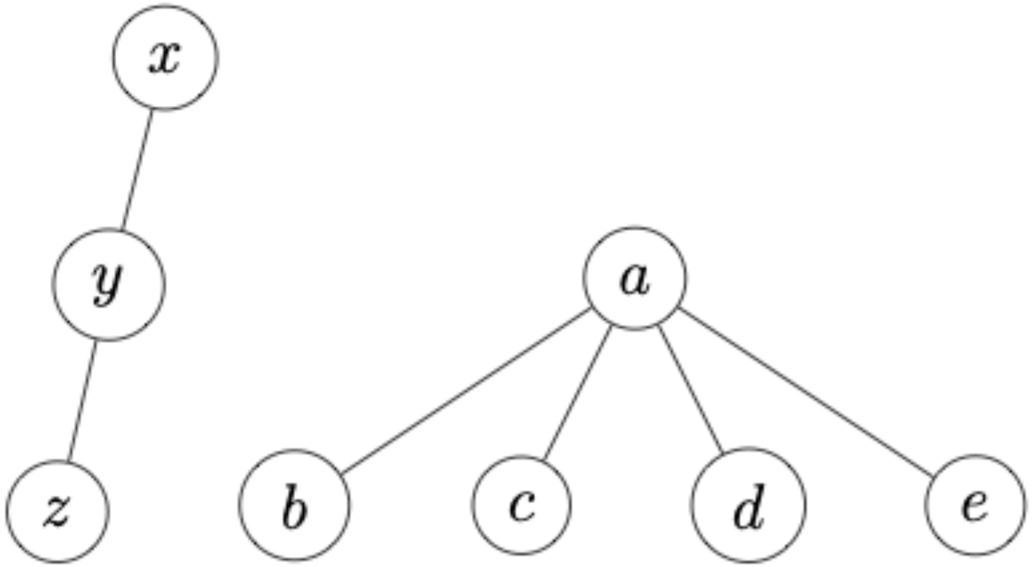
EECS 4020 Algorithms

HW5

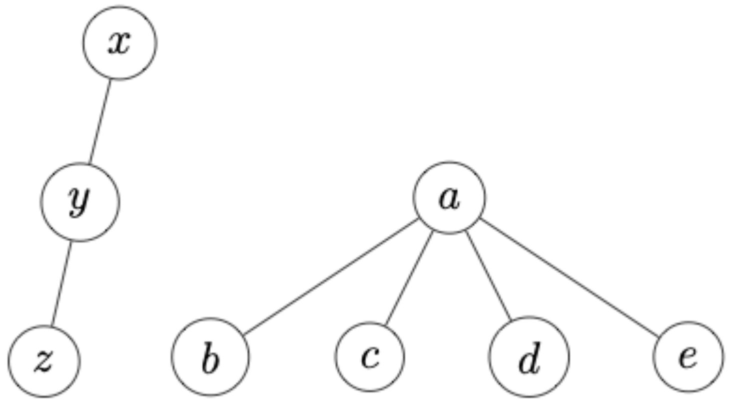
I. Disjoint Sets

Q1

Consider maintaining a set by trees as follows:



Q1(a)

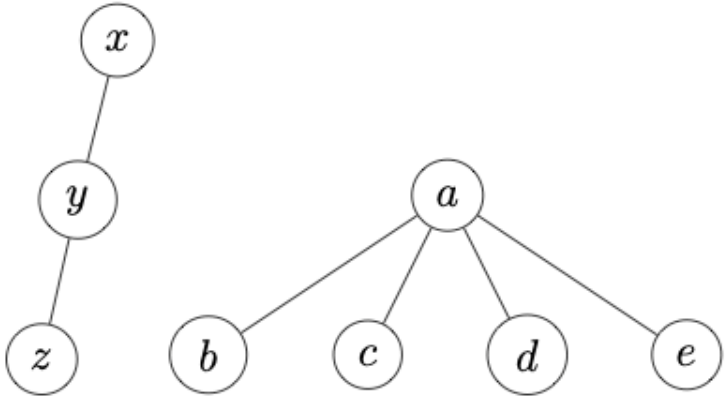


Using Union-by-Size and Path Compression

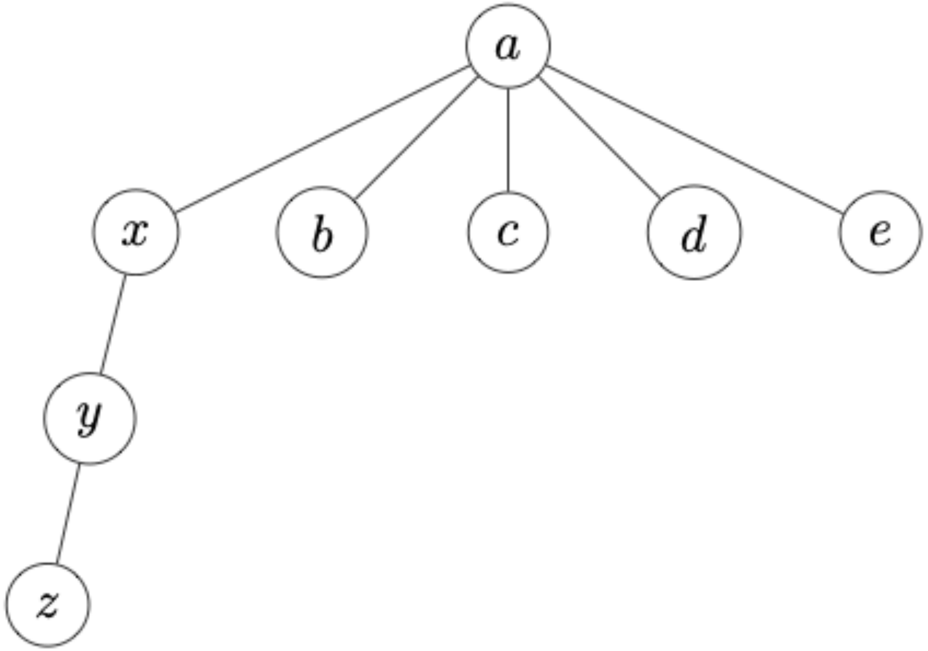
what will happen

after Union(a, x) and then Find(y) ?

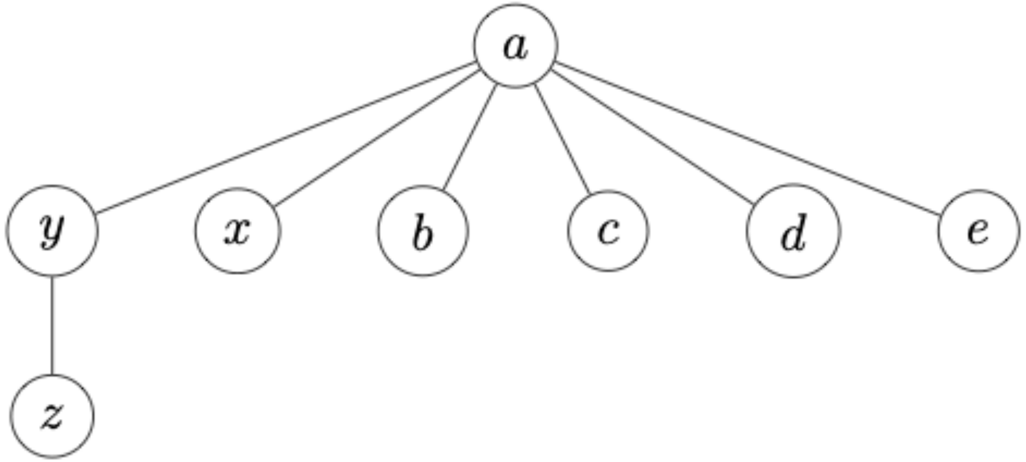
Q1(a) [solution]



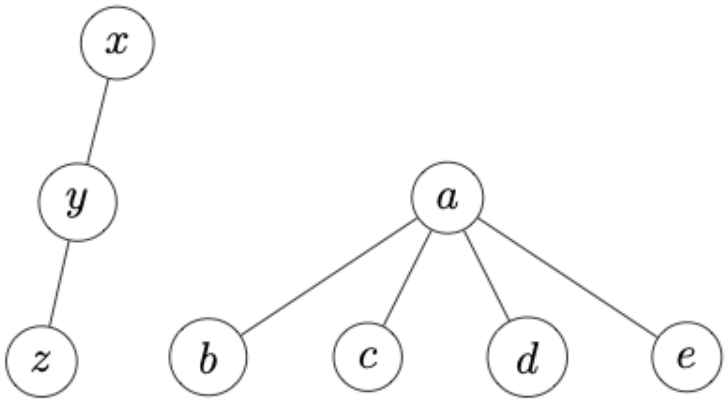
After Union(a, x) :



Then Find(y) :



Q1(b)



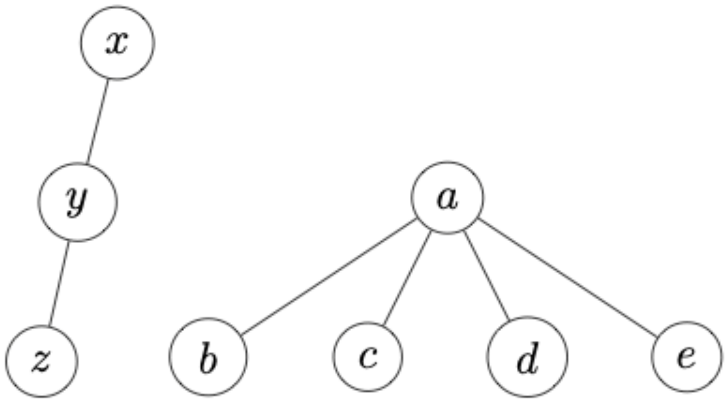
Assume rank(x) > rank(a).

Using Union-by-Rank and Path Compression

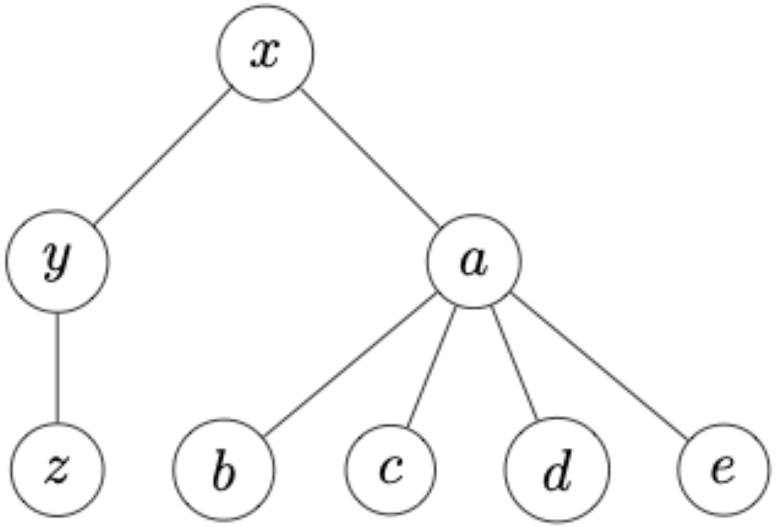
what will happen

after Union(a, x) and then Find(y) ?

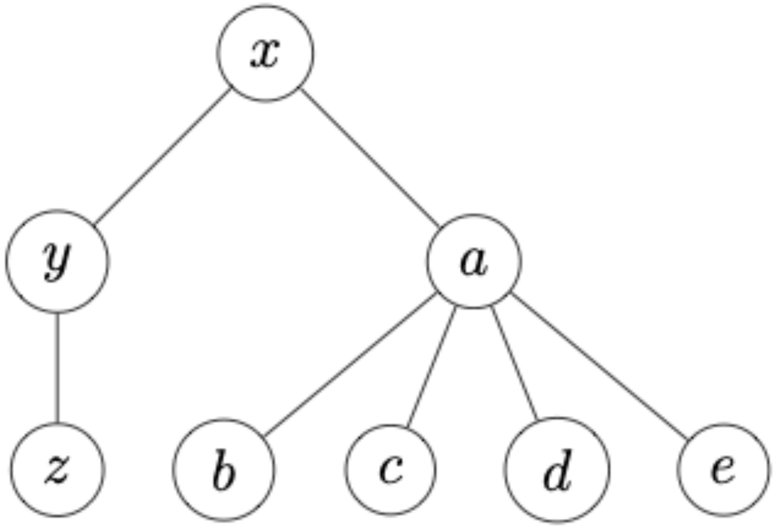
Q1(b) [solution]



After Union(a, x) :



Then Find(y) :



Q2

We use a set of trees to maintain Union-Find

- Currently, the tree has k edges
- Heuristic: Path Compression

Show that if we next perform n find operations

total time = O(n + k)

Q2 [solution]

- Call an edge a root edge if it is directly connected to some root of a tree
- Give $1 to each non-root edge initially
- Each find involves a couple of non-root edges and at most one root edge

  ➔ non-root edge becomes root edge afterwards

Q2 [solution]

- Cost of accessing a non-root edge can be paid by the initial \$1

➔ Total cost to access non-root edge = $O(k)$

➔ Total cost to access root edge = $O(n)$

➔ n operations has total cost $O(n + k)$

II. BFS and DFS

Q1

- Let T be an undirected tree
- The diameter of T is the distance (# edges) between the farthest two nodes in T

How to find diameter of T in linear time?

Q1 [solution]

Idea 1: Use DP

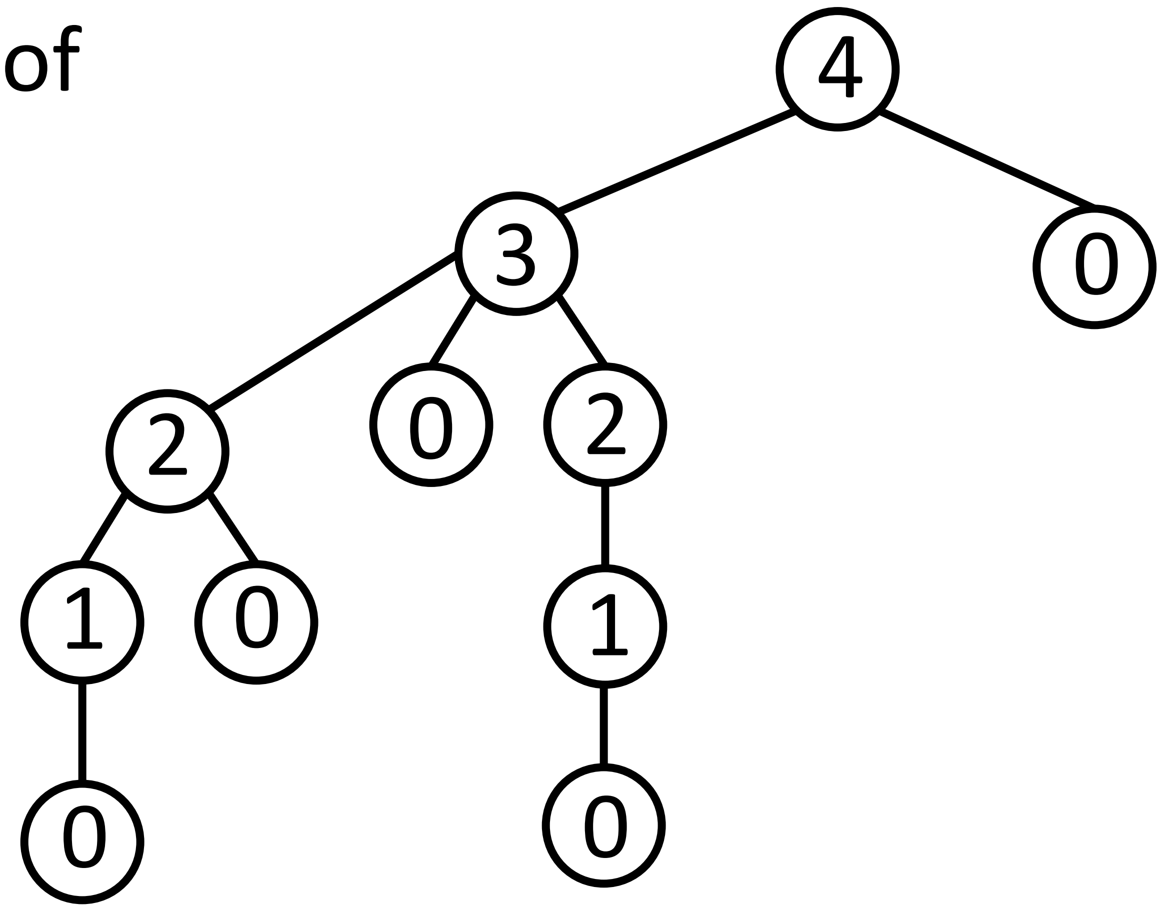
- Turn T into a rooted tree and perform DFS on T

➔ Root x of each subtree computes the distance of the farthest leaf from x

➔ Distance between farthest leaves that pass through x can be computed in O(deg(x)) time

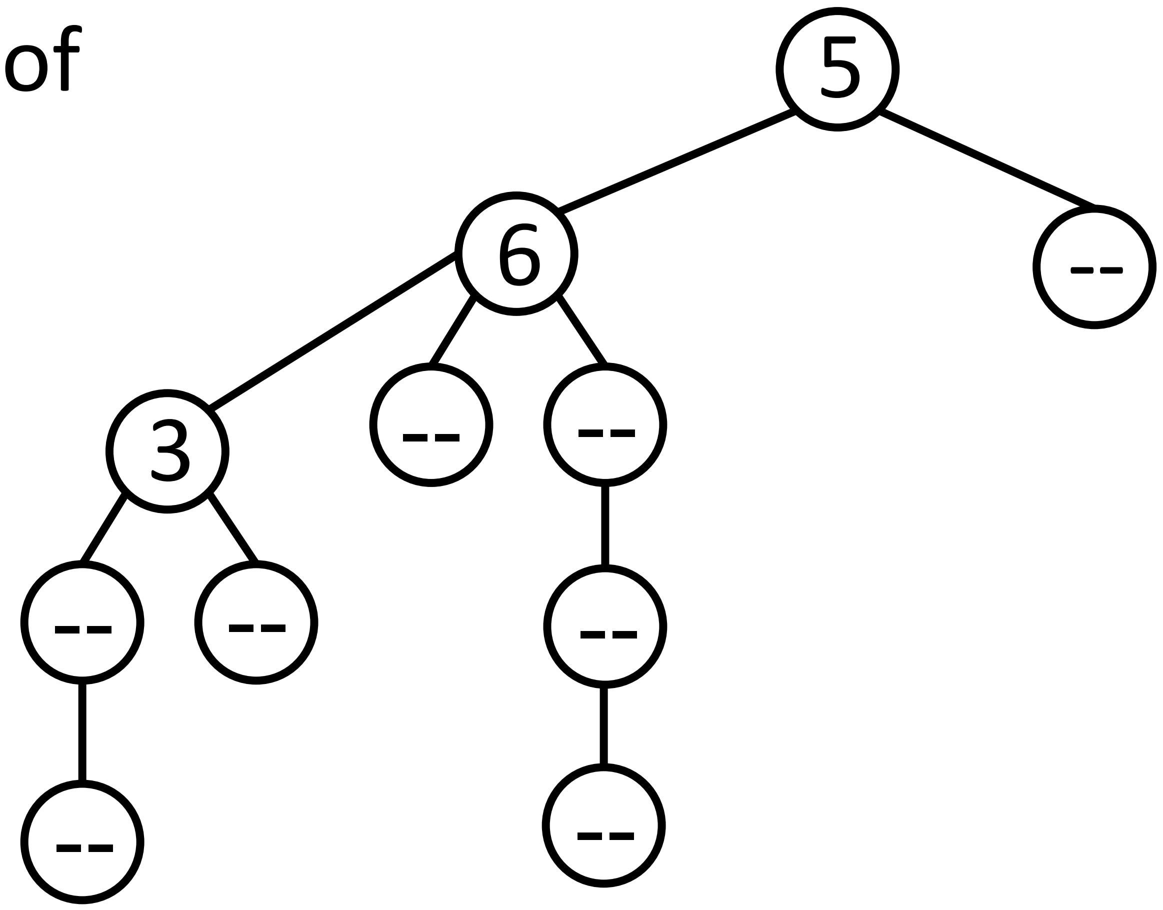
Q1 [solution]

1. Compute distance of
   of farthest leaf



Q1 [solution]

2. Compute distance of
   of farthest leaves
   passing through



Q1 [solution]

Idea 2: Use BFS twice

- Turn T into a rooted tree and perform BFS on T

  ➔ Find the farthest leaf y from the root of T

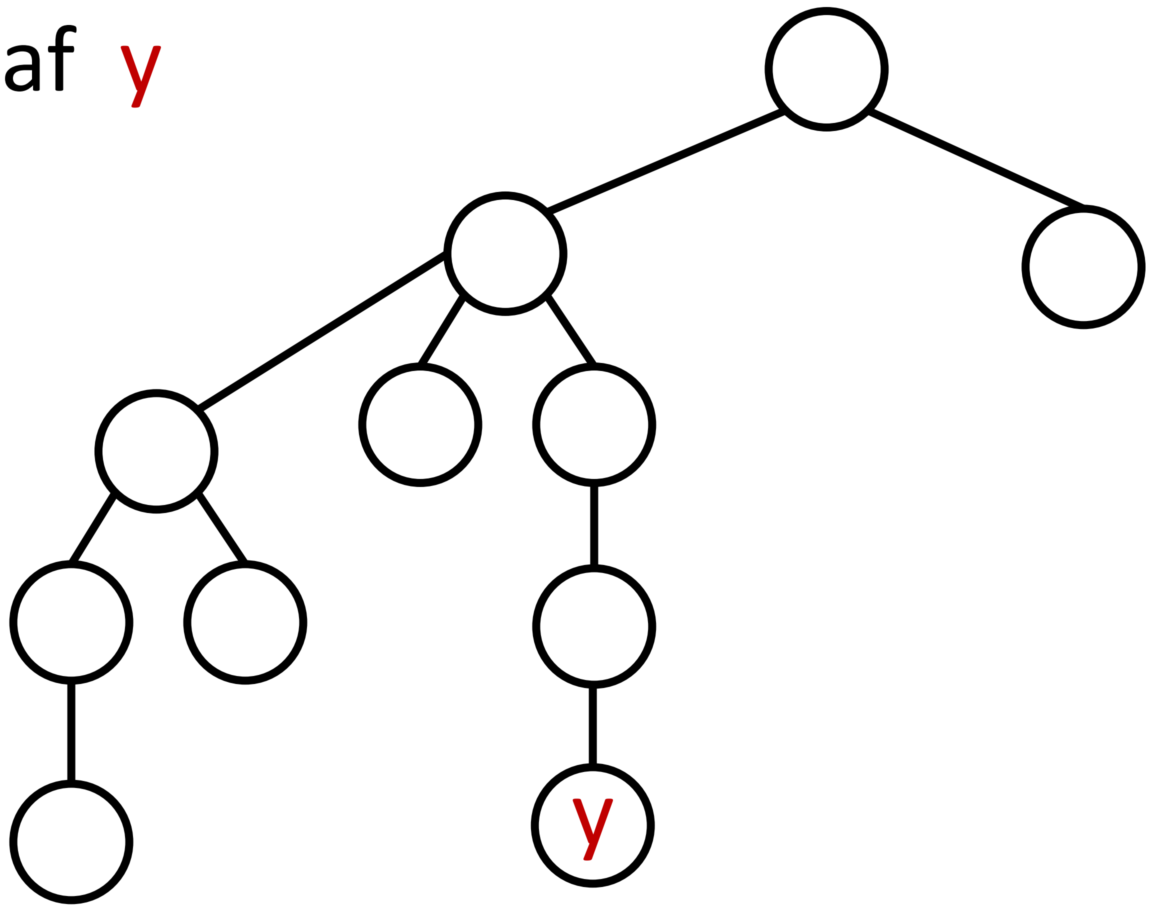
- Perform BFS from y

  ➔ Find the farthest leaf z from y

  ➔ y and z are the farthest nodes

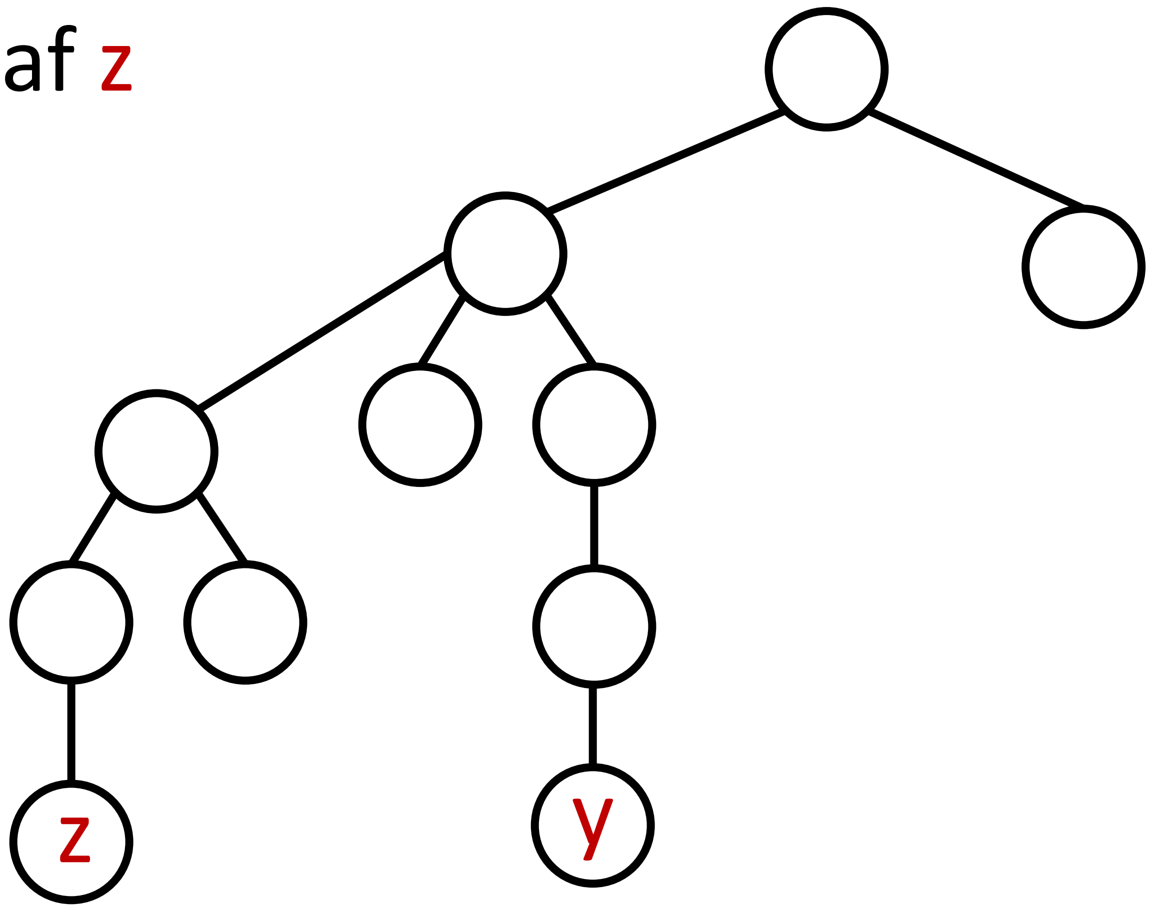
Q1 [solution]

1. Find the farthest leaf y



Q1 [solution]

2. Find the farthest leaf z
   from y



Q1 [solution]

Idea 2: Use BFS twice

- Why does this algorithm work?
- Did you see that this is a Greedy Algorithm?

➔ leaf y is always a good choice as one of the two nodes farthest apart

Q2

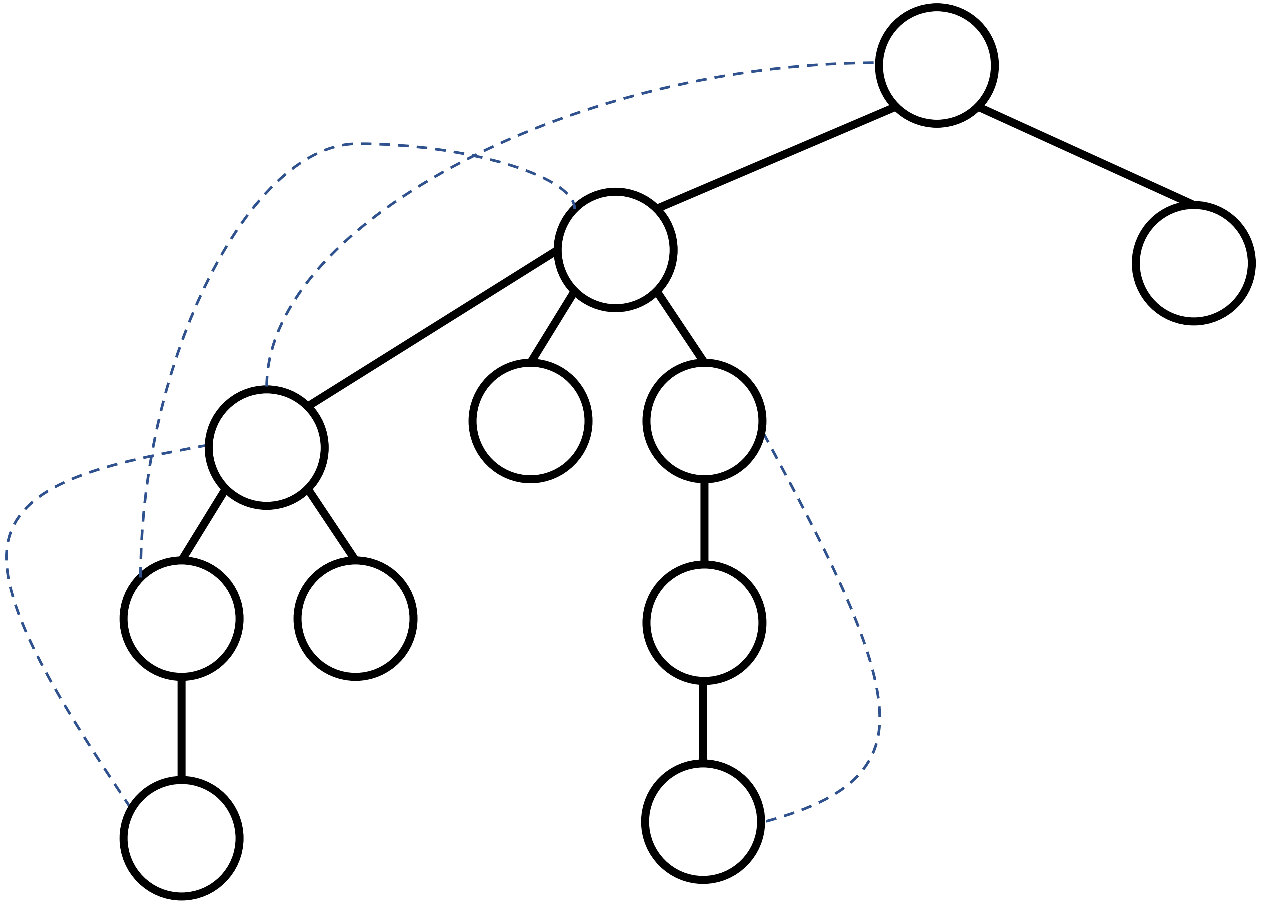
- Let G be a connected undirected graph
- A node v is an articulation point (AP)

  if removing v from G makes G disconnected

How to find all articulation points ?

Q2 [solution]

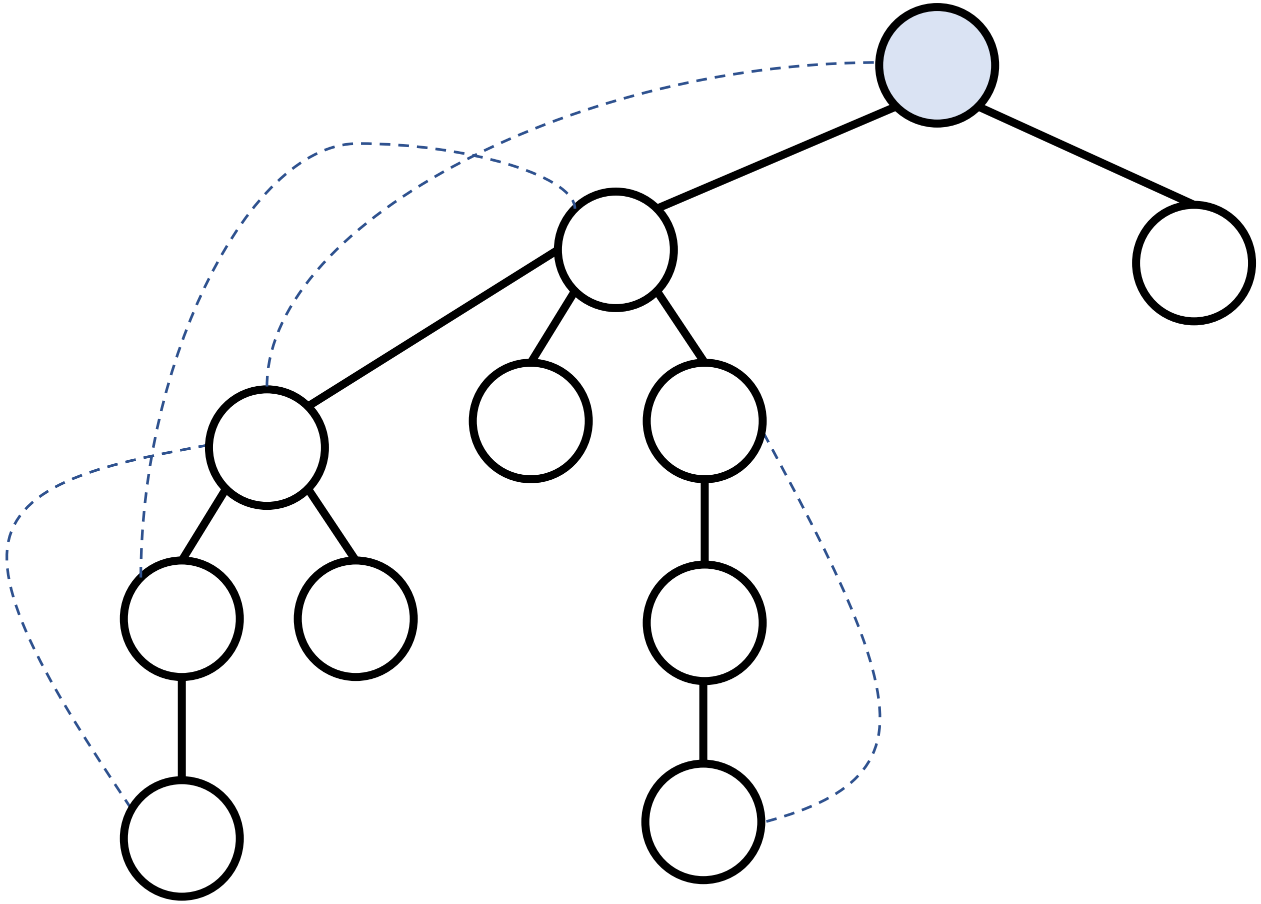
1. Perform DFS and get DFS tree T



Dotted lines are back edges

Q2 [solution]

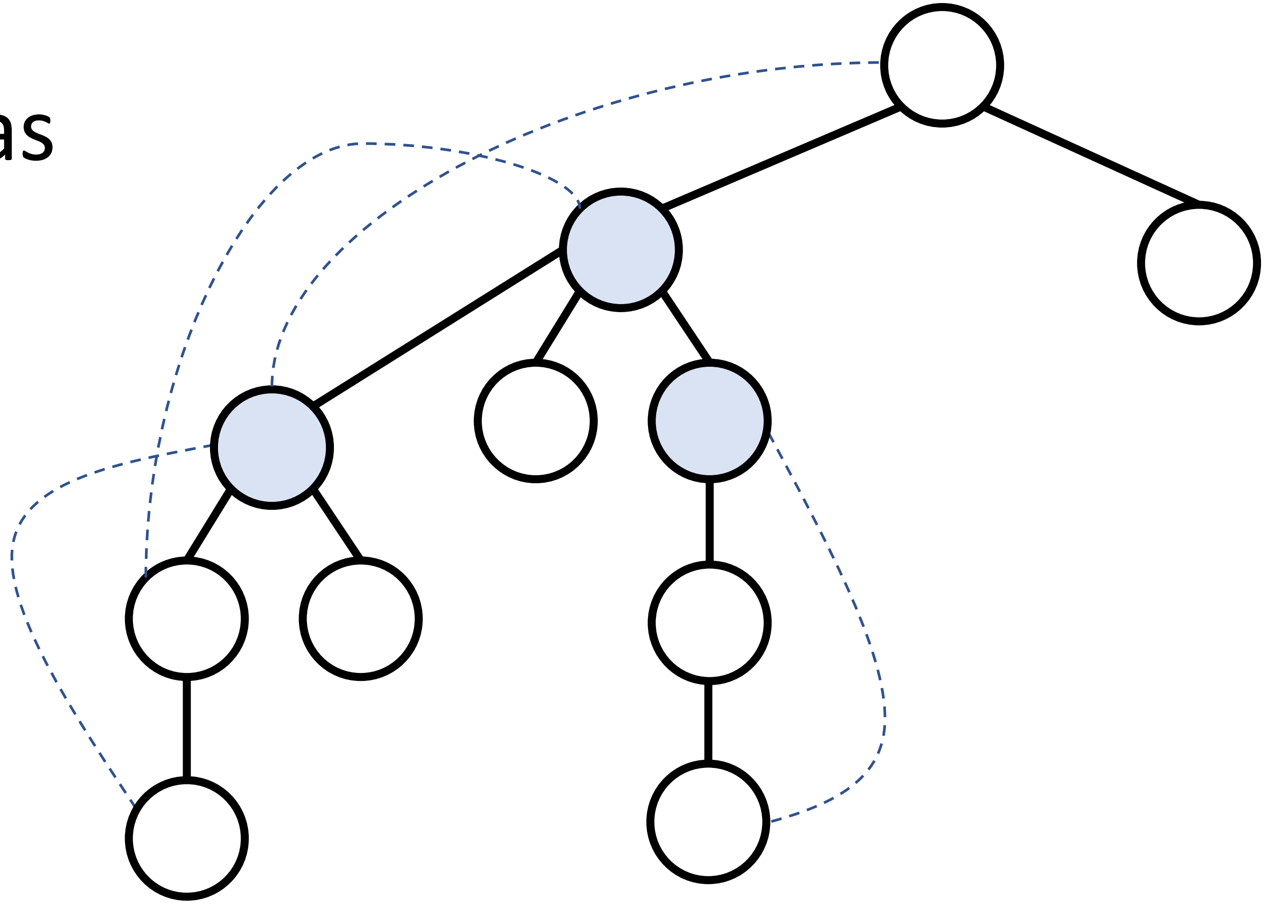
2. Root r of T has 2 or more children $\Leftrightarrow$ r is an AP



Dotted lines are back edges

Q2 [solution]

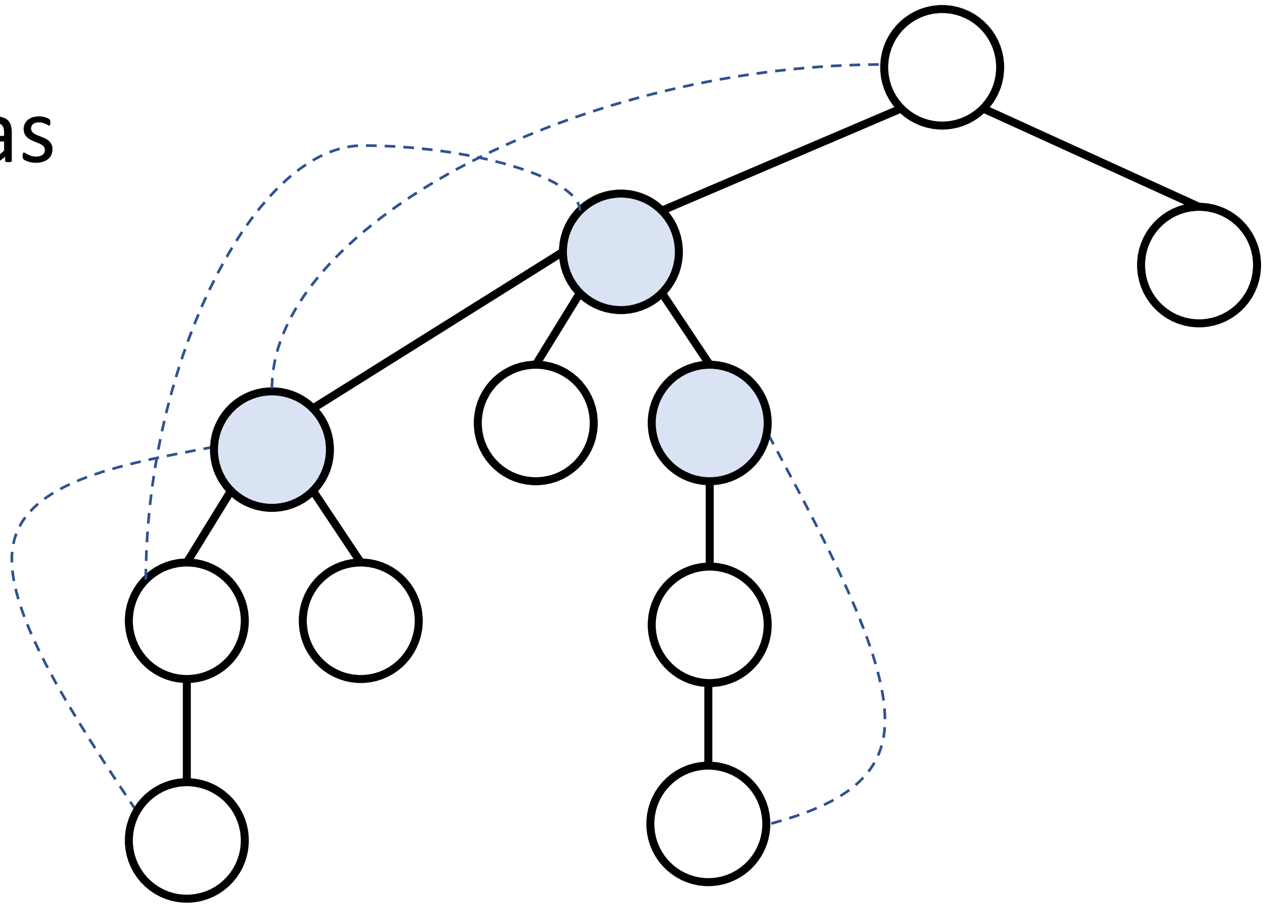
3. For non-leaf v :
   some child of v has no descendant to link back to any proper ancestor of v ⇔ v is an AP



Q2 [solution]

3. For non-leaf v :

   some child of v has no descendant to link back to any proper ancestor

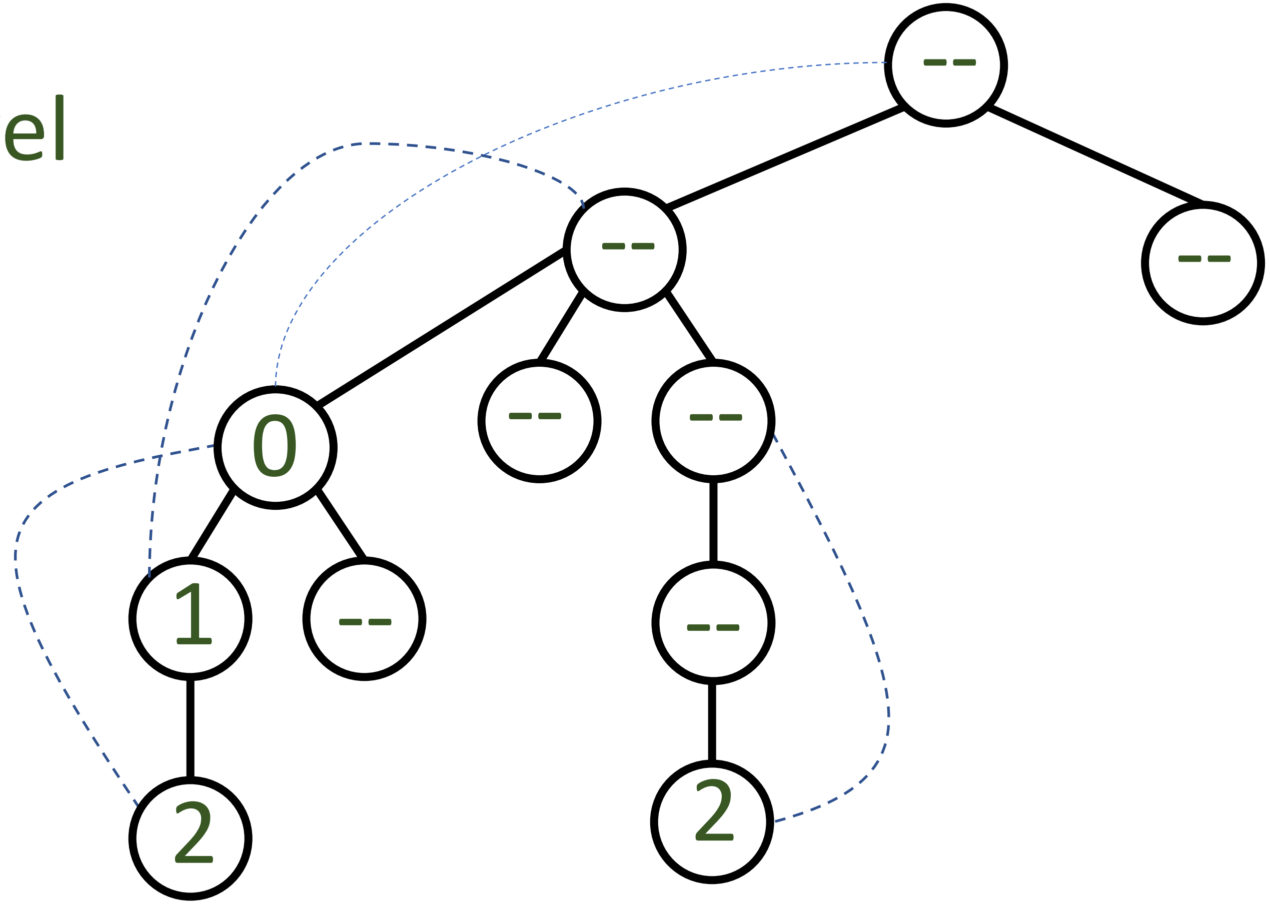


How to check ?

Q2 [solution]

Idea:

Each node finds level of highest ancestor it connects with



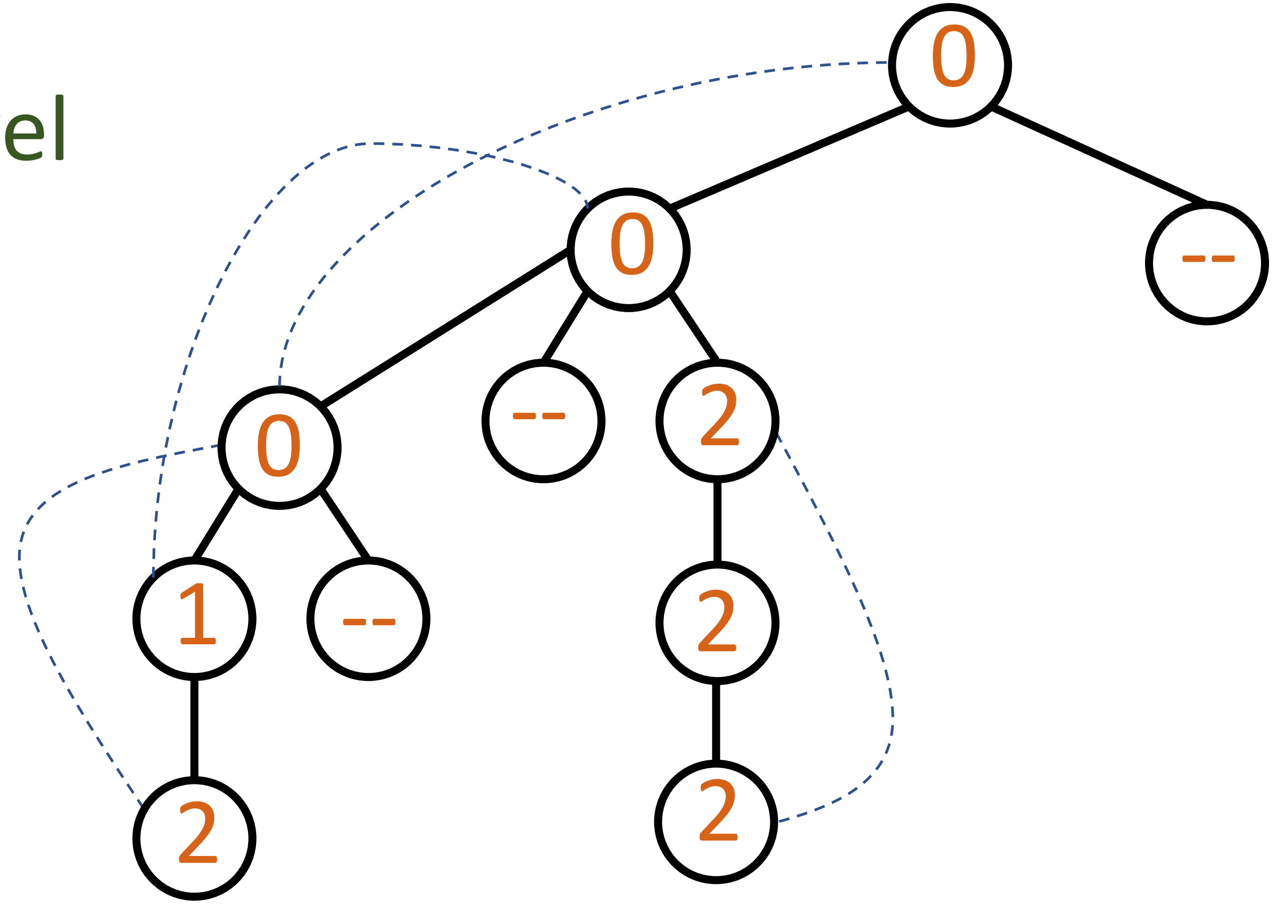
Step 1

Q2 [solution]

Idea:

Each node finds level of highest ancestor its descendants or itself connect with

Step 2

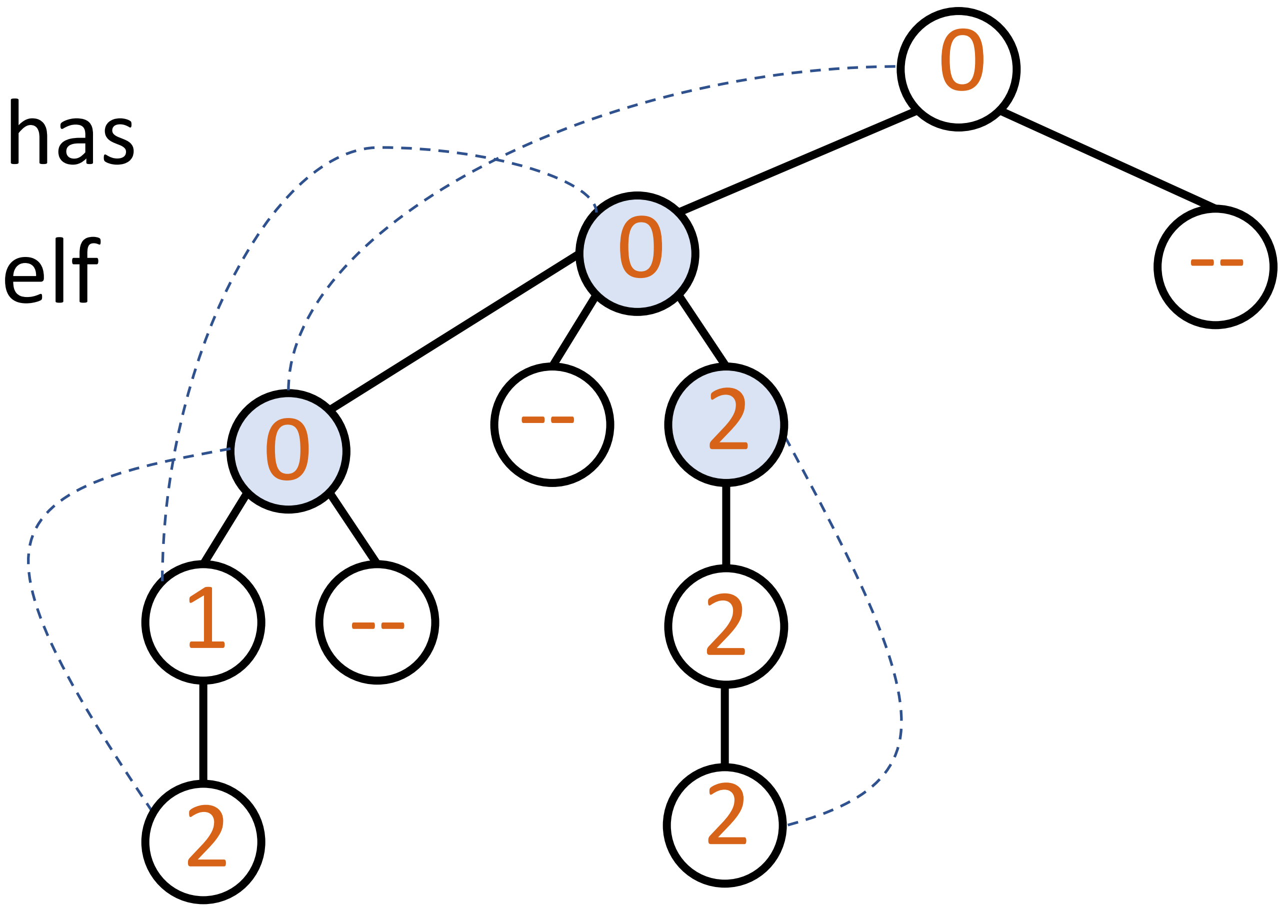


Q2 [solution]

Idea:

Check if each child has a descendant or itself that connects with a proper ancestor

Step 3



Q2 [solution]

In fact, since we are using DFS

- For two nodes with ancestor-descendant relationship:

relative level == relative discovery time

- Use min discovery time instead of min level

III. Topological Sort and SCC

Q1 [solution]

How to perform Topological Sort in BFS way ?

1. Keep a queue of nodes without incoming edges
2. Remove these nodes successively
   - Whenever some node does not have incoming edges after updating ➔ Add it to the queue

Q1 [solution]

Correctness (assume input graph is a DAG):

- For every DAG, there must be a node without incoming edges (why?)

  ➔ Queue is never empty unless we are done

- If the process ends, every edge must come from some node that is removed earlier

  ➔ Topologically sorted

Q1 [solution]

Running Time:

- O(E) time to compute in-degree of every node
- O(V) time to initialize the queue
- Removing of a node v takes O(deg(v)) time

➔ Each node is removed once

➔ Total time to remove nodes is O(E)

Q1 [solution]

What if the input graph is not a DAG ?

If this occurs:

- some node never enters the queue
  - ➔ can be checked by counting total # nodes entering the queue

Q2

- Let G be a directed graph
- We call G to be semi-connected if

for any two nodes u and v, either

u is connected to v by a directed path,

or v is connected to u by a directed path,

or both

Q2

How to check if G is semi-connected ?

Key Observation:

G is semi-connected $\Leftrightarrow$

G^{SCC} has a directed path joining all vertices

Q2 [solution]

Proof of Key Observation:

(=>) By contradiction

(<=) By checking each pair of vertices

Q2 [solution]

Algorithm :

1. Obtain SCC graph G^{SCC} from G
2. Topological sort G^{SCC}

➔ Check whether each node in topo-sorted order connects to next node

Running time: Linear time