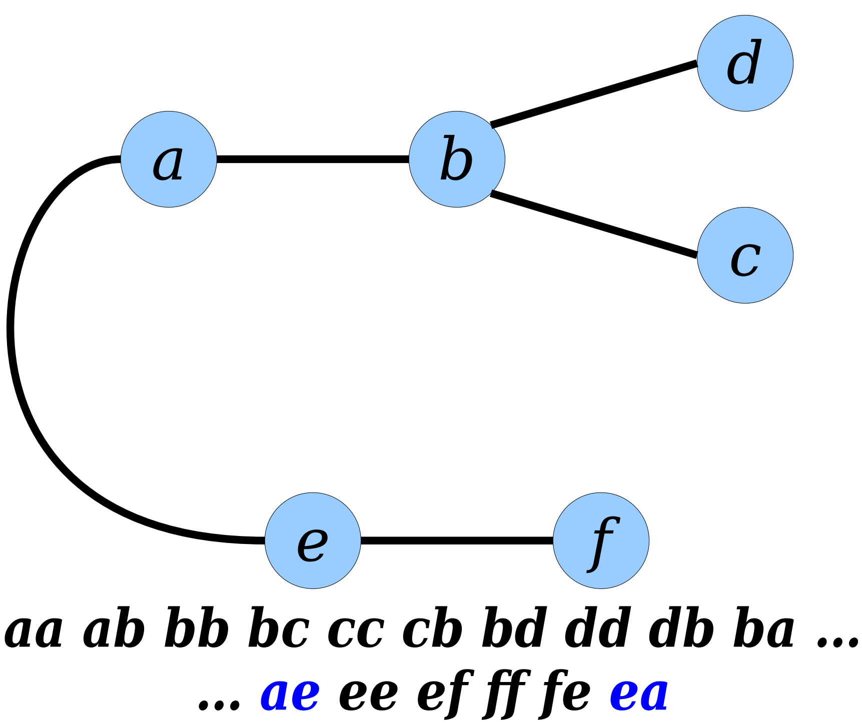
Binary Search(less) Trees

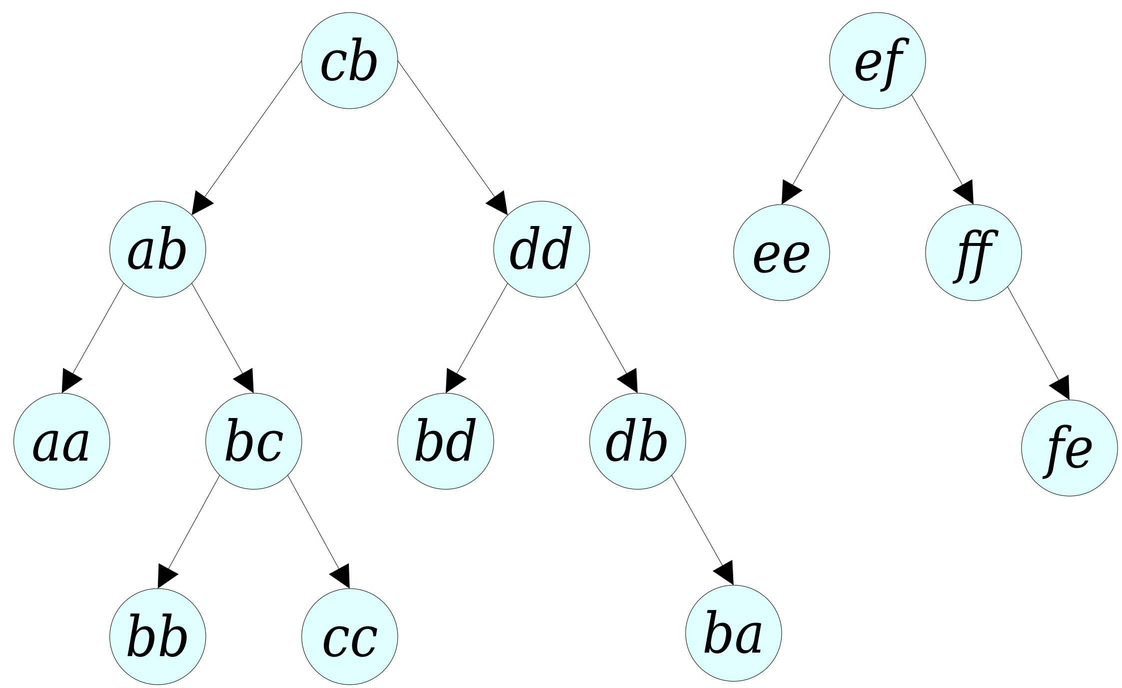
- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).



Binary Search(less) Trees

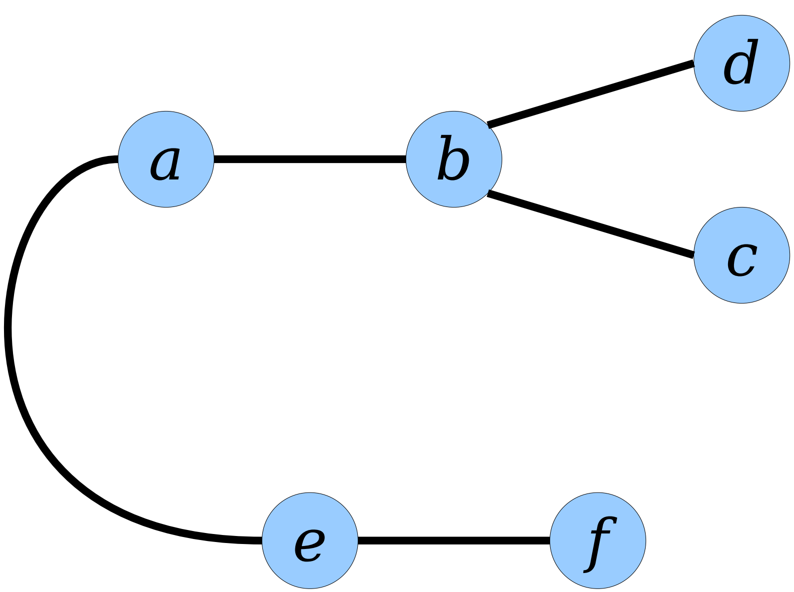
- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).



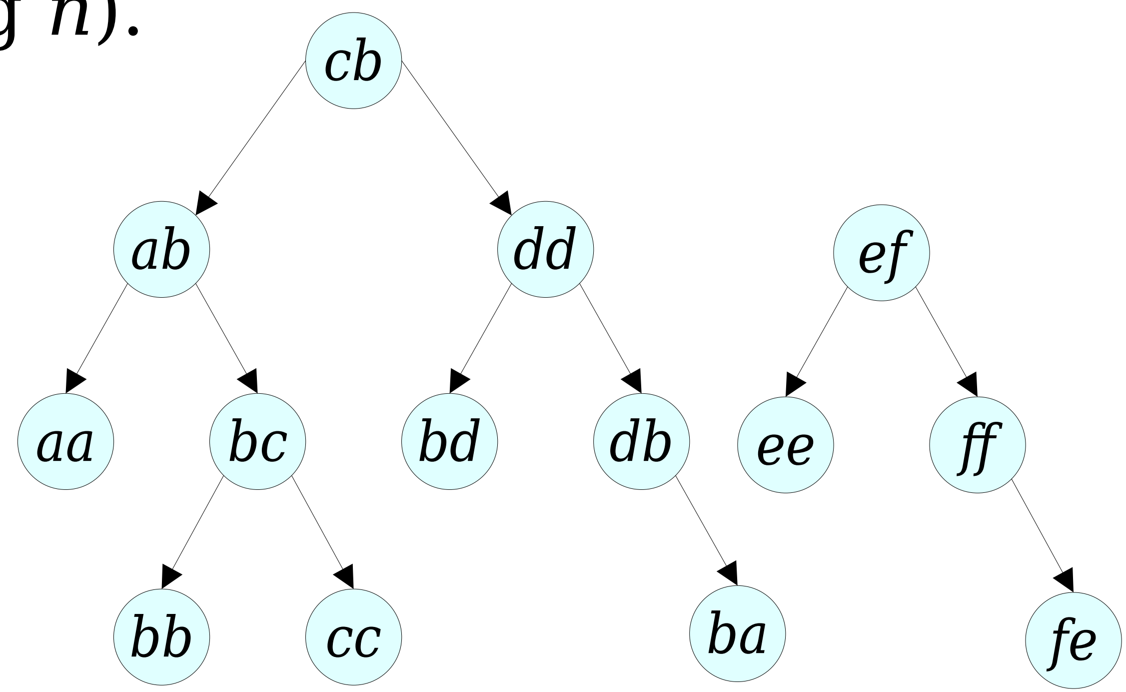


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).

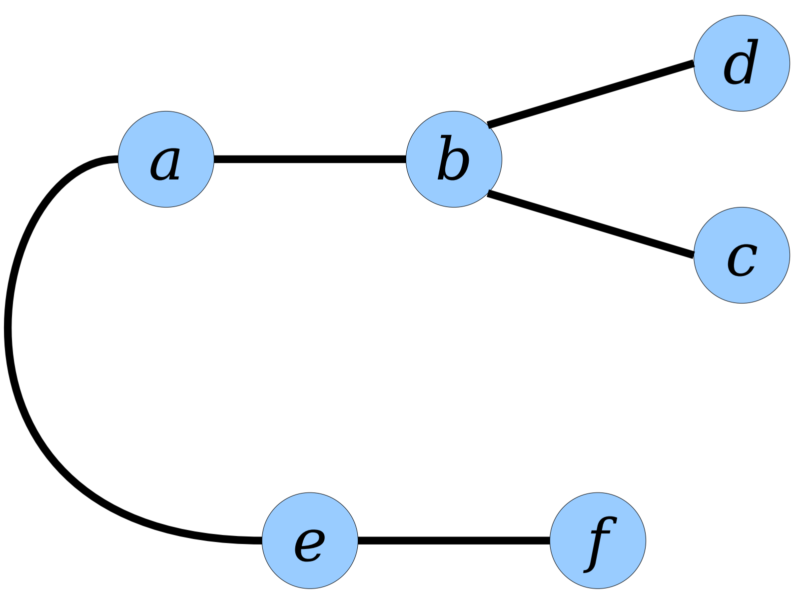


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

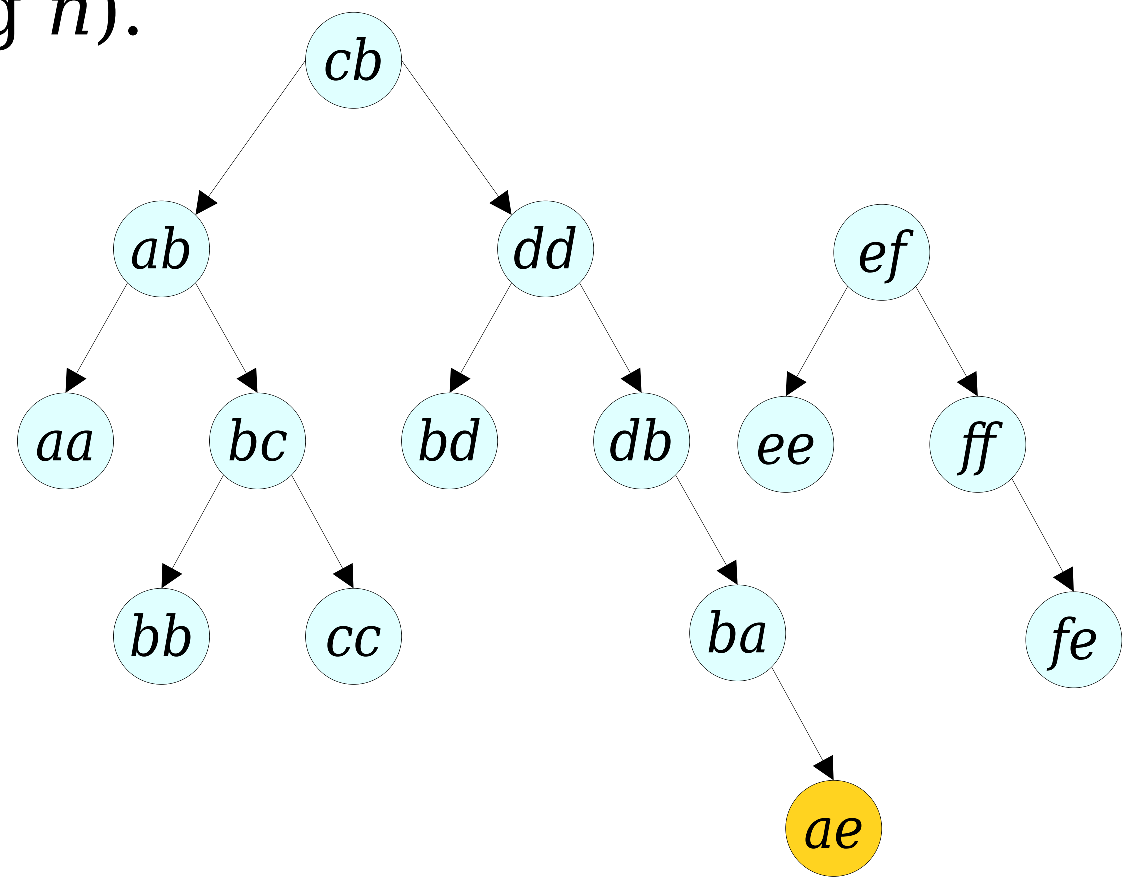


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).

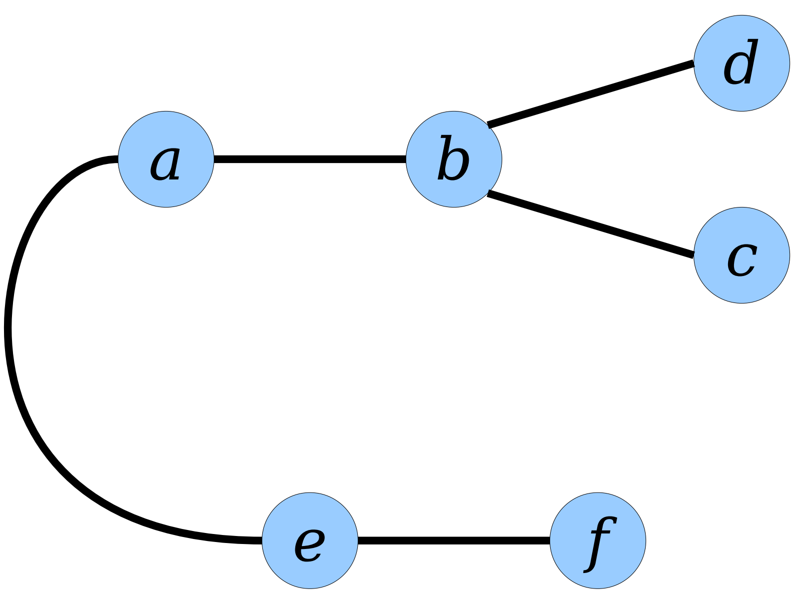


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

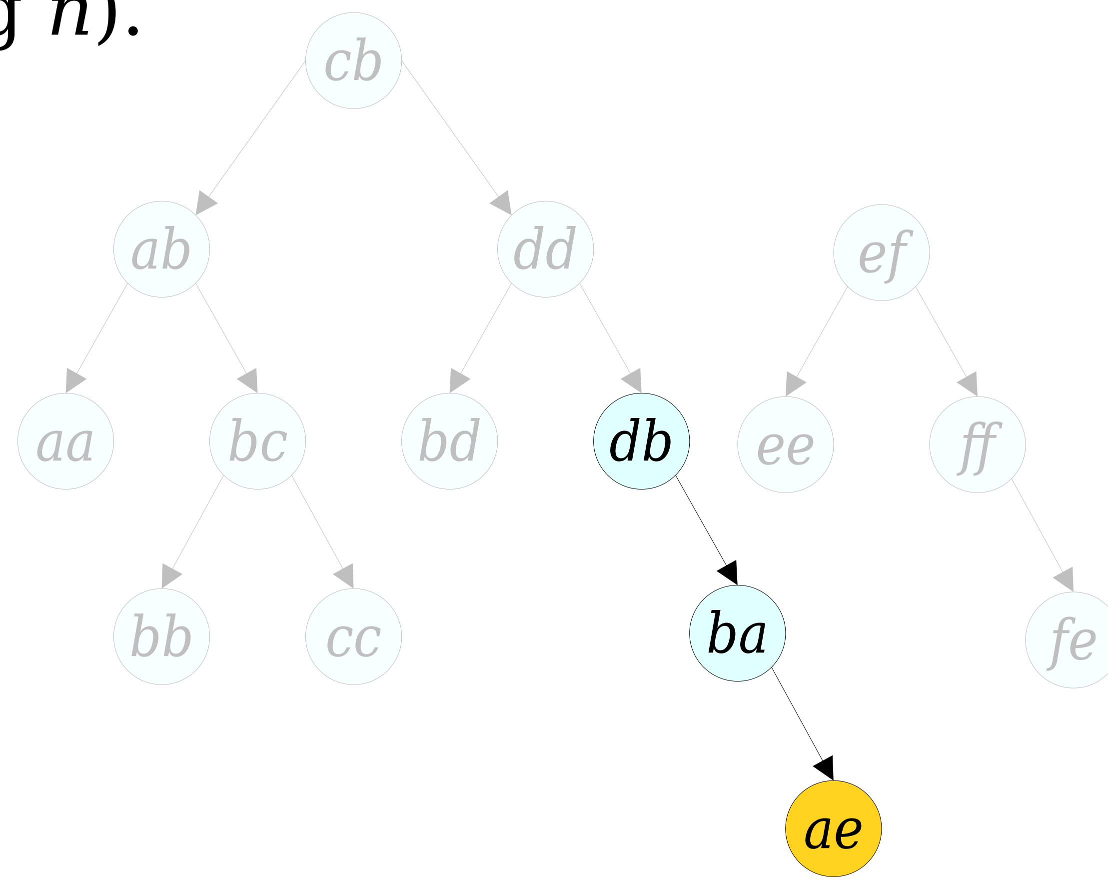


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

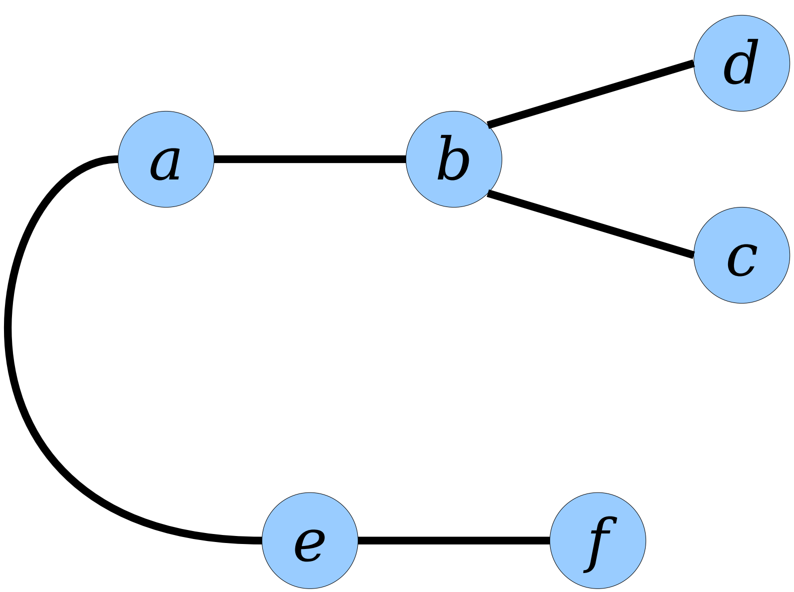


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

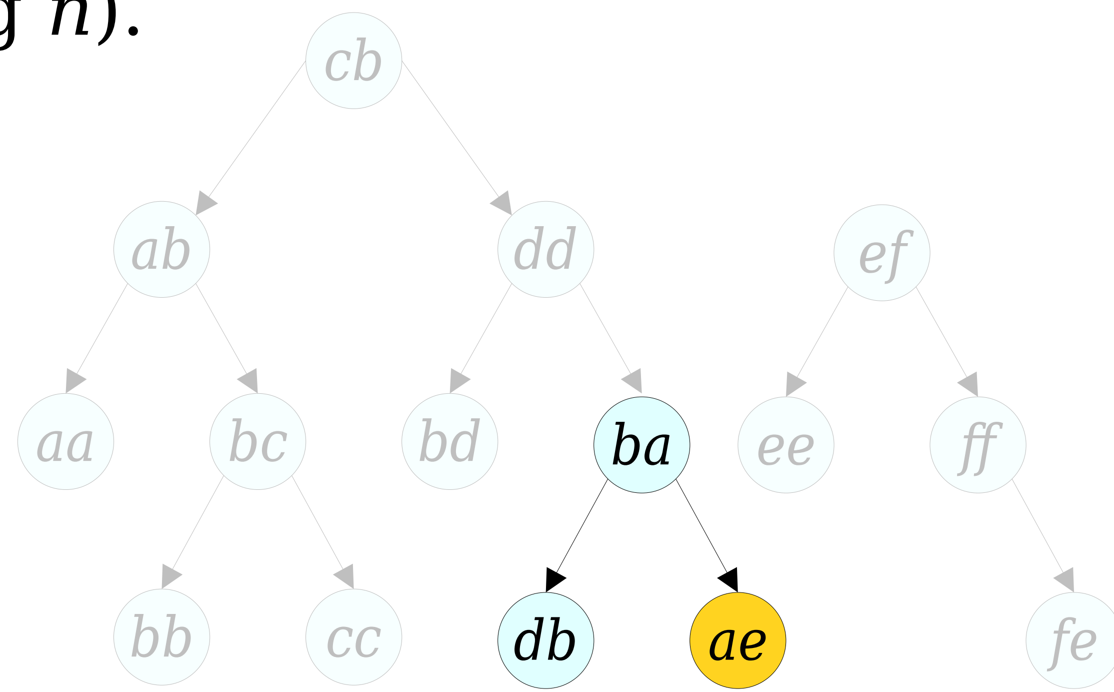


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

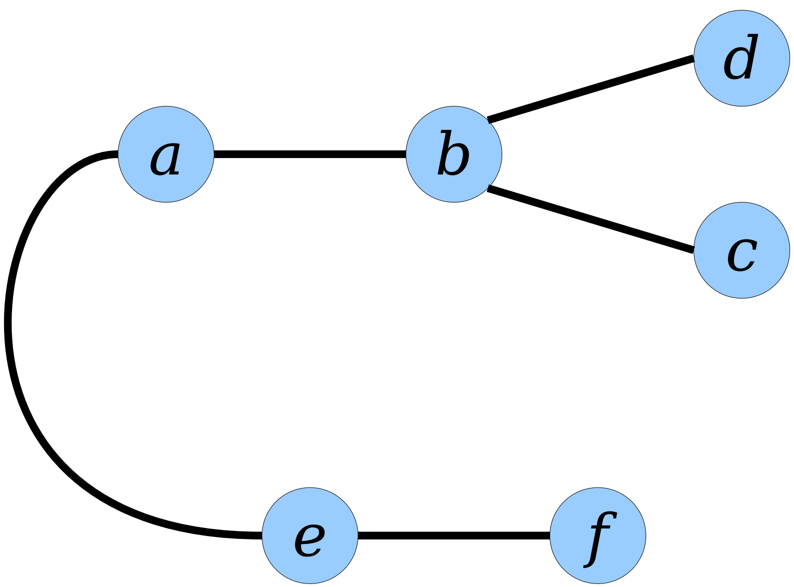


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

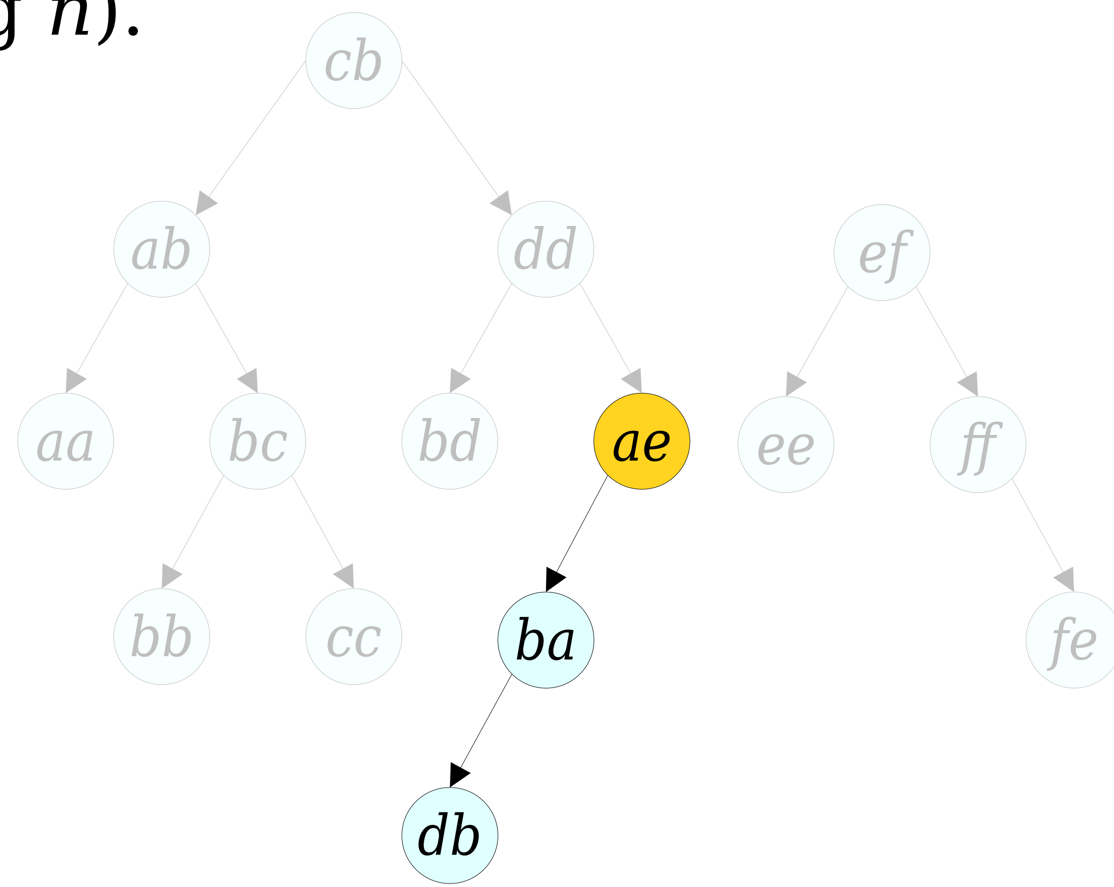


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).

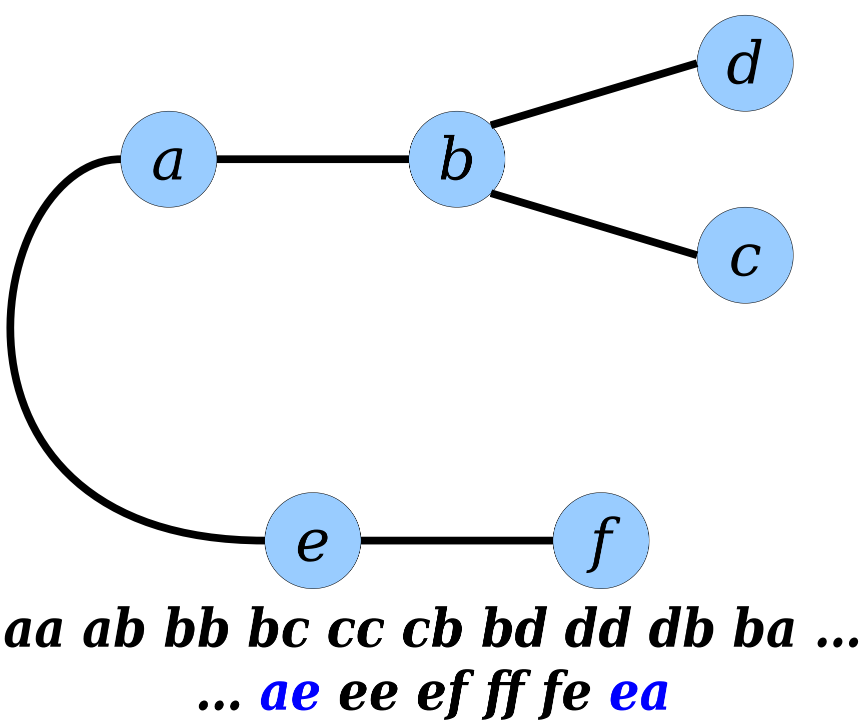


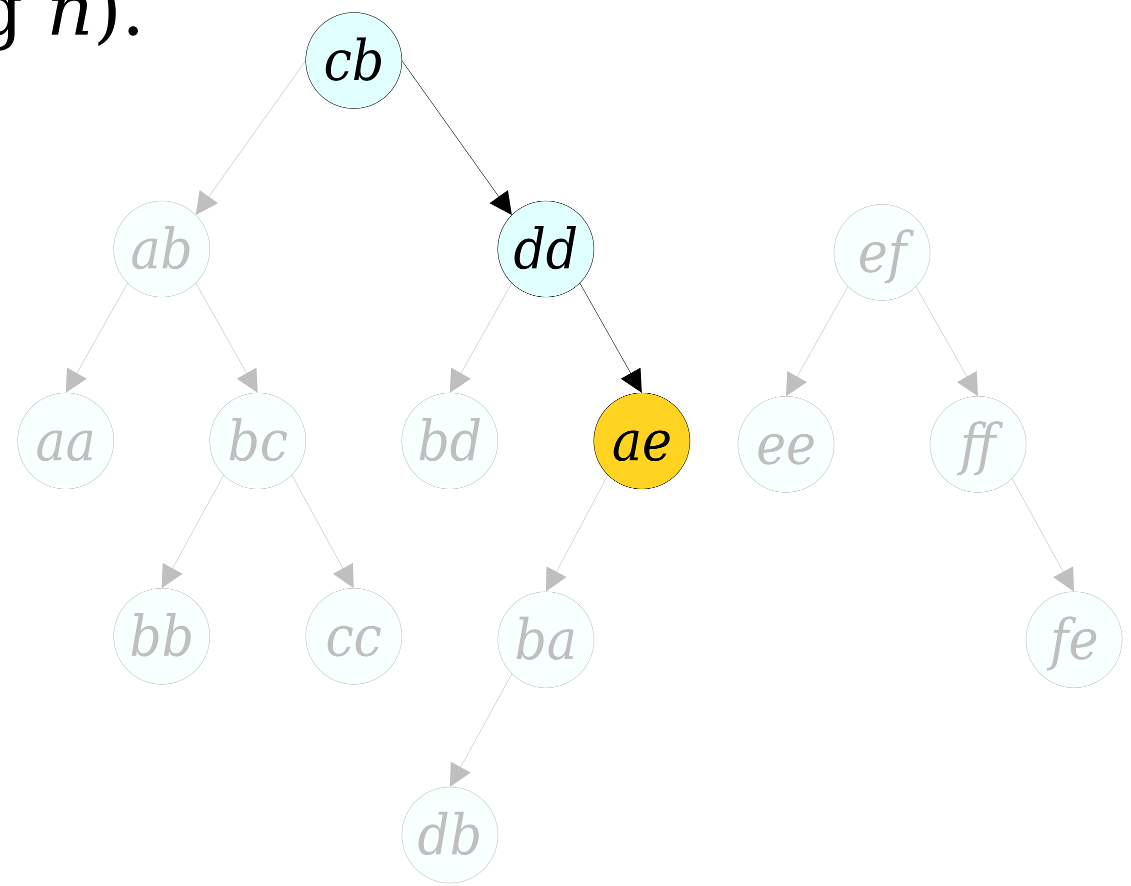
aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea



Binary Search(less) Trees

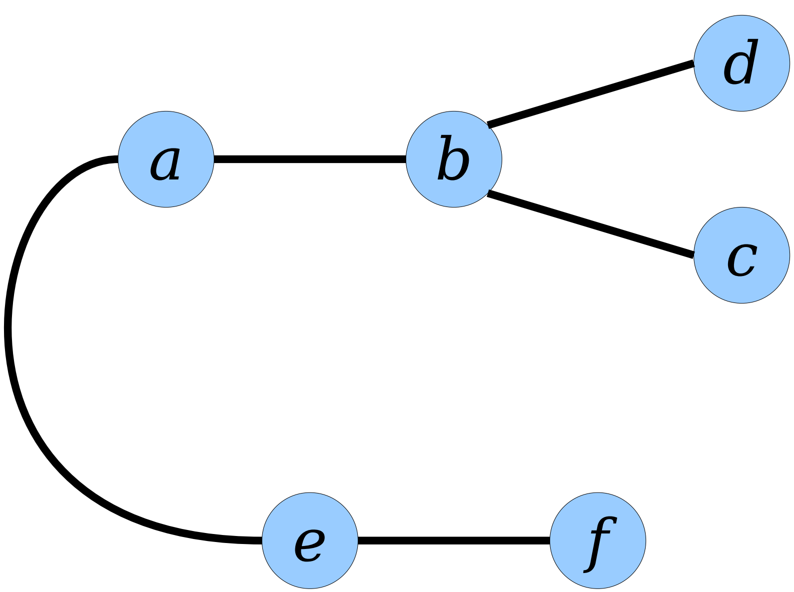
- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).



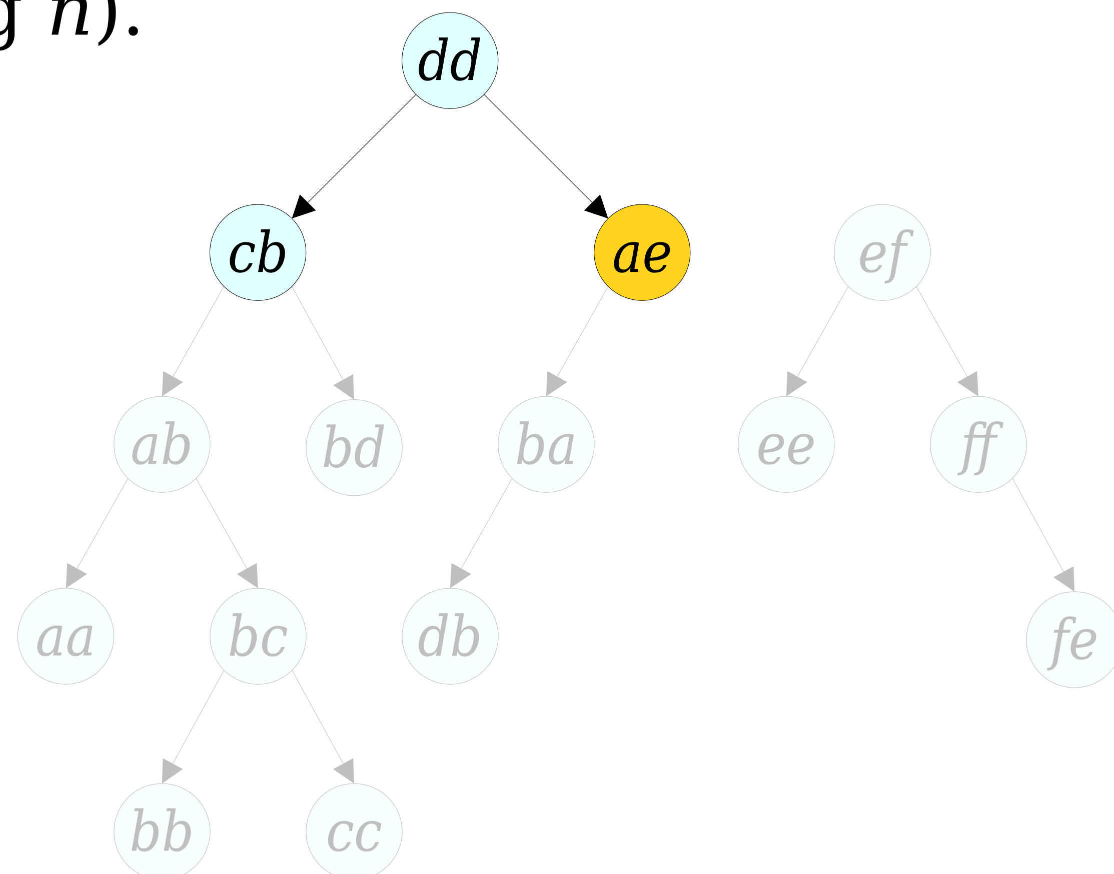


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

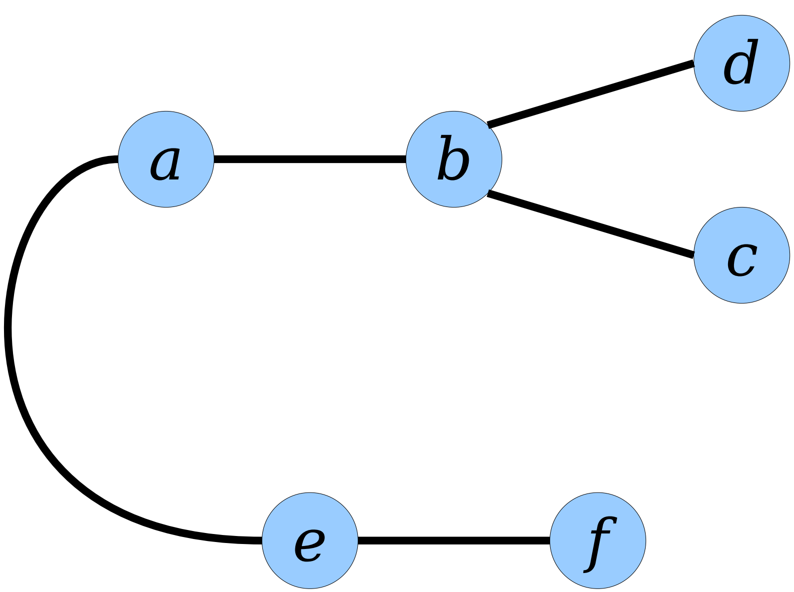


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

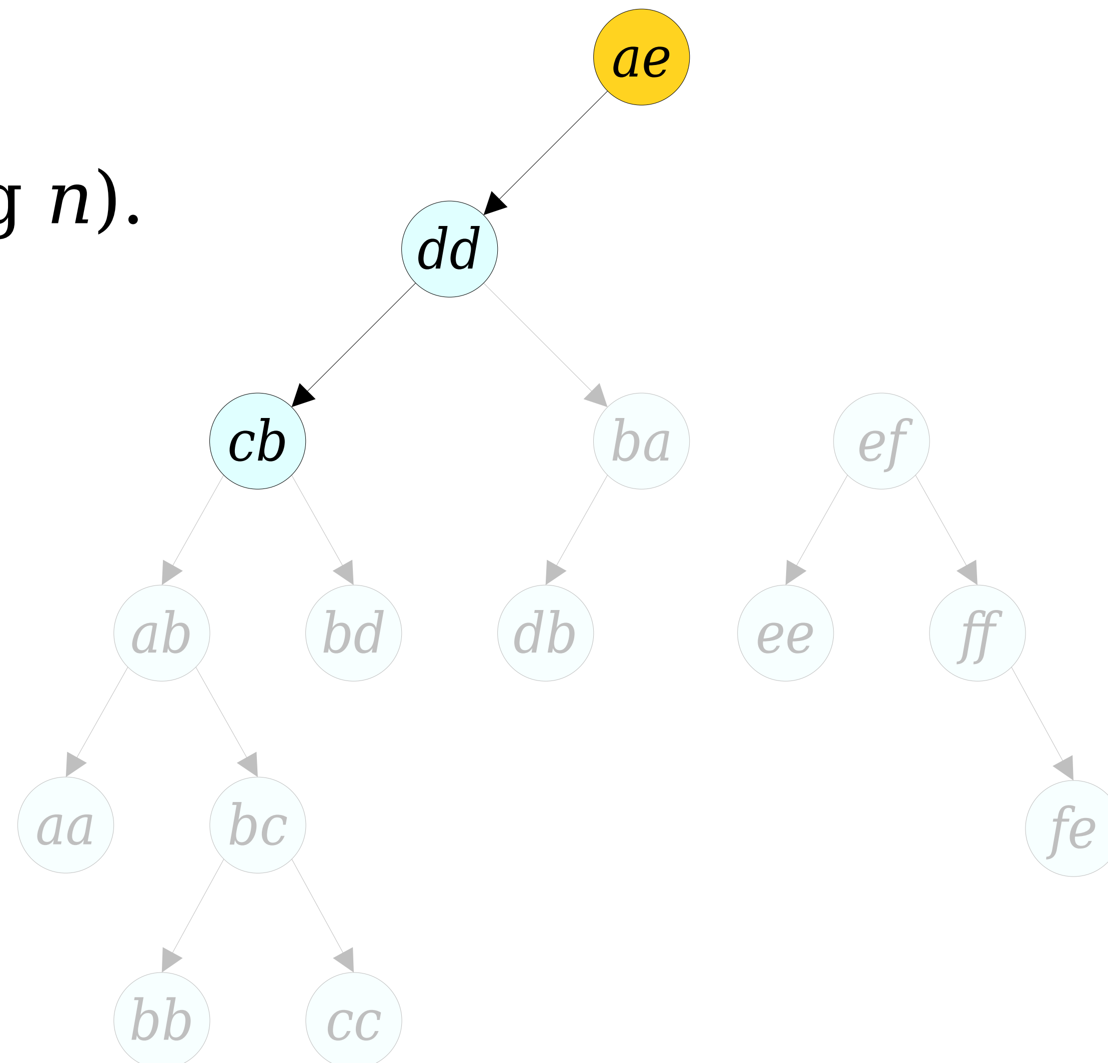


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

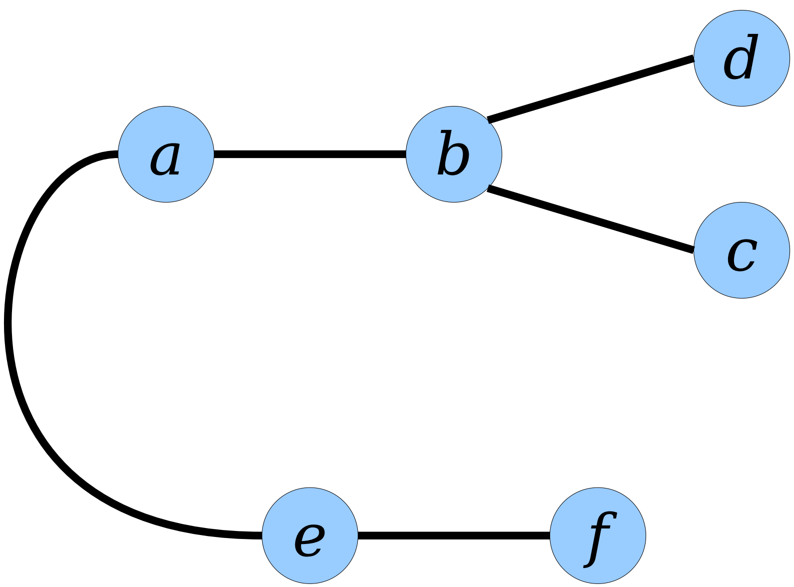


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

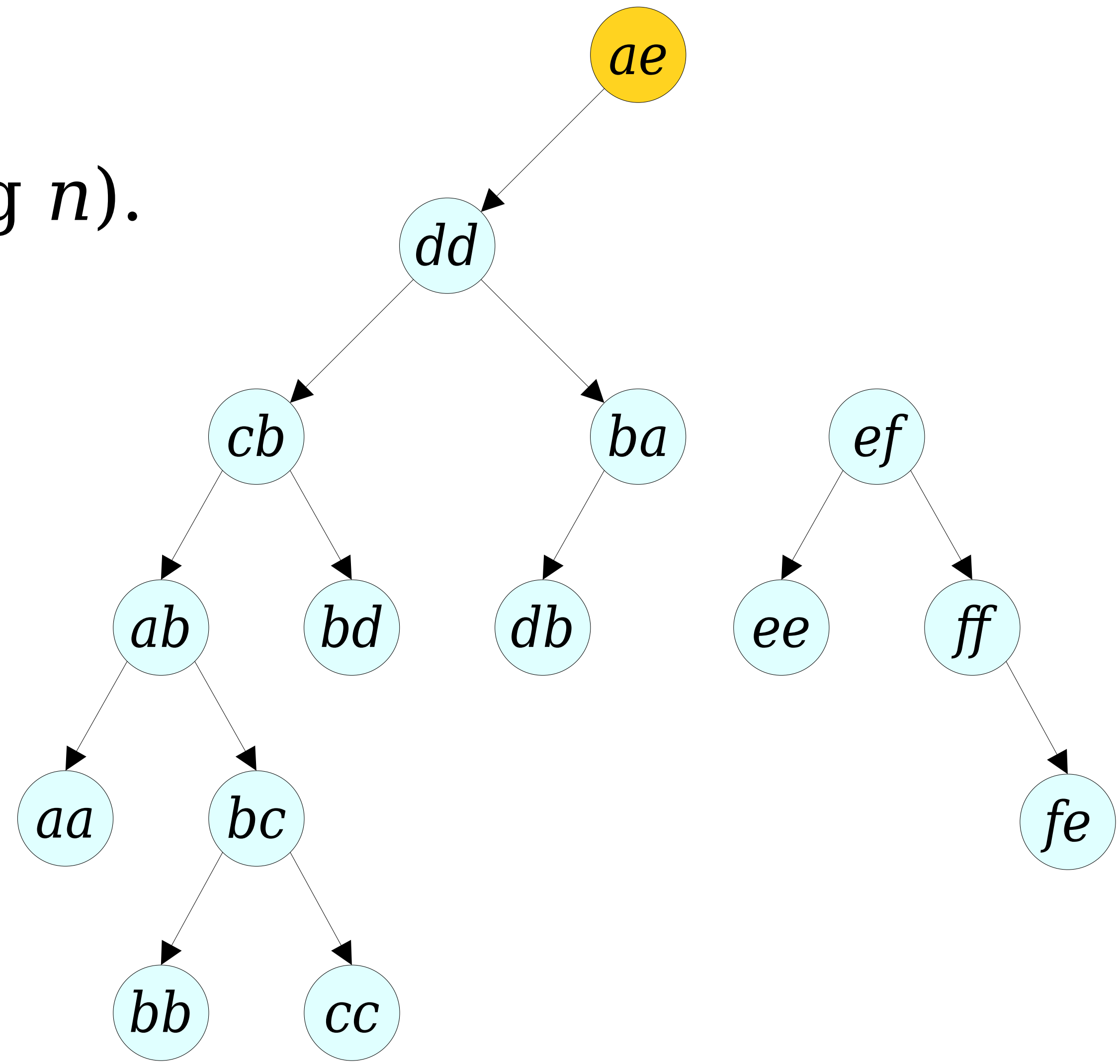


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

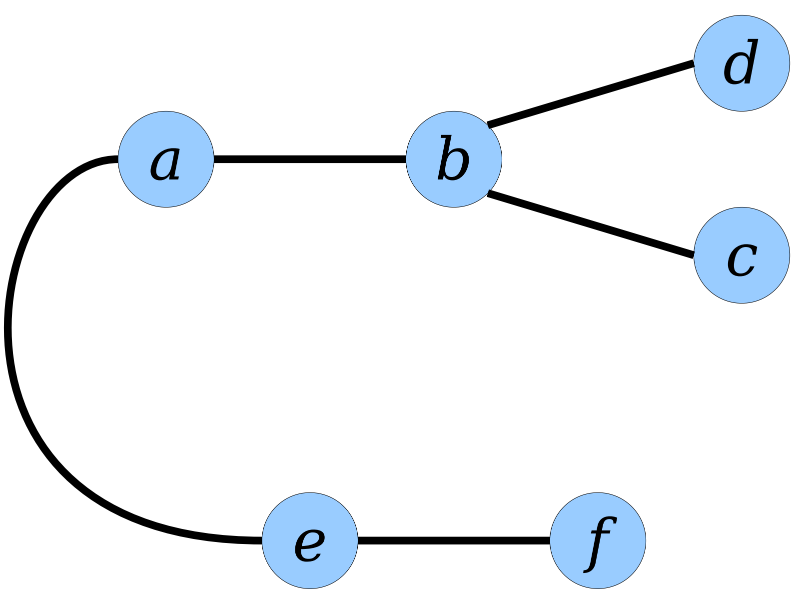


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

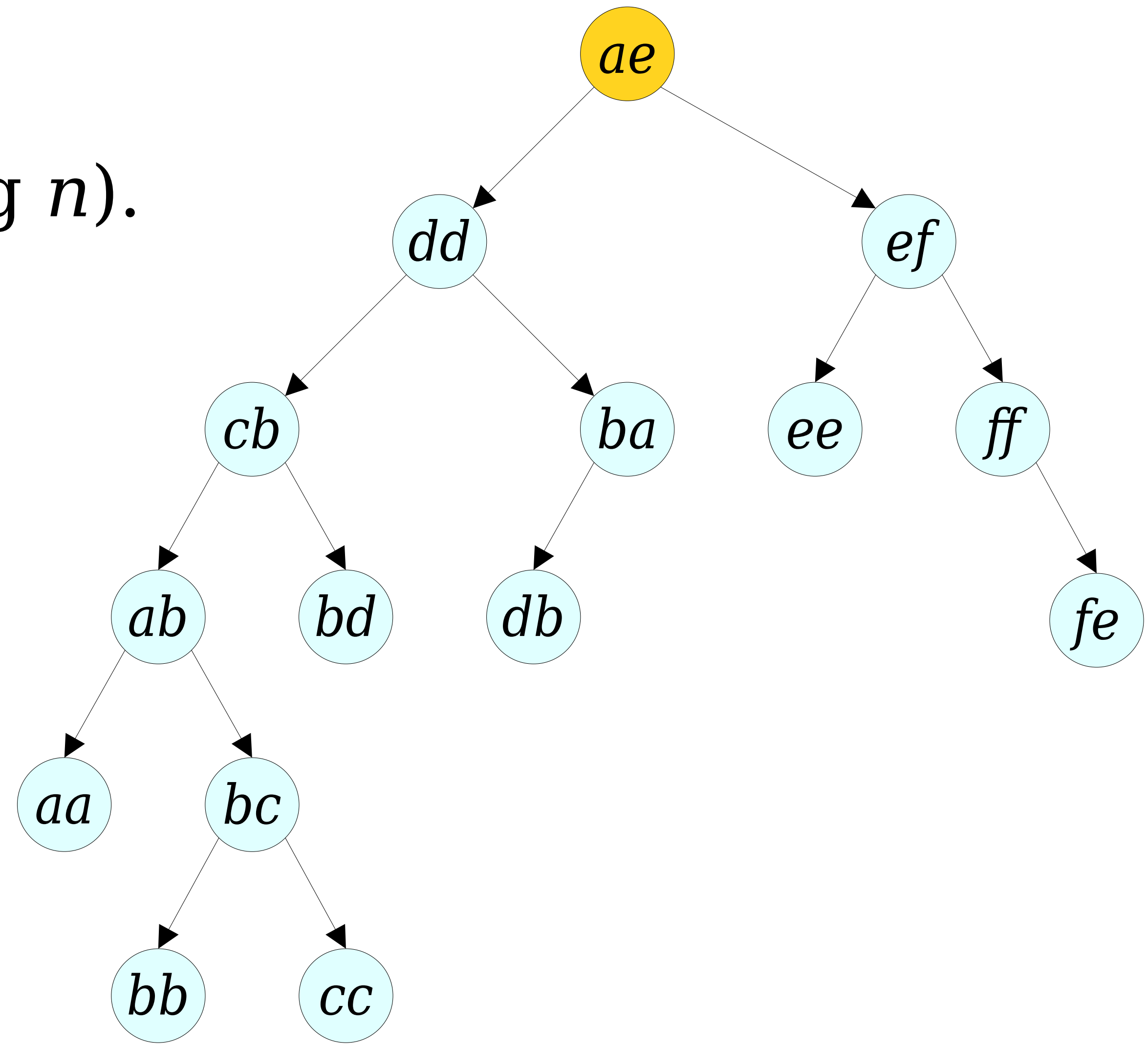


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

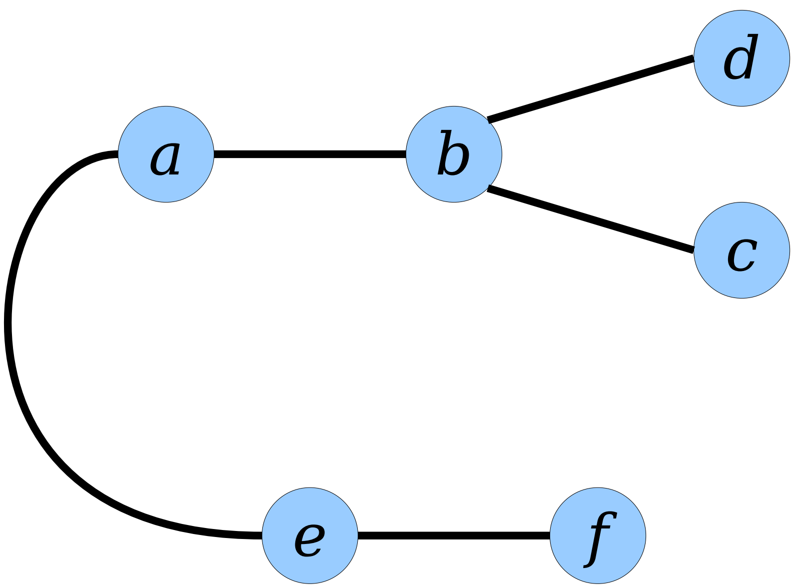


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

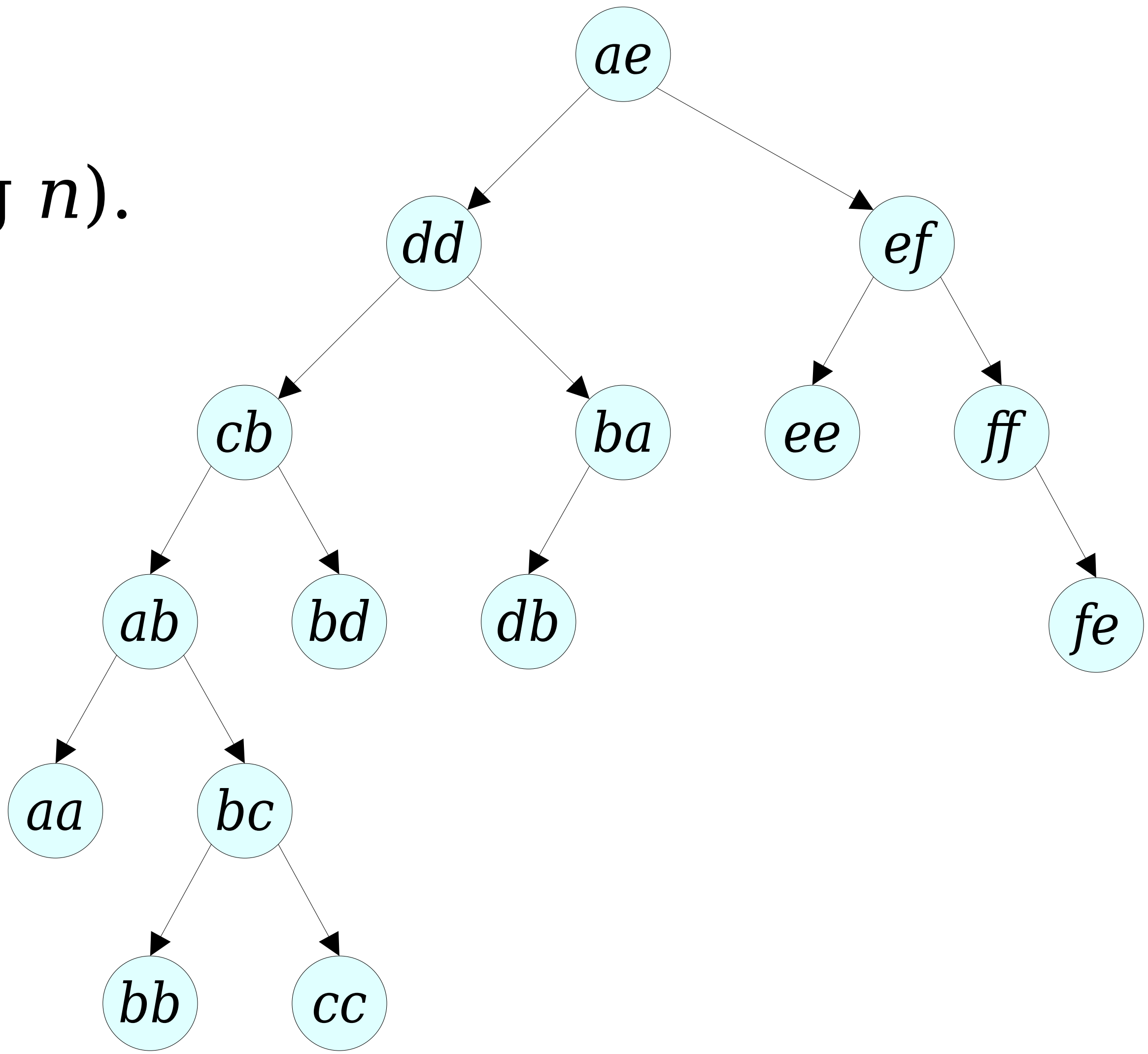


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

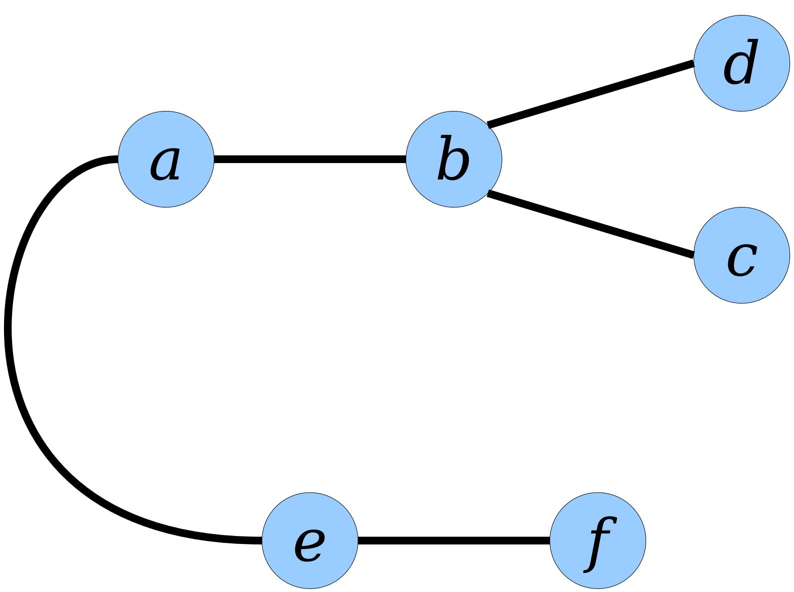


aa ab bb bc cc cb bd dd db ba … … ae ee ef ff fe ea

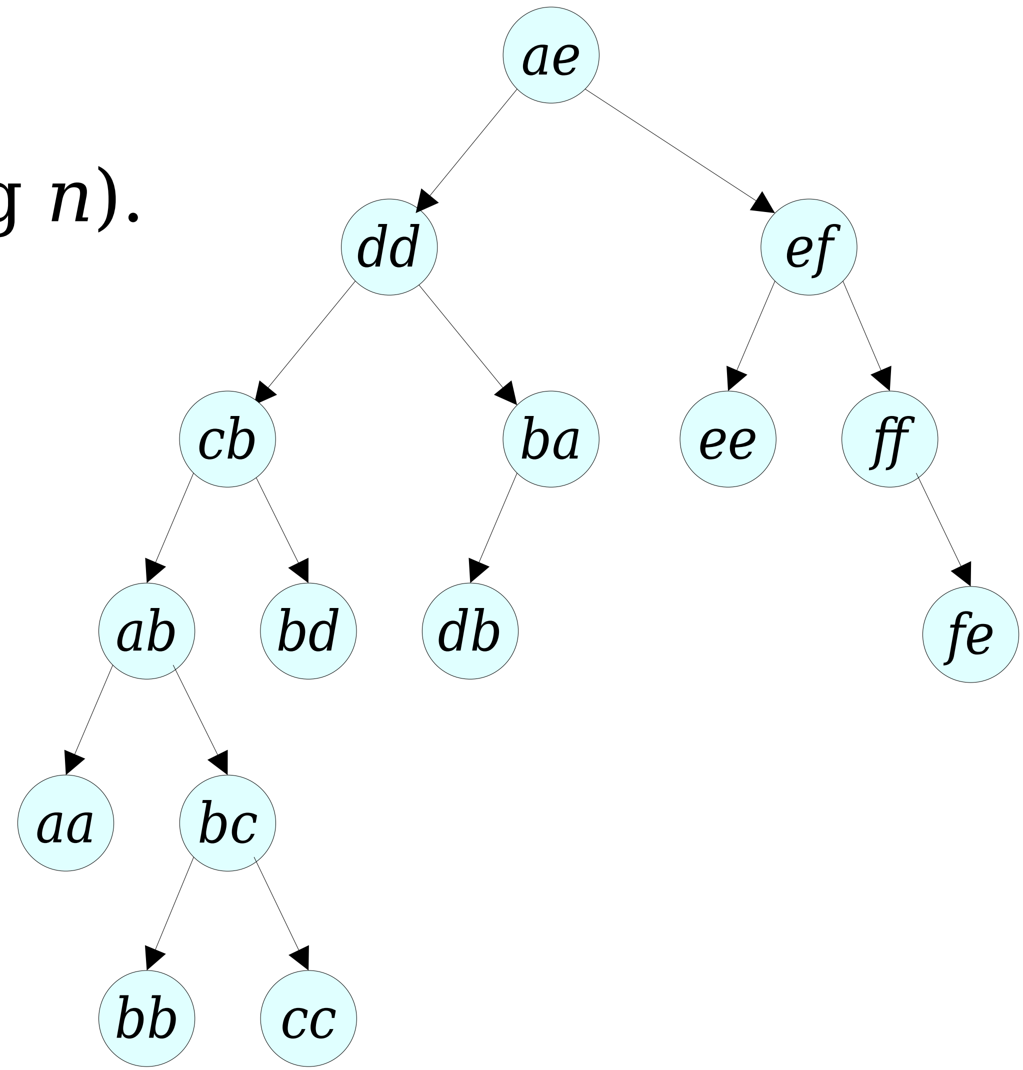


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

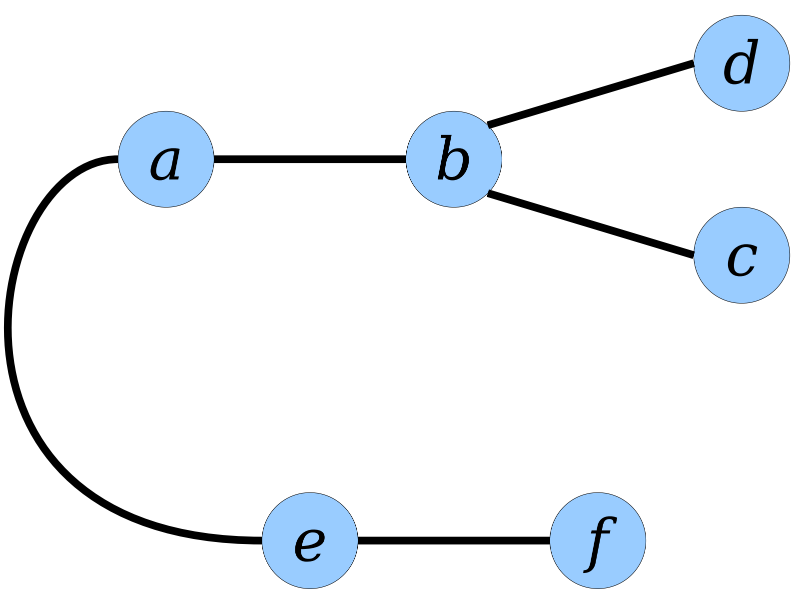


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

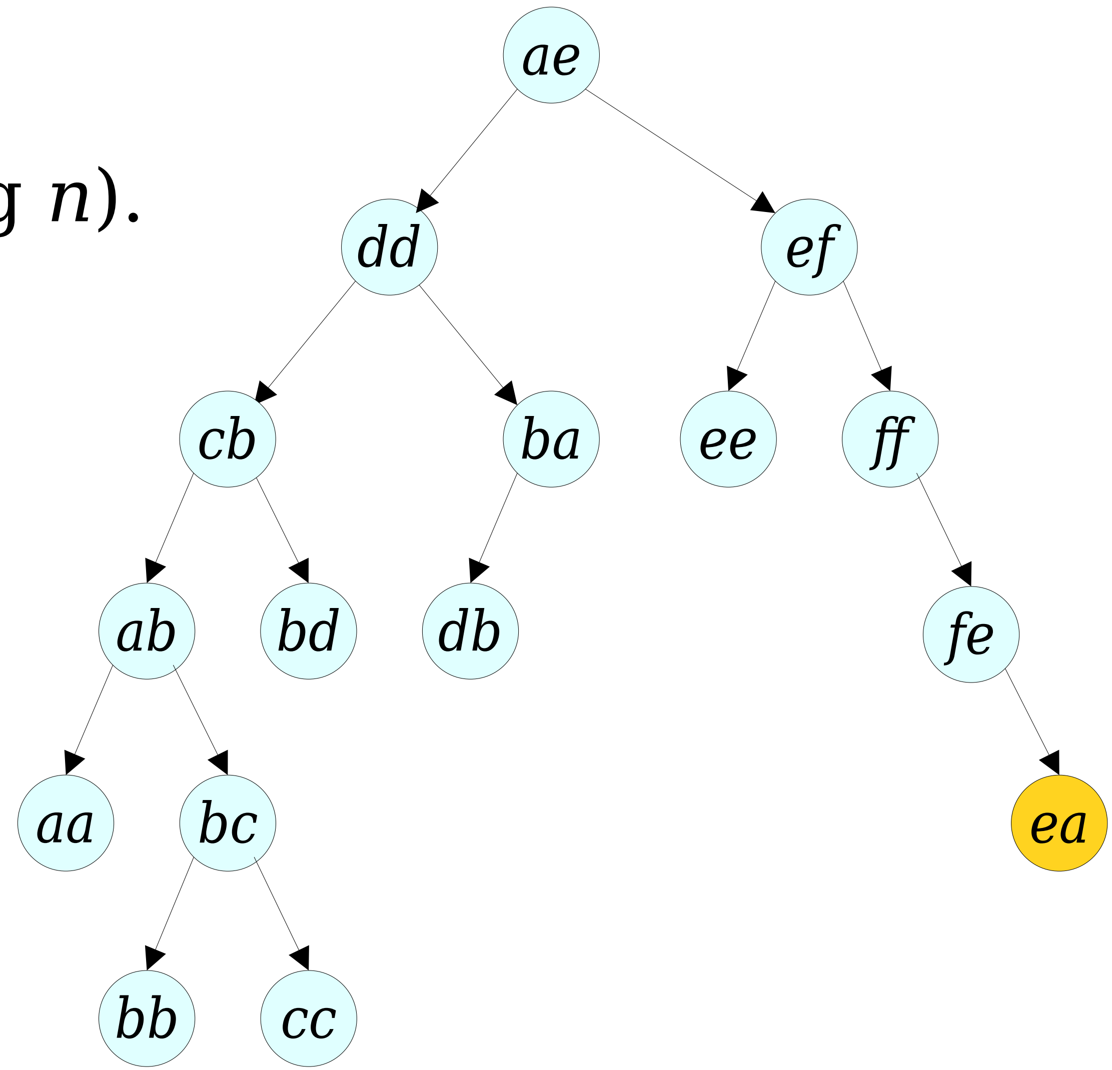


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

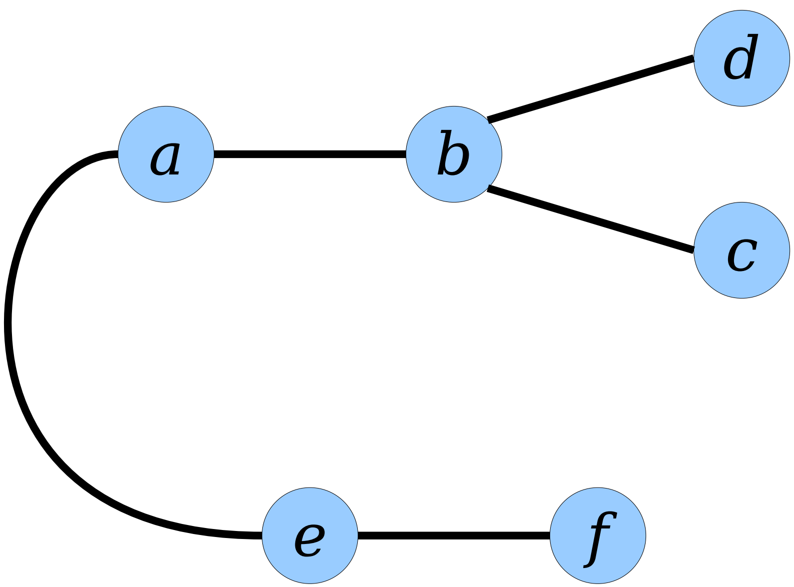


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

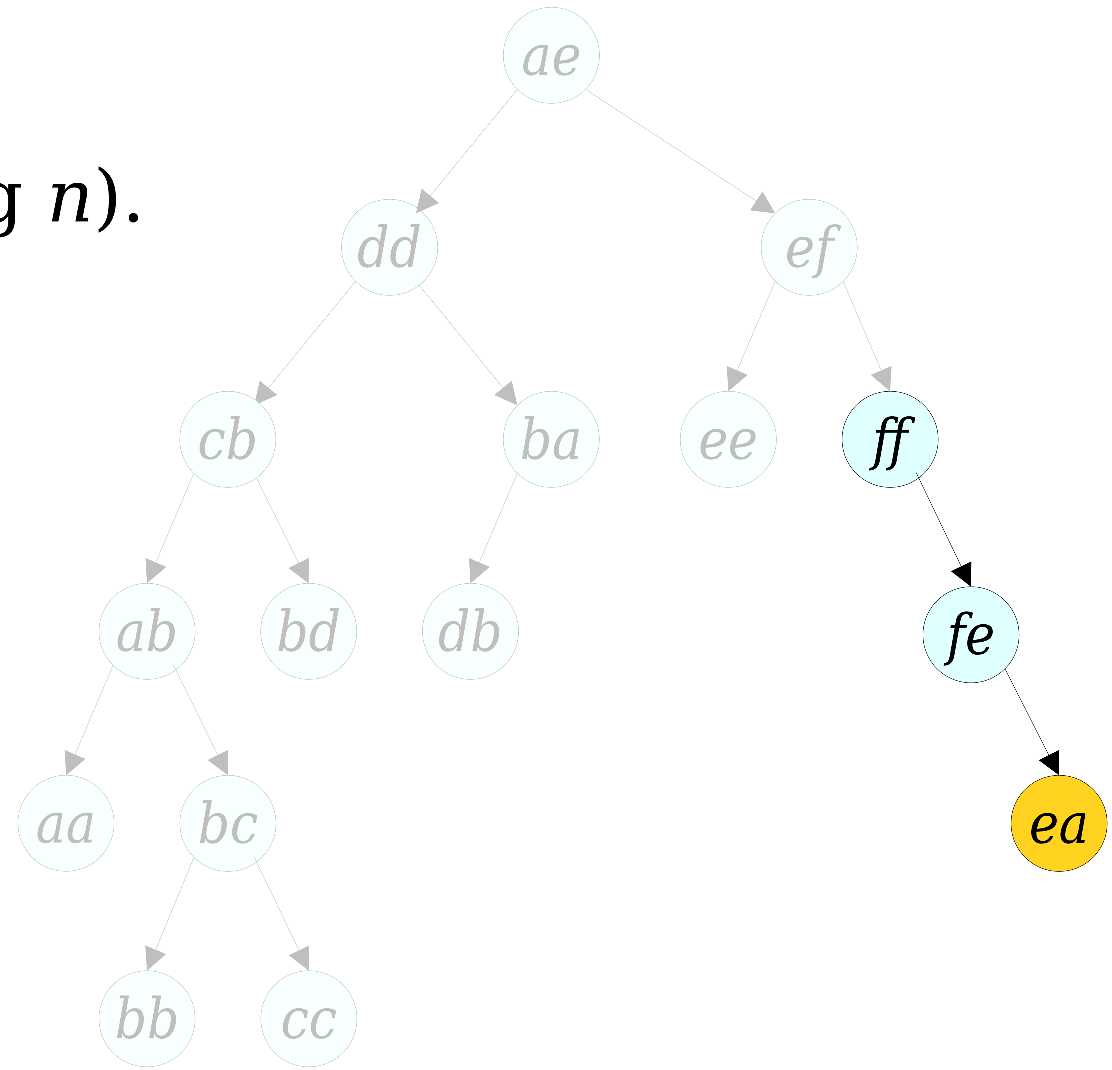


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

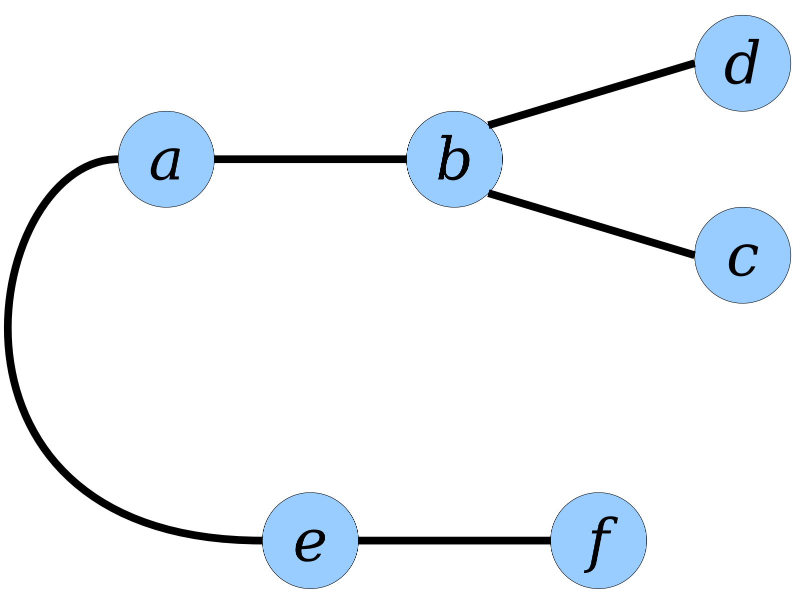


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

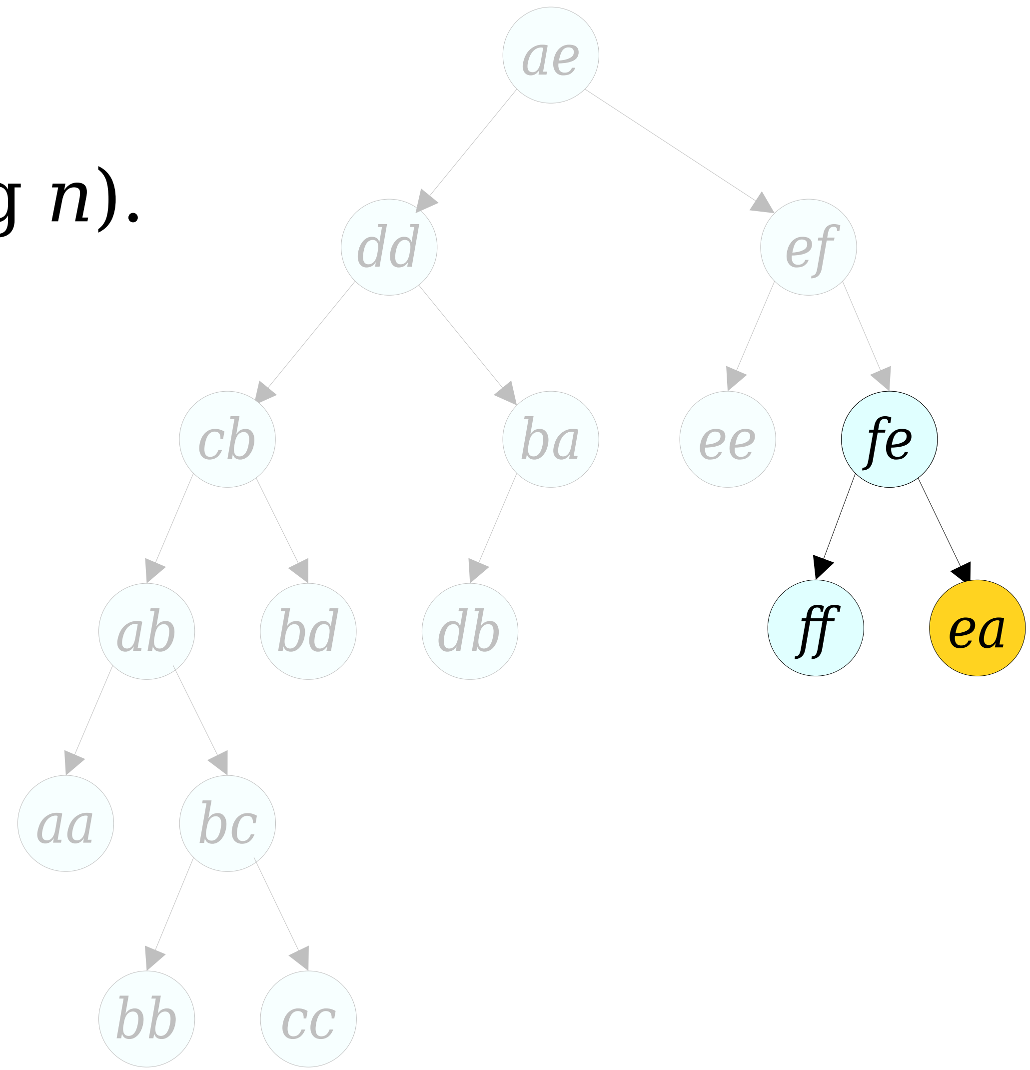


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

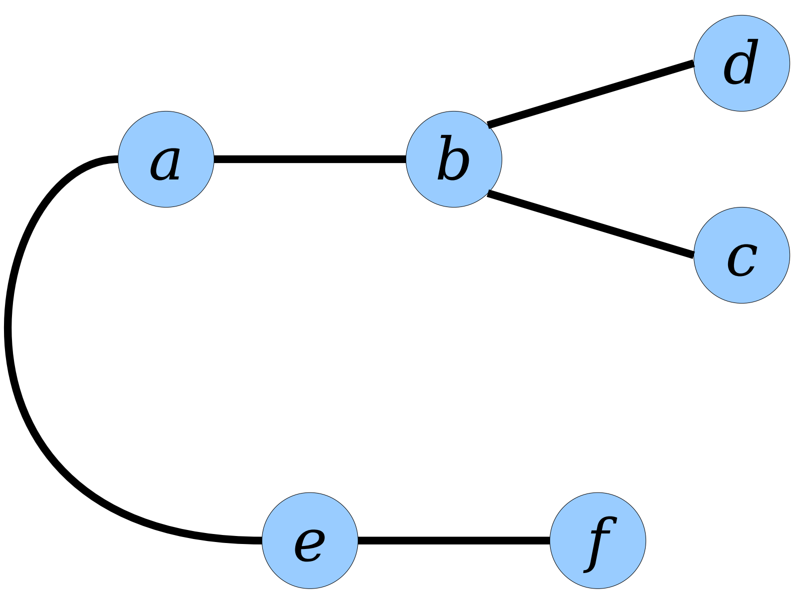


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

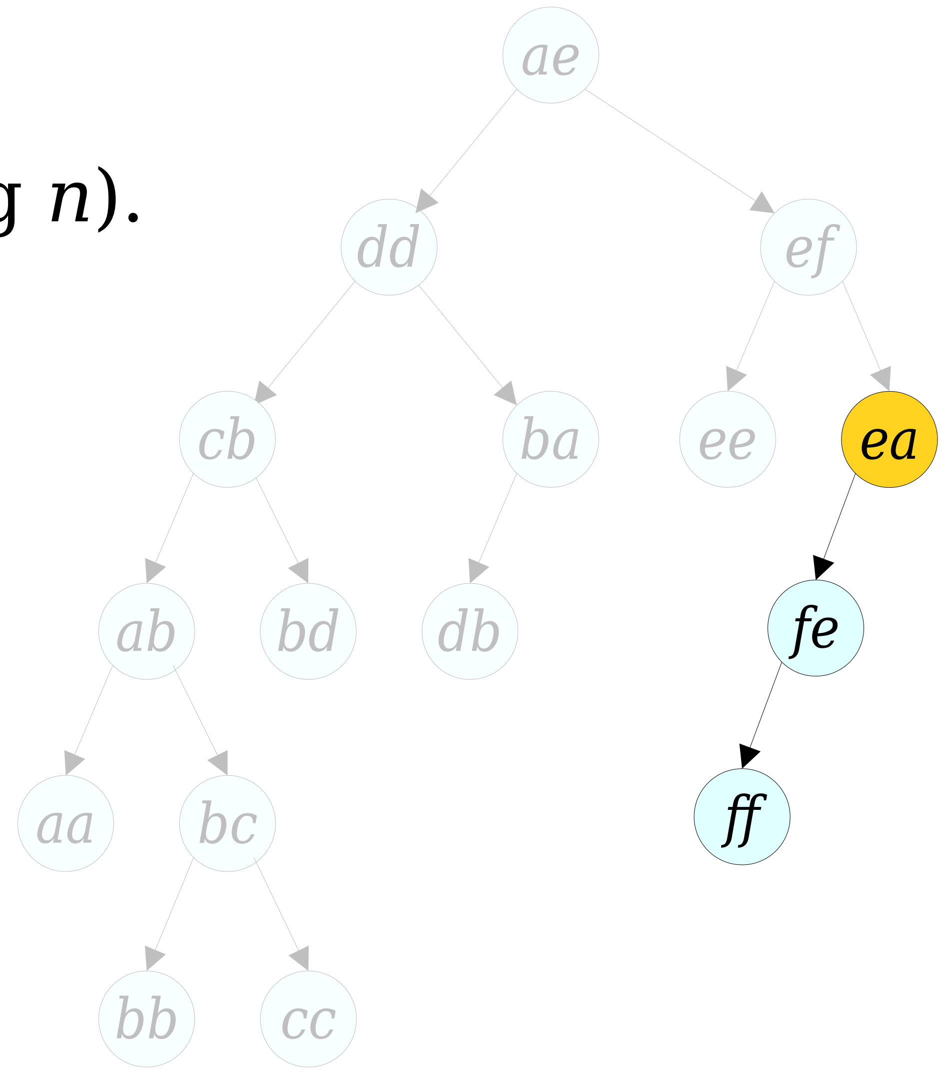


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).

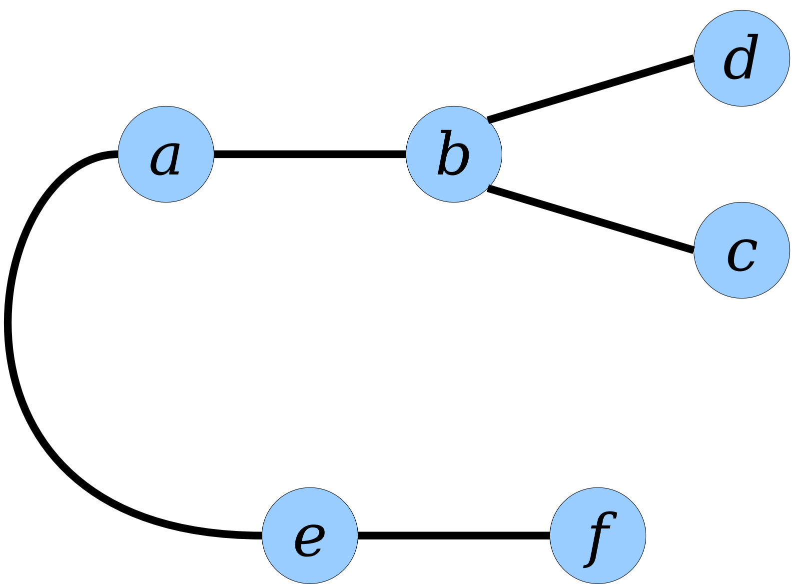


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

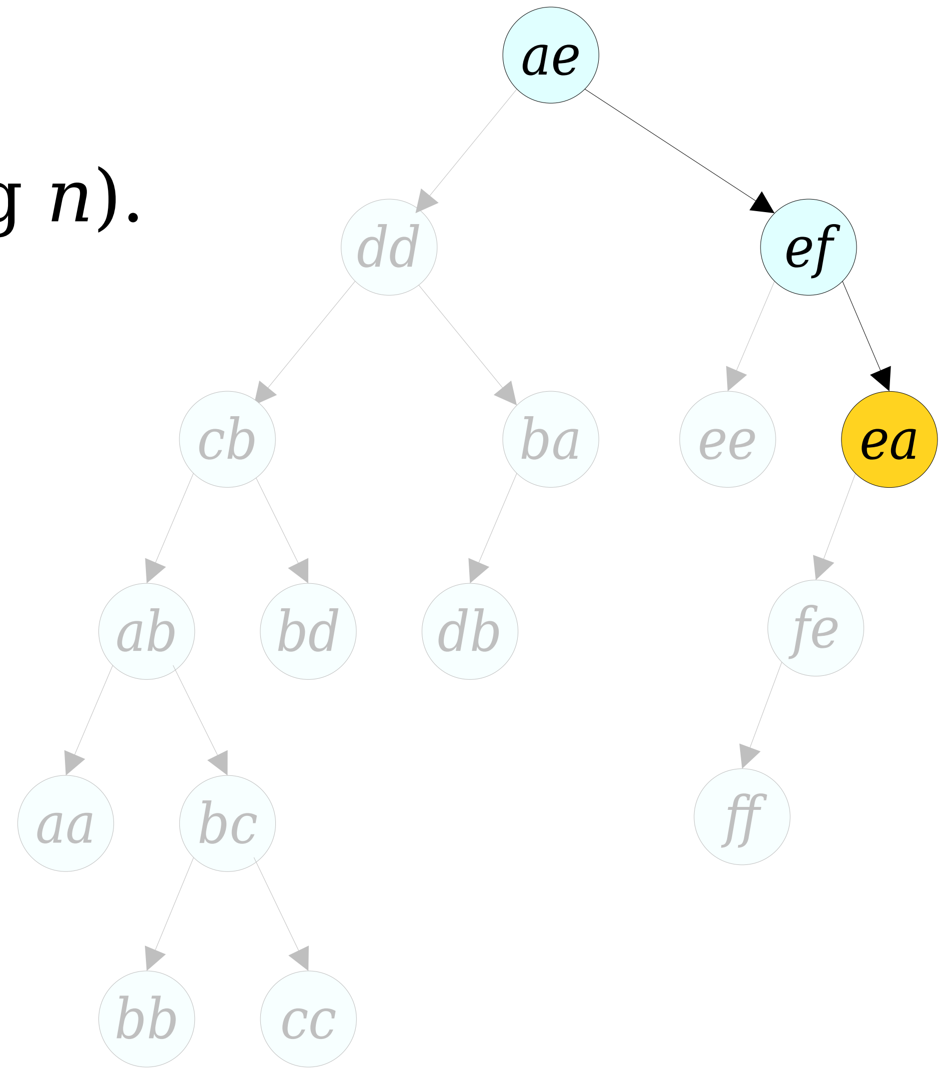


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).

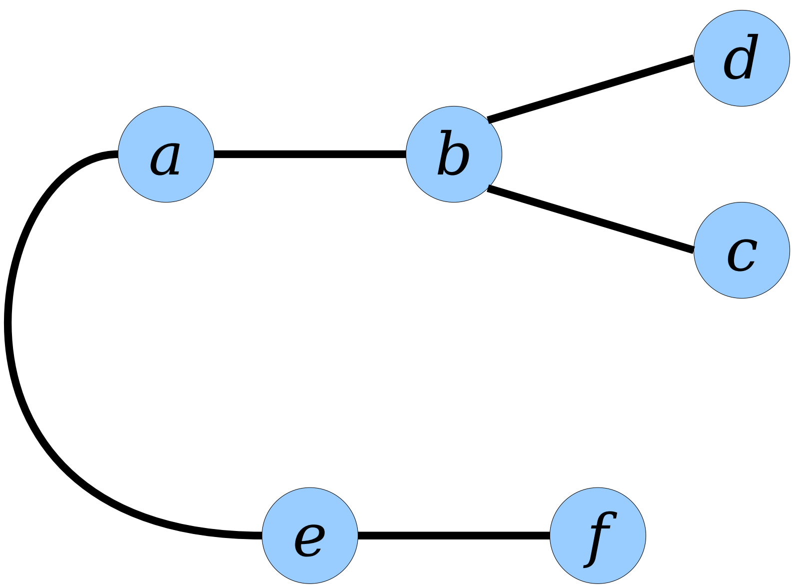


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

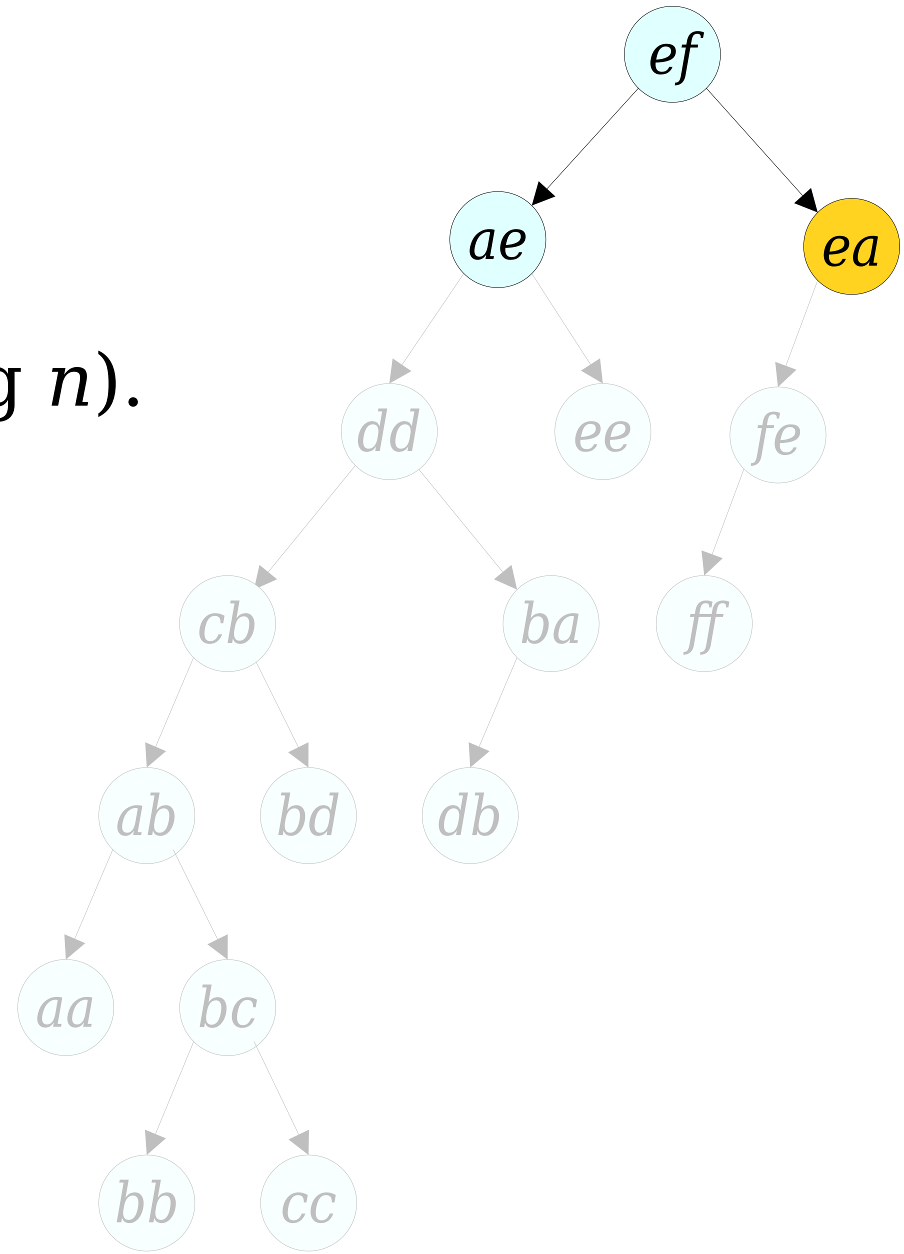


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

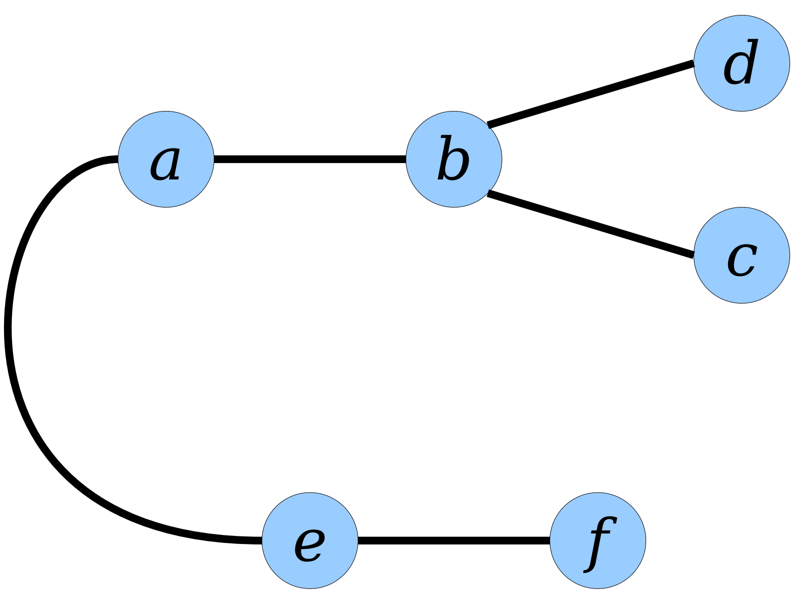


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

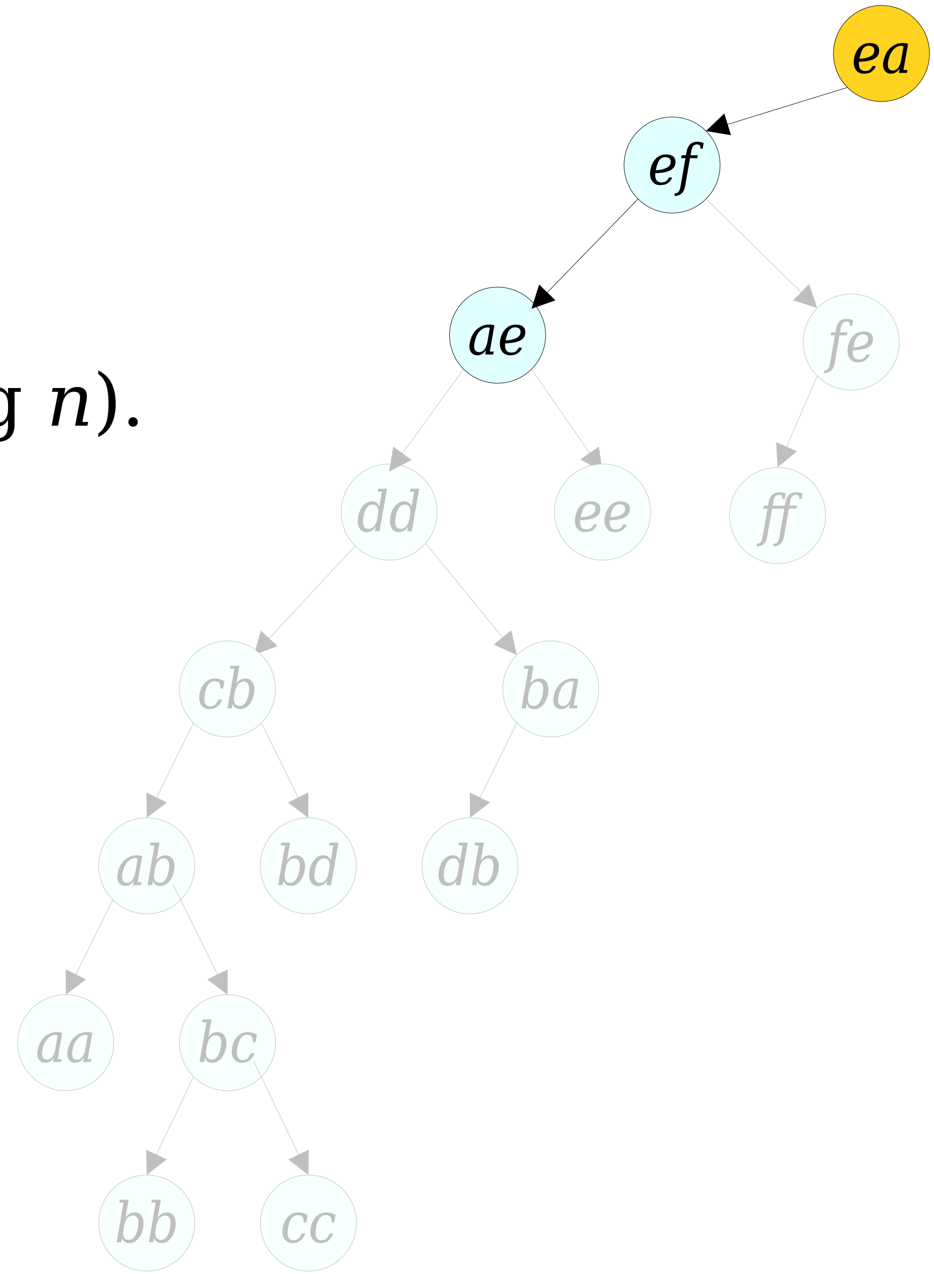


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log *n*).

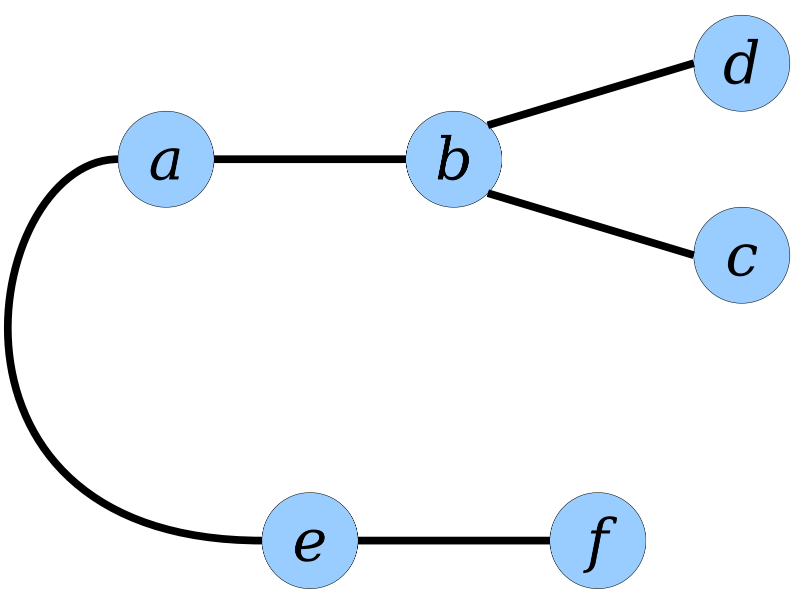


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

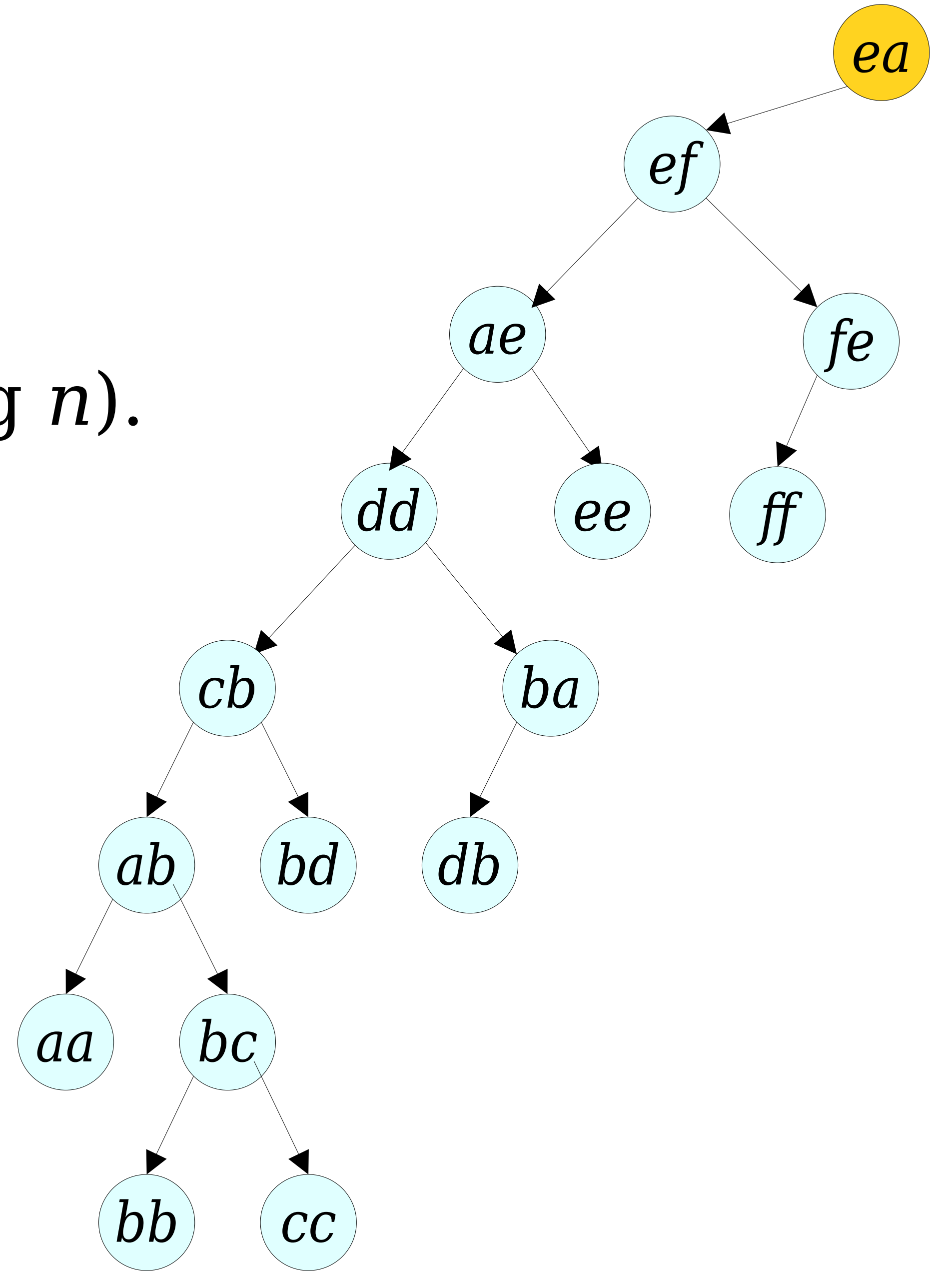


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).

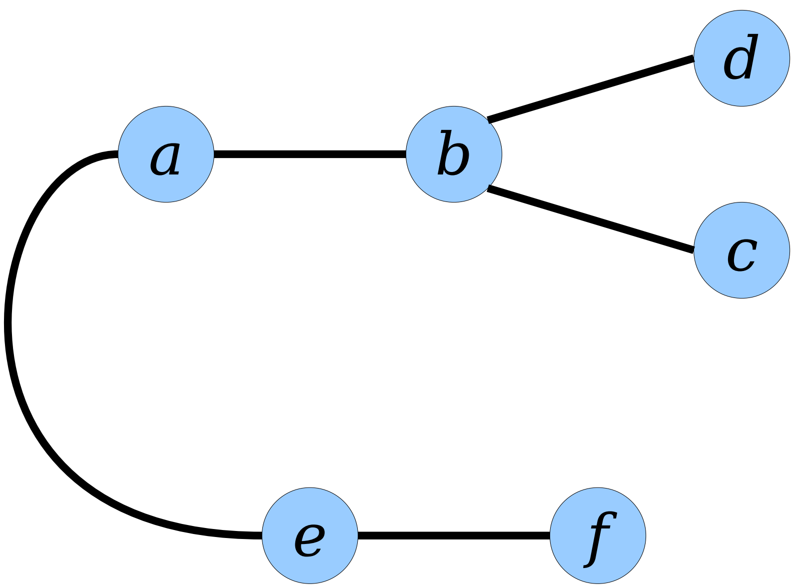


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea

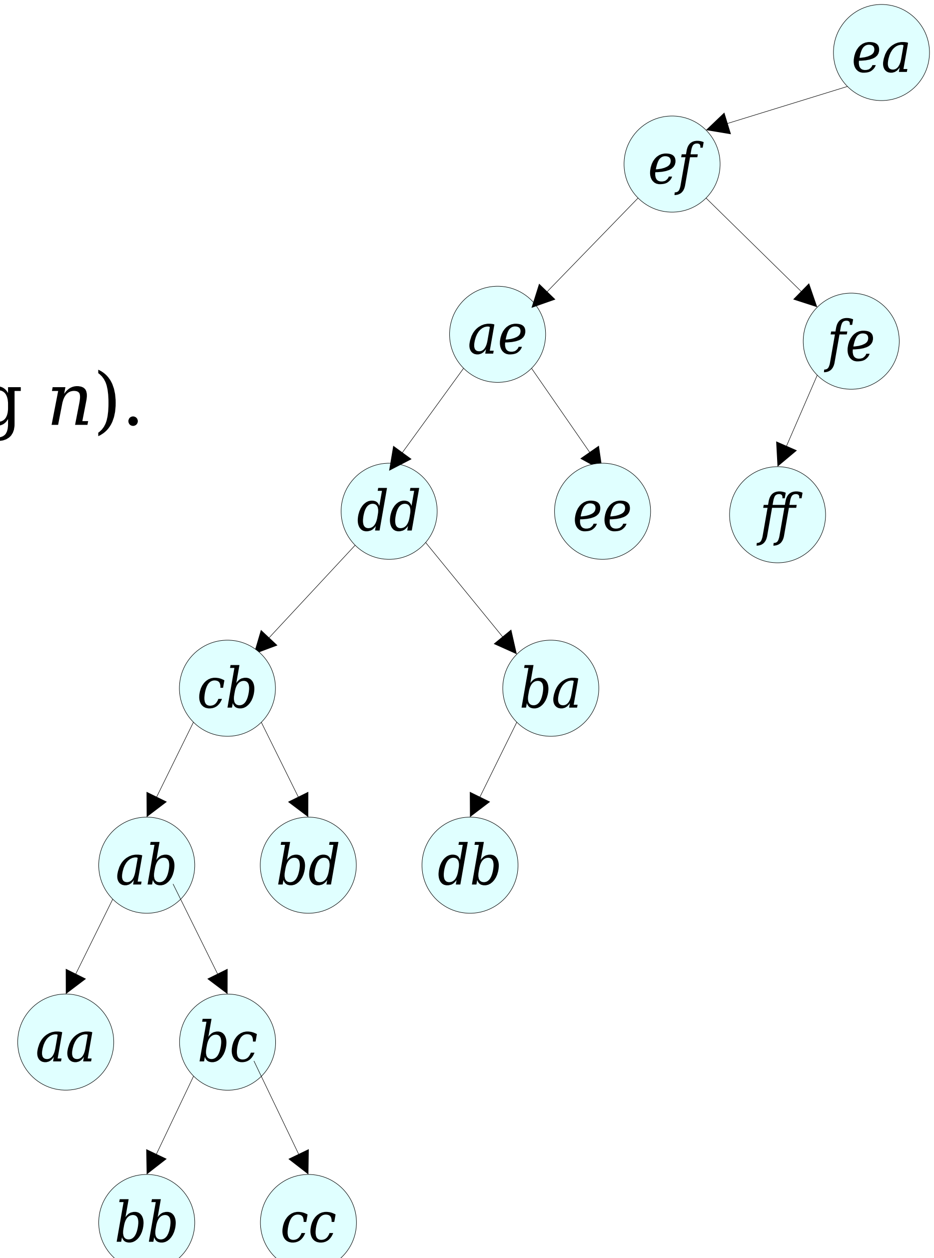


Binary Search(less) Trees

- ***Answer:*** Use splay trees! They support these operations in amortized time O(log n).

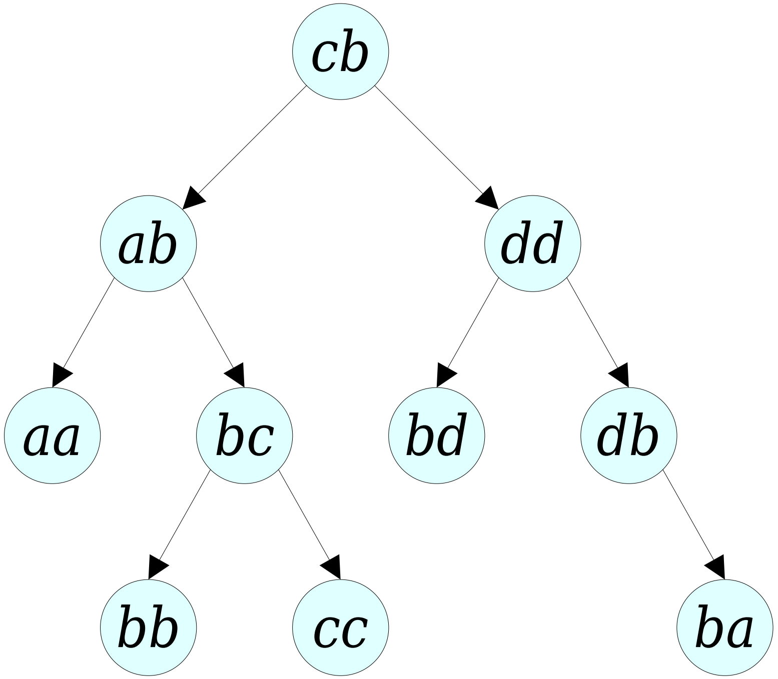


aa ab bb bc cc cb bd dd db ba …
… ae ee ef ff fe ea



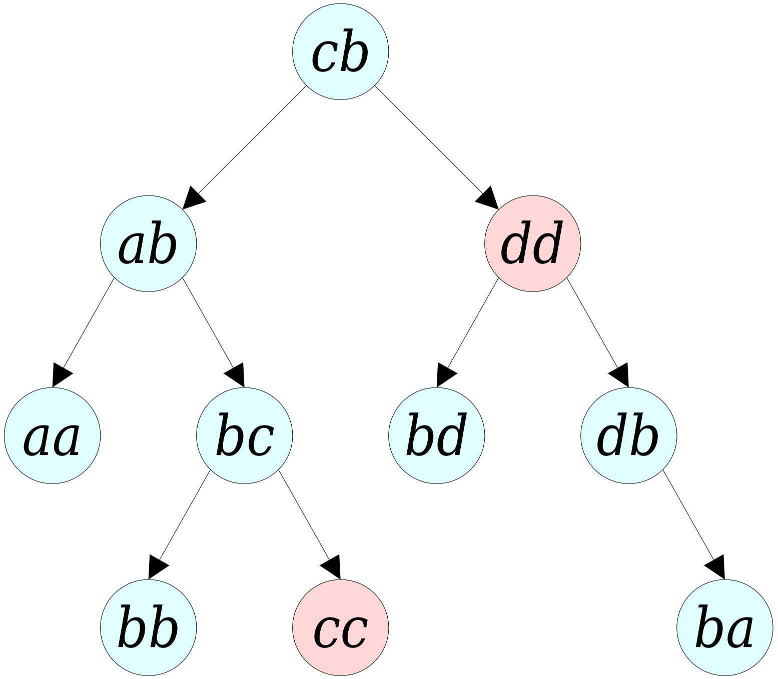
Euler Tour Trees

- To answer ***are-connected***(x, y):



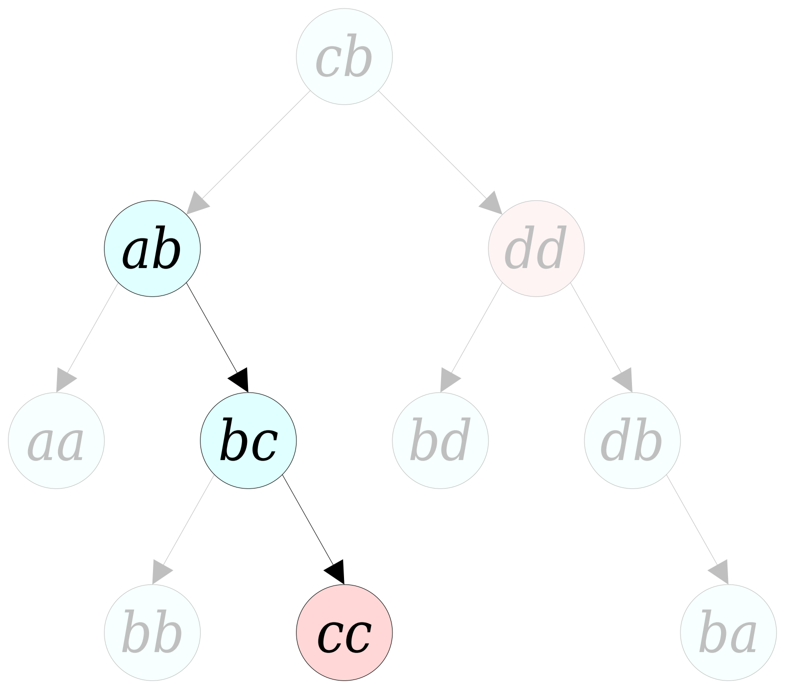
Euler Tour Trees

- To answer ***are-connected***(x, y):



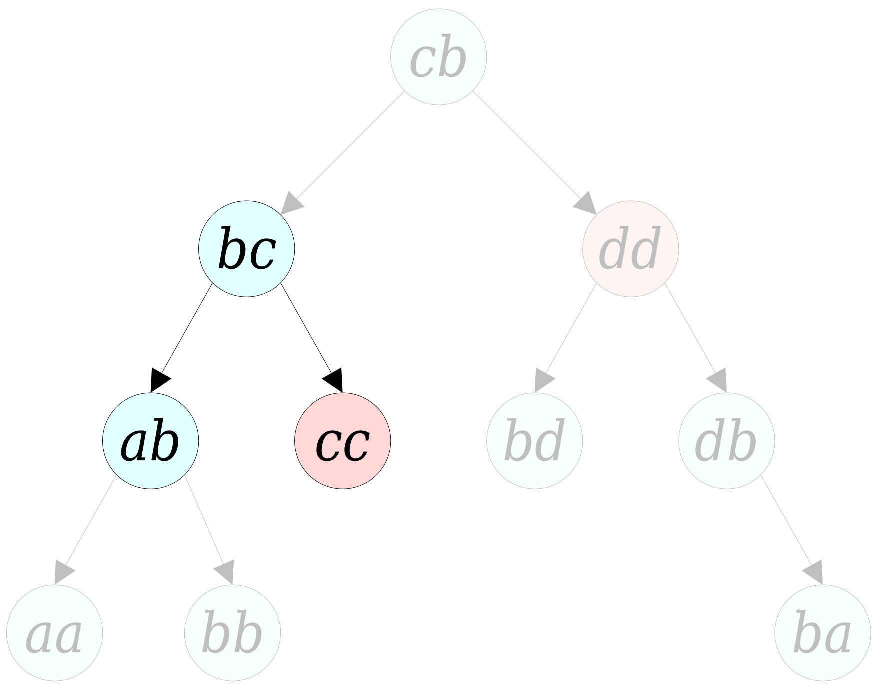
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.



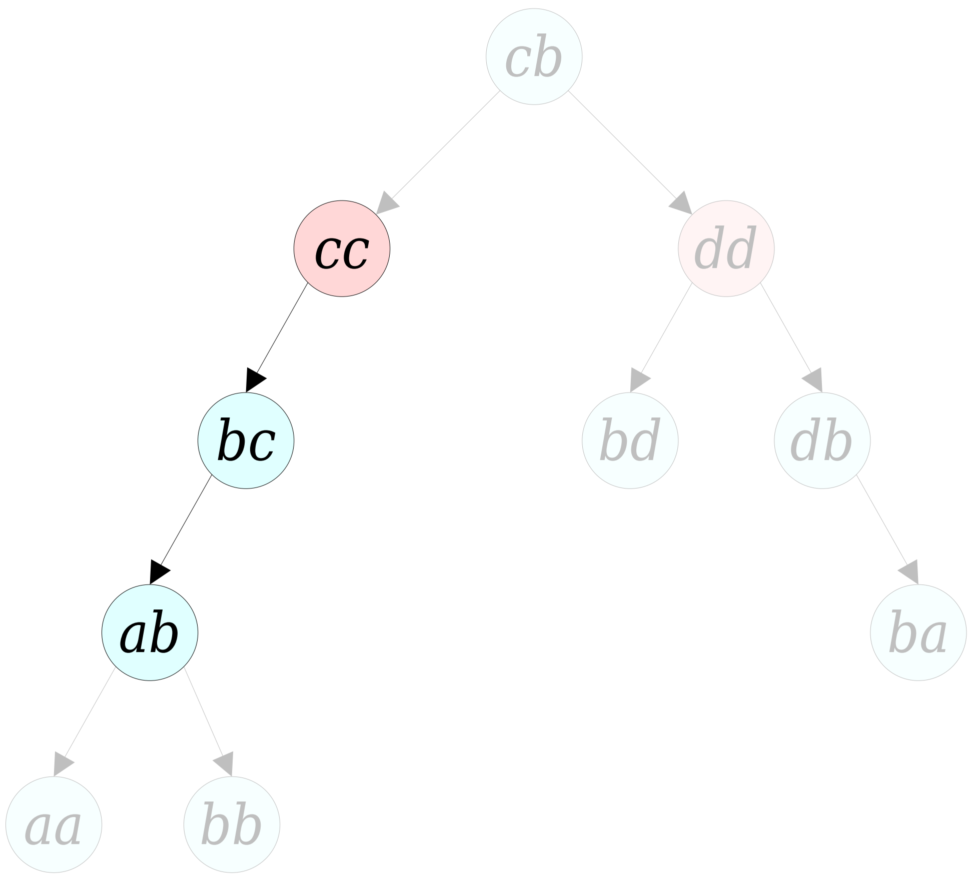
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.



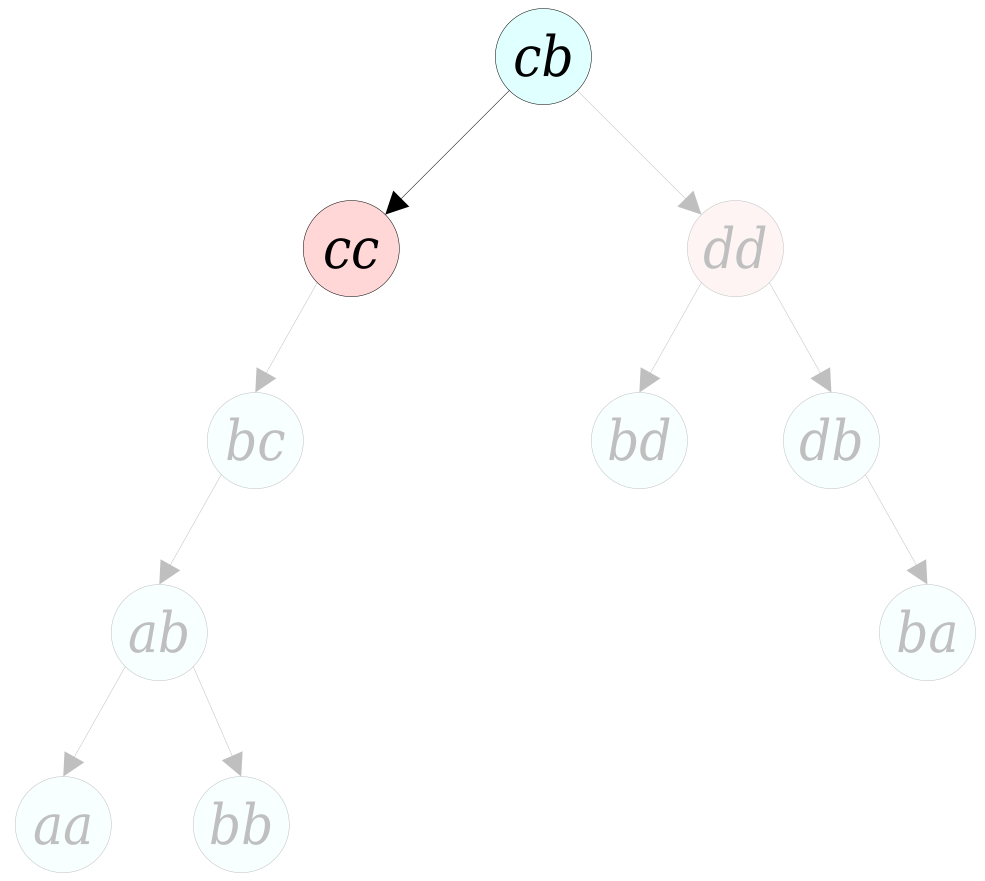
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.



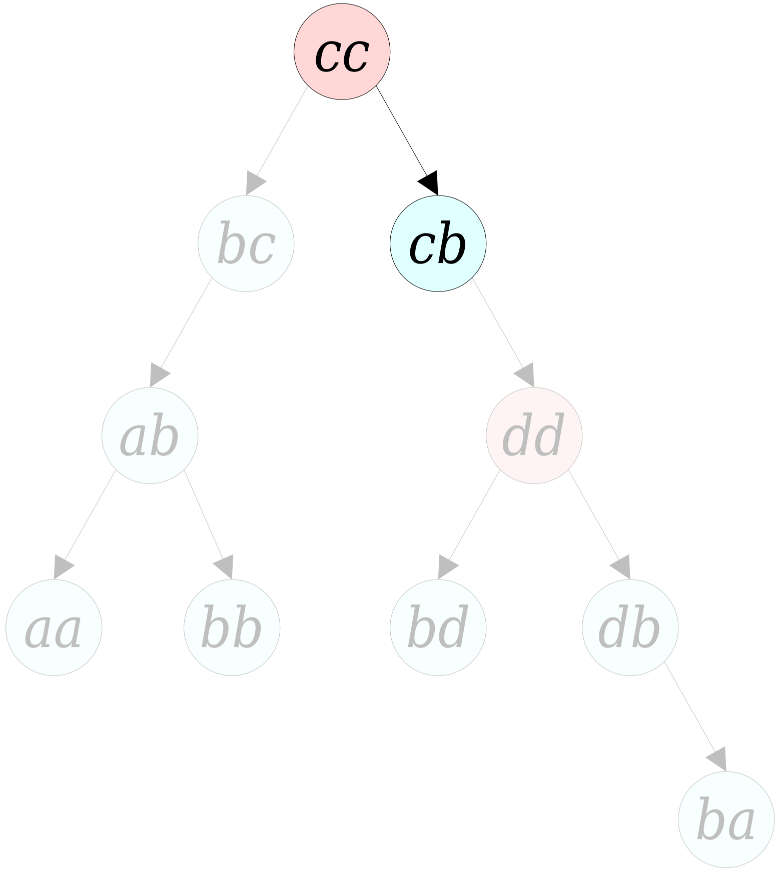
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.



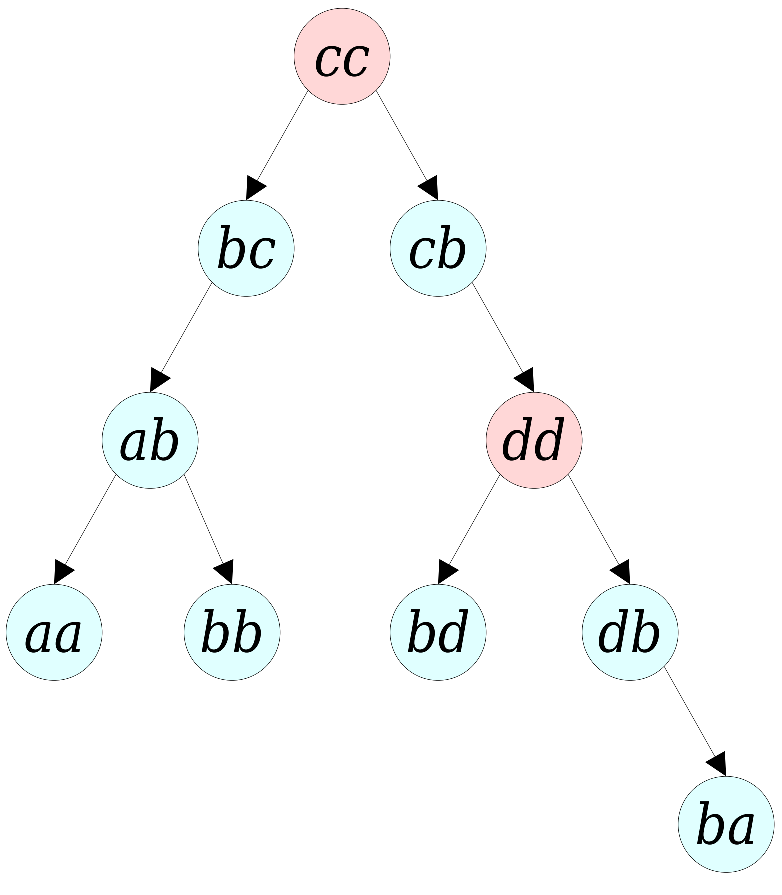
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.



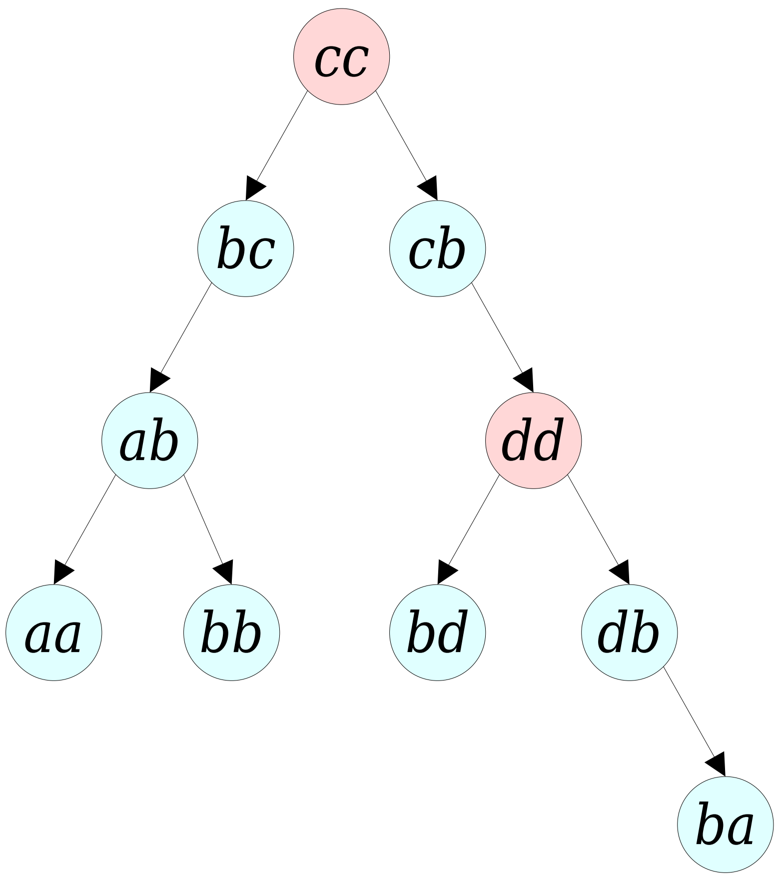
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.



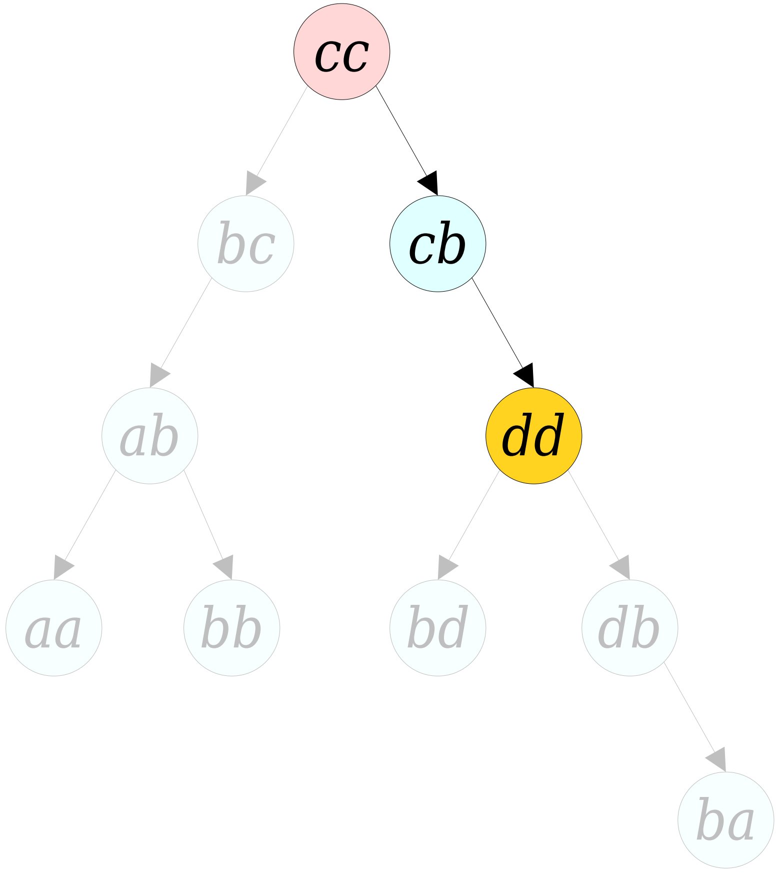
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.
  - Splay yy.



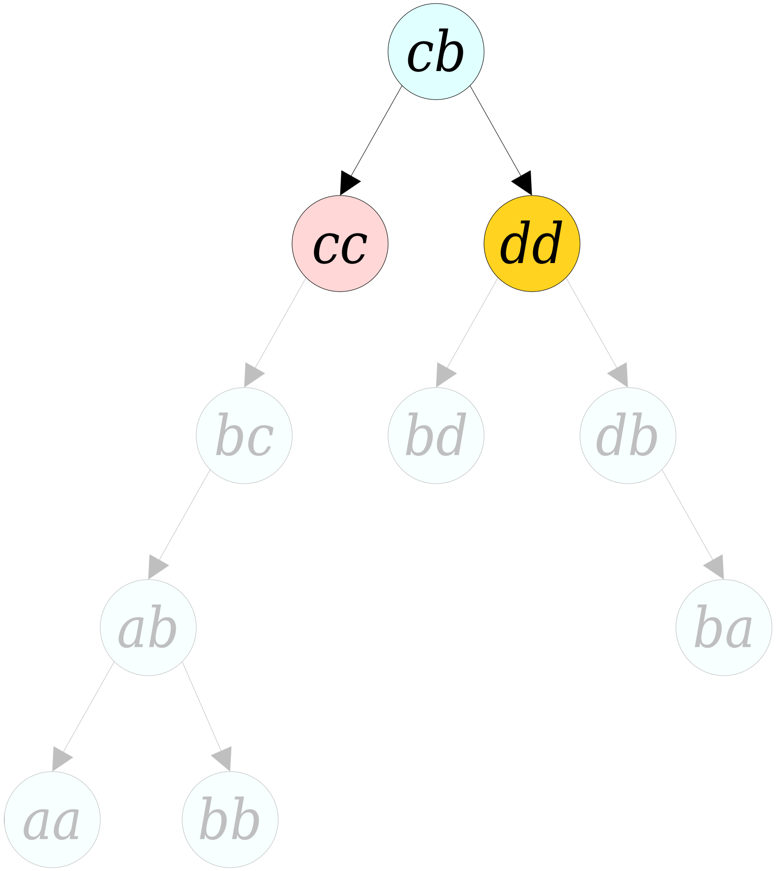
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.
  - Splay yy.



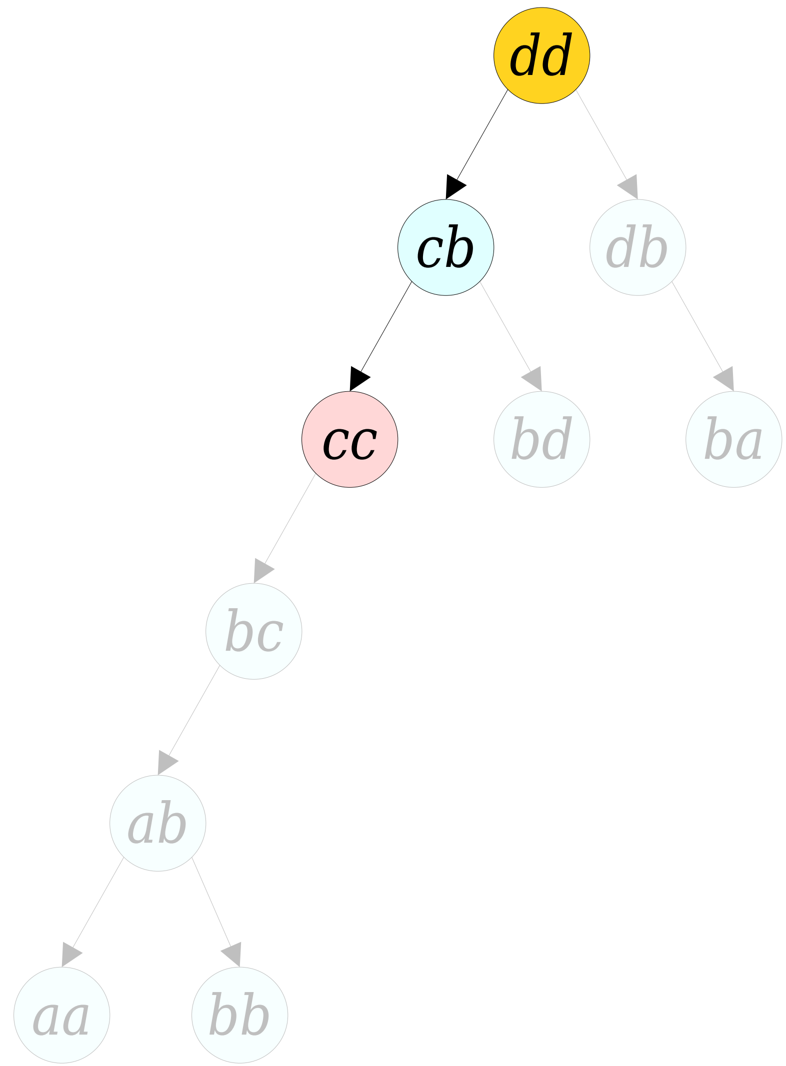
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.
  - Splay yy.



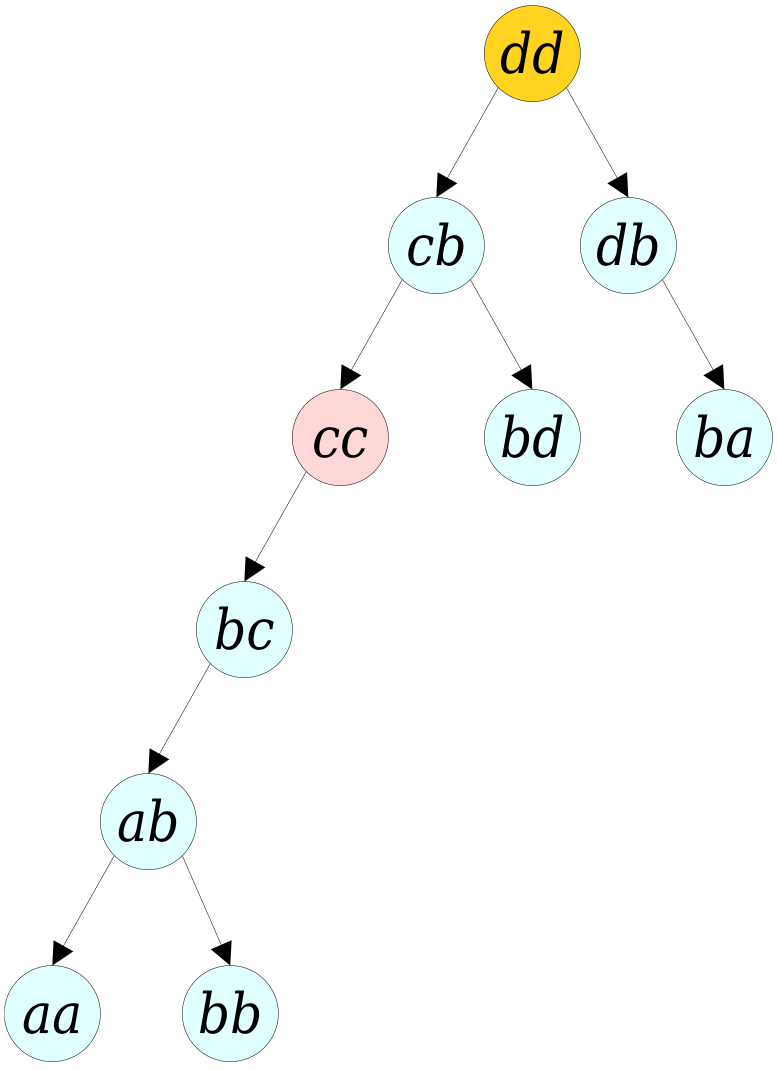
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.
  - Splay yy.



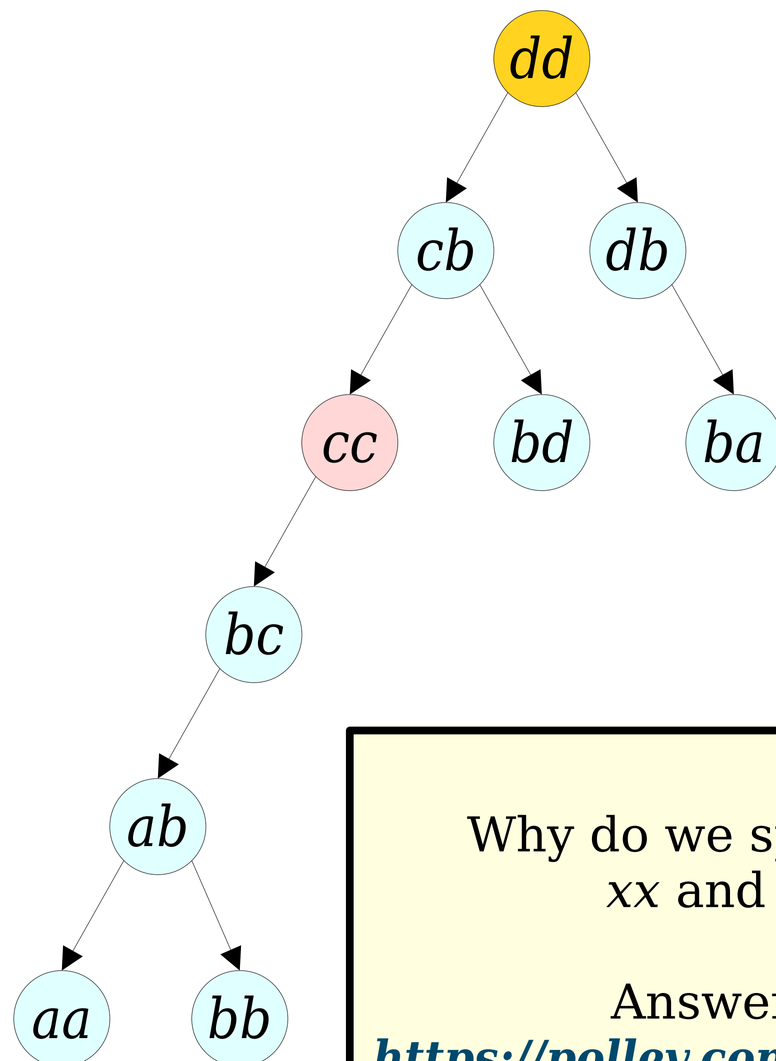
Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay xx.
  - Splay yy.



Euler Tour Trees

- To answer ***are-connected***(x, y):
  - Splay *xx*.
  - Splay *yy*.
  - Return whether *xx* was encountered on the second splay.
- Amortized cost: **O(log *n*)**.

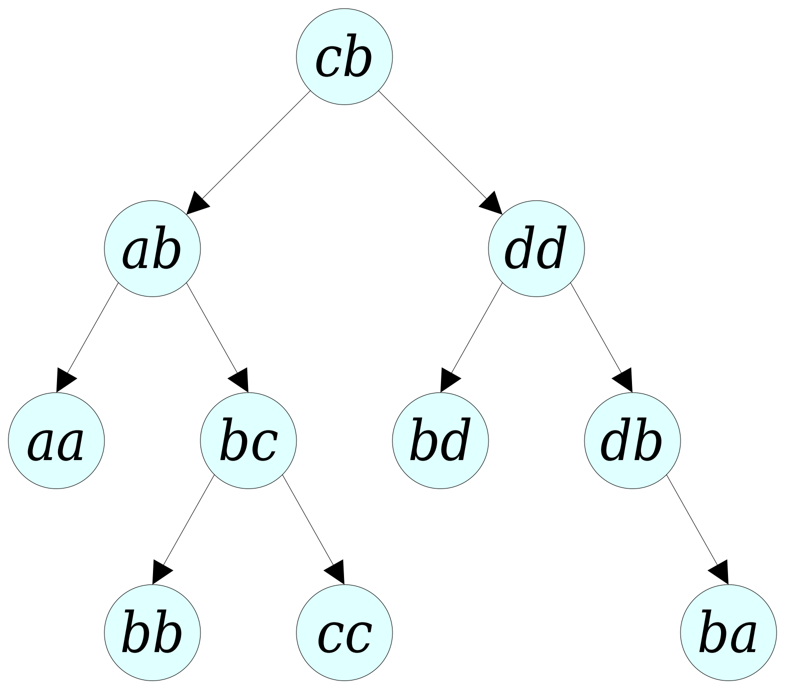


Why do we splay both *xx* and *yy*?

Answer at ***https://pollev.com/cs166spr23***

Euler Tour Trees

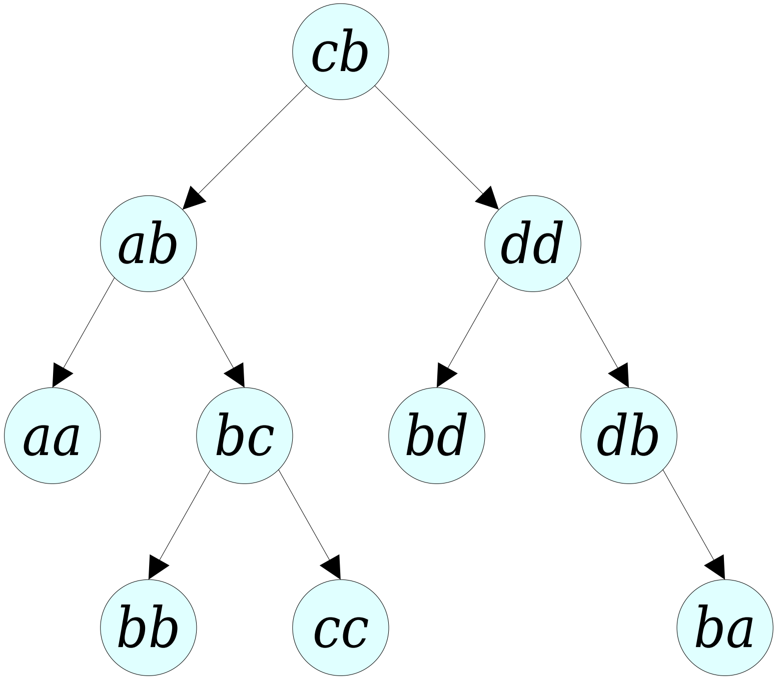
- To ***reroot***(x):



aa ab bb bc cc cb bd dd db ba

Euler Tour Trees

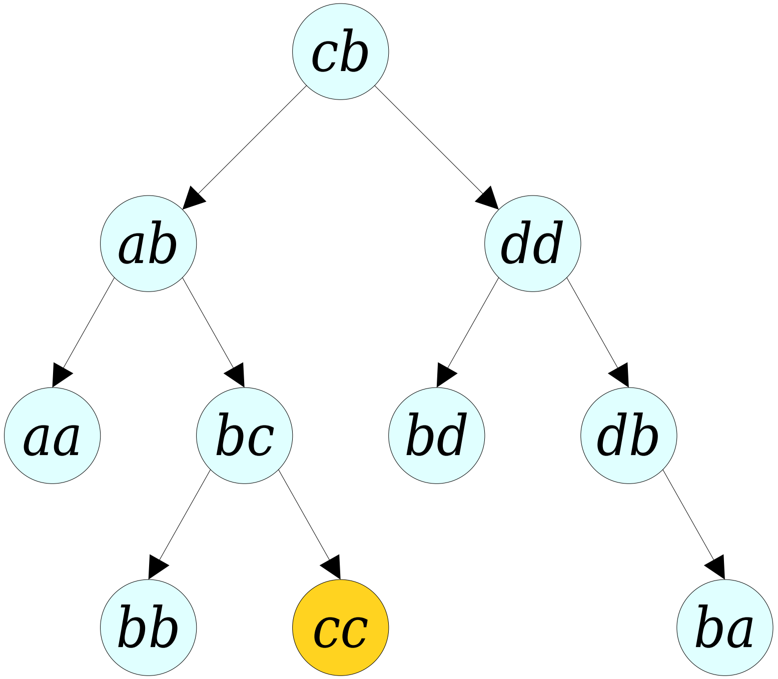
- To ***reroot***(x):
  - Splay xx.



aa ab bb bc cc cb bd dd db ba

Euler Tour Trees

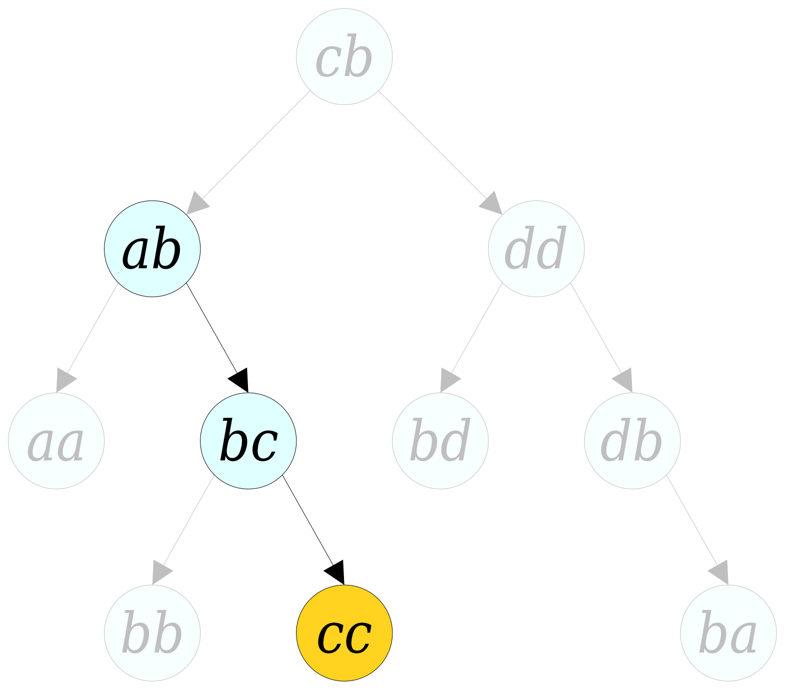
- To ***reroot***(x):
  - Splay xx.



aa ab bb bc cc cb bd dd db ba

Euler Tour Trees

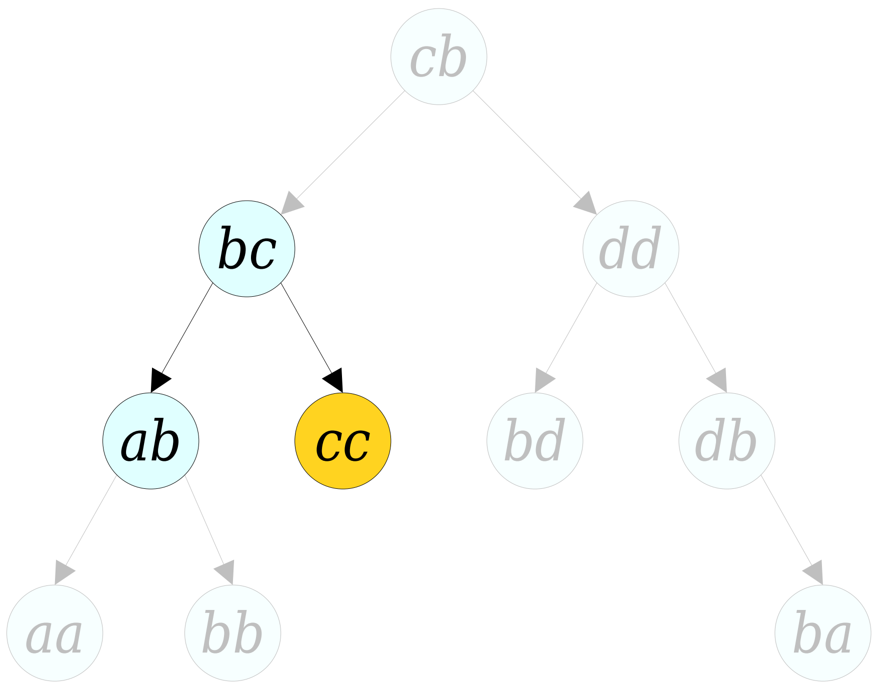
- To ***reroot***(x):
  - Splay xx.



aa ab bb bc cc cb bd dd db ba

Euler Tour Trees

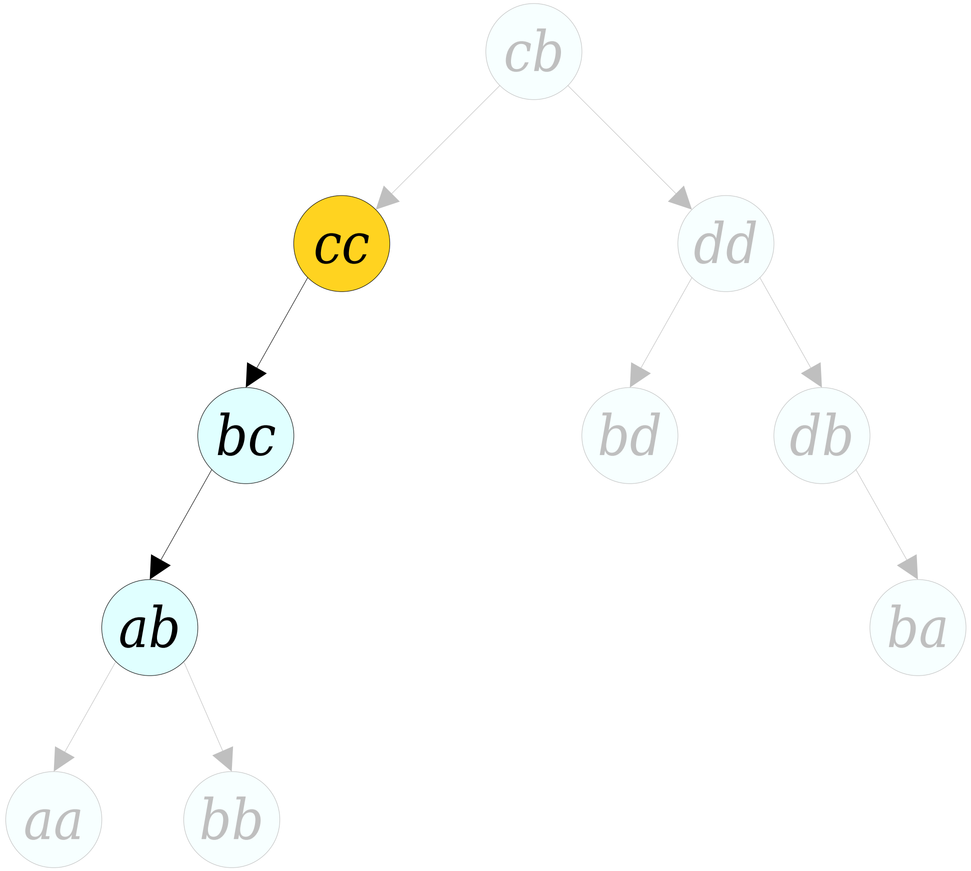
- To ***reroot***(x):
  - Splay xx.



aa ab bb bc cc cb bd dd db ba

Euler Tour Trees

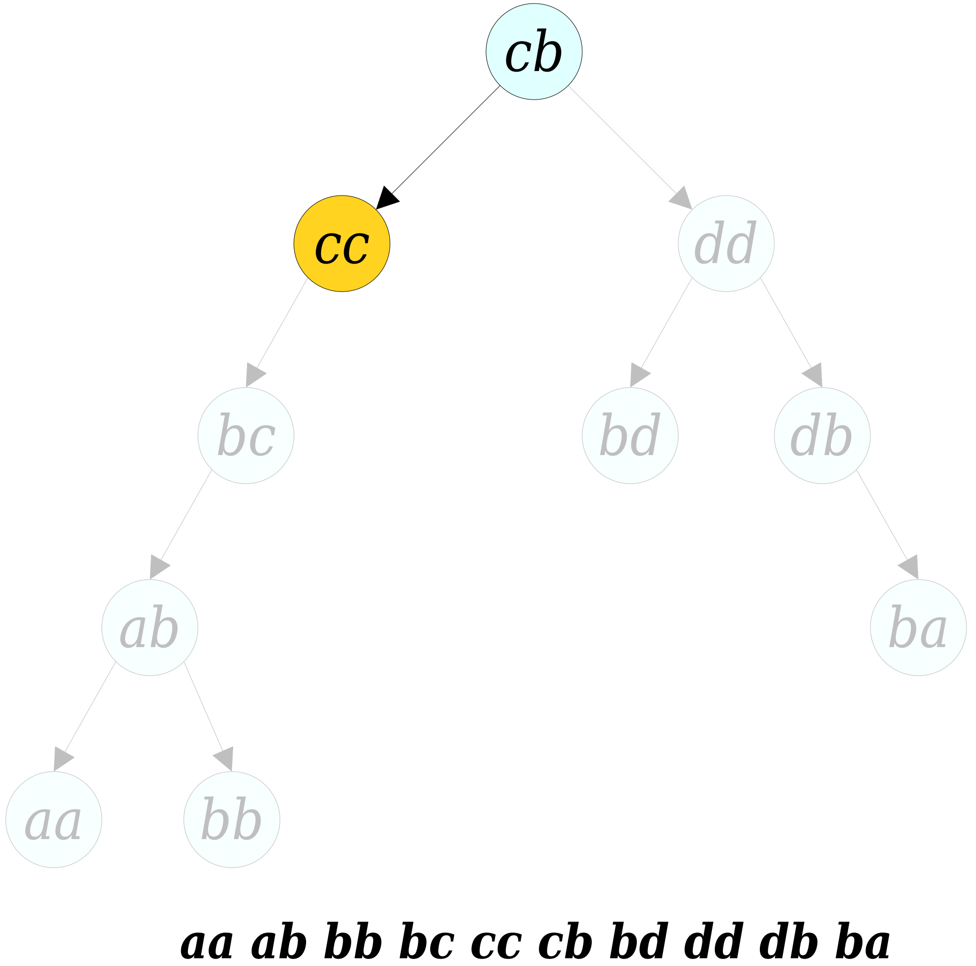
- To ***reroot***(x):
  - Splay xx.



aa ab bb bc cc cb bd dd db ba

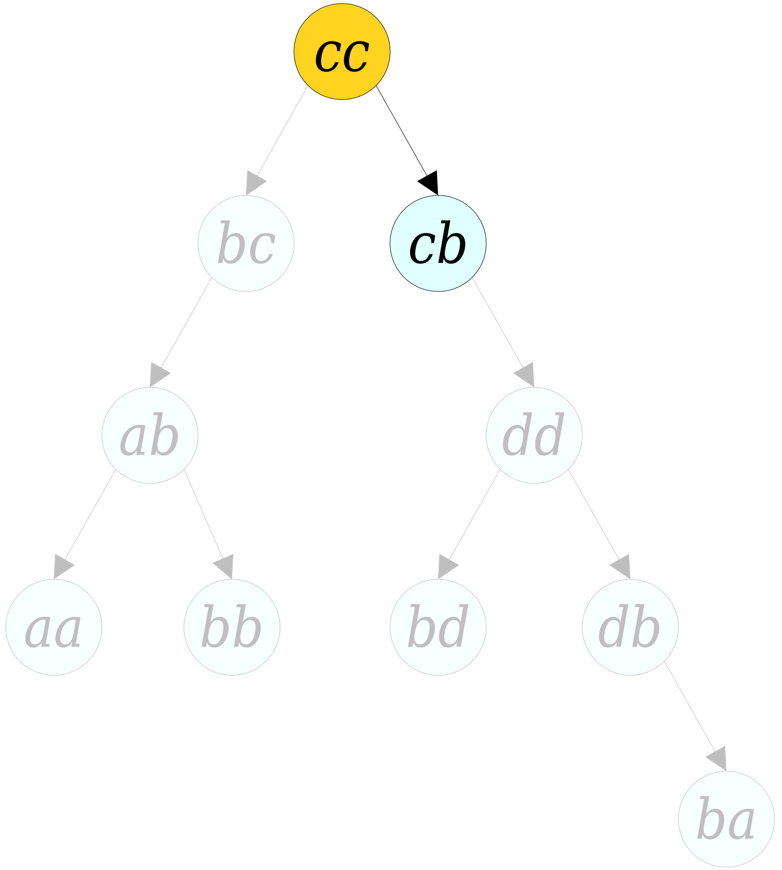
Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.



Euler Tour Trees

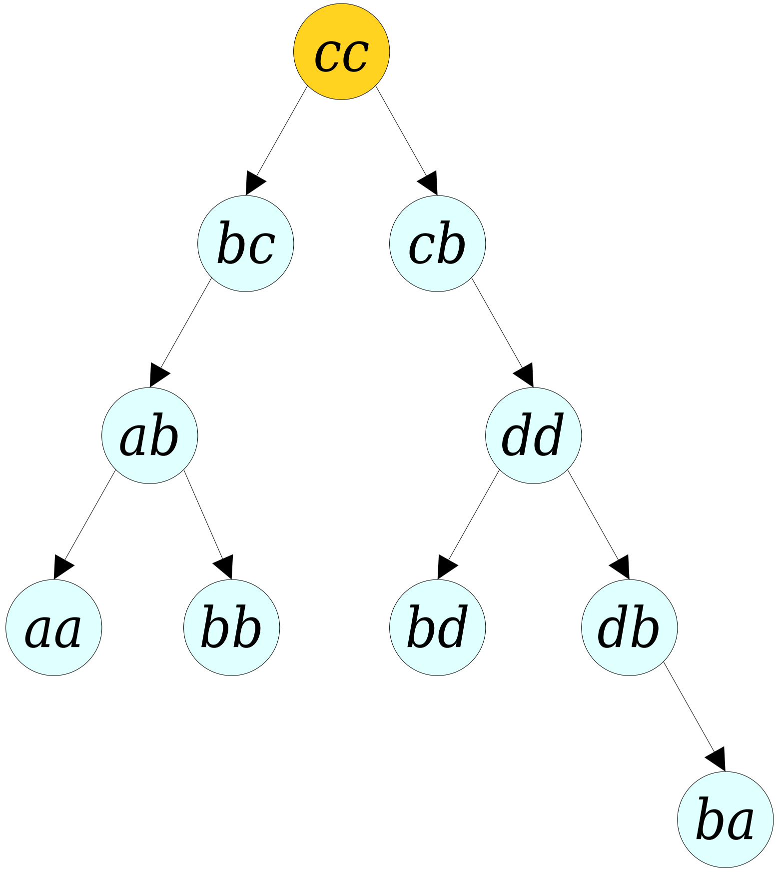
- To ***reroot***(x):
  - Splay xx.



aa ab bb bc cc cb bd dd db ba

Euler Tour Trees

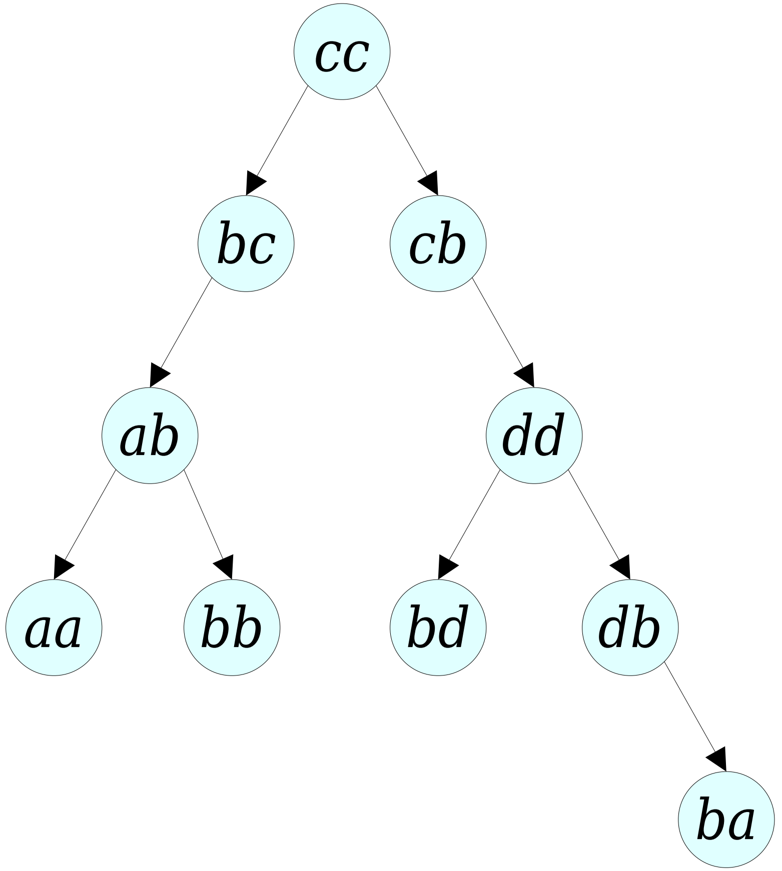
- To ***reroot***(x):
  - Splay xx.



aa ab bb bc cc cb bd dd db ba

Euler Tour Trees

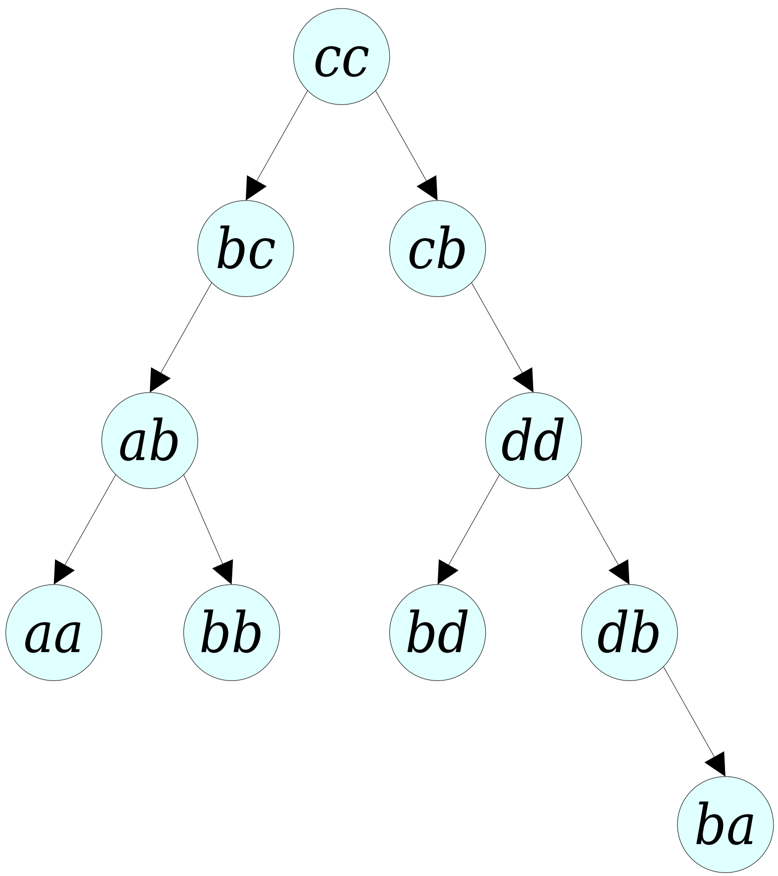
- To ***reroot***(x):
  - Splay xx.



aa ab bb bc cc cb bd dd db ba

Euler Tour Trees

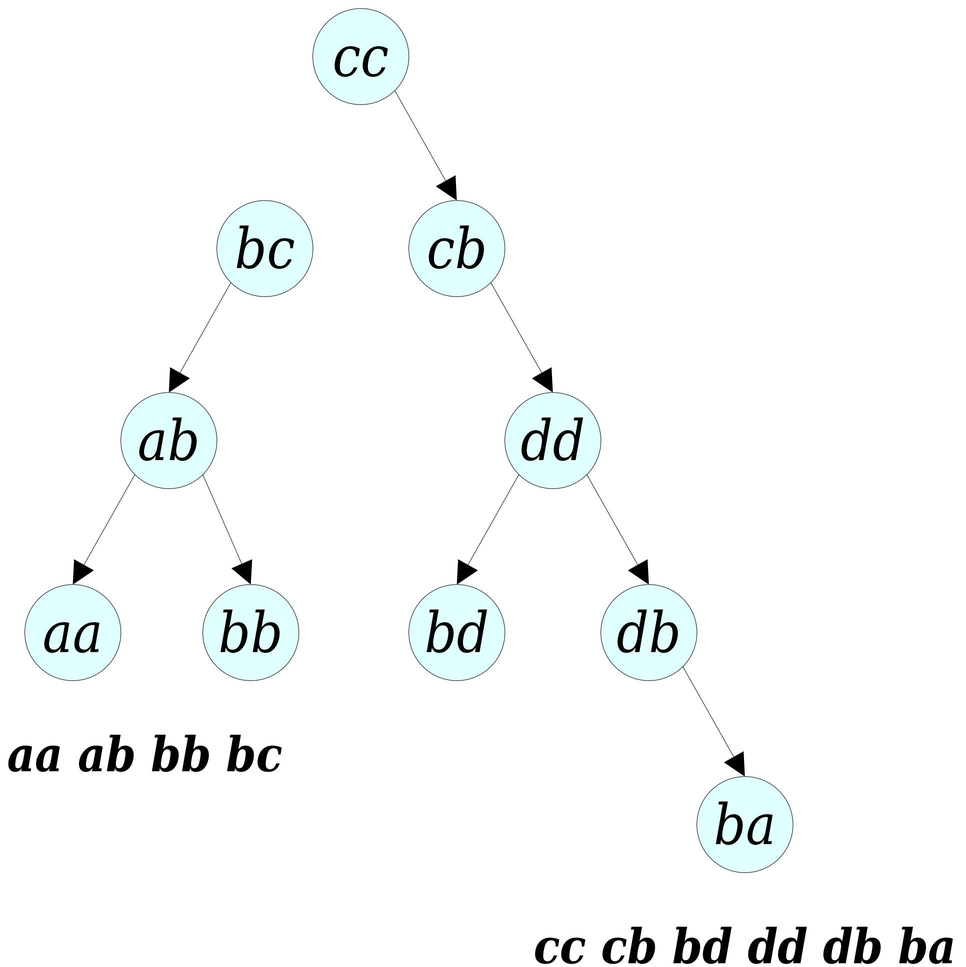
- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.



aa ab bb bc cc cb bd dd db ba

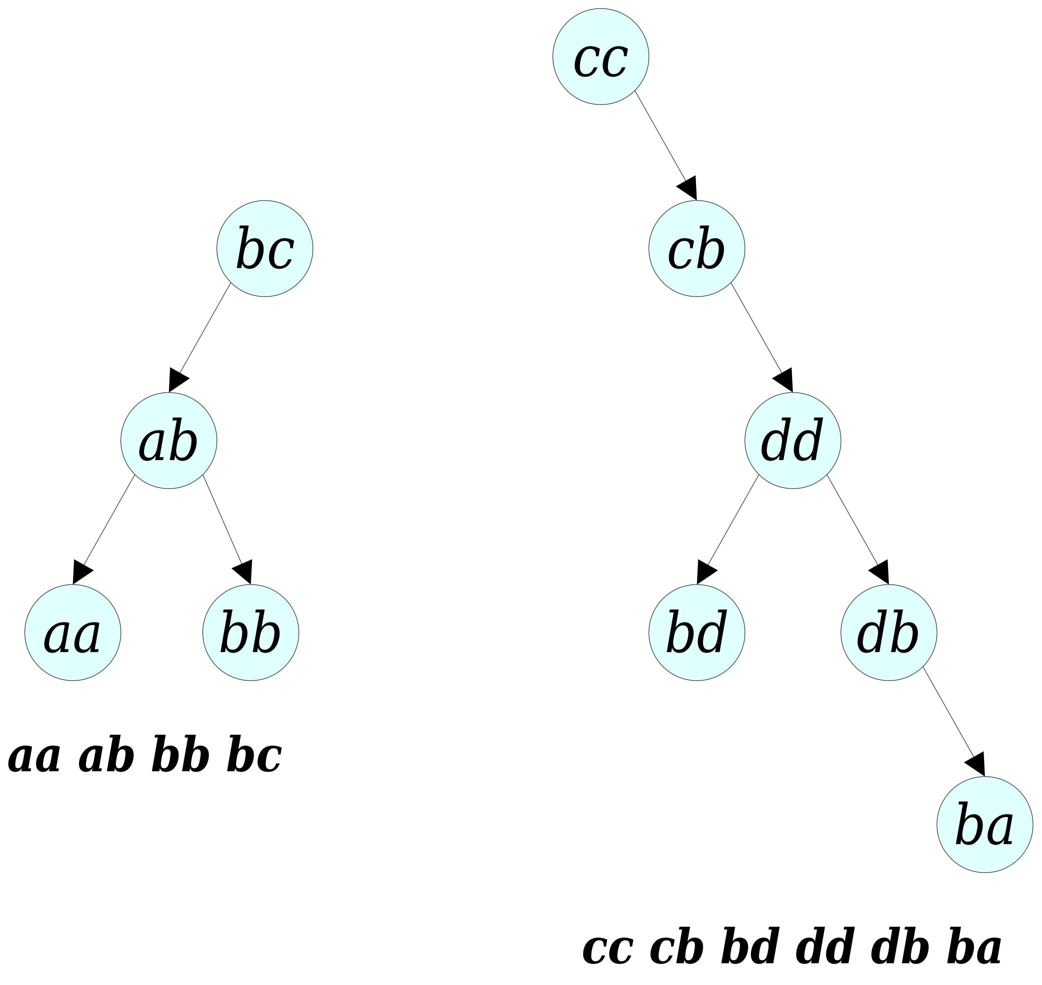
Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.



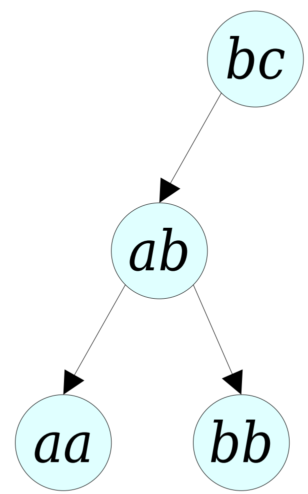
Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.

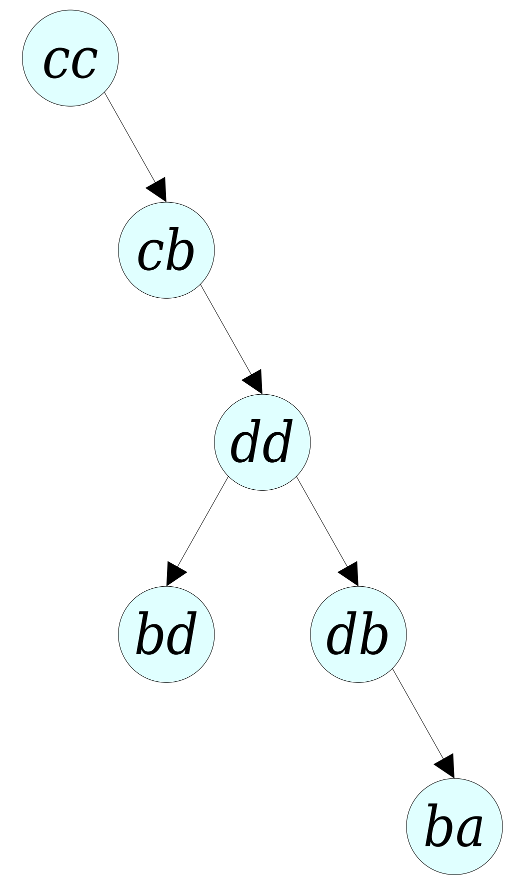


Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.



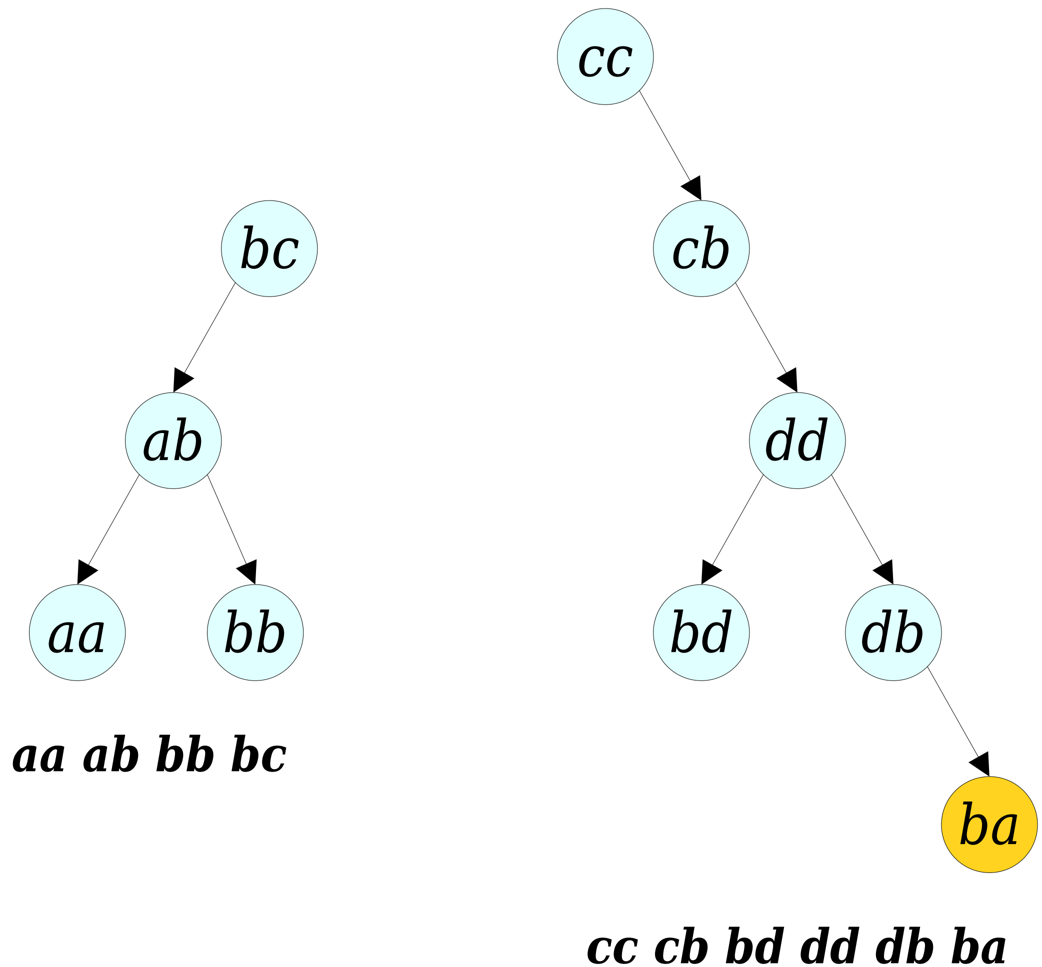
aa ab bb bc



cc cb bd dd db ba

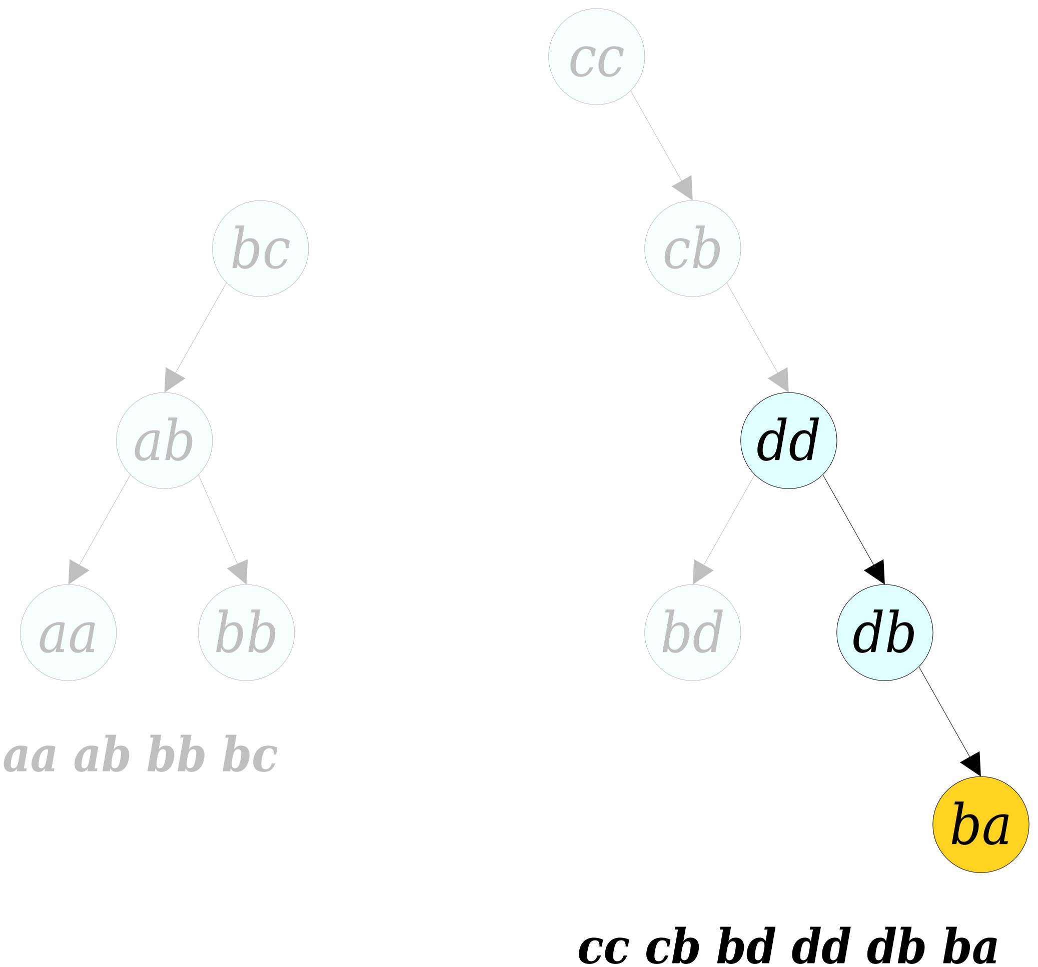
Euler Tour Trees

- To ***reroot***(x):
  - Splay *xx*.
  - Disconnect *xx*’s left child tree *T*.
  - Splay the rightmost node in *xx*’s subtree.



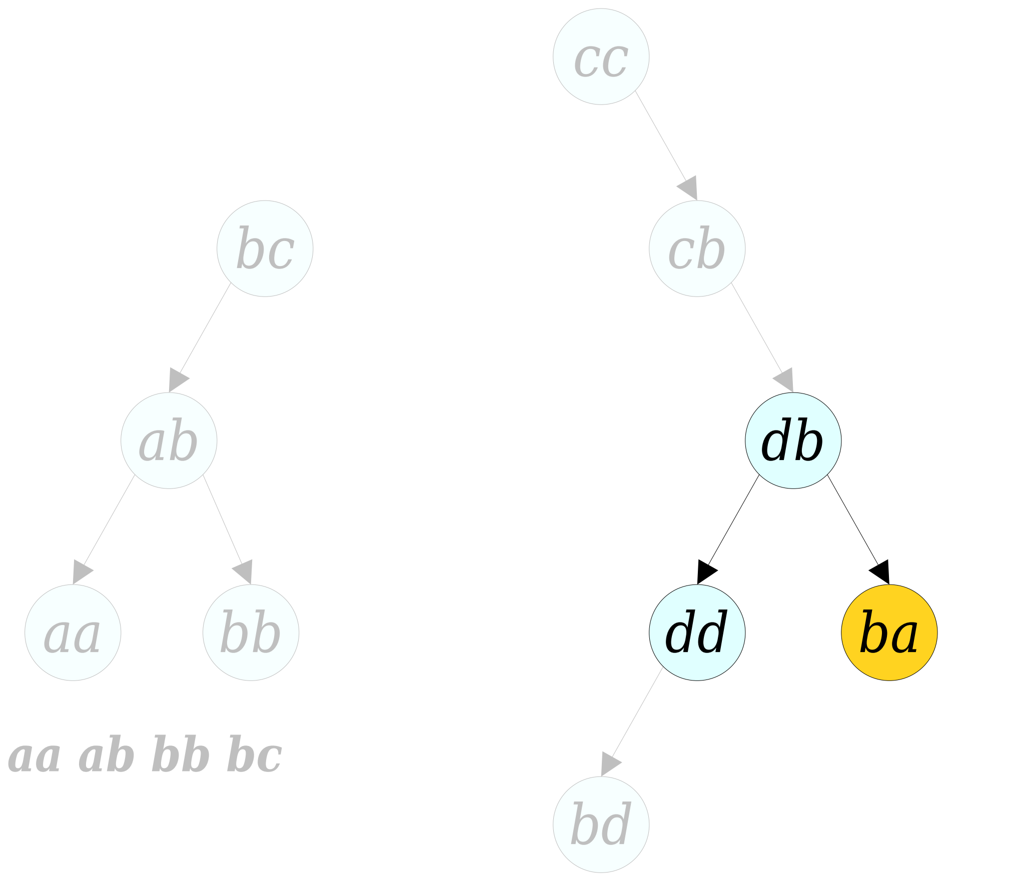
Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.



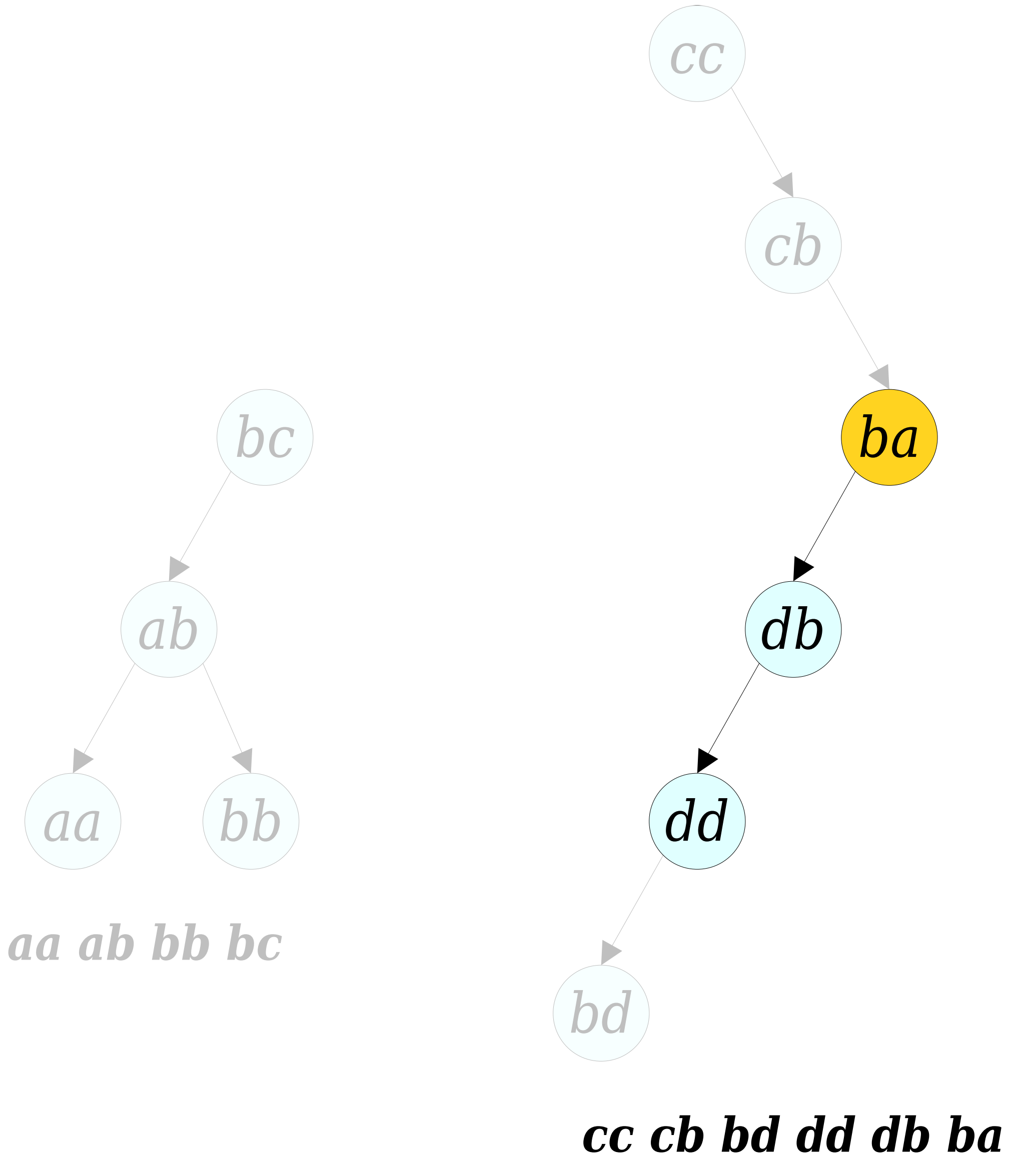
Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.



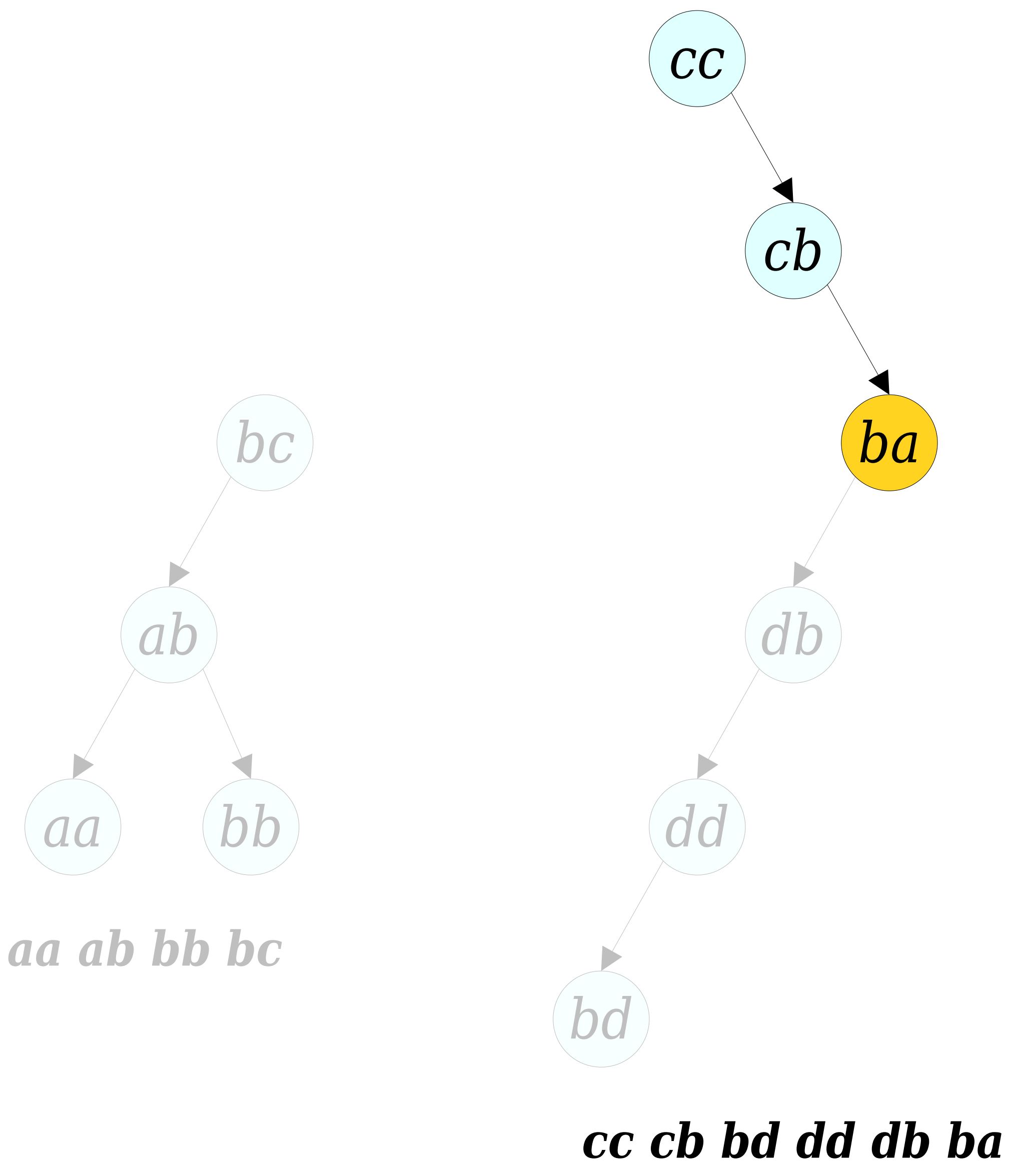
Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.



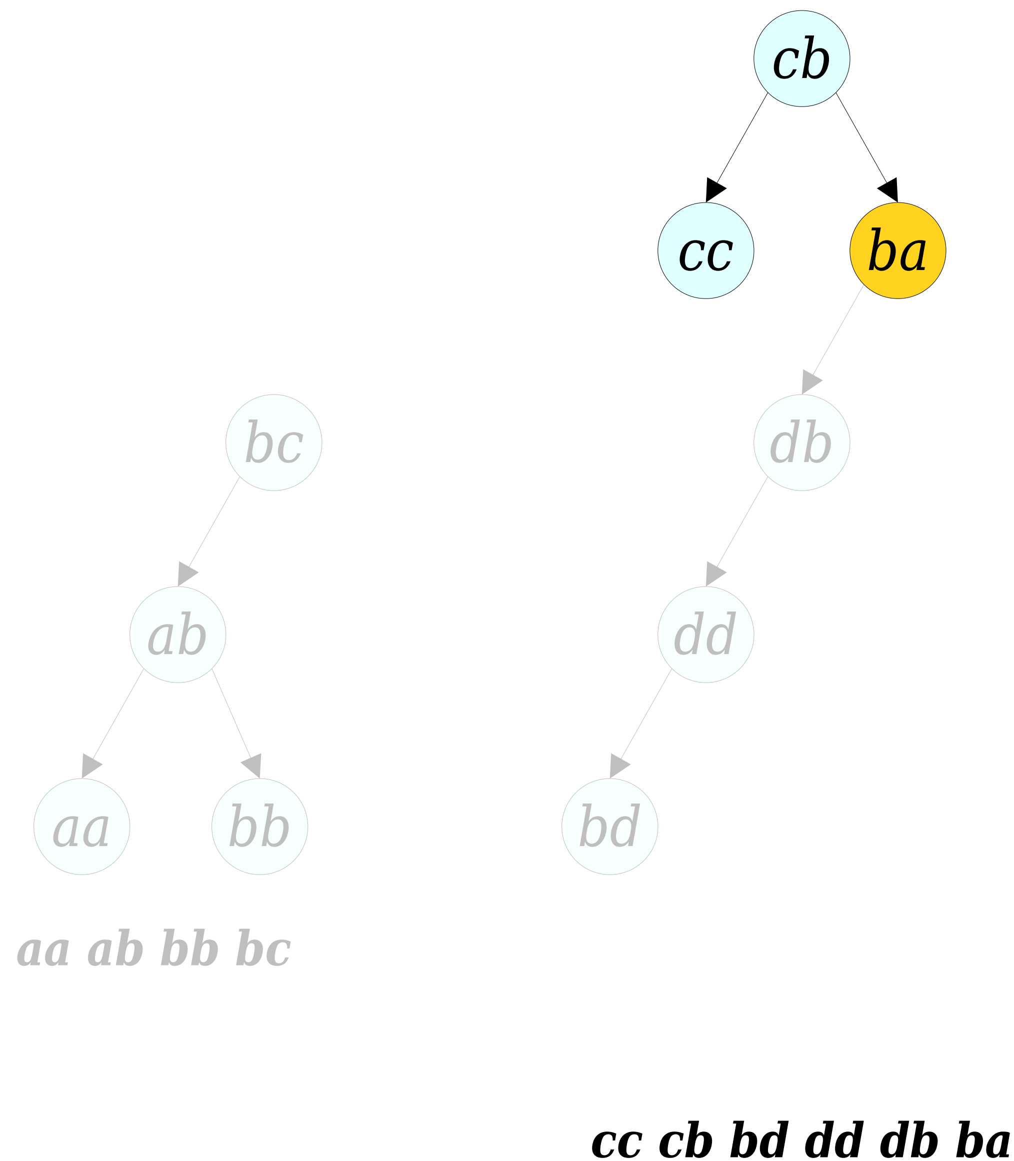
Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.



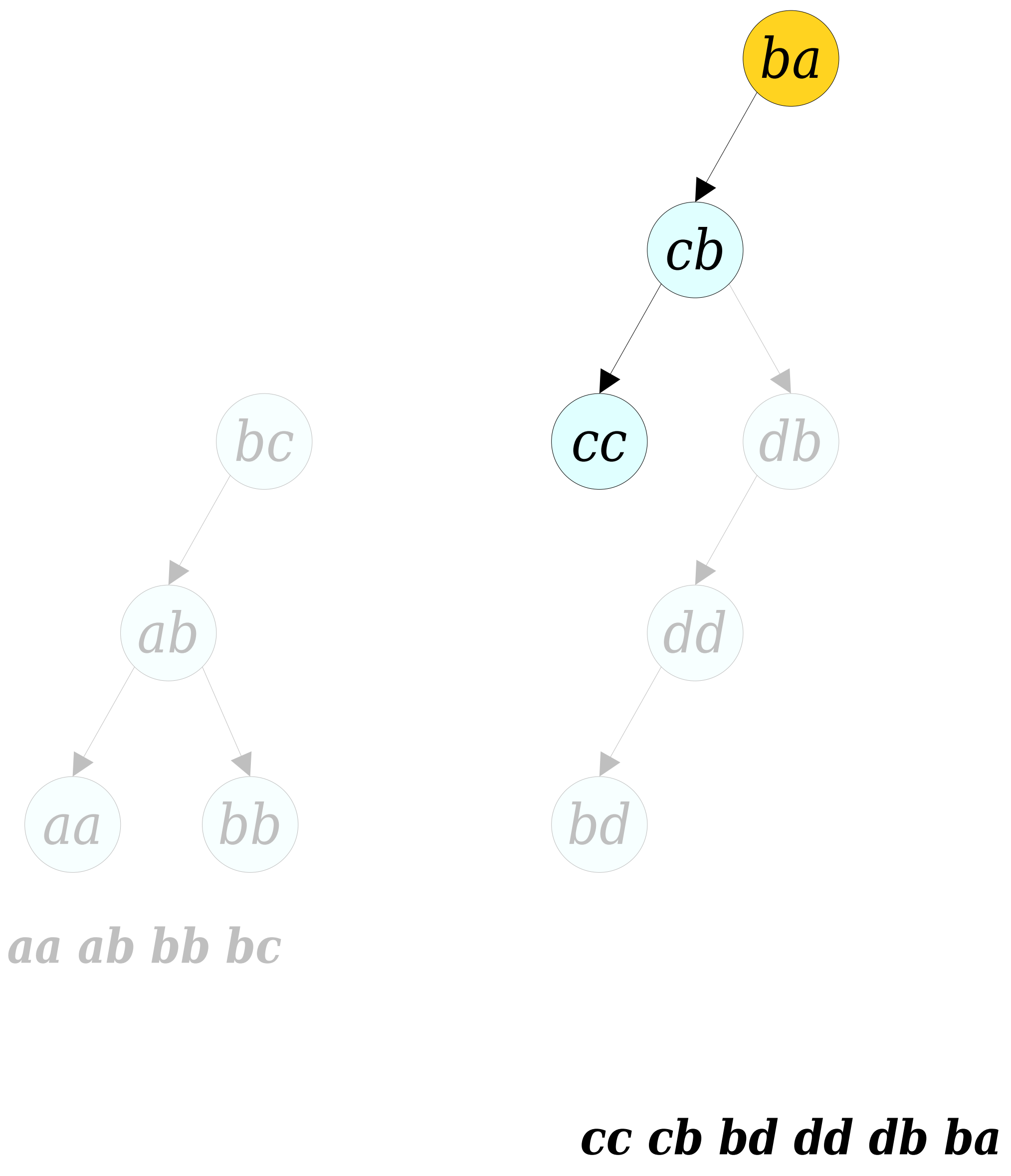
Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.



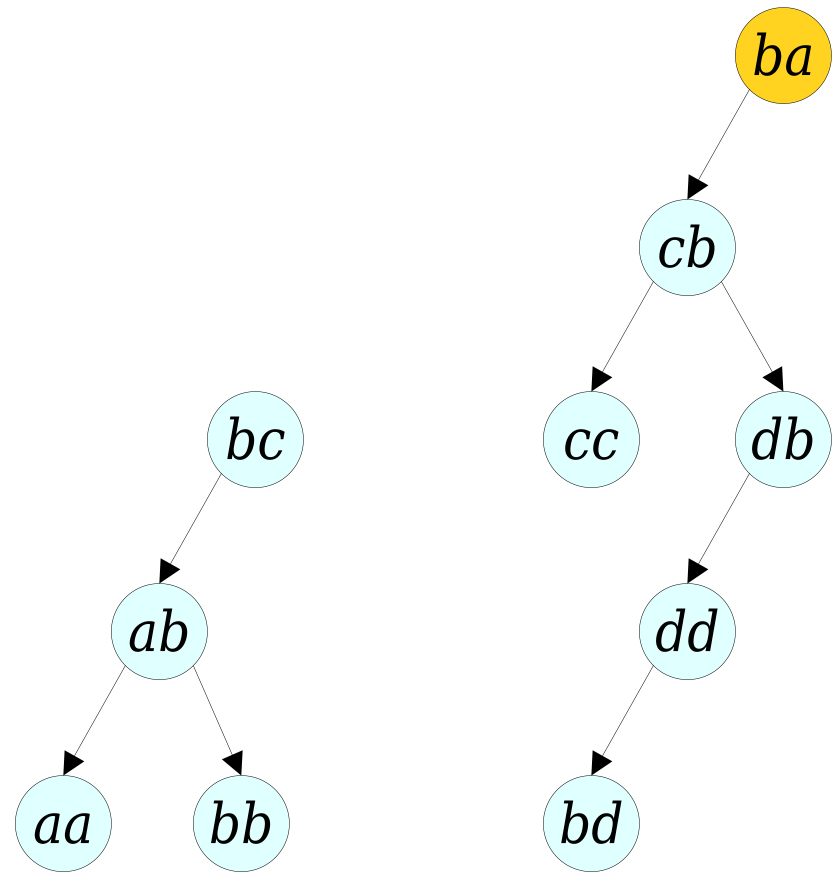
Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.



Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx’s left child tree T.
  - Splay the rightmost node in xx’s subtree.

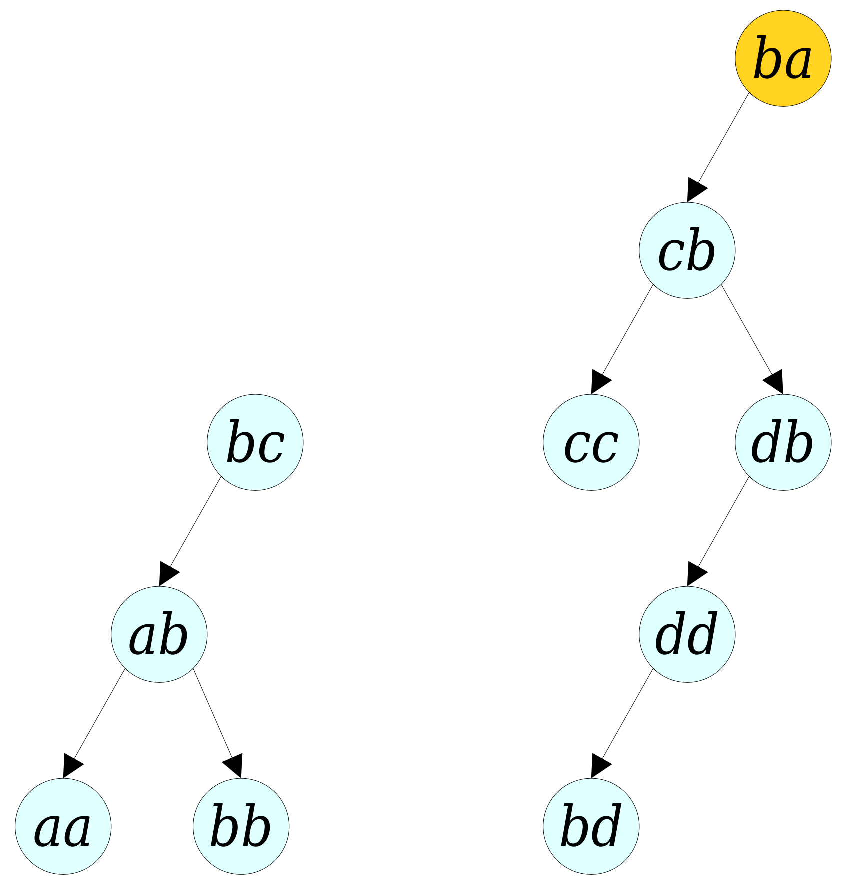


aa ab bb bc

cc cb bd dd db ba

Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.
  - Make T the right child of the root.

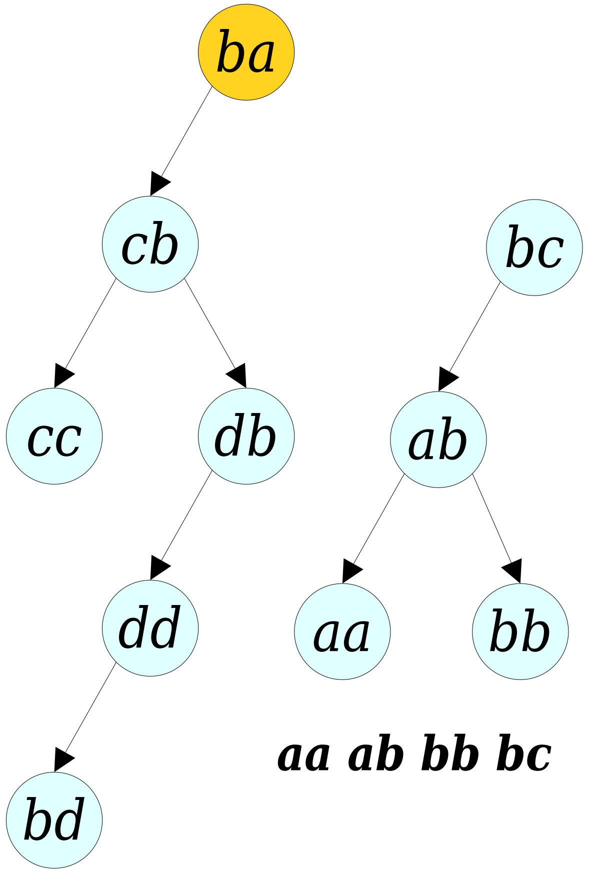


aa ab bb bc

cc cb bd dd db ba

Euler Tour Trees

- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.
  - Make T the right child of the root.

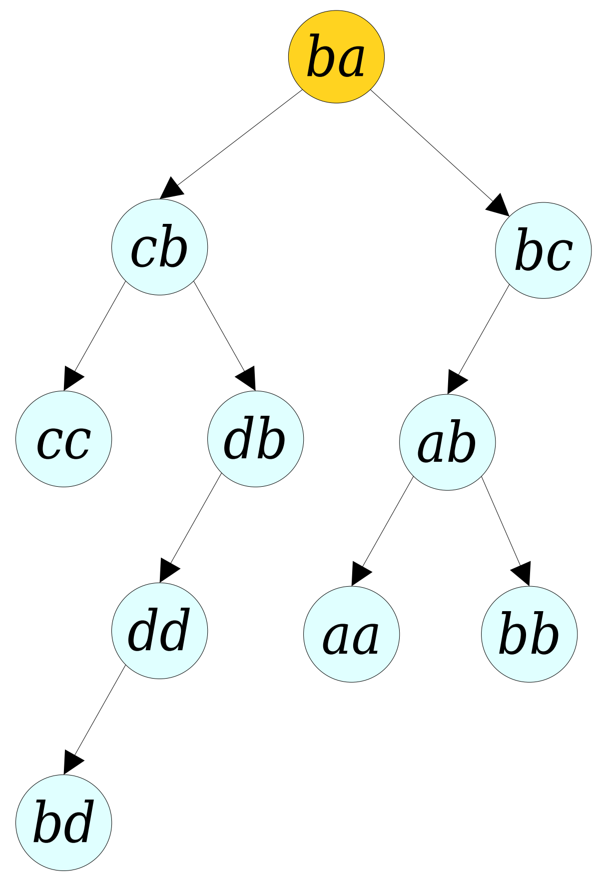


aa ab bb bc

cc cb bd dd db ba

Euler Tour Trees

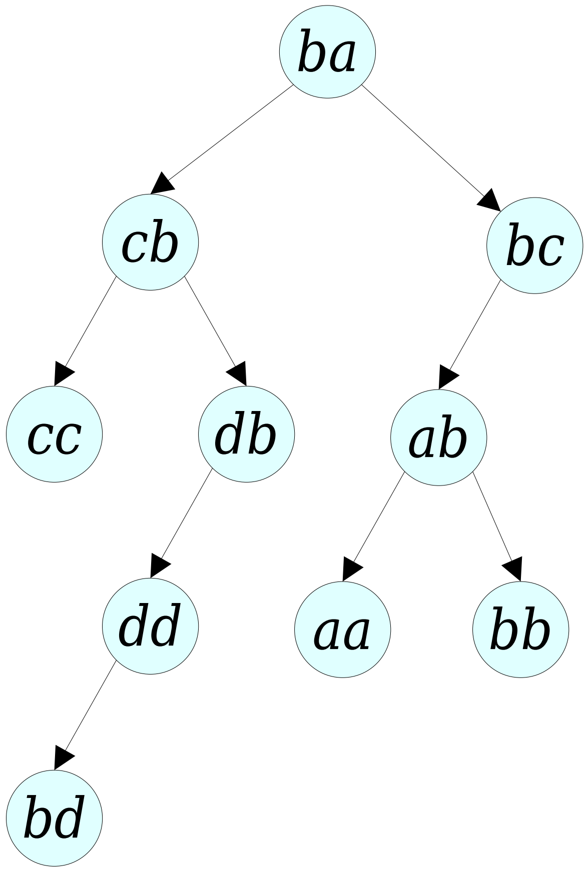
- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.
  - Make T the right child of the root.



cc cb bd dd db ba aa ab bb bc

Euler Tour Trees

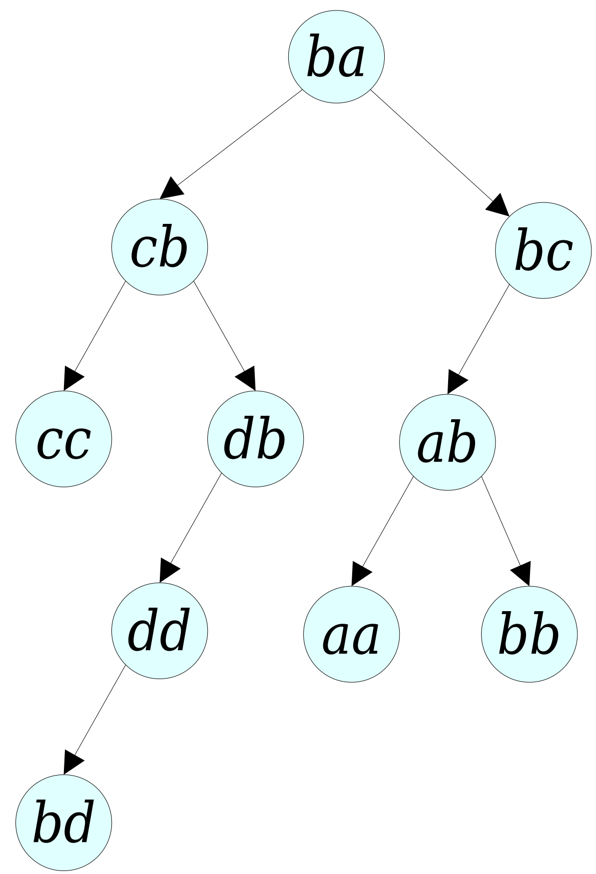
- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.
  - Make T the right child of the root.



cc cb bd dd db ba aa ab bb bc

Euler Tour Trees

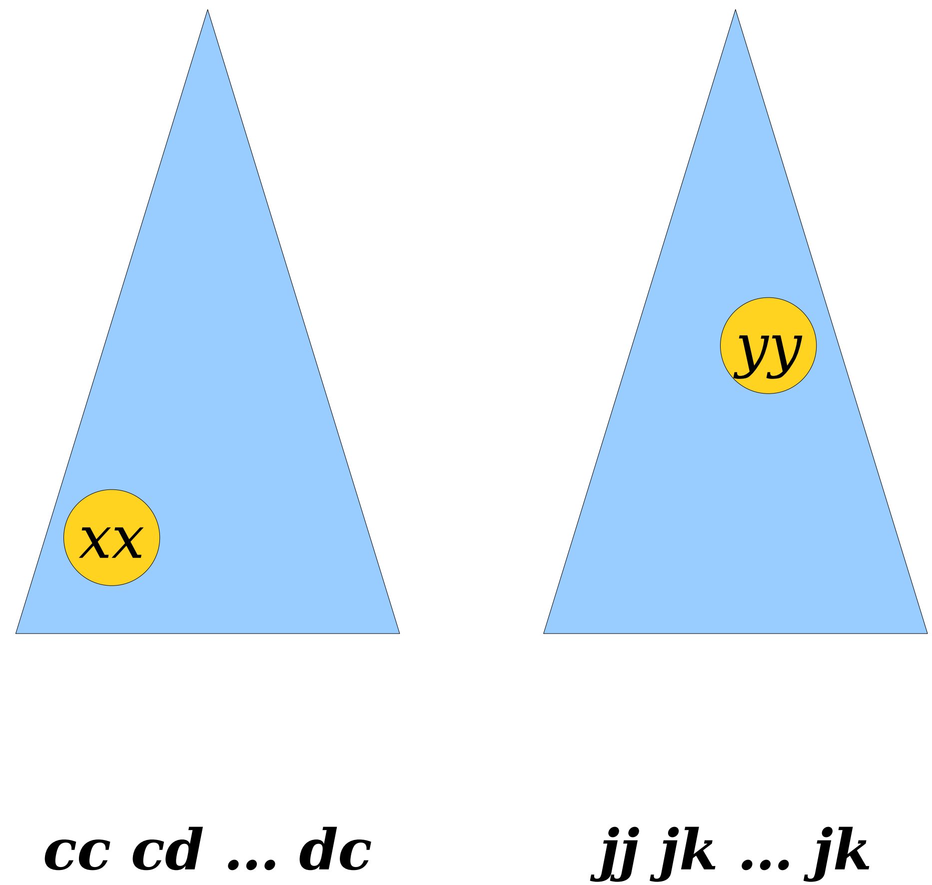
- To ***reroot***(x):
  - Splay xx.
  - Disconnect xx's left child tree T.
  - Splay the rightmost node in xx's subtree.
  - Make T the right child of the root.
- Amortized cost: **O(log *n*)**.



cc cb bd dd db ba aa ab bb bc

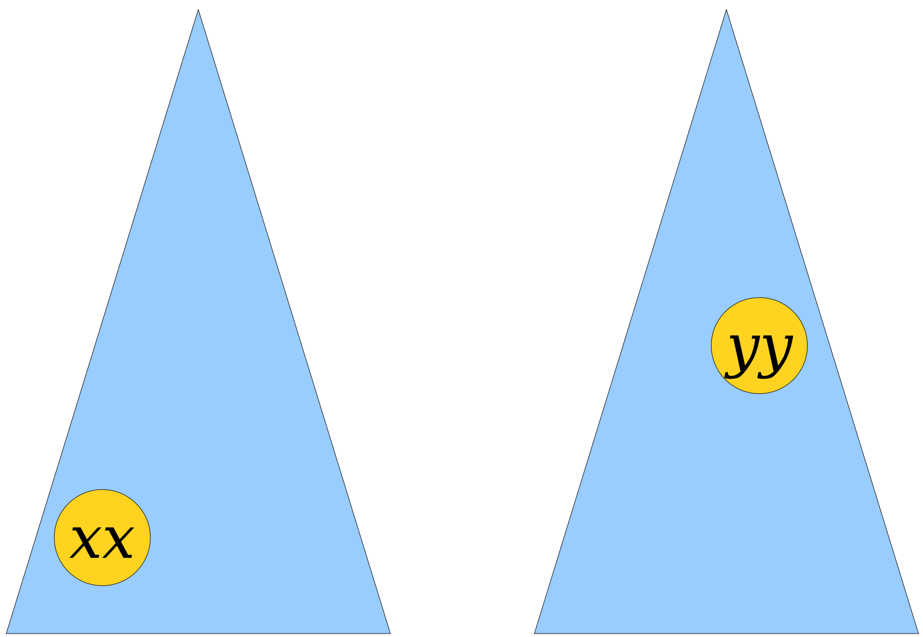
Euler Tour Trees

- To ***link***(x, y):



Euler Tour Trees

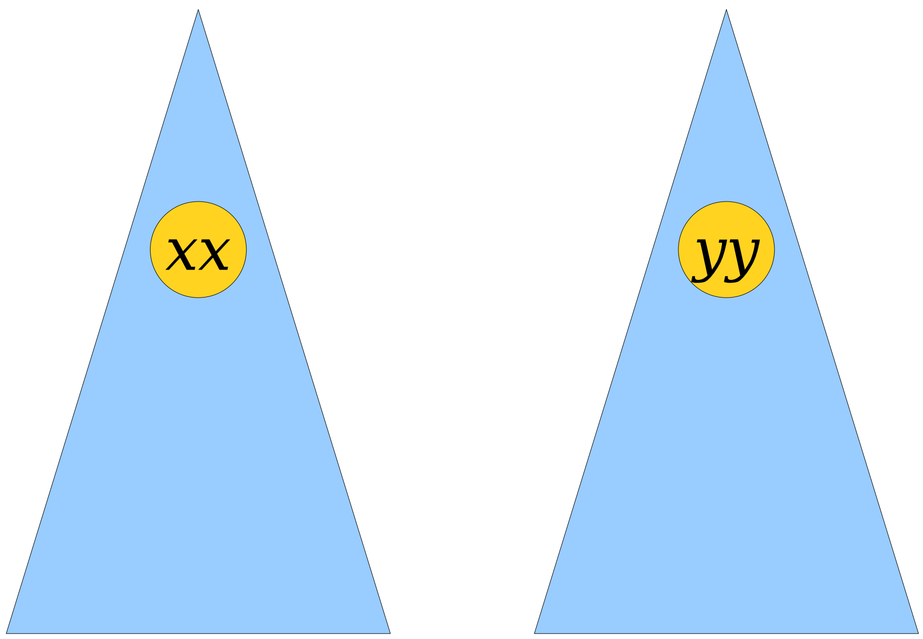
- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).



cc cd … dc ***jj jk … jk***

Euler Tour Trees

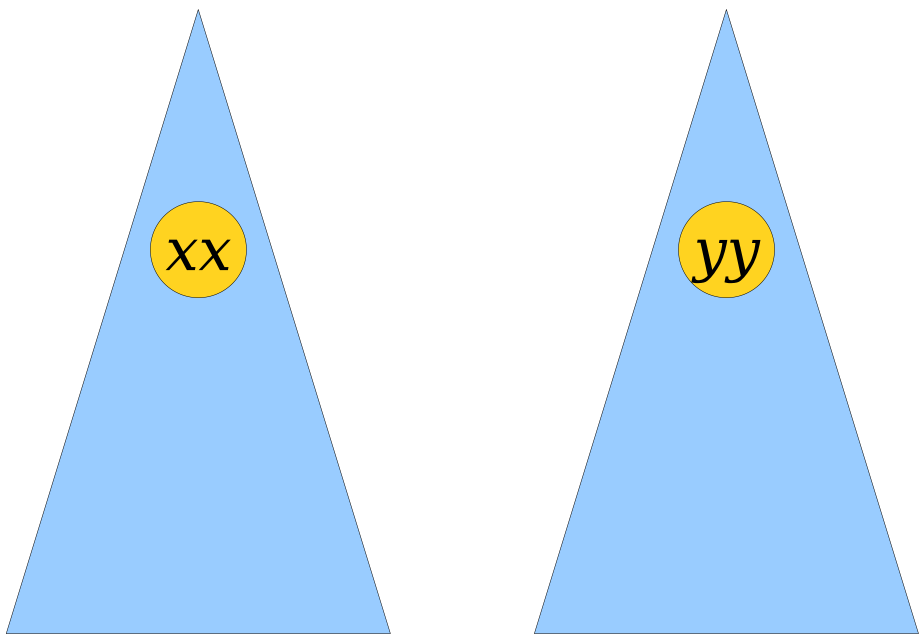
- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).



xx xa … ax ***yy yf … fy***

Euler Tour Trees

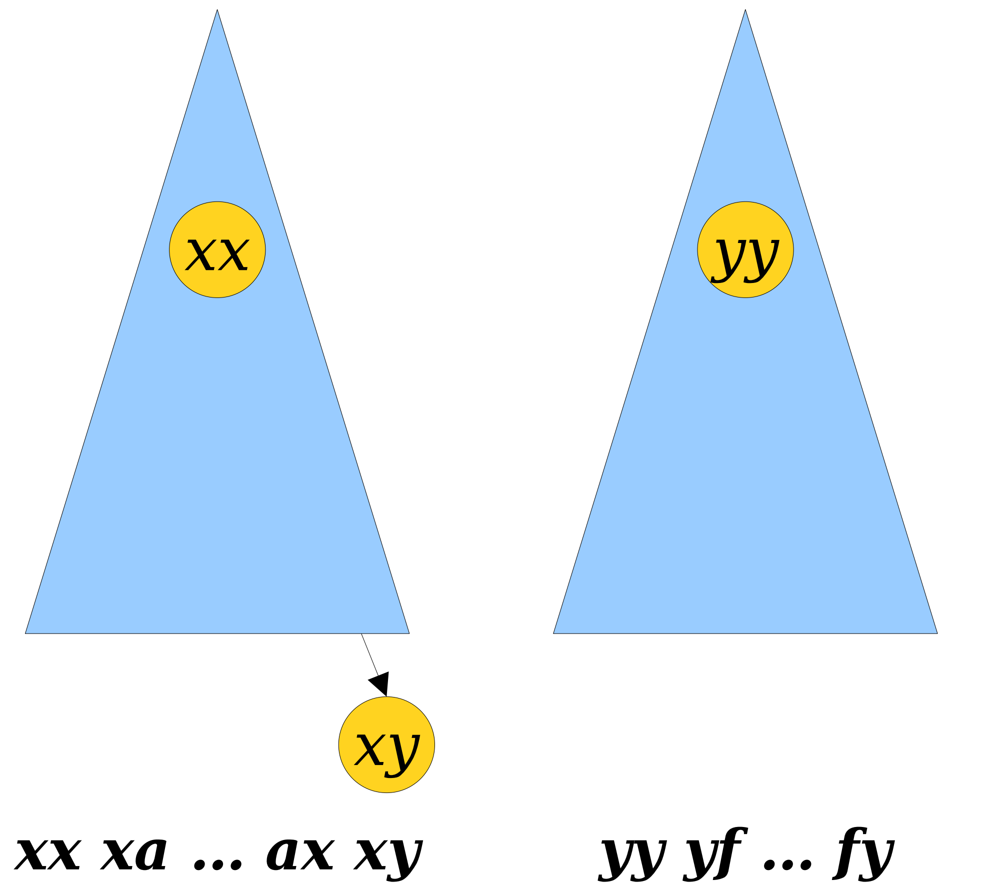
- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).
  - Add xy as the rightmost node of x's tree.



xx xa … ax ***yy yf … fy***

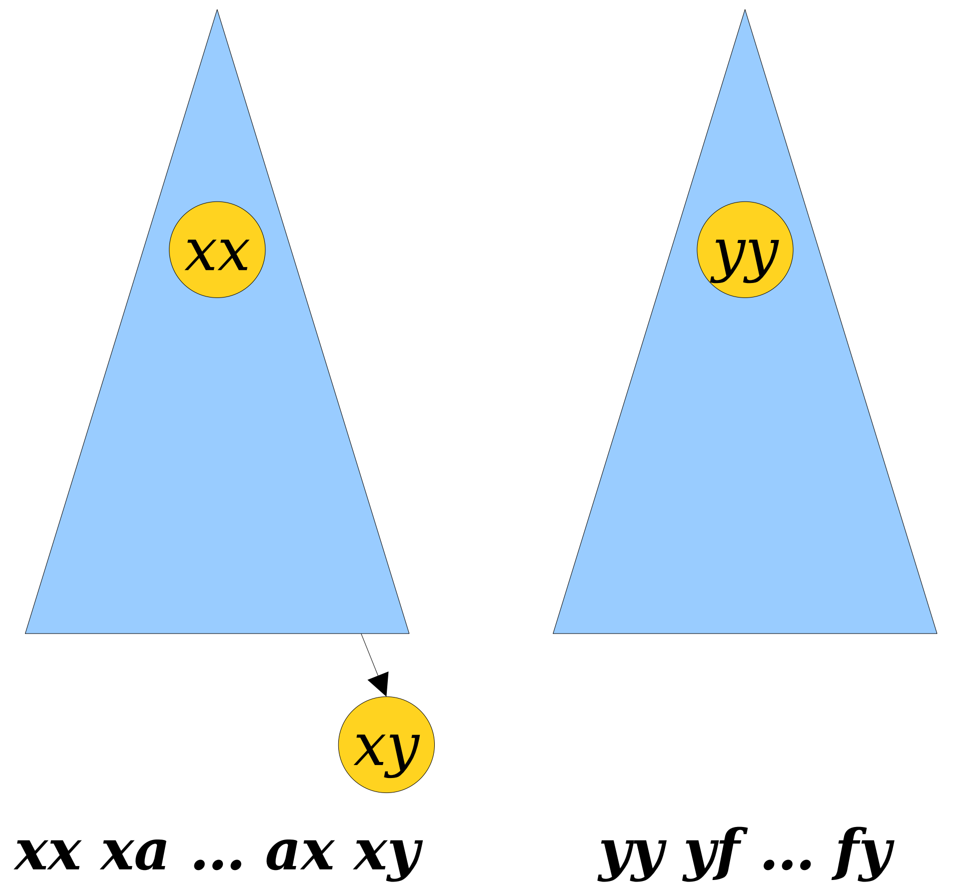
Euler Tour Trees

- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).
  - Add xy as the rightmost node of x's tree.



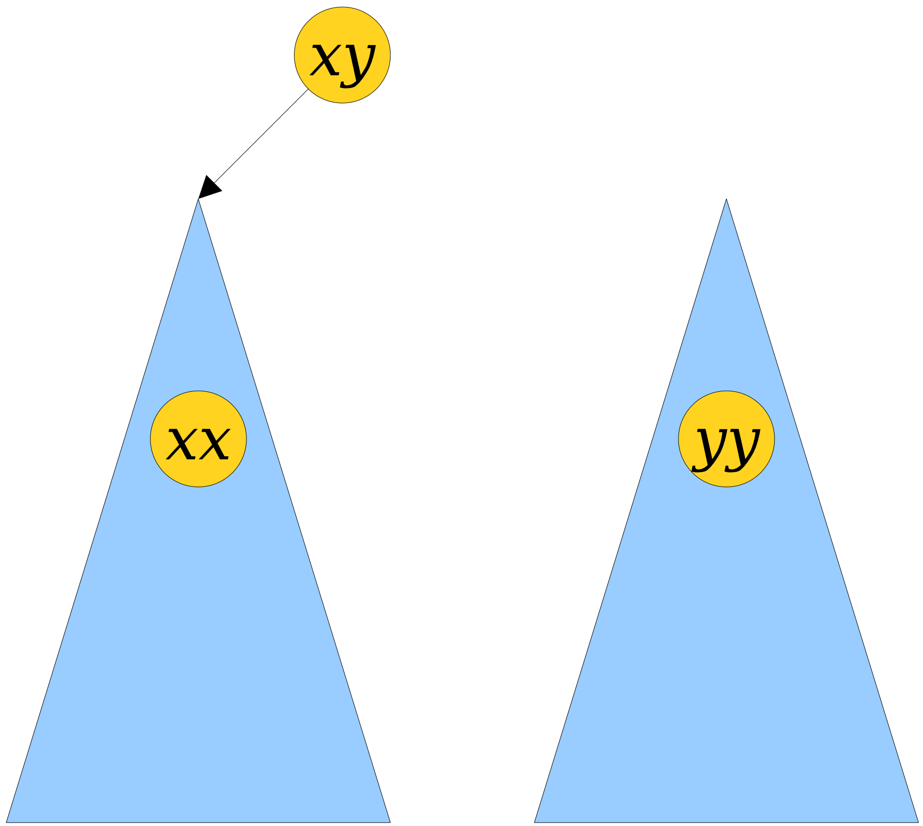
Euler Tour Trees

- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).
  - Add xy as the rightmost node of x's tree.
  - Splay xy.



Euler Tour Trees

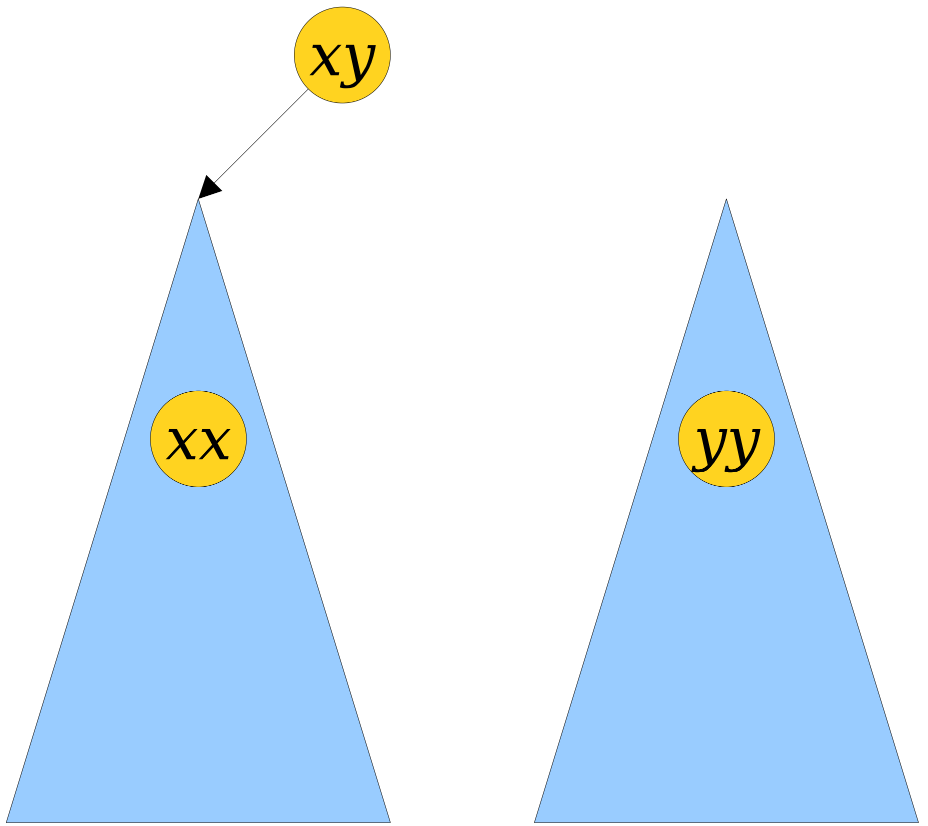
- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).
  - Add xy as the rightmost node of x's tree.
  - Splay xy.



xx xa … ax xy   ***yy yf … fy***

Euler Tour Trees

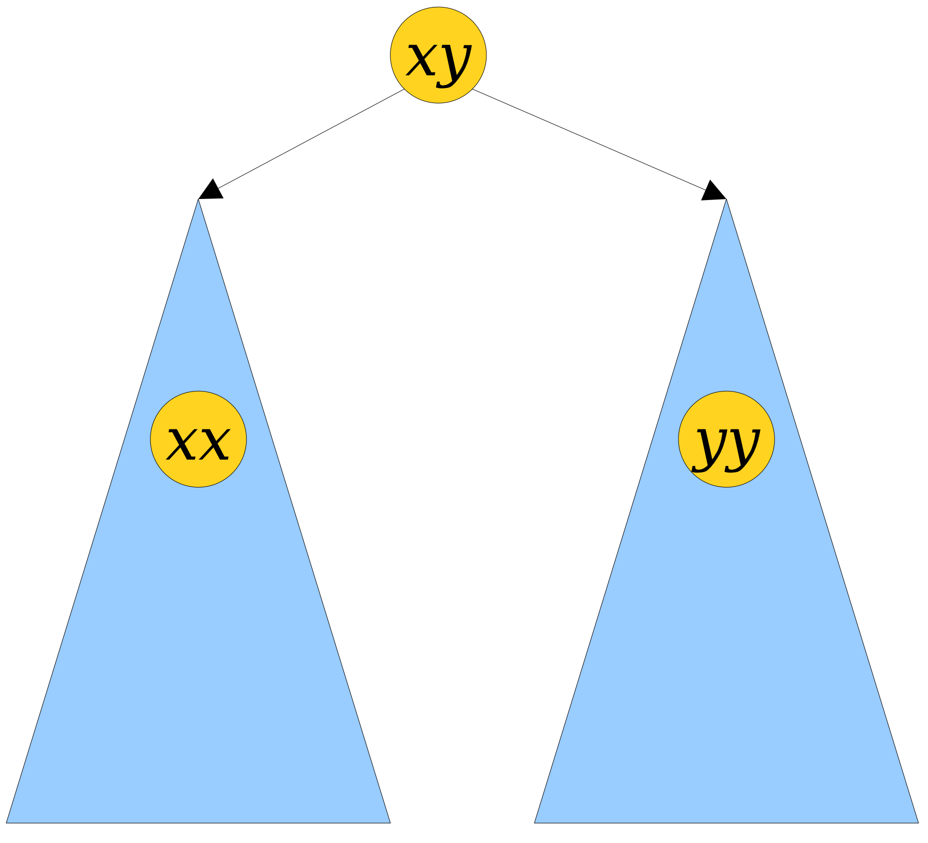
- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).
  - Add xy as the rightmost node of x's tree.
  - Splay xy.
  - Set yy's tree as xy's right child.



xx xa … ax xy ***yy yf … fy***

Euler Tour Trees

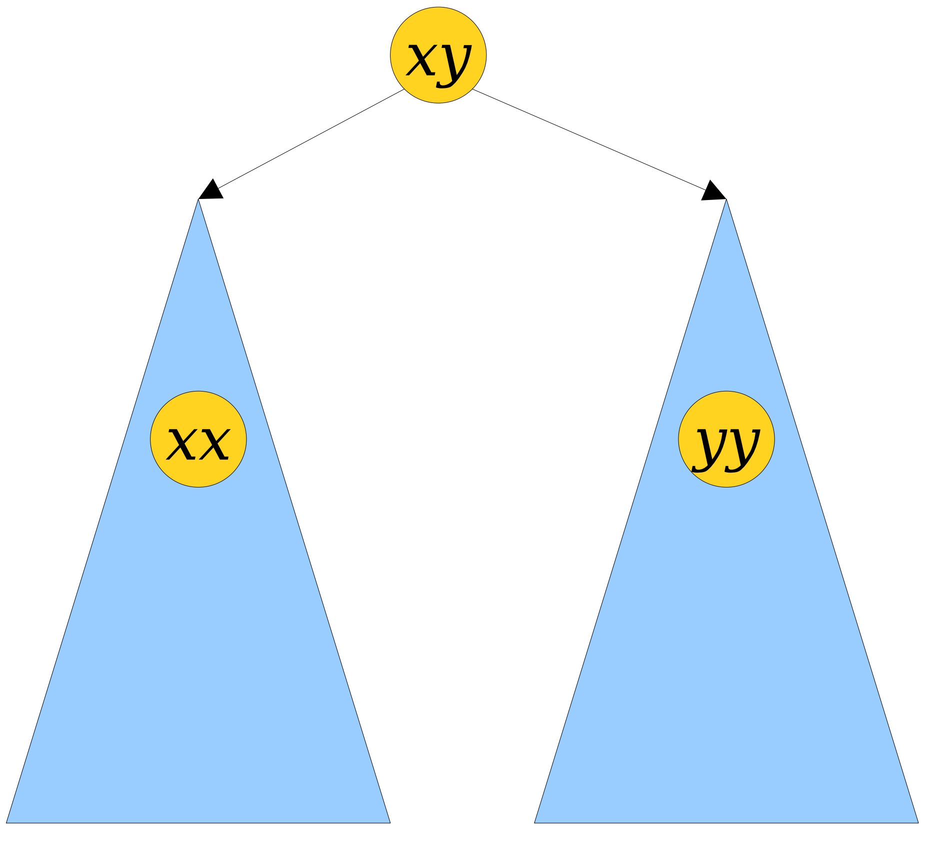
- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).
  - Add xy as the rightmost node of x's tree.
  - Splay xy.
  - Set yy's tree as xy's right child.



xx xa … ax xy yy yf … fy

Euler Tour Trees

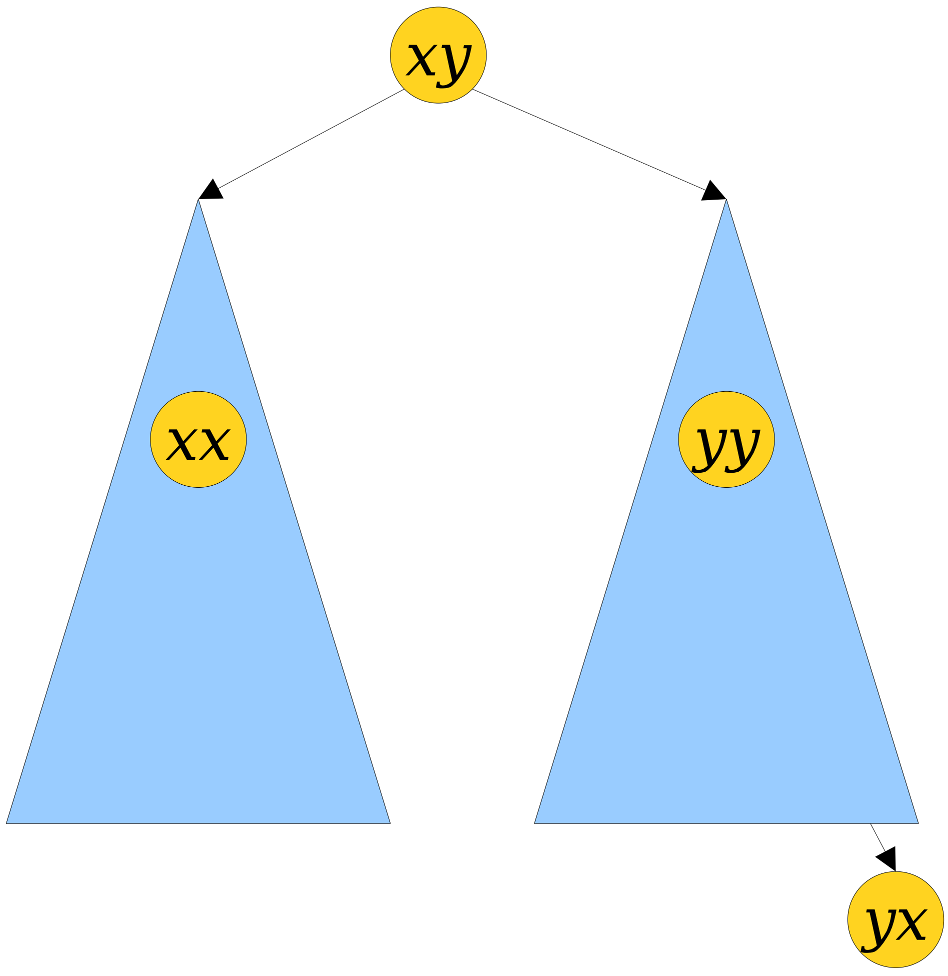
- To ***link***(*x*, *y*):
  - ***reroot***(*x*) and ***reroot***(*y*).
  - Add *xy* as the rightmost node of *x*'s tree.
  - Splay *xy*.
  - Set *yy*'s tree as *xy*'s right child.
  - Add *yx* as the rightmost node of the tree.



xx xa … ax xy yy yf … fy

Euler Tour Trees

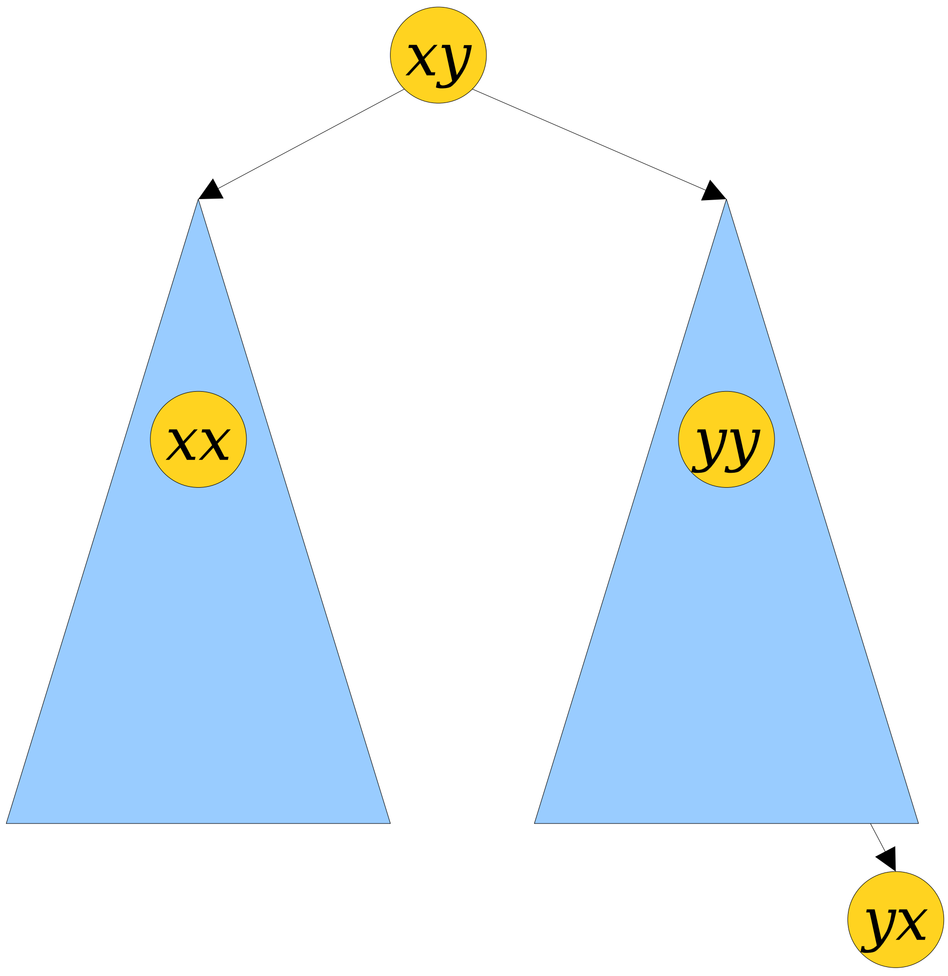
- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).
  - Add xy as the rightmost node of x's tree.
  - Splay xy.
  - Set yy's tree as xy's right child.
  - Add yx as the rightmost node of the tree.



xx xa … ax xy yy yf … fy yx

Euler Tour Trees

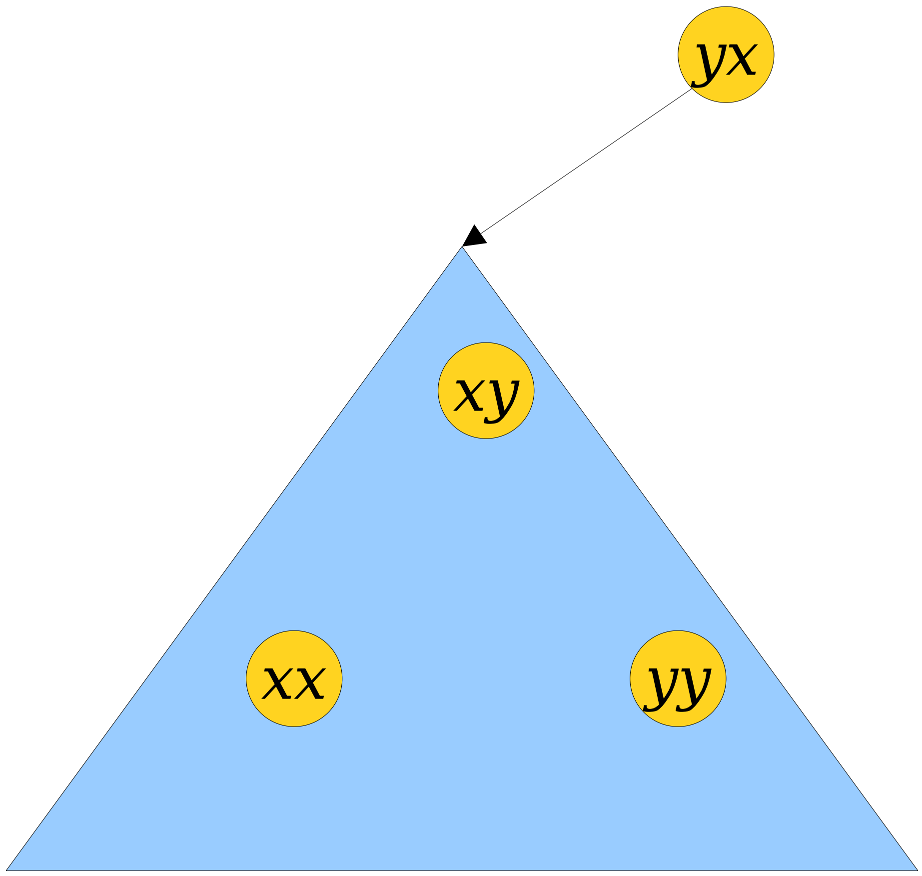
- To ***link***(x, y):
  - ***reroot***(x) and ***reroot***(y).
  - Add xy as the rightmost node of x's tree.
  - Splay xy.
  - Set yy's tree as xy's right child.
  - Add yx as the rightmost node of the tree.
  - Splay yx.



xx xa … ax xy yy yf … fy yx

Euler Tour Trees

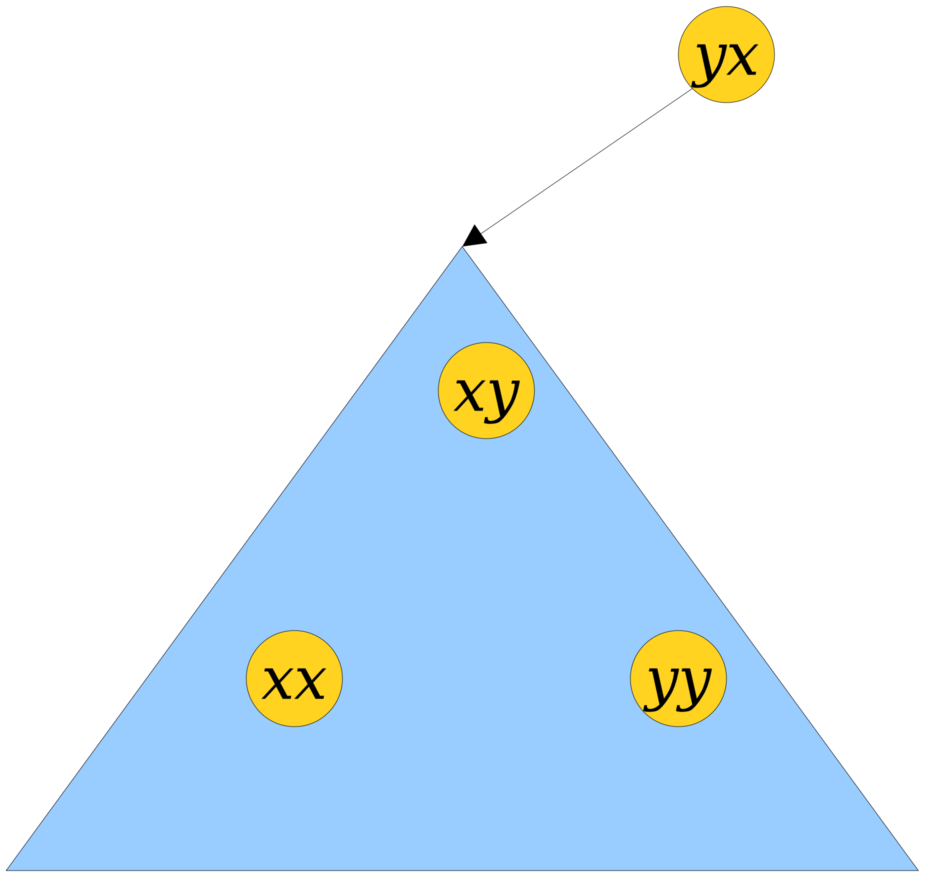
- To ***link***(*x*, *y*):
  - ***reroot***(*x*) and ***reroot***(*y*).
  - Add *xy* as the rightmost node of *x*'s tree.
  - Splay *xy*.
  - Set *yy*'s tree as *xy*'s right child.
  - Add *yx* as the rightmost node of the tree.
  - Splay *yx*.



xx xa … ax xy yy yf … fy yx

Euler Tour Trees

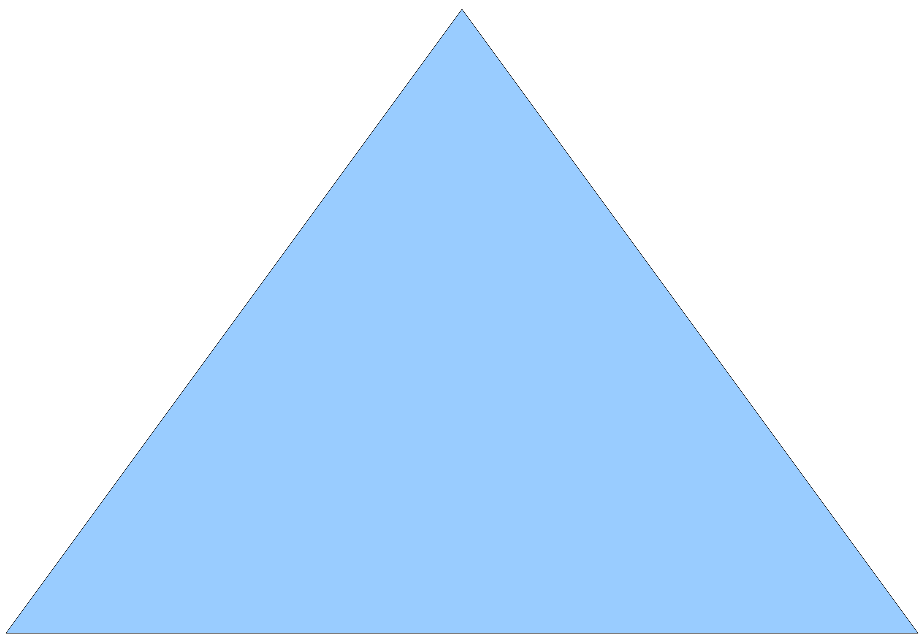
- To ***link***(*x*, *y*):
  - ***reroot***(*x*) and ***reroot***(*y*).
  - Add *xy* as the rightmost node of *x*'s tree.
  - Splay *xy*.
  - Set *yy*'s tree as *xy*'s right child.
  - Add *yx* as the rightmost node of the tree.
  - Splay *yx*.
- Amortized cost: **O(log *n*)**.



xx xa … ax xy yy yf … fy yx

Euler Tour Trees

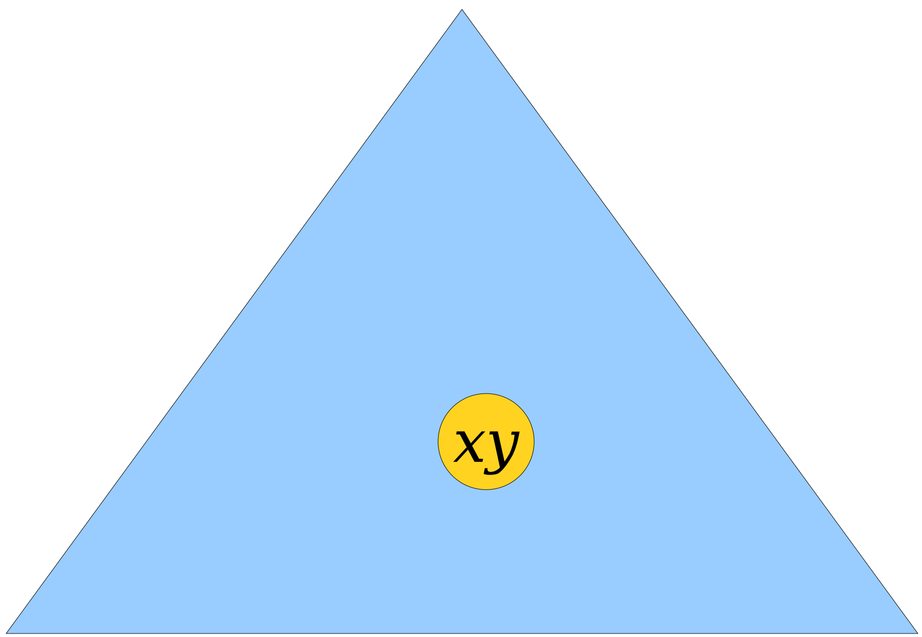
- To ***cut***(x, y):



aa ab … cx xy yy yf … fy yx xt … ba

Euler Tour Trees

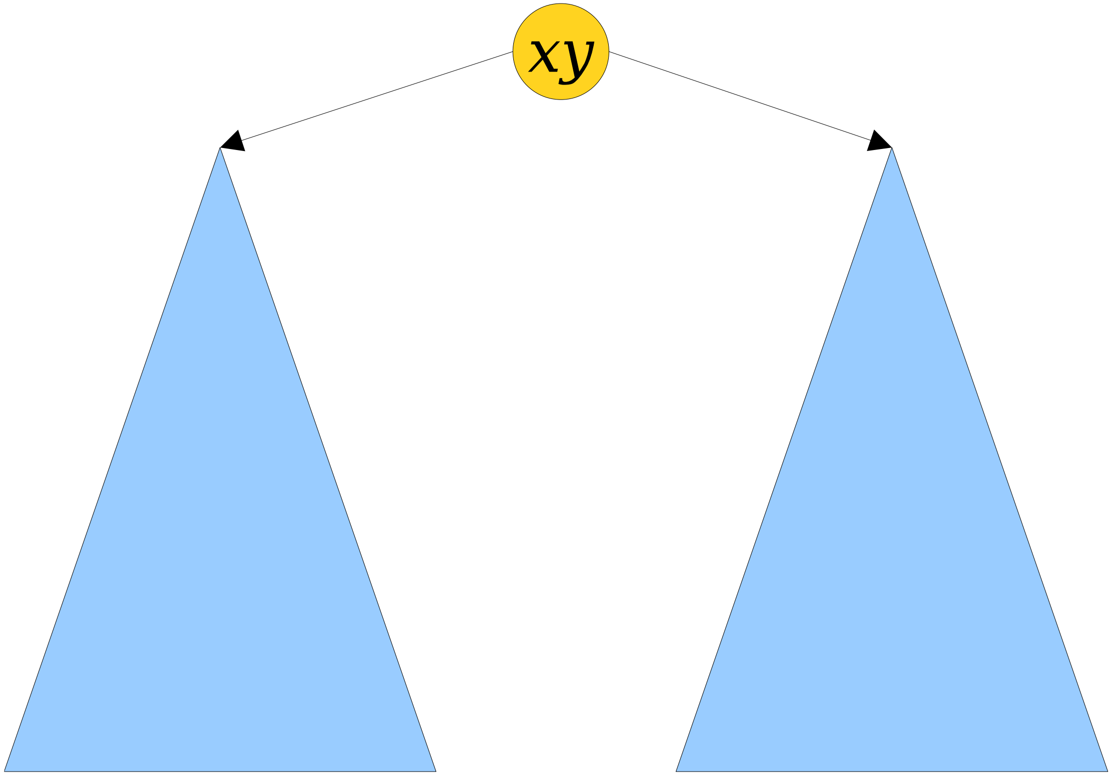
- To ***cut***(x, y):
  - Splay xy.



aa ab … cx xy yy yf … fy yx xt … ba

Euler Tour Trees

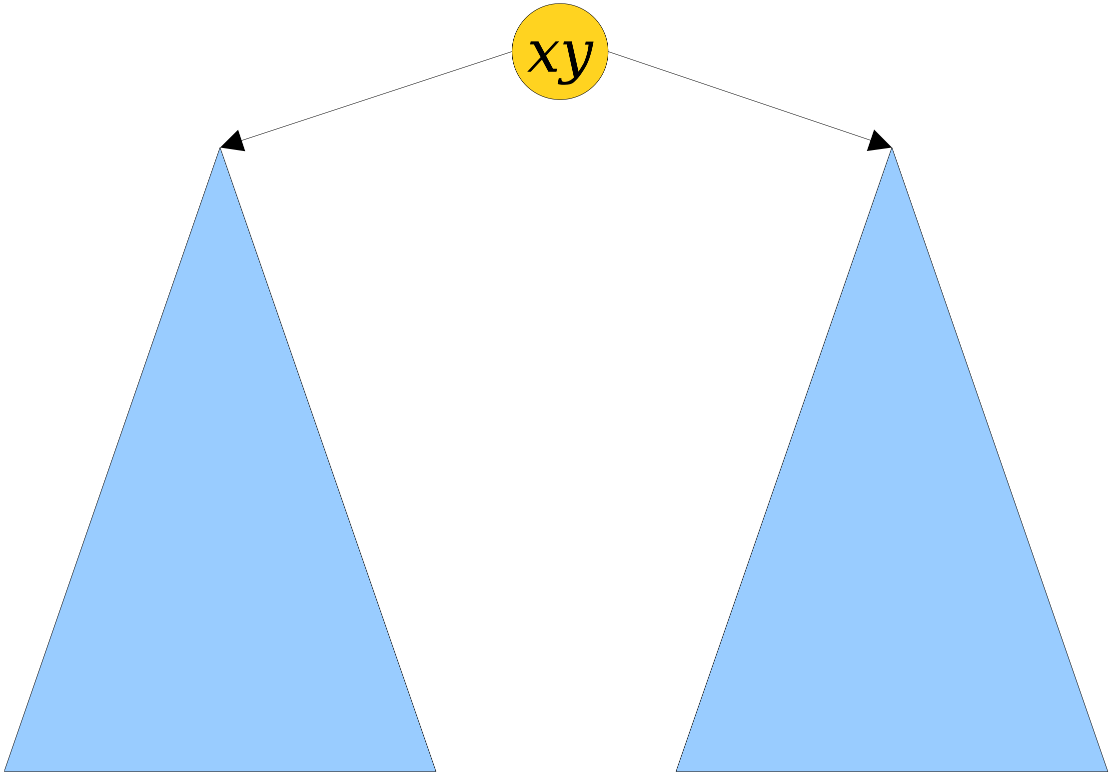
- To ***cut***(x, y):
  - Splay xy.



aa ab … cx xy yy yf … fy yx xt … ba

Euler Tour Trees

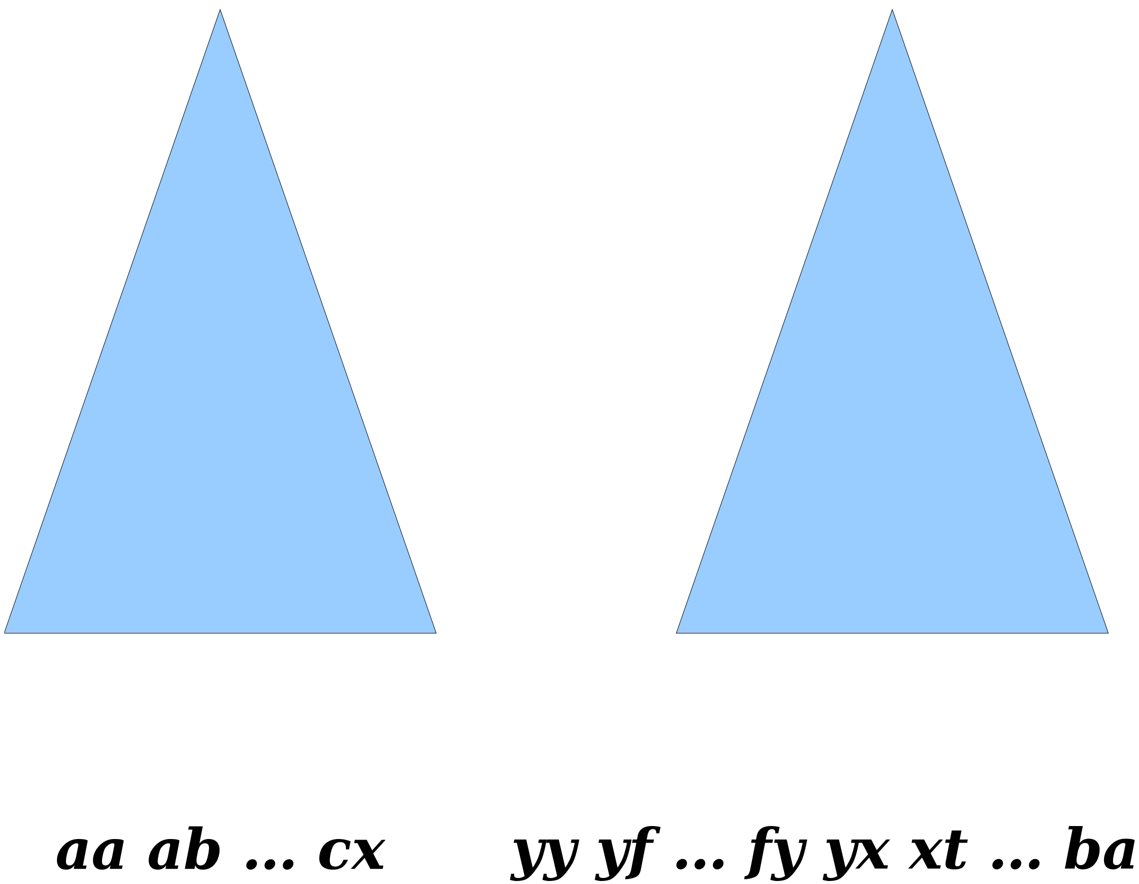
- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.



aa ab … cx xy yy yf … fy yx xt … ba

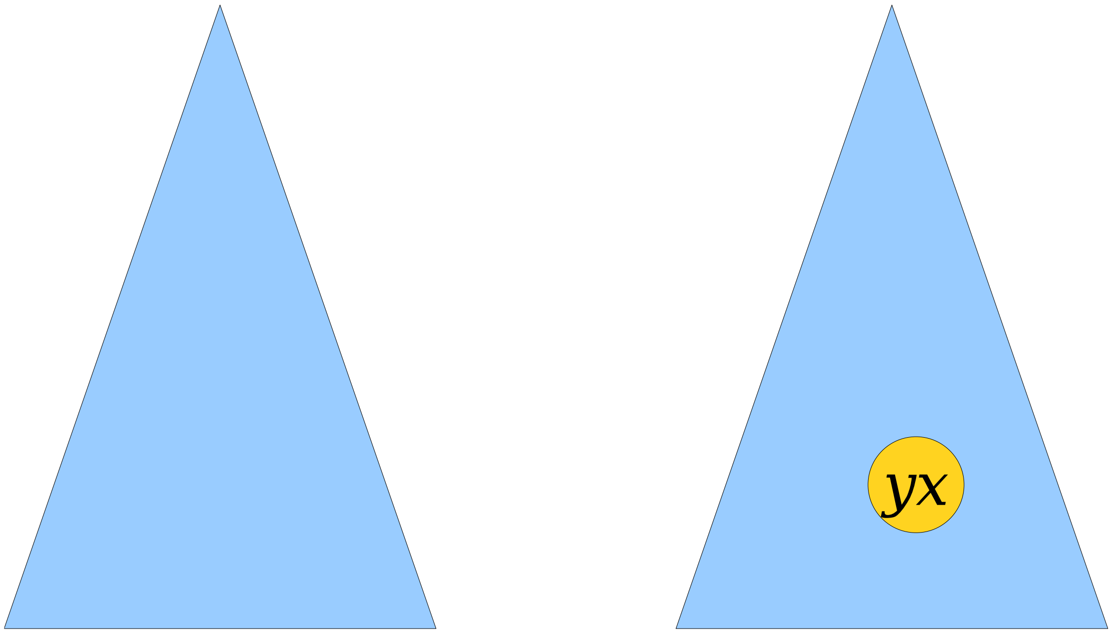
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.



Euler Tour Trees

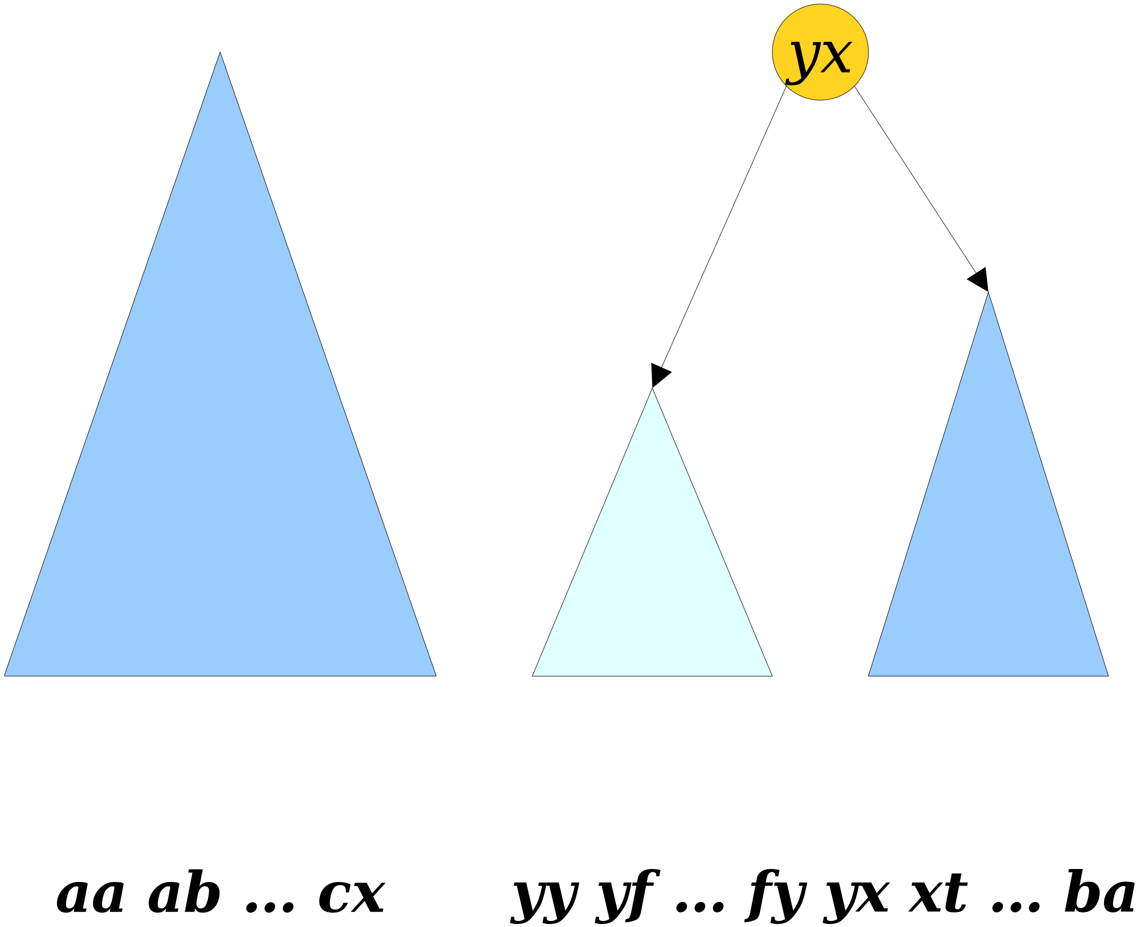
- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.



aa ab … cx ***yy yf … fy yx xt … ba***

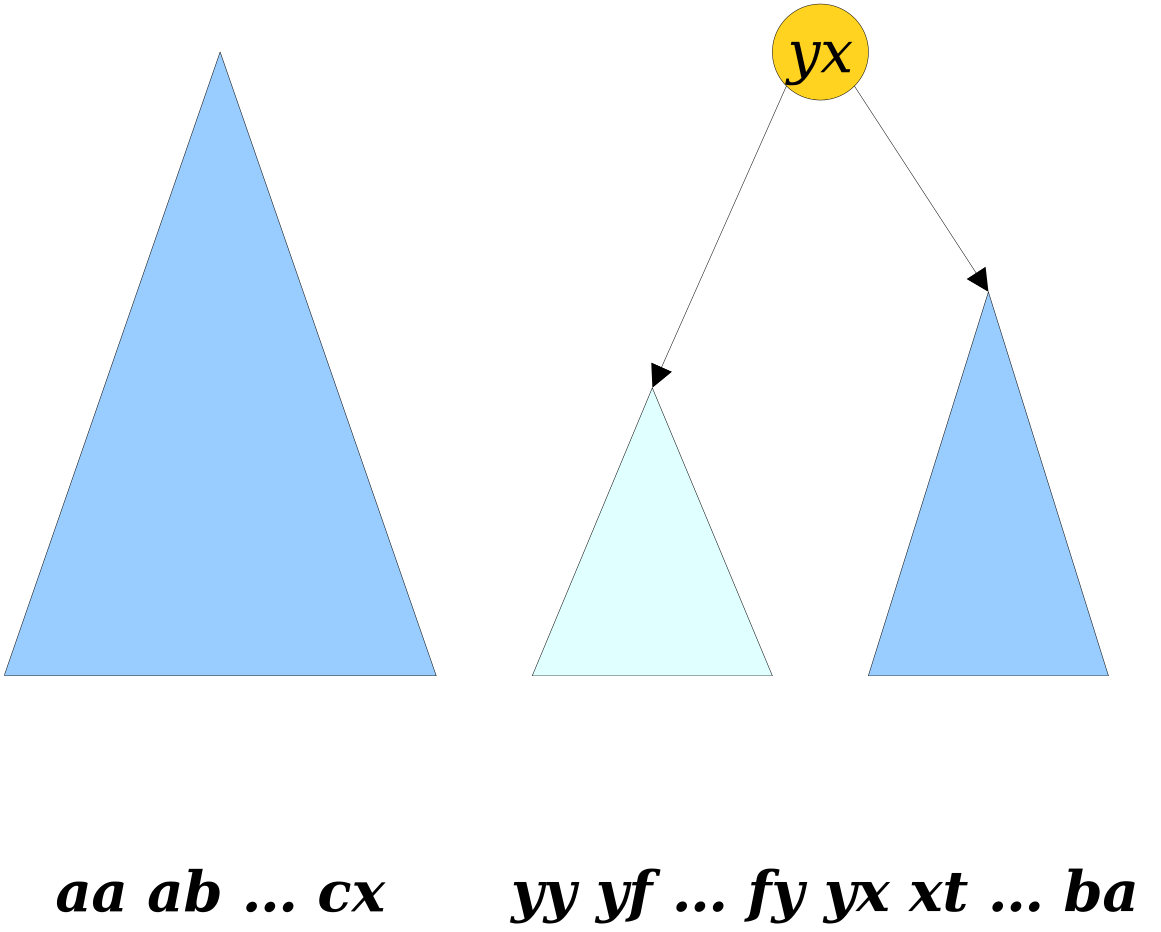
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.



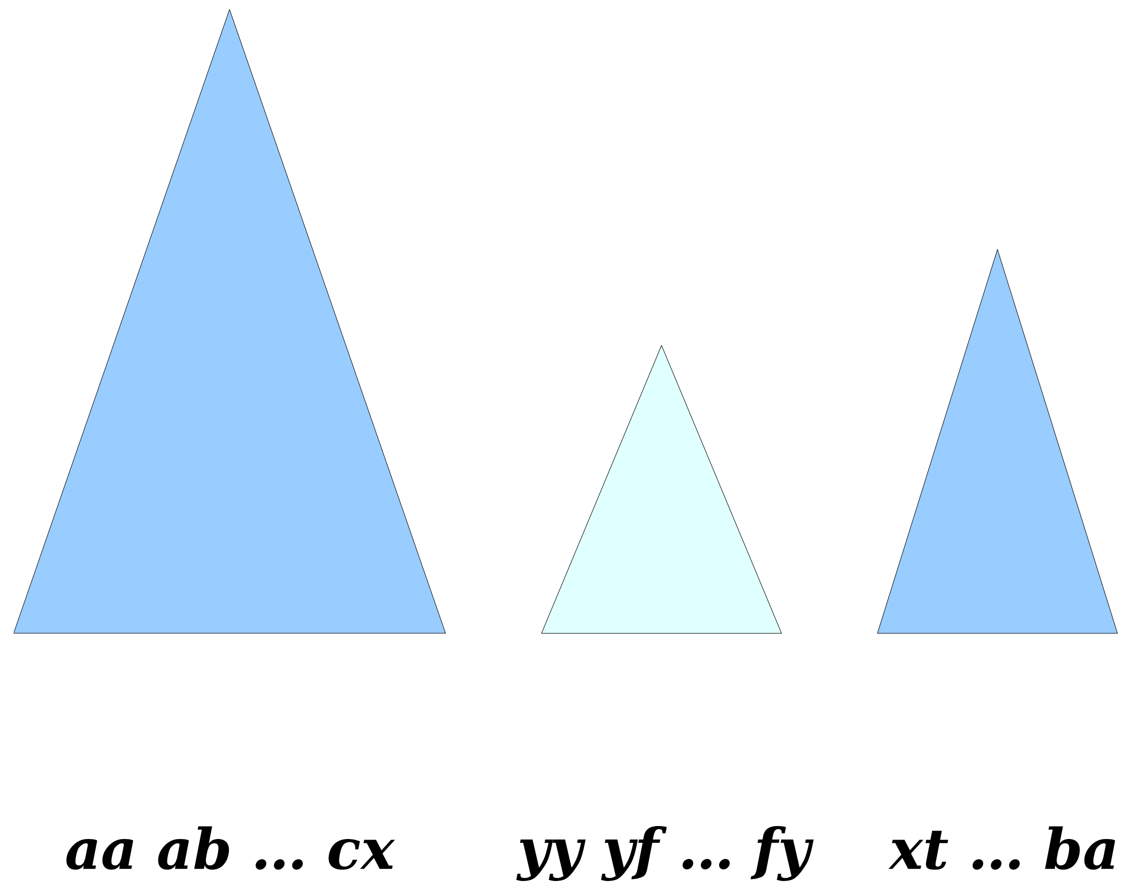
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.



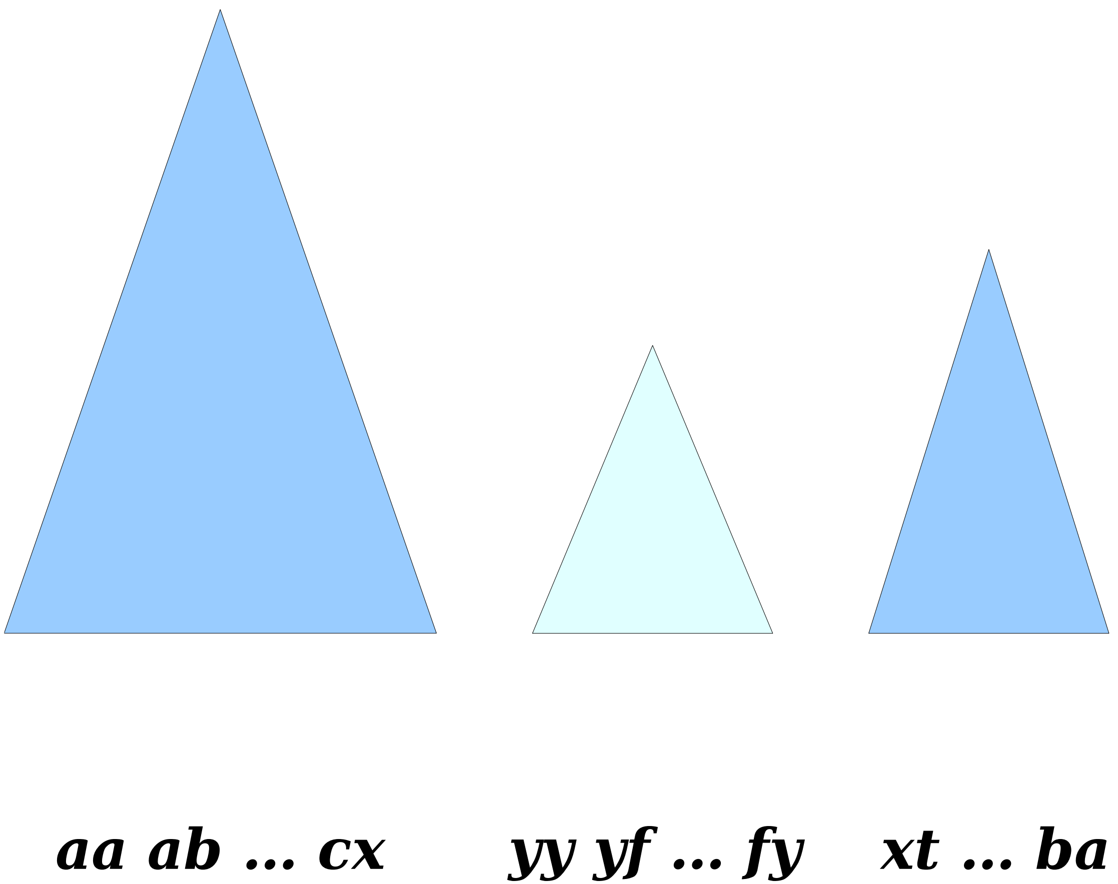
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.



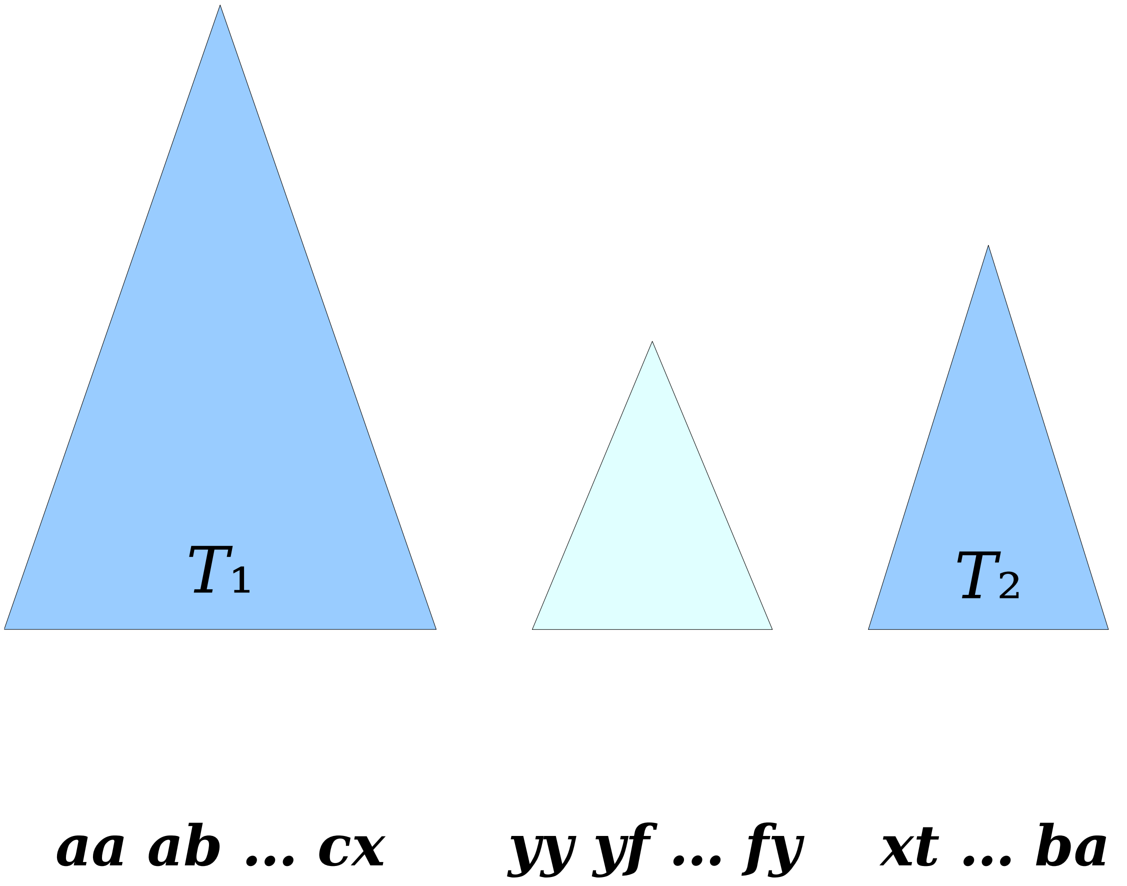
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.
  - Let T_1 and T_2 be the trees on the left and right.



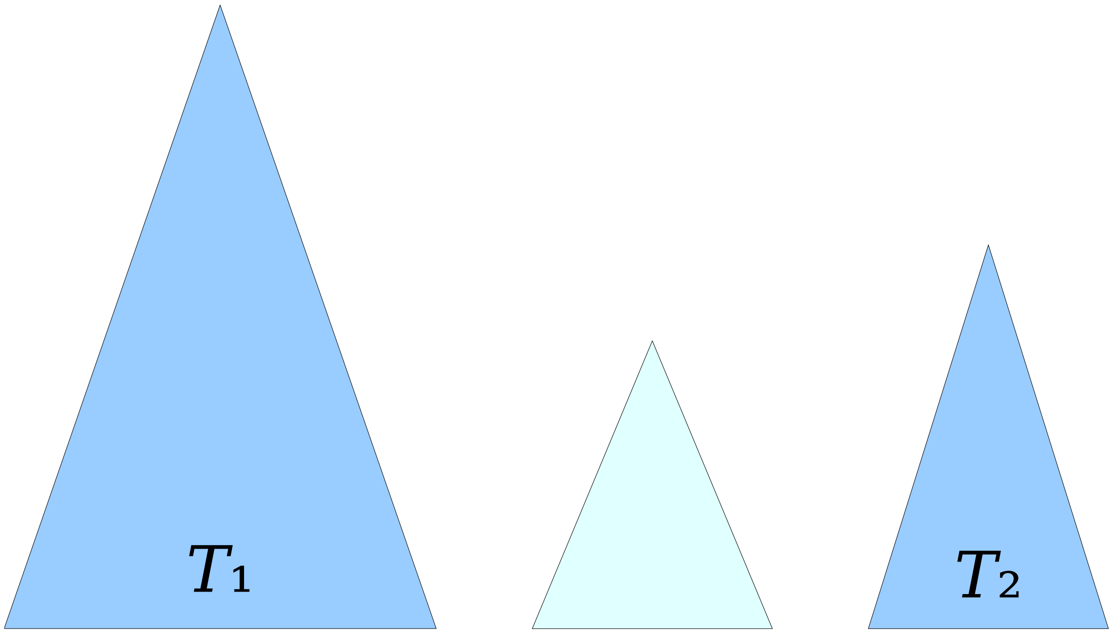
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.
  - Let T_1 and T_2 be the trees on the left and right.



Euler Tour Trees

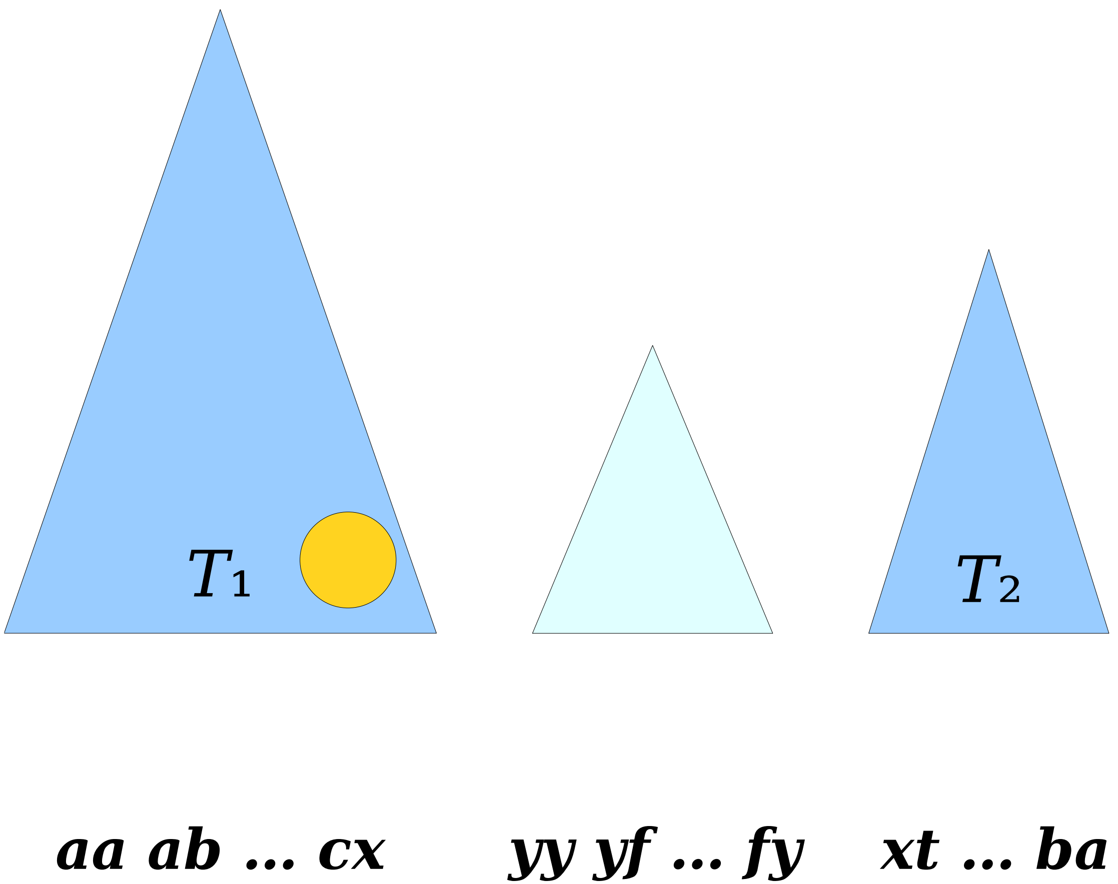
- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.
  - Let T_1 and T_2 be the trees on the left and right.
  - Splay the rightmost node of T_1.



aa ab … cx ***yy yf … fy*** ***xt … ba***

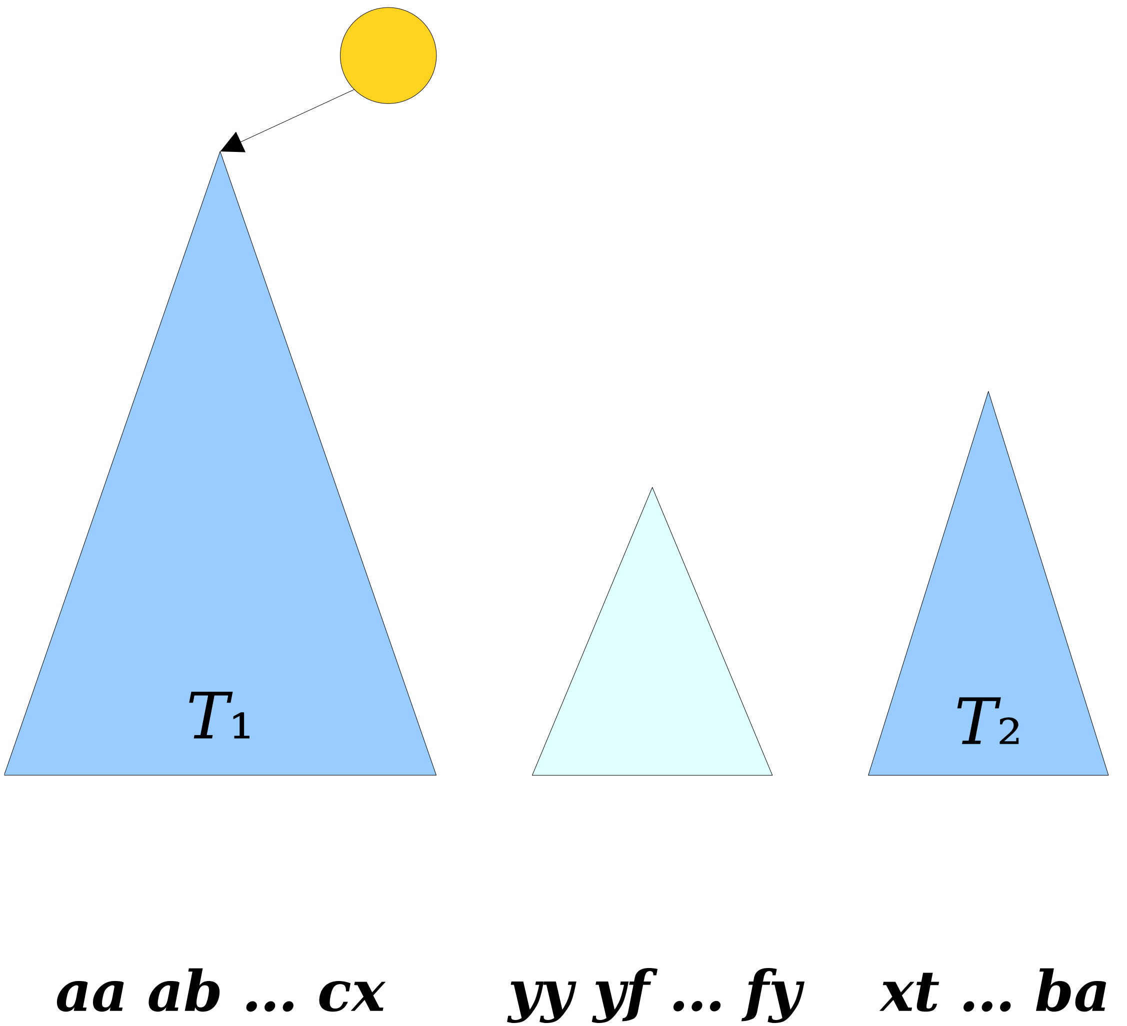
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.
  - Let T_1 and T_2 be the trees on the left and right.
  - Splay the rightmost node of T_1.



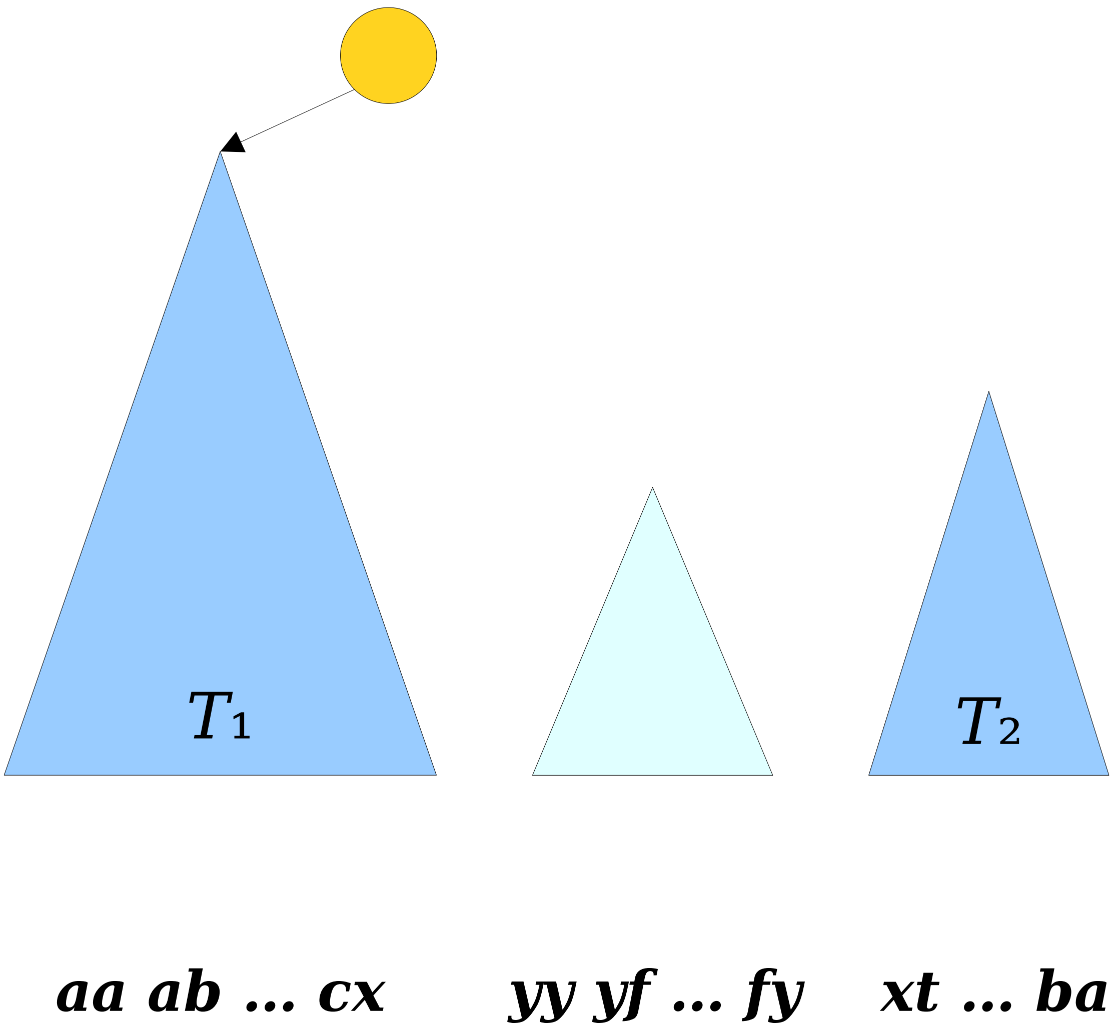
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.
  - Let T_1 and T_2 be the trees on the left and right.
  - Splay the rightmost node of T_1.



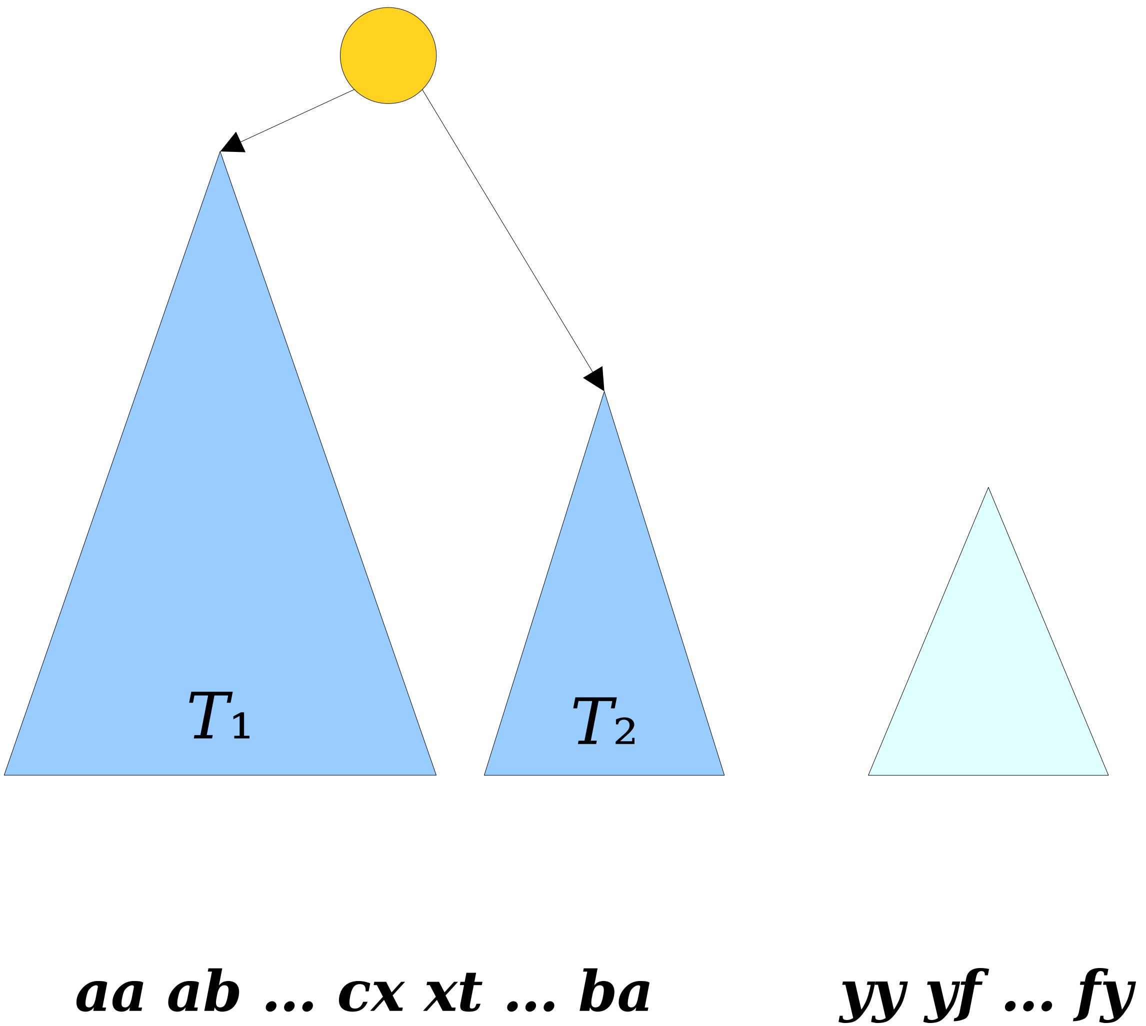
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.
  - Let T_1 and T_2 be the trees on the left and right.
  - Splay the rightmost node of T_1.
  - Attach T_2 as the right child of that node.



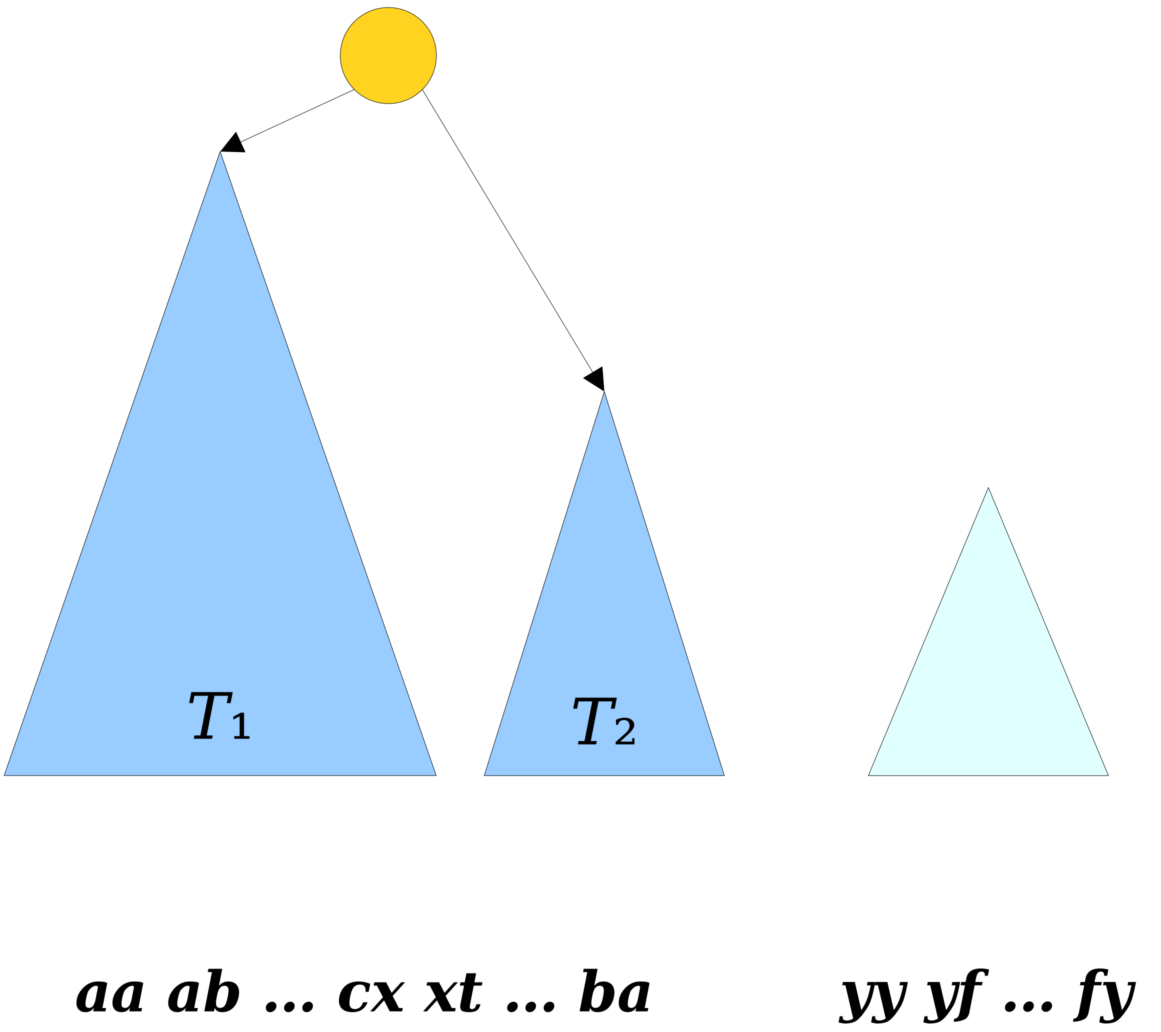
Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.
  - Let T_1 and T_2 be the trees on the left and right.
  - Splay the rightmost node of T_1.
  - Attach T_2 as the right child of that node.



Euler Tour Trees

- To ***cut***(x, y):
  - Splay xy.
  - Delete xy.
  - Splay yx.
  - Delete yx.
  - Let T_1 and T_2 be the trees on the left and right.
  - Splay the rightmost node of T_1.
  - Attach T_2 as the right child of that node.
- Amortized cost: **O(log *n*)**.



Euler Tour Trees

- With all things said and done, we get the following amortized runtimes for each operation:
  - ***are-connected***: O(log *n*)
  - ***link***: O(log *n*)
  - ***cut***: O(log *n*)
- These bounds can be made worst-case efficient using different types of balanced BSTs instead of splay trees, but splaying is probably the fastest way to do this.

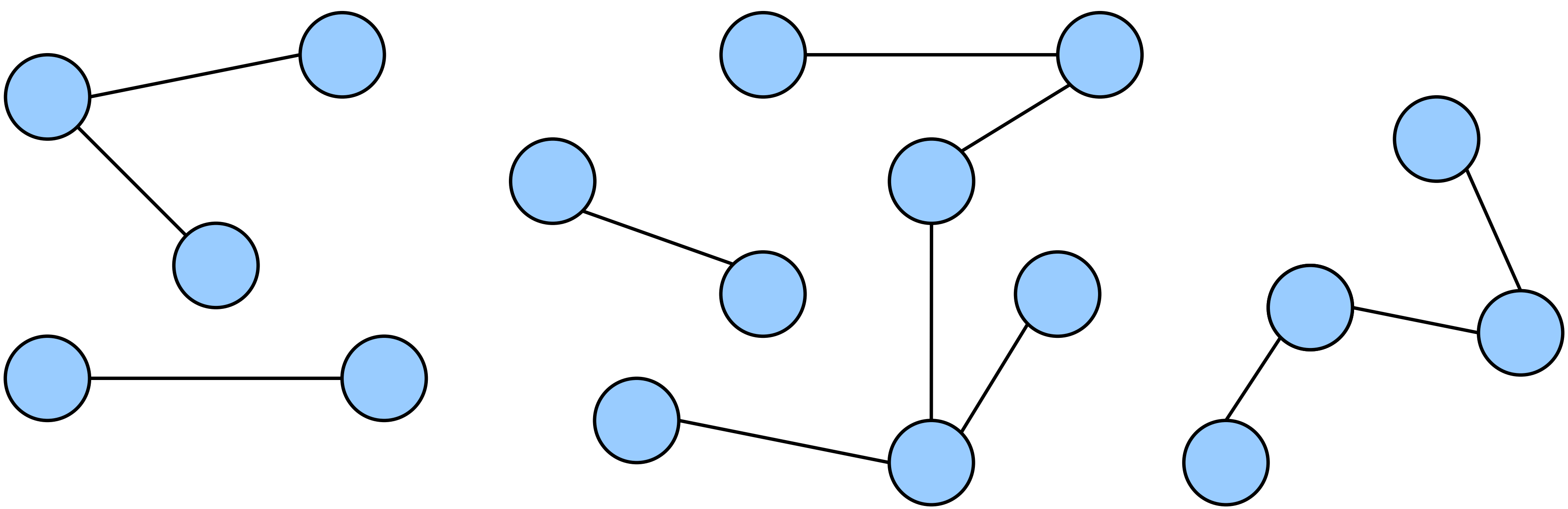
Extending Euler Tour Trees

Extending Euler Tour Trees

- We now have a (relatively) simple and fast data structure for solving dynamic connectivity in forests.
- What else can we do with them?

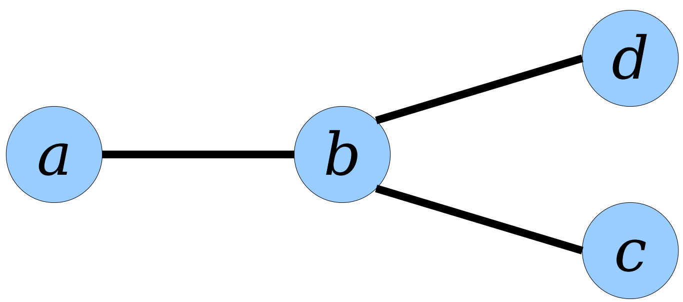
Extending Euler Tour Trees

- Suppose we want to add an operation ***size***(x) that returns the number of nodes in the tree containing x.
- How might we accomplish this?



Tree Sizes

- We can determine ***size***(x) as follows:
  - Figure out which Euler tour xx is in.
  - Count how many nodes of the form zz it contains.
- A naive implementation of this algorithm might take time $\Theta(n)$ if all nodes are in the same tree. Can we do better?



aa ab bb bc cc cb bd dd db ba

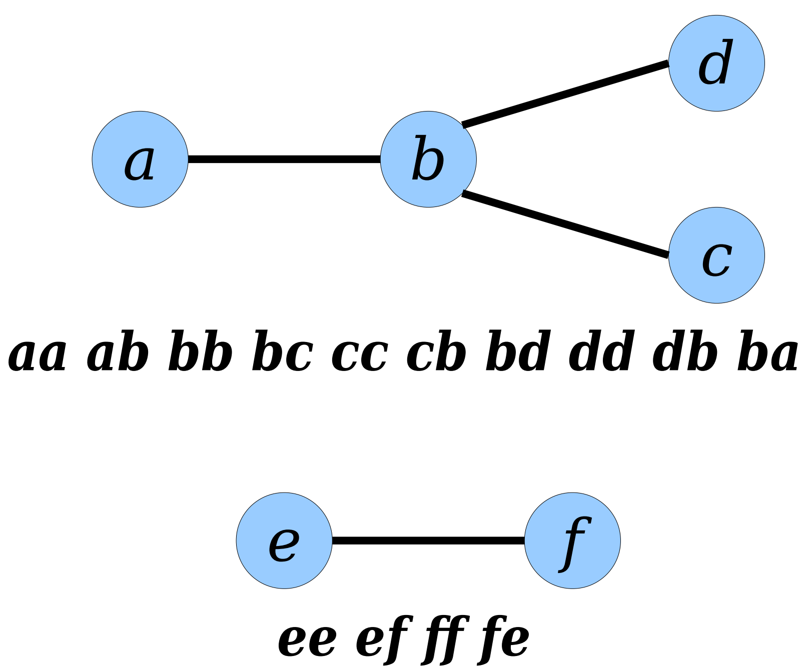
ee ef ff fe

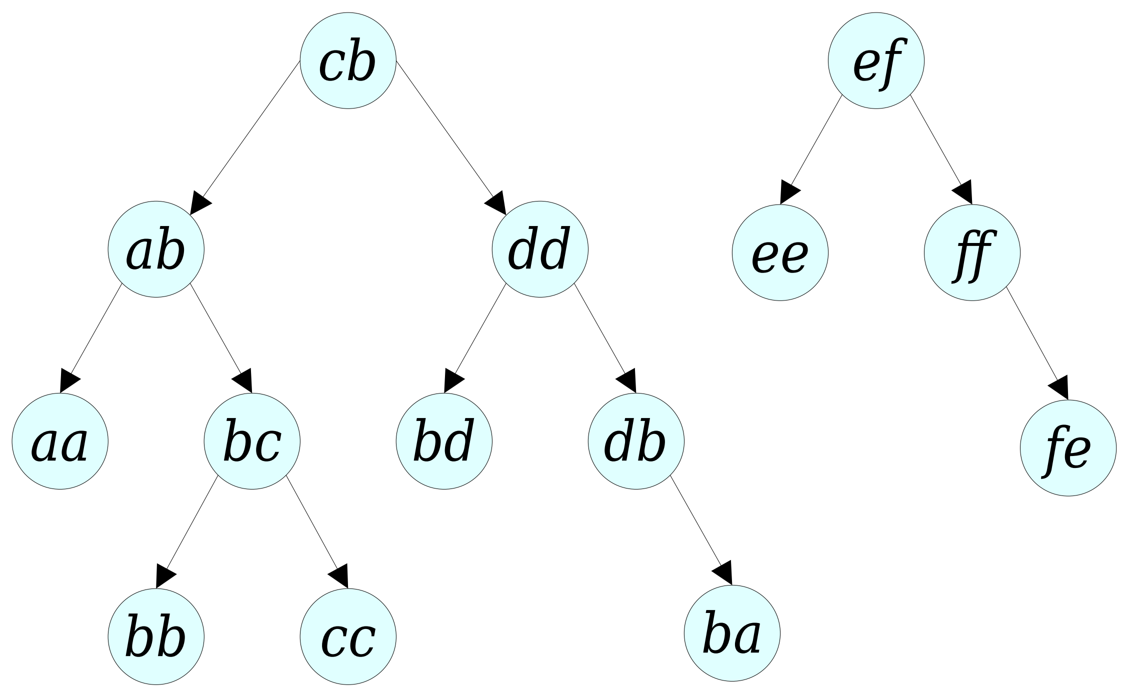
Tree Sizes

- We’re storing our Euler tours in balanced BSTs.
- We want to be able to answer the following question about a given BST:

How many nodes of the form xx are in this BST?

- This can be done in time O(log n). How?



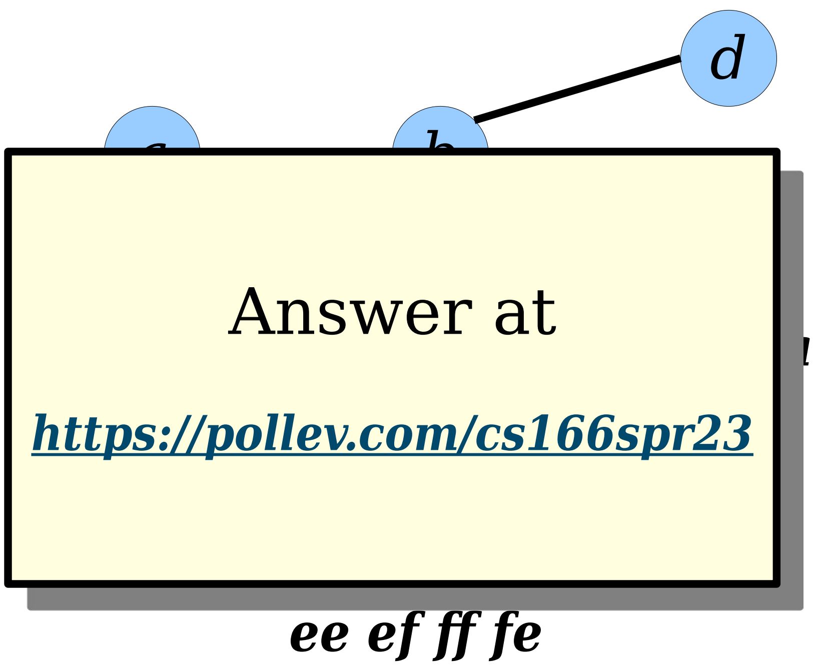


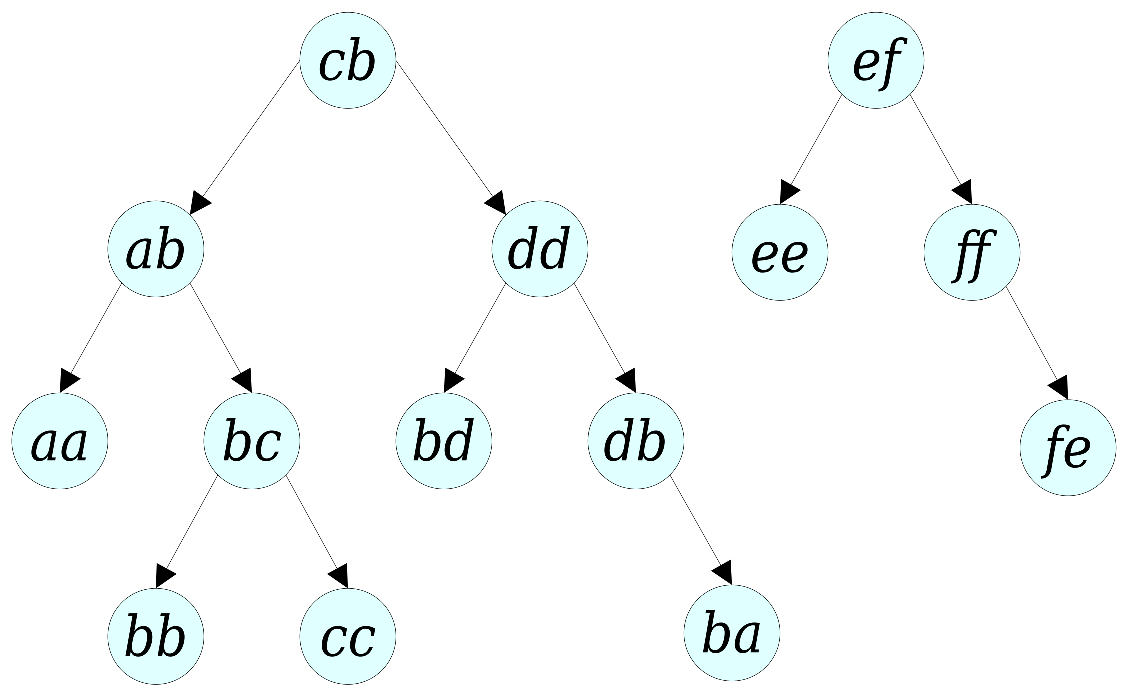
Tree Sizes

- We’re storing our Euler tours in balanced BSTs.
- We want to be able to answer the following question about a given BST:

How many nodes of the form xx are in this BST?

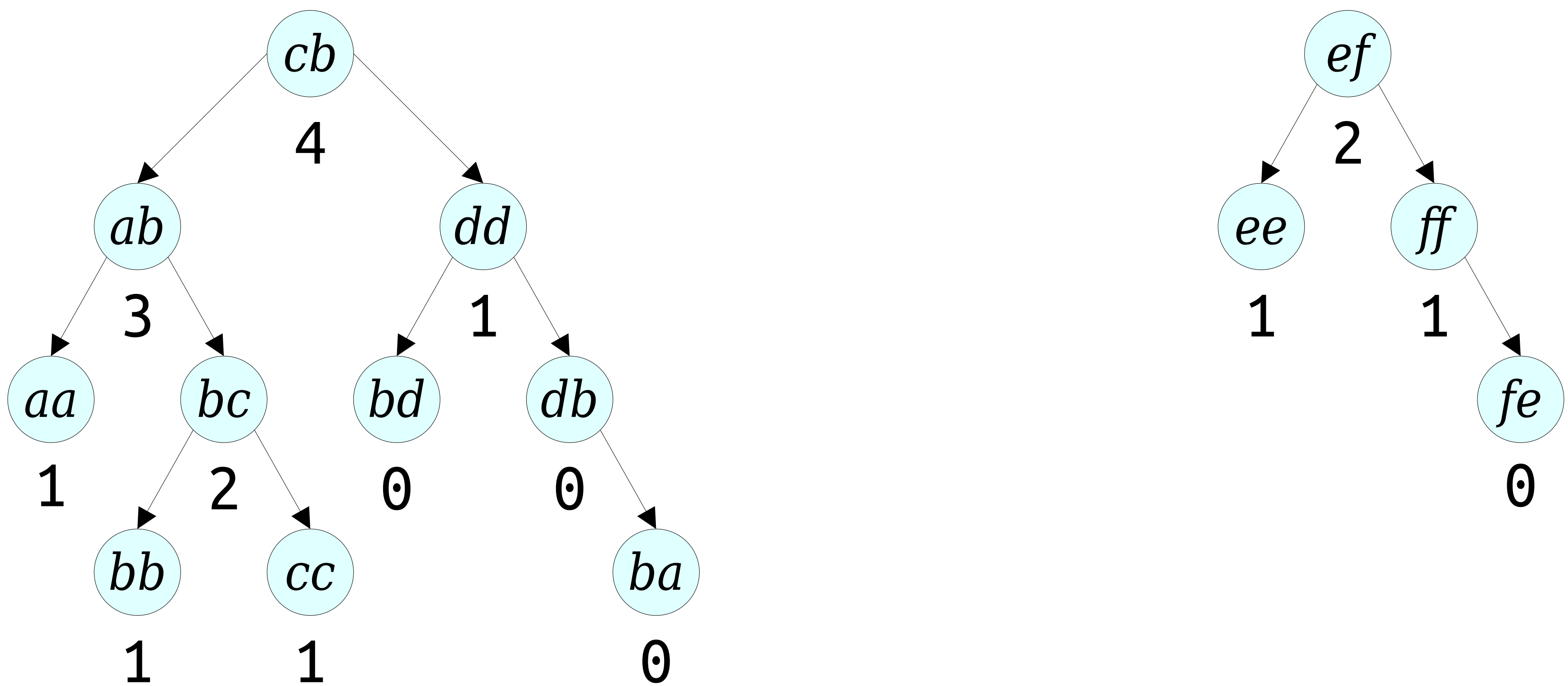
- This can be done in time O(log *n*). How?





Tree Sizes

- ***Idea:*** Augment the BSTs holding our Euler tours.
- Specifically, each node stores the number of self-loops at or below it in the tree.
- This information can be maintained through rotations and after each splay tree operation.

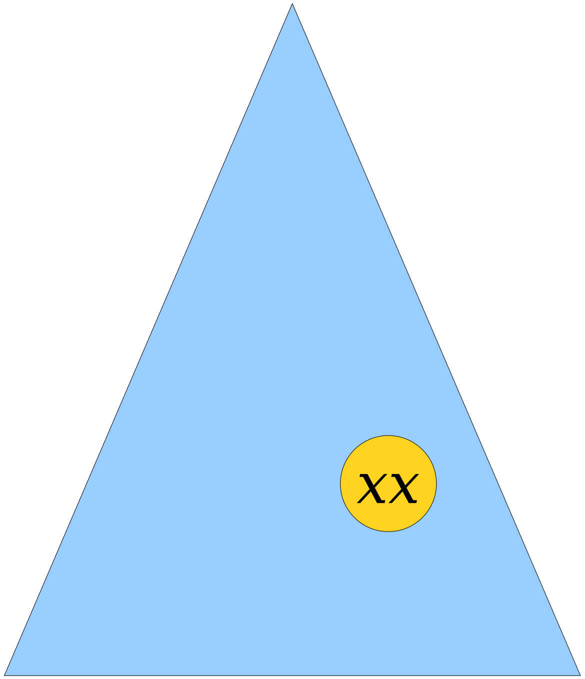


Tree Sizes

- To determine ***size***(x):

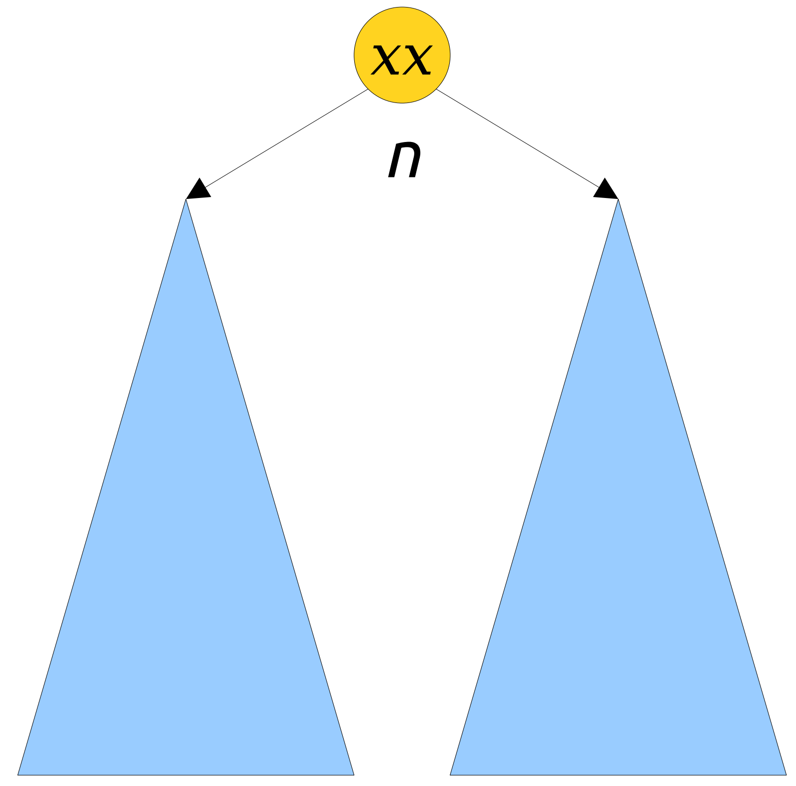
Tree Sizes

- To determine ***size***(x):
  - Splay xx.



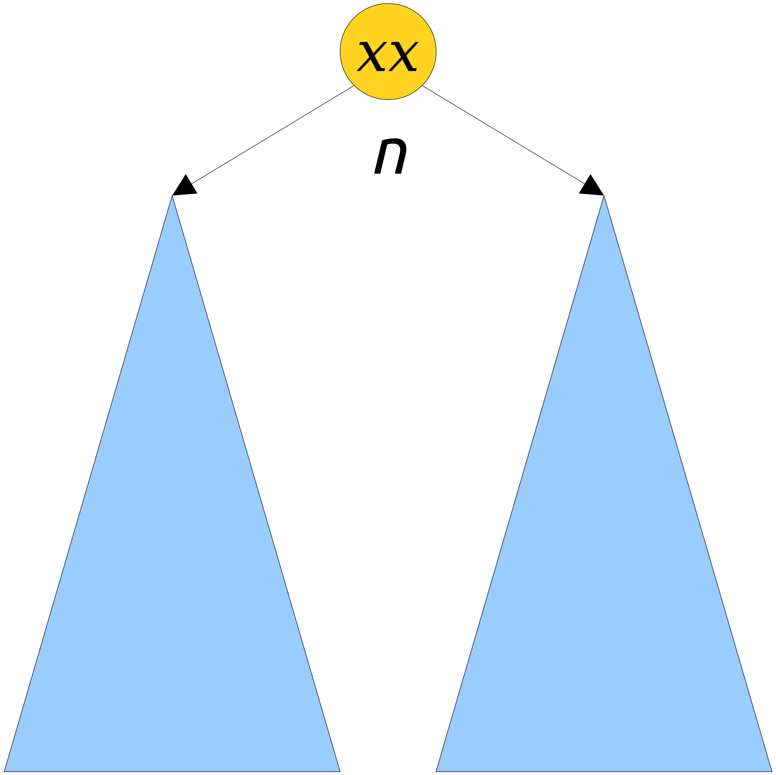
Tree Sizes

- To determine ***size***(x):
  - Splay x.



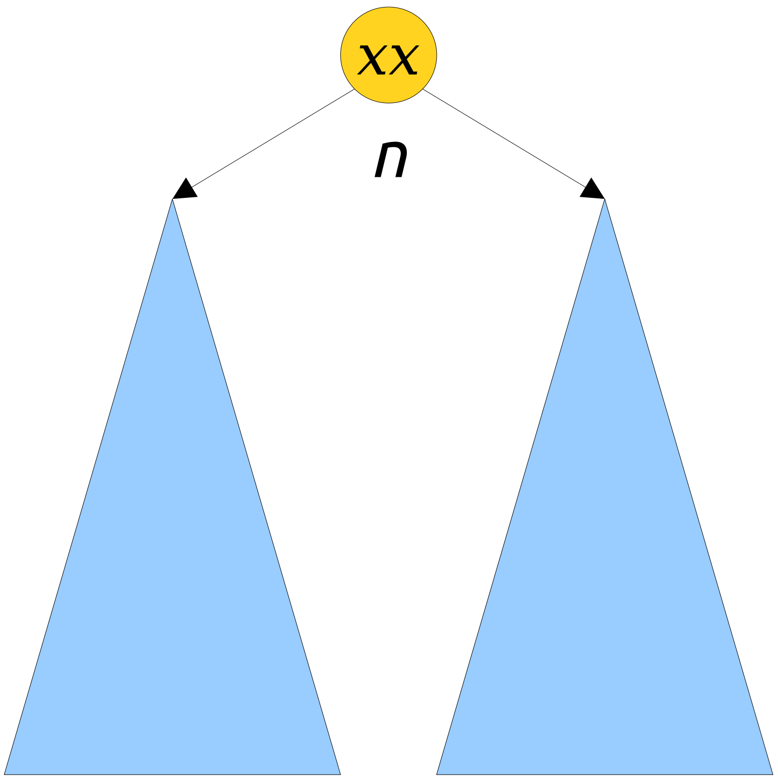
Tree Sizes

- To determine ***size***(x):
  - Splay x.
  - Return the augmented value in the node for x.



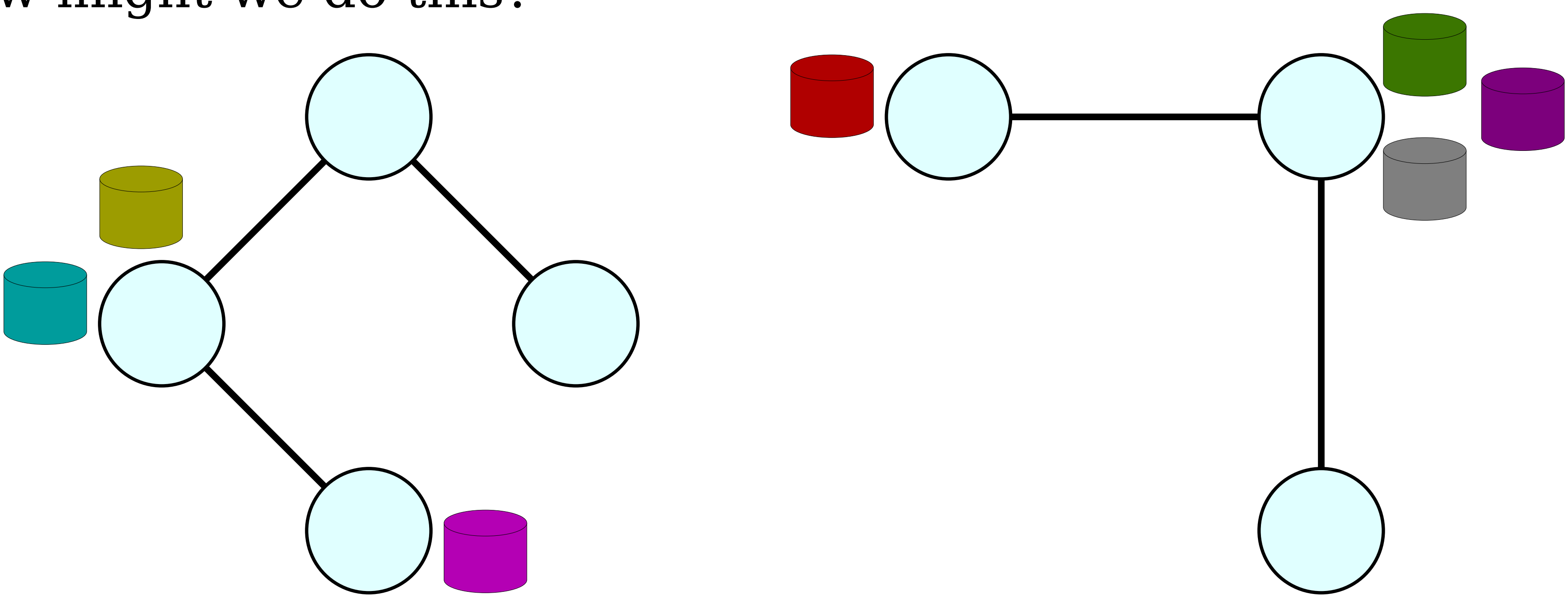
Tree Sizes

- To determine ***size***(x):
  - Splay xx.
  - Return the augmented value in the node for xx.
- Amortized cost: **O(log *n*)**.



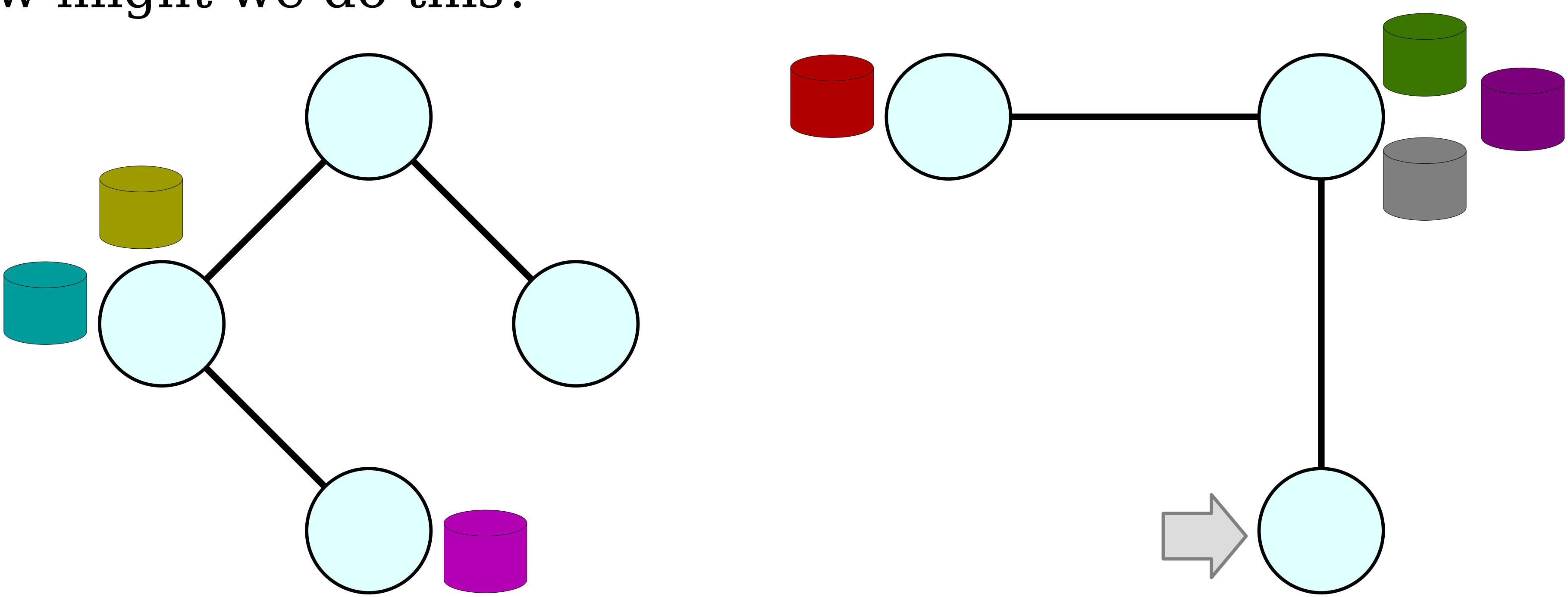
Extending Euler Tour Trees

- Suppose that each node represents a network router.
- We want to add these two operations:
  - ***add-packet***(x, p), which attaches packet p to node x; and
  - ***remove-packet***(x), which removes and returns some packet reachable from x, chosen arbitrarily from all the options.
- How might we do this?



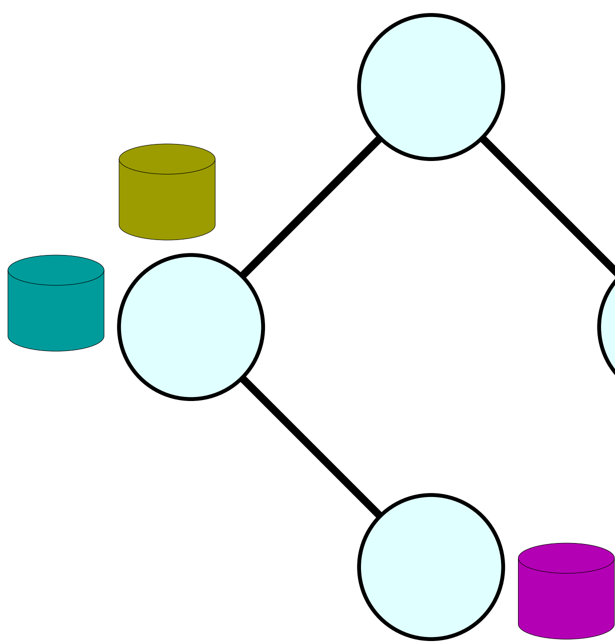
Extending Euler Tour Trees

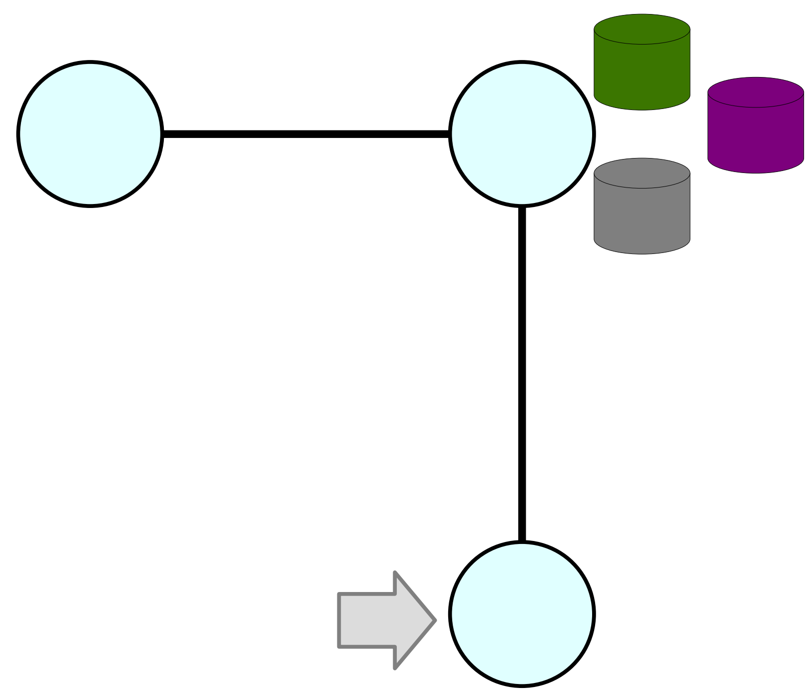
- Suppose that each node represents a network router.
- We want to add these two operations:
  - ***add-packet***(x, p), which attaches packet p to node x; and
  - ***remove-packet***(x), which removes and returns some packet reachable from x, chosen arbitrarily from all the options.
- How might we do this?



Extending Euler Tour Trees

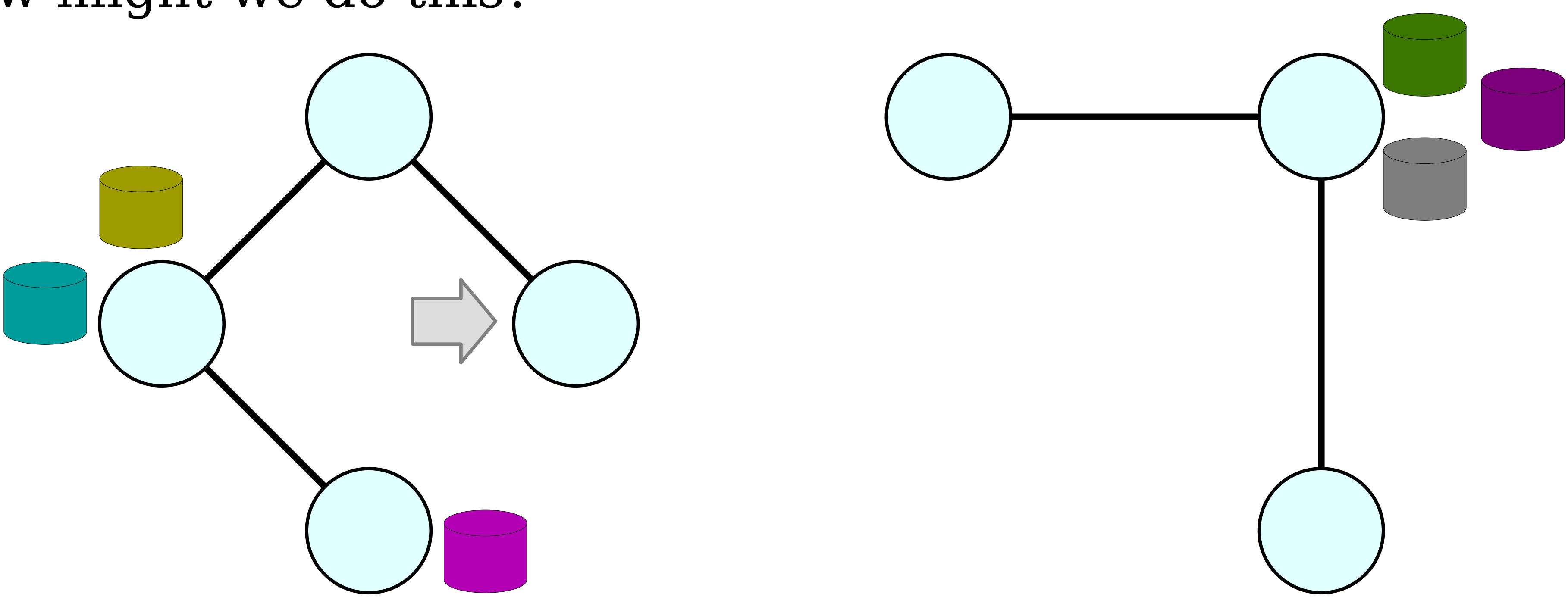
- Suppose that each node represents a network router.
- We want to add these two operations:
  - ***add-packet***(x, p), which attaches packet p to node x; and
  - ***remove-packet***(x), which removes and returns some packet reachable from x, chosen arbitrarily from all the options.
- How might we do this?





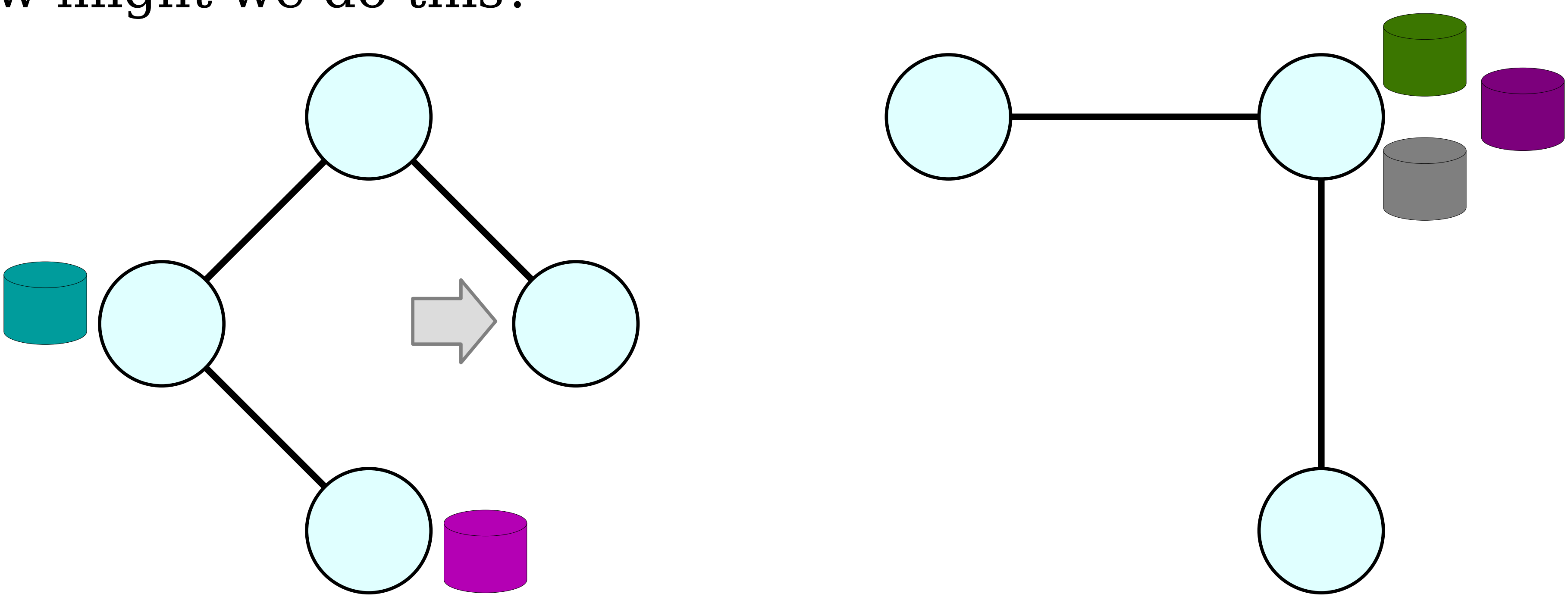
Extending Euler Tour Trees

- Suppose that each node represents a network router.
- We want to add these two operations:
  - ***add-packet***(x, p), which attaches packet p to node x; and
  - ***remove-packet***(x), which removes and returns some packet reachable from x, chosen arbitrarily from all the options.
- How might we do this?



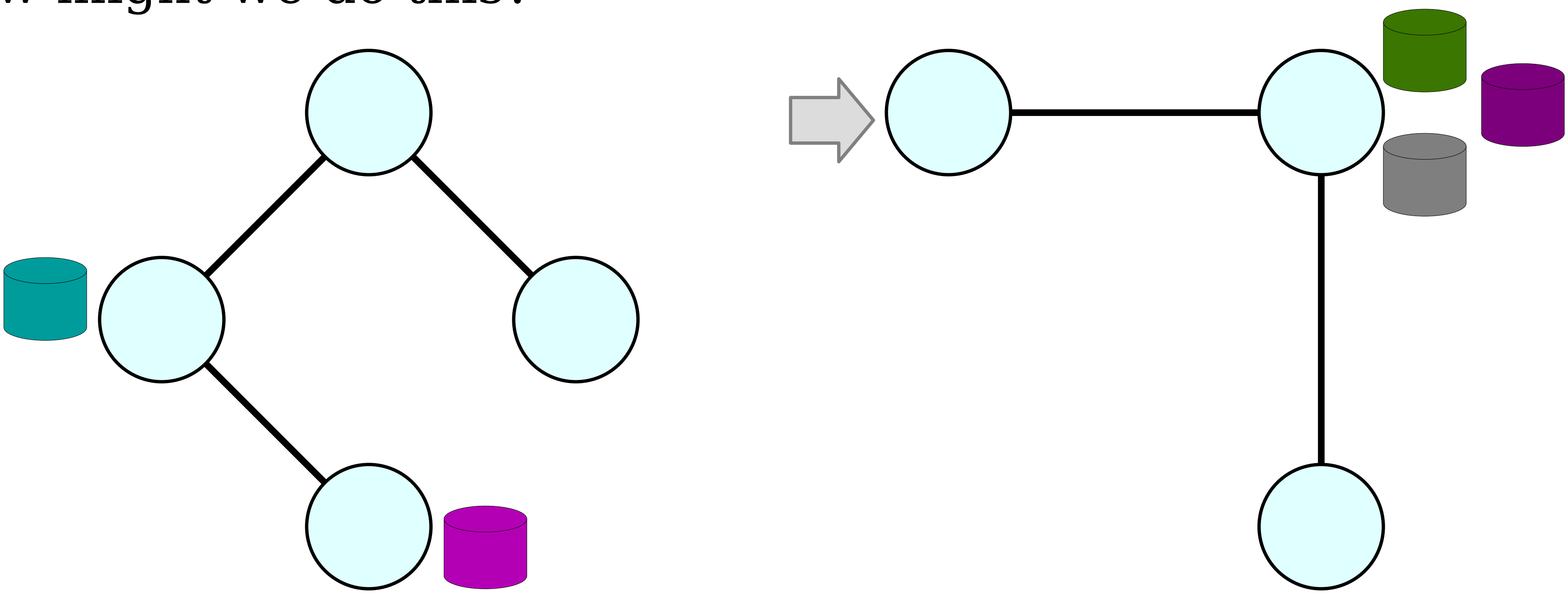
Extending Euler Tour Trees

- Suppose that each node represents a network router.
- We want to add these two operations:
  - ***add-packet***(x, p), which attaches packet p to node x; and
  - ***remove-packet***(x), which removes and returns some packet reachable from x, chosen arbitrarily from all the options.
- How might we do this?



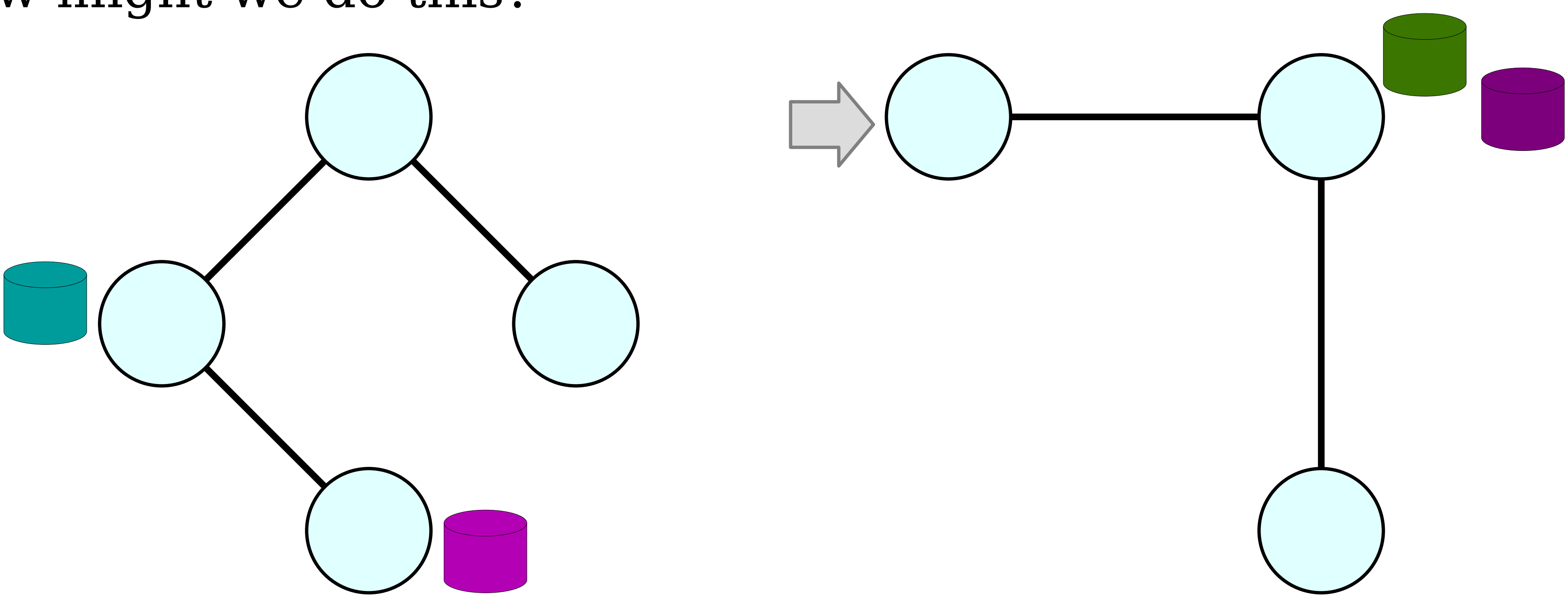
Extending Euler Tour Trees

- Suppose that each node represents a network router.
- We want to add these two operations:
  - ***add-packet***(x, p), which attaches packet p to node x; and
  - ***remove-packet***(x), which removes and returns some packet reachable from x, chosen arbitrarily from all the options.
- How might we do this?



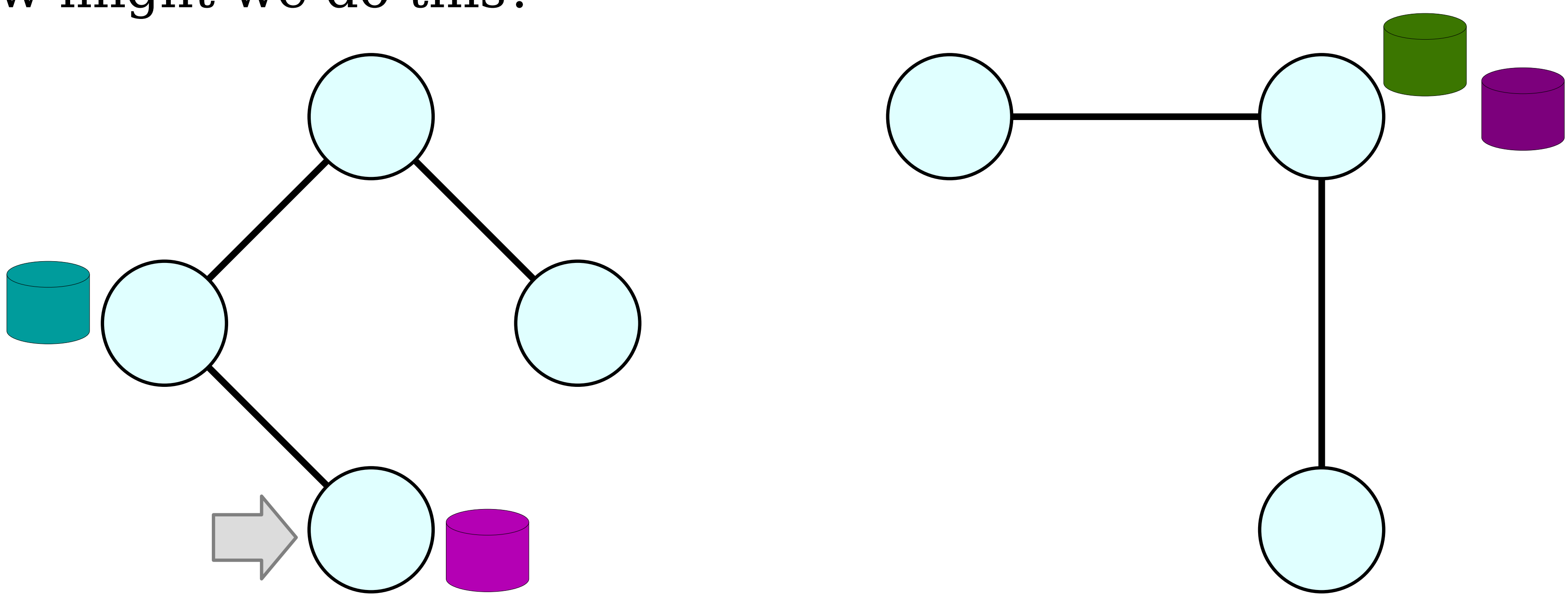
Extending Euler Tour Trees

- Suppose that each node represents a network router.
- We want to add these two operations:
  - ***add-packet***(x, p), which attaches packet p to node x; and
  - ***remove-packet***(x), which removes and returns some packet reachable from x, chosen arbitrarily from all the options.
- How might we do this?



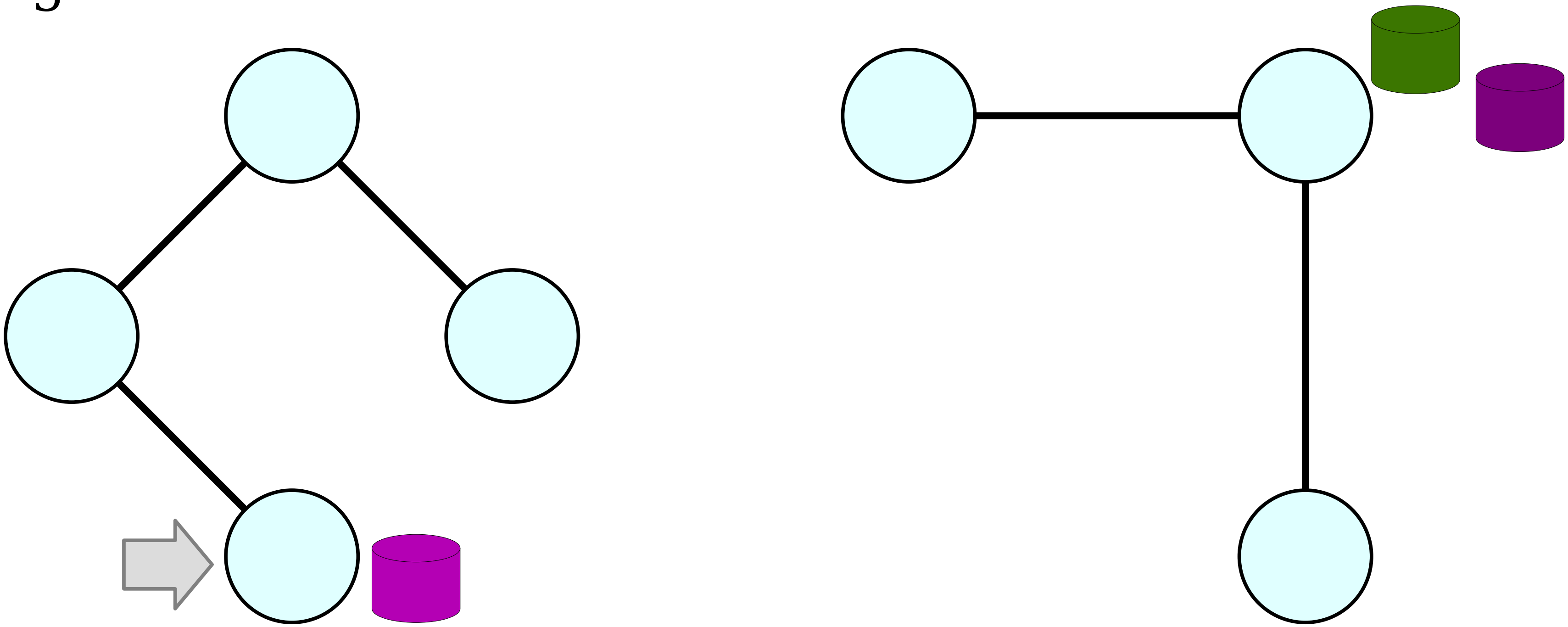
Extending Euler Tour Trees

- Suppose that each node represents a network router.
- We want to add these two operations:
  - ***add-packet***(x, p), which attaches packet p to node x; and
  - ***remove-packet***(x), which removes and returns some packet reachable from x, chosen arbitrarily from all the options.
- How might we do this?



Extending Euler Tour Trees

- Suppose that each node represents a network router.
- We want to add these two operations:
  - ***add-packet***(x, p), which attaches packet p to node x; and
  - ***remove-packet***(x), which removes and returns some packet reachable from x, chosen arbitrarily from all the options.
- How might we do this?



Packet Finding

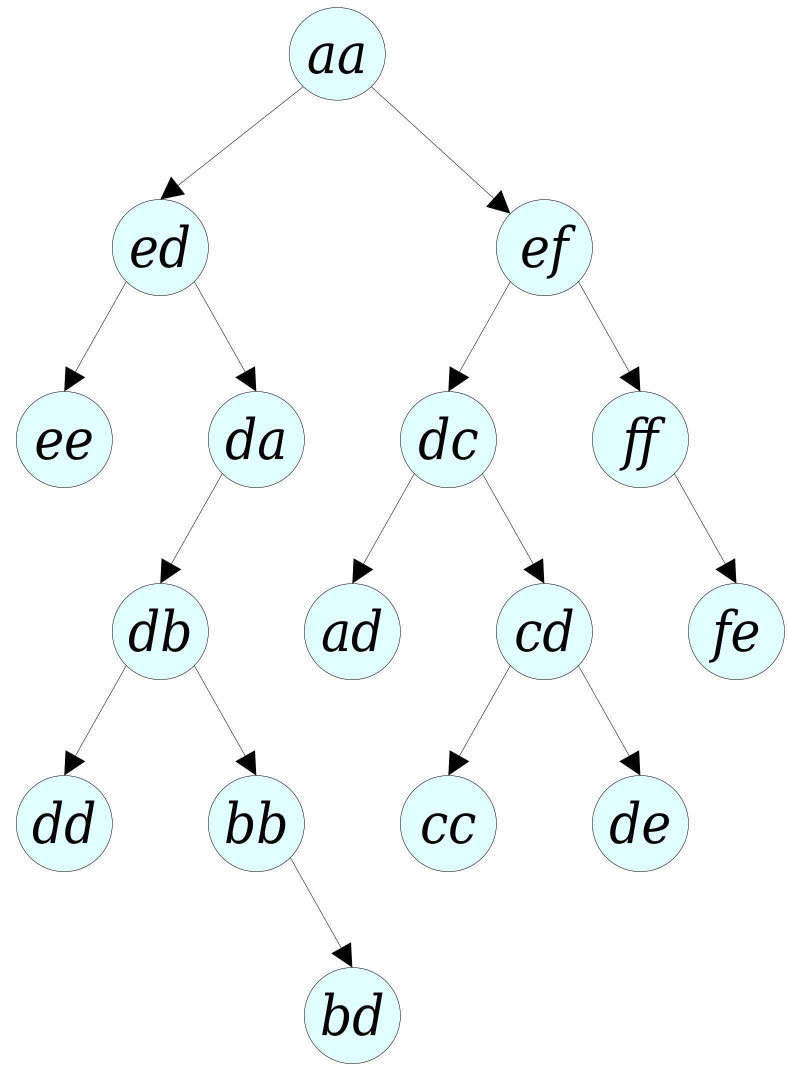
- Given the Euler tour representation of our trees, this essentially boils down to the following:

Augment a BST containing nodes and edges so that we can quickly identify a node with a packet.

- How might we do this?

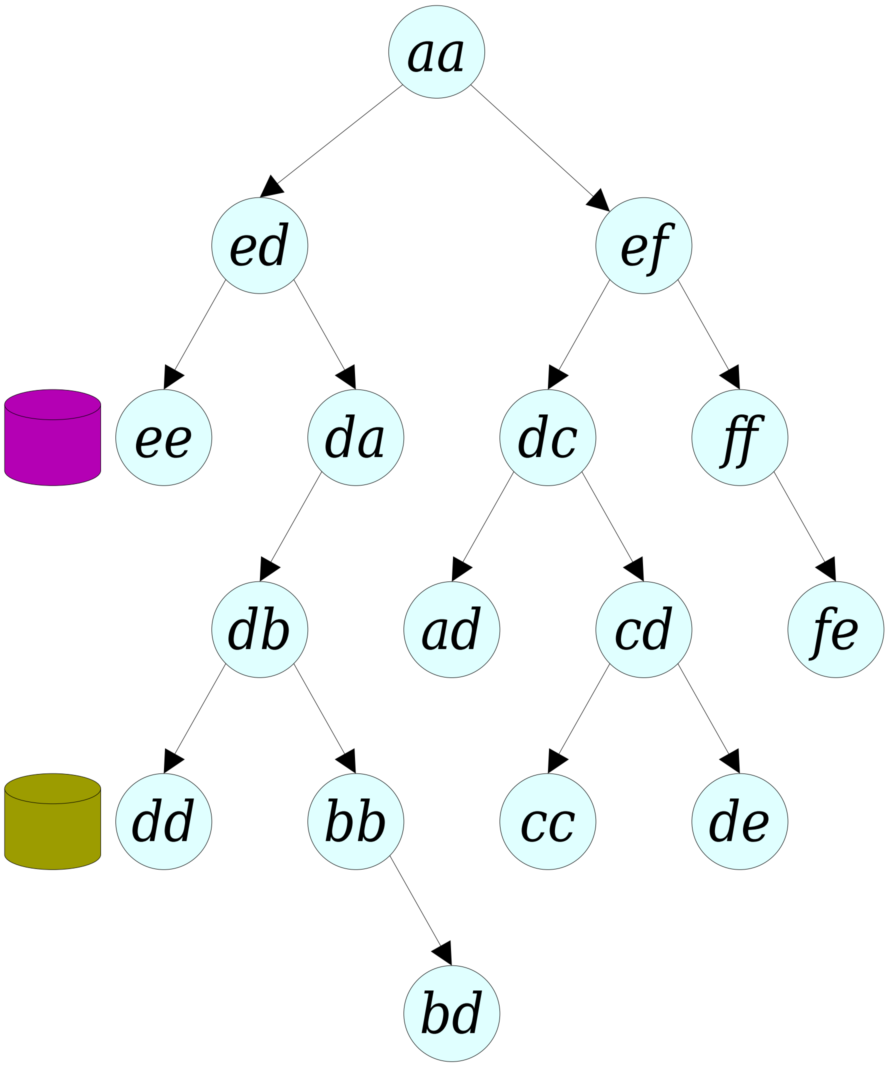
Packet Finding

- Augment each node *xx* with a list of the packets it stores.



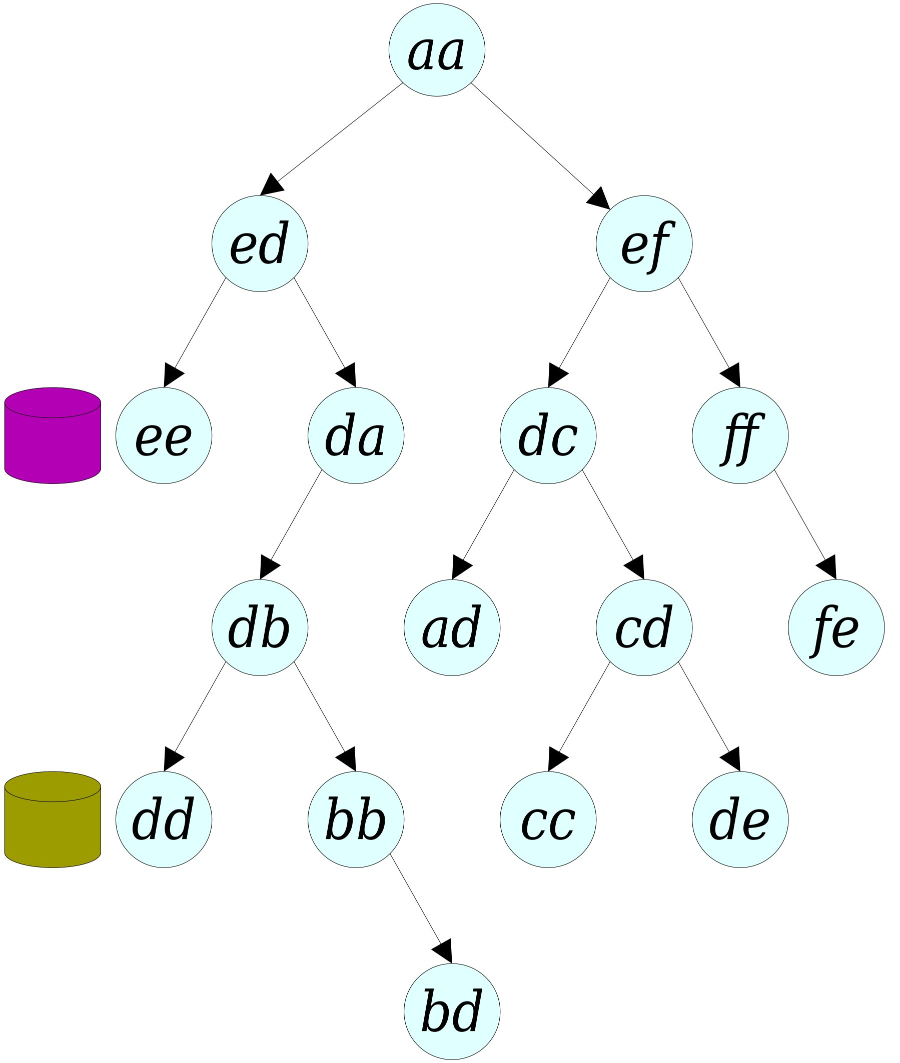
Packet Finding

- Augment each node *xx* with a list of the packets it stores.



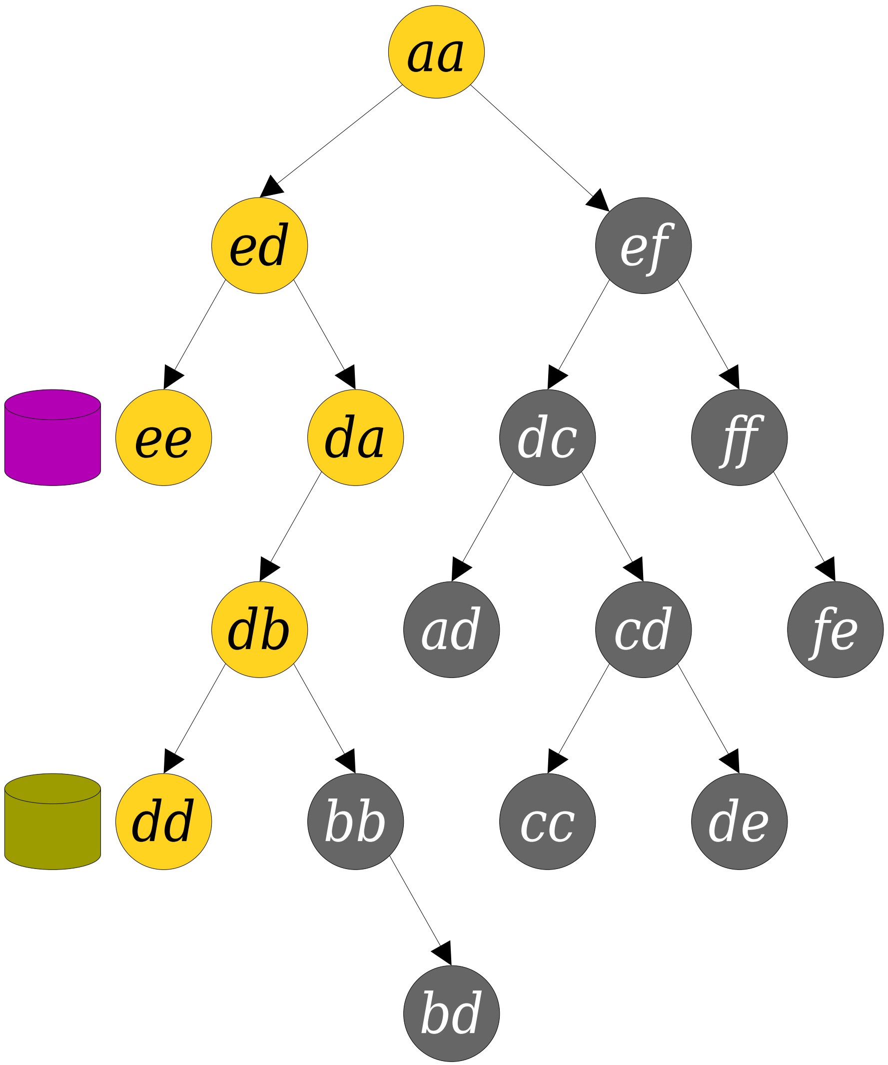
Packet Finding

- Augment each node *xx* with a list of the packets it stores.
- Augment each tree node with a bit indicating whether there's a packet in its subtree.



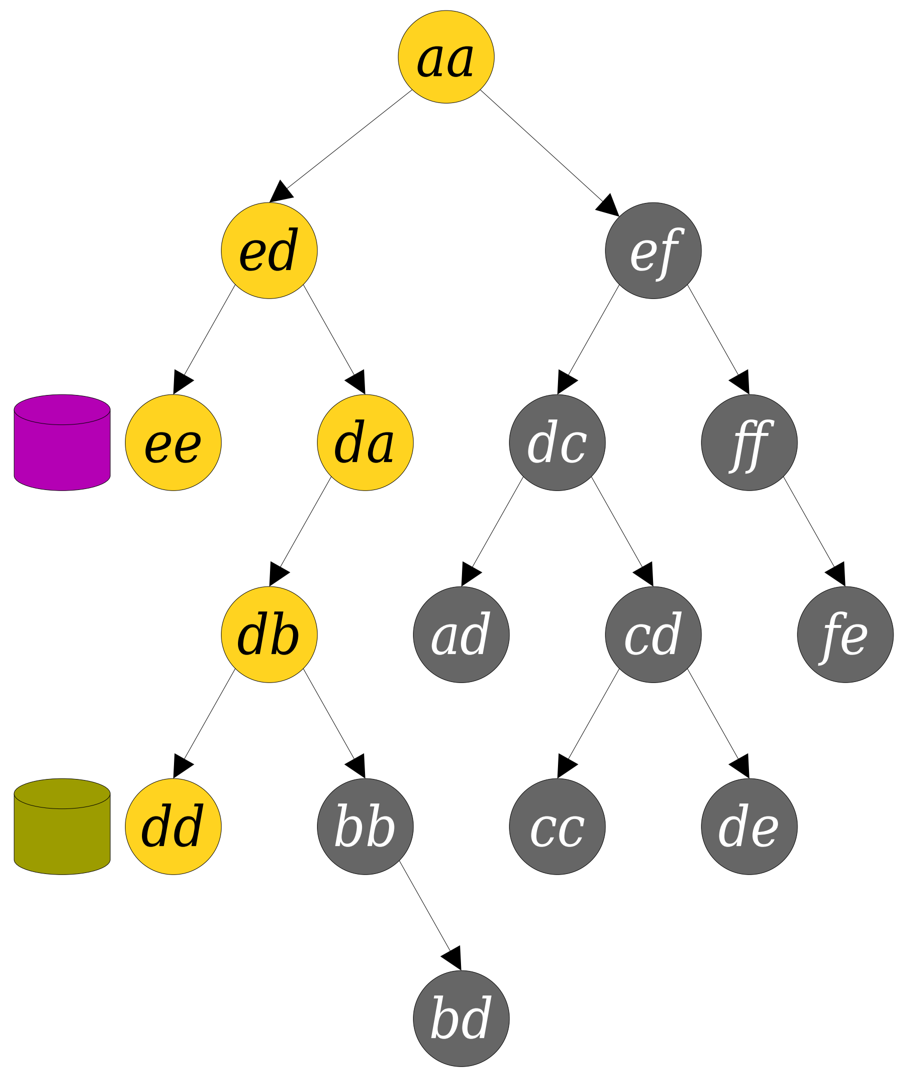
Packet Finding

- Augment each node *xx* with a list of the packets it stores.
- Augment each tree node with a bit indicating whether there's a packet in its subtree.



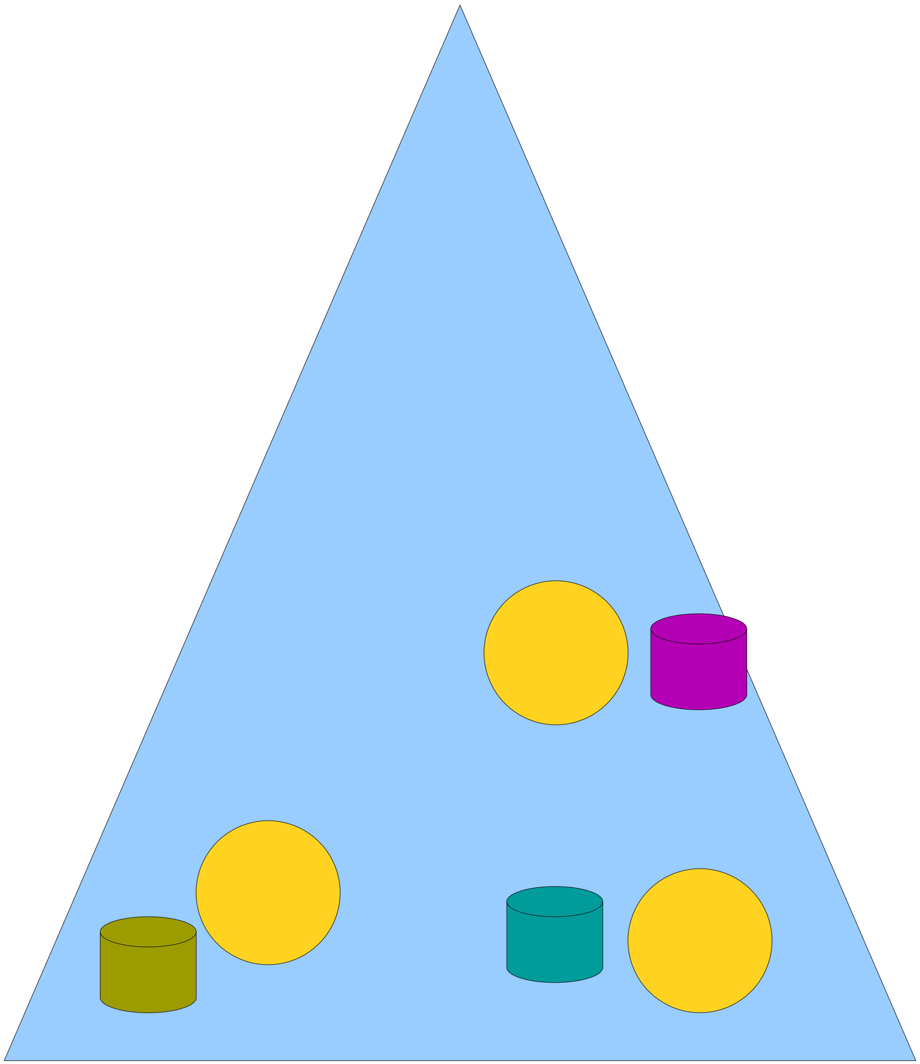
Packet Finding

- Augment each node *xx* with a list of the packets it stores.
- Augment each tree node with a bit indicating whether there's a packet in its subtree.
- We can use this latter information to quickly find nodes holding packets.



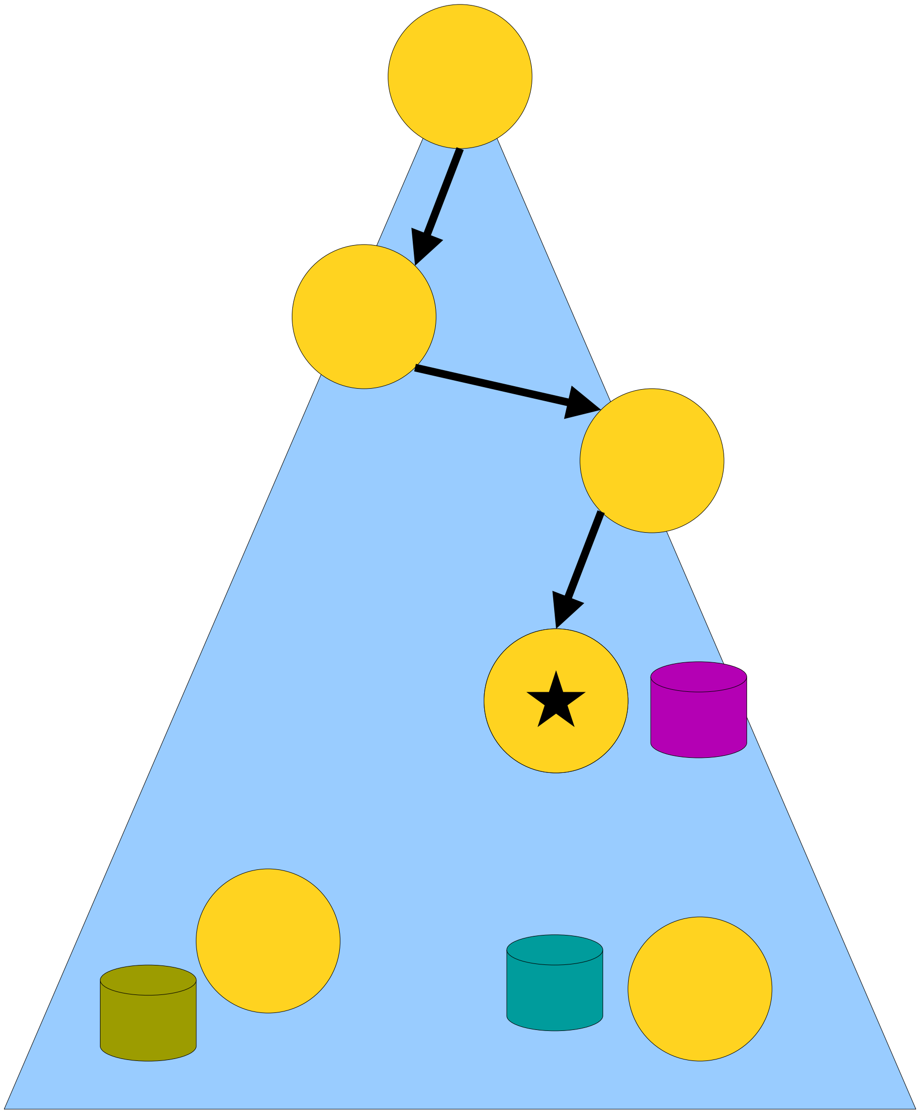
Packet Finding

- To find and remove a packet:



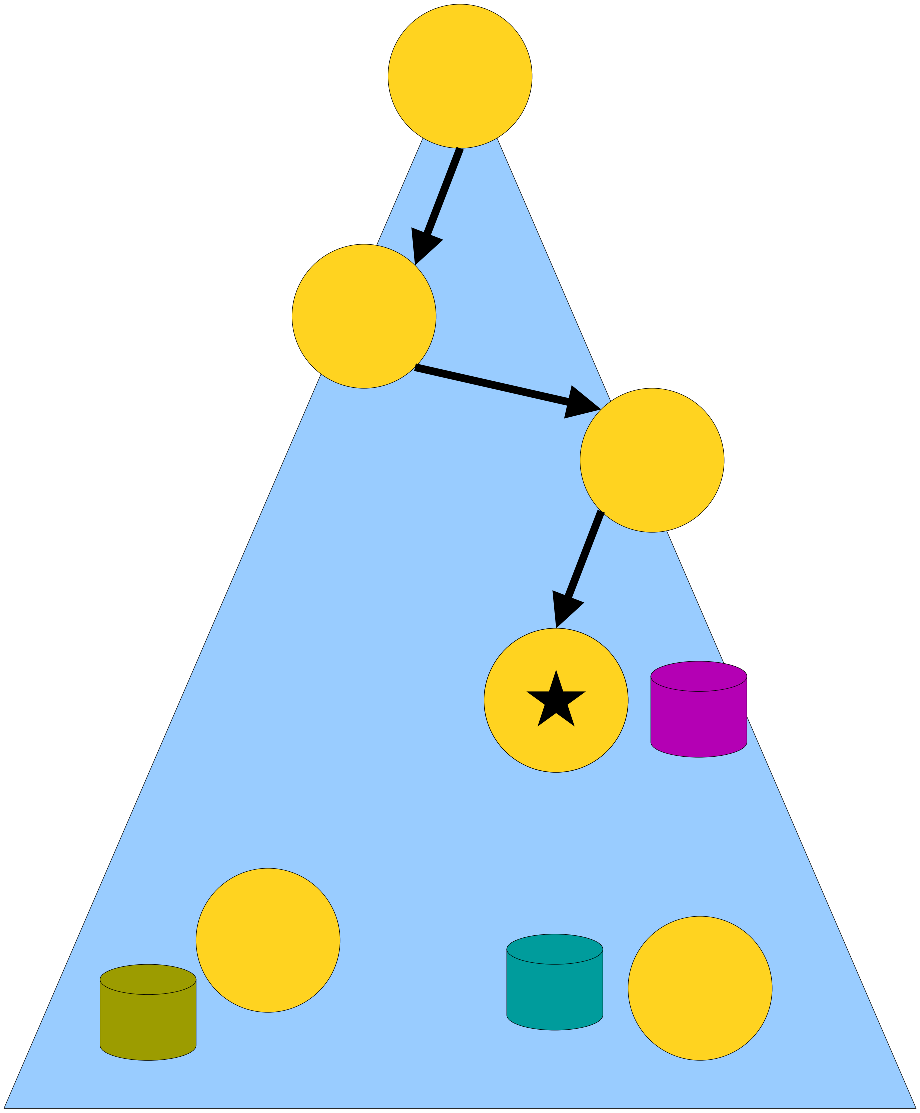
Packet Finding

- To find and remove a packet:
  - Walk from the root to any node containing a packet, using the augmentation to guide the search.



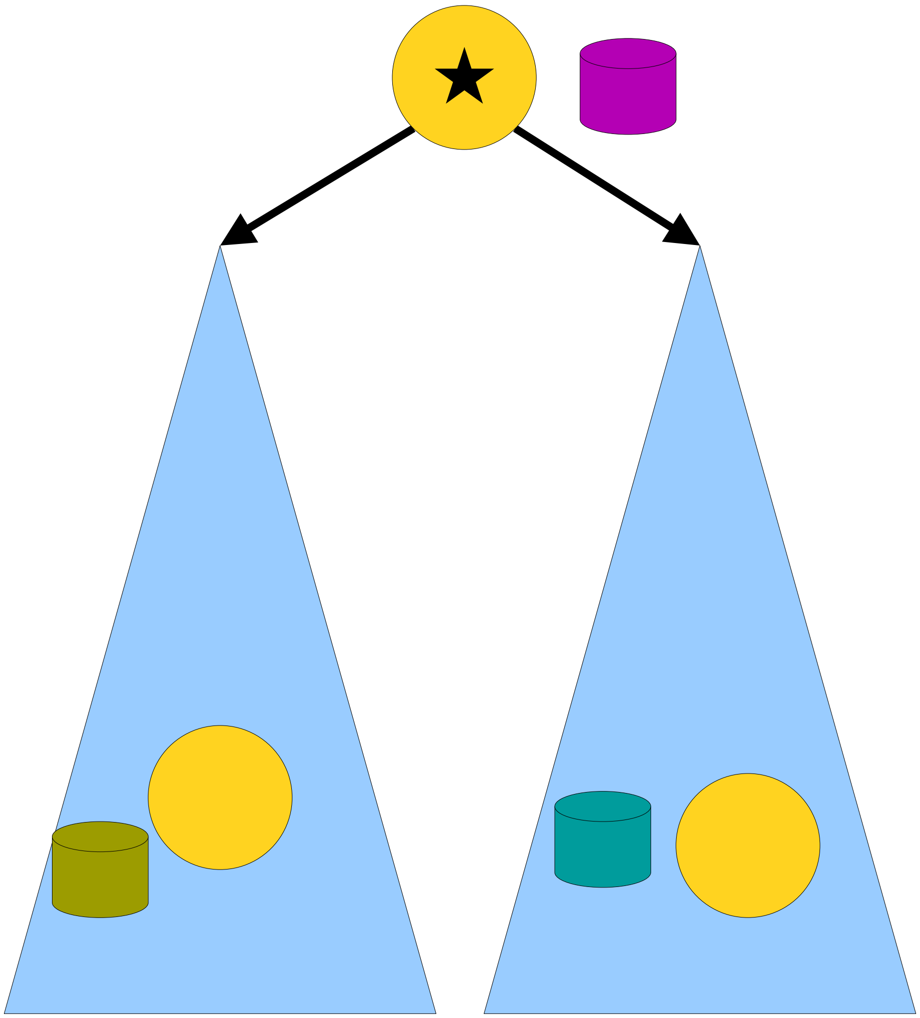
Packet Finding

- To find and remove a packet:
  - Walk from the root to any node containing a packet, using the augmentation to guide the search.
  - Splay that node to the root.



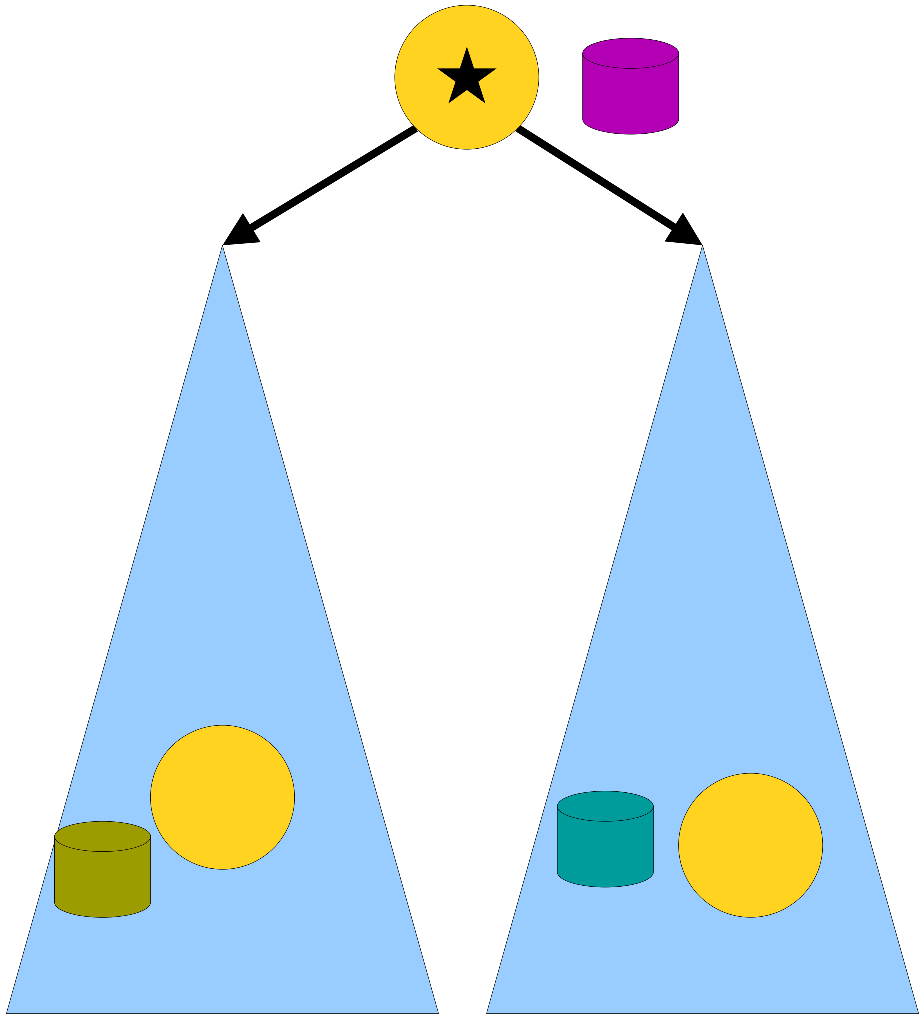
Packet Finding

- To find and remove a packet:
  - Walk from the root to any node containing a packet, using the augmentation to guide the search.
  - Splay that node to the root.



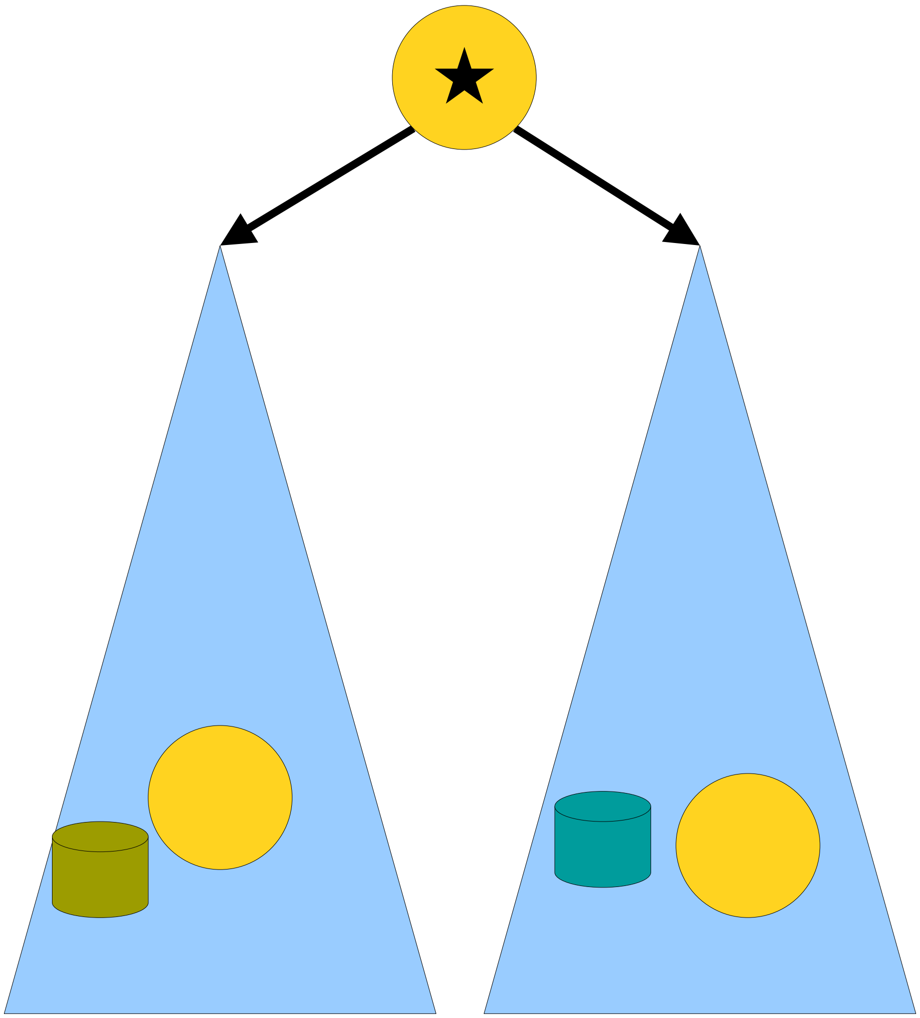
Packet Finding

- To find and remove a packet:
  - Walk from the root to any node containing a packet, using the augmentation to guide the search.
  - Splay that node to the root.
  - Remove a packet from it, updating the root's augmentation.



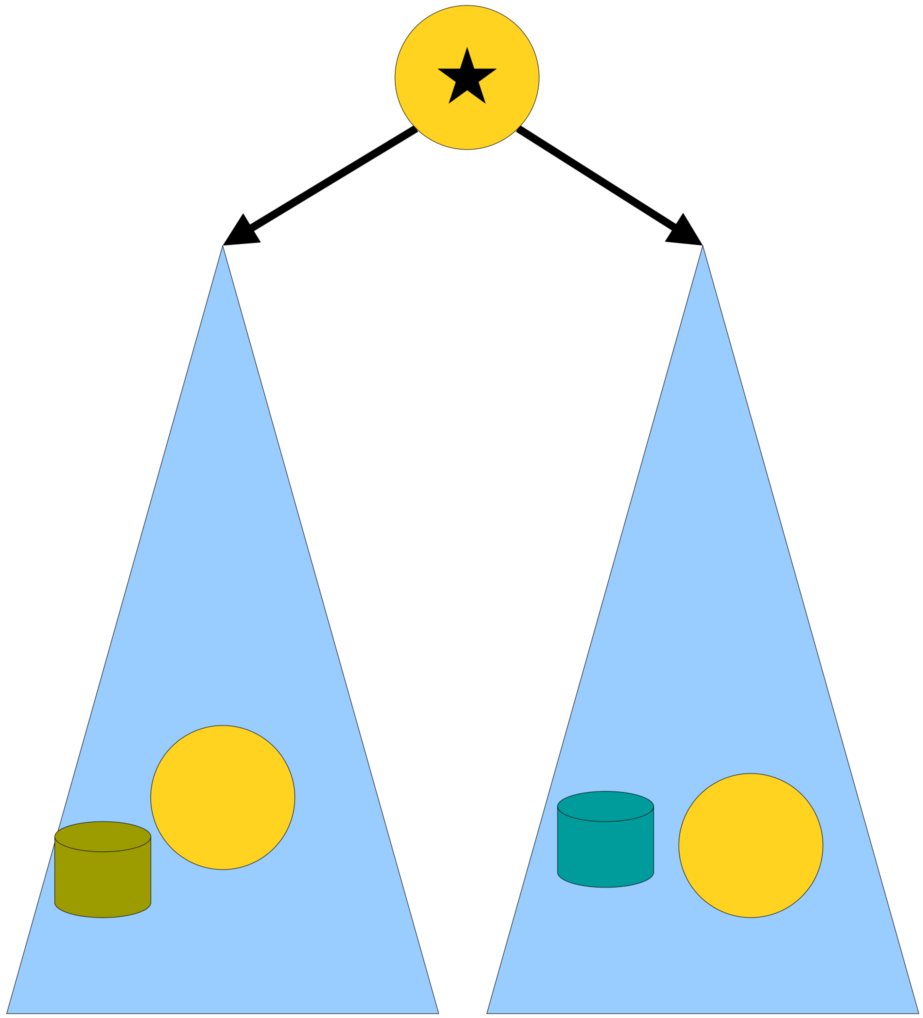
Packet Finding

- To find and remove a packet:
  - Walk from the root to any node containing a packet, using the augmentation to guide the search.
  - Splay that node to the root.
  - Remove a packet from it, updating the root's augmentation.



Packet Finding

- To find and remove a packet:
  - Walk from the root to any node containing a packet, using the augmentation to guide the search.
  - Splay that node to the root.
  - Remove a packet from it, updating the root's augmentation.
- Amortized cost: **O(log *n*)**.



Generalizing This Idea

- More generally, Euler tour trees play well with augmentations that care about global properties of individual trees.
- There's another way to use splay trees to encode dynamic trees (***st-trees***, also called ***link/cut trees***, though the later name is ambiguous) that works well for augmenting over ***paths*** in trees rather than trees as a whole.
- (Check out the Sleator/Tarjan paper for more details.)

Next Time

- ***Fully-Dynamic Connectivity***
  - Solving connectivity in general graphs, not just forests.
- ***“Blame It On The Little Guy”***
  - A surprisingly versatile algorithmic strategy.
- ***Holm’s Structure***
  - An elegant way to solve dynamic connectivity by harnessing augmented ETTs.