CS202 – Data Structures

LECTURE-22

Graphs and Disjoint Sets

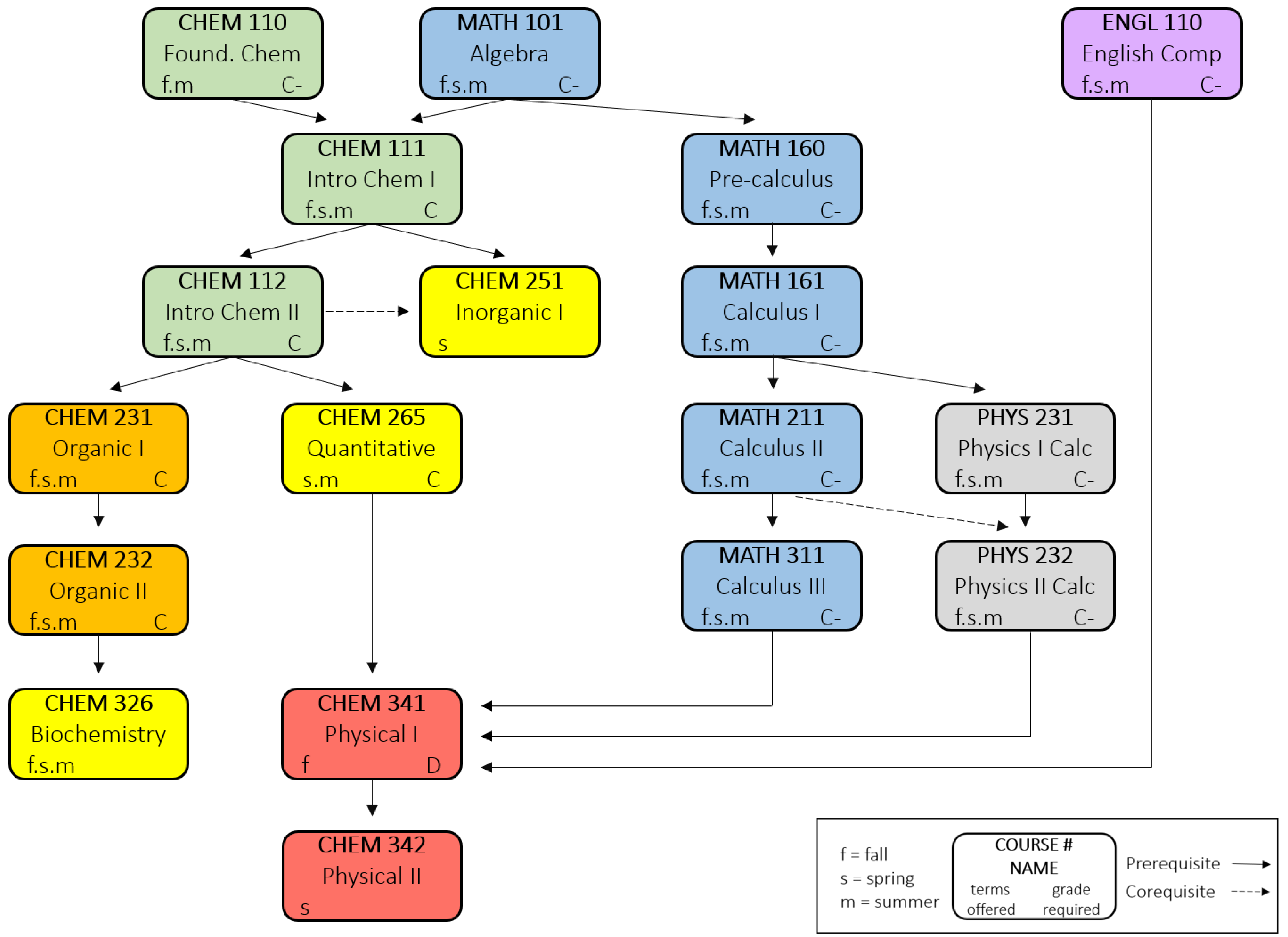
Topological Sort, Set Operations

Dr. Maryam Abdul Ghafoor
Assistant Professor
Department of Computer Science, SBASSE

Agenda

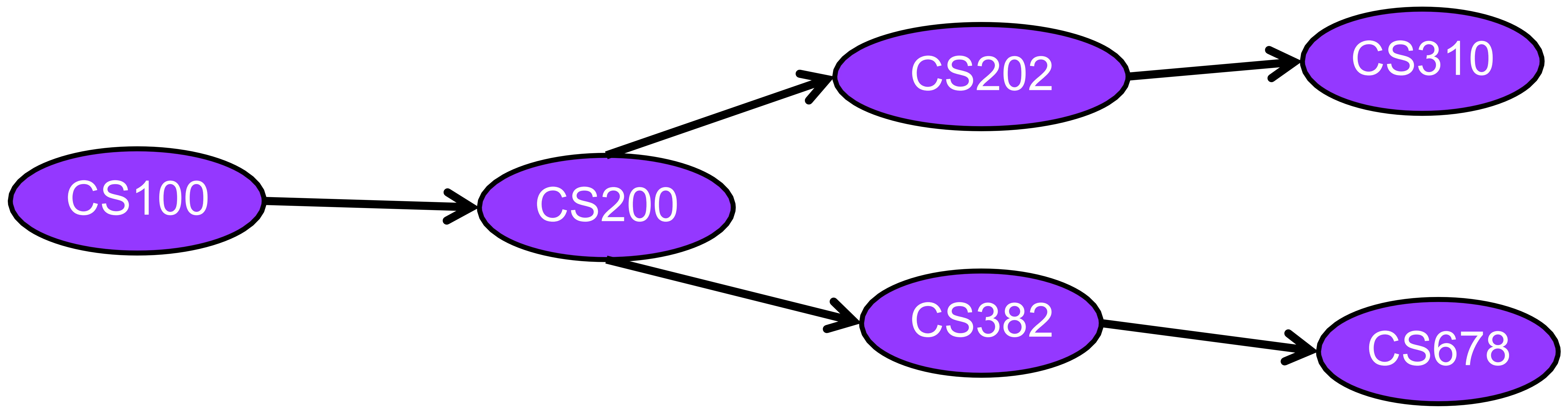
- Topological Sort
- Sets ADT
  - Disjoint Sets
  - Set Operations

Topological Sort



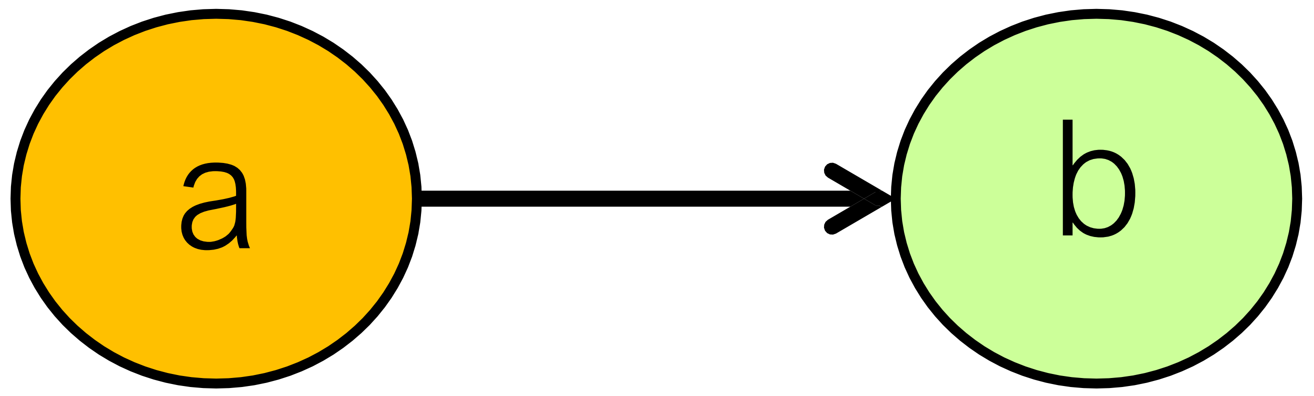
Graph Problem

- Finding course pre-requisites



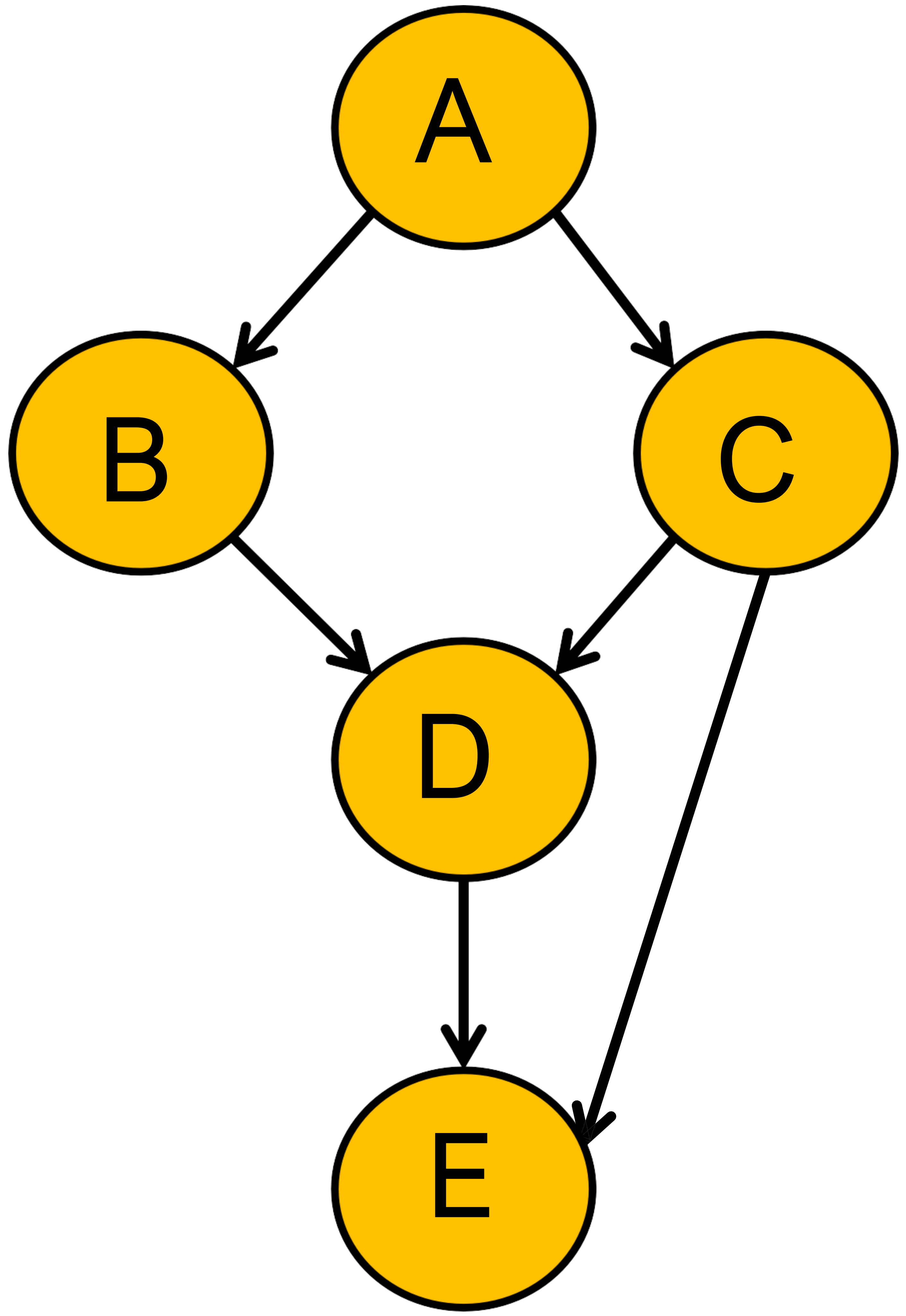
Topological Sorts

- It is a sorting of vertices in a directed graph such that if there is an edge from a to b, then a has to be before b in the topological sort



- A way to linearly order vertices
- Used to represent dependency constraints
- The graph of dependencies cannot have a cycle

Topological Sort – Examples


$$\begin{tikzcd} & A \arrow[dl] \arrow[dr] & \\ B \arrow[dr] & & C \arrow[dl] \arrow[ddl] \\ & D \arrow[d] & \\ & E & \end{tikzcd}$$

A B C D E

A C B D E

Directed Acyclic Graphs (DAGs)

- Topological sorts are **only valid** for DAGs
  - Recall: a DAG has no cycles

- If G is a DAG, then G has a node with no outgoing edges (also called the sink node)

DAG Properties

- Lemma: If G is a DAG, then G has a node with no outgoing edges
- Lemma: If G is a DAG, then G has a node with no incoming edges

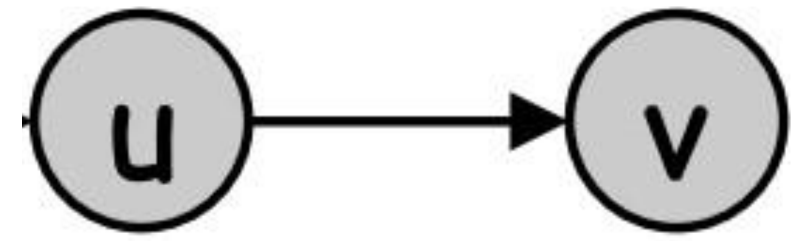
Proof by Contradiction:

- Suppose G is a DAG and every node has at least one incoming edge.

DAG Properties

- Lemma: If G is a DAG, then G has a node with no incoming edges

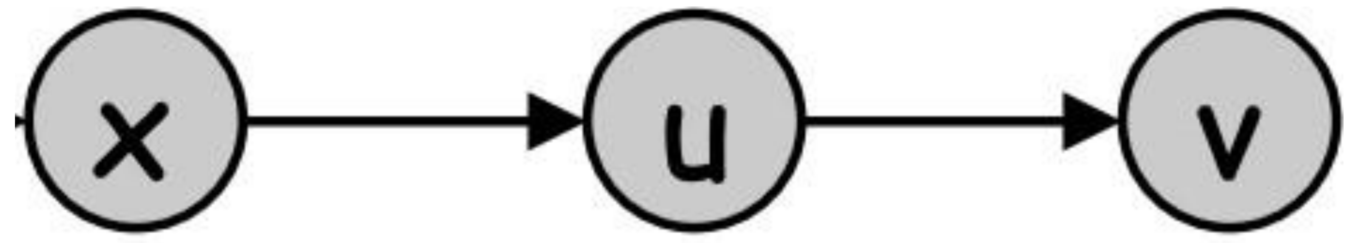
- Suppose G is a DAG and every node has at least one incoming edge.



DAG Properties

- Lemma: If G is a DAG, then G has a node with no incoming edges

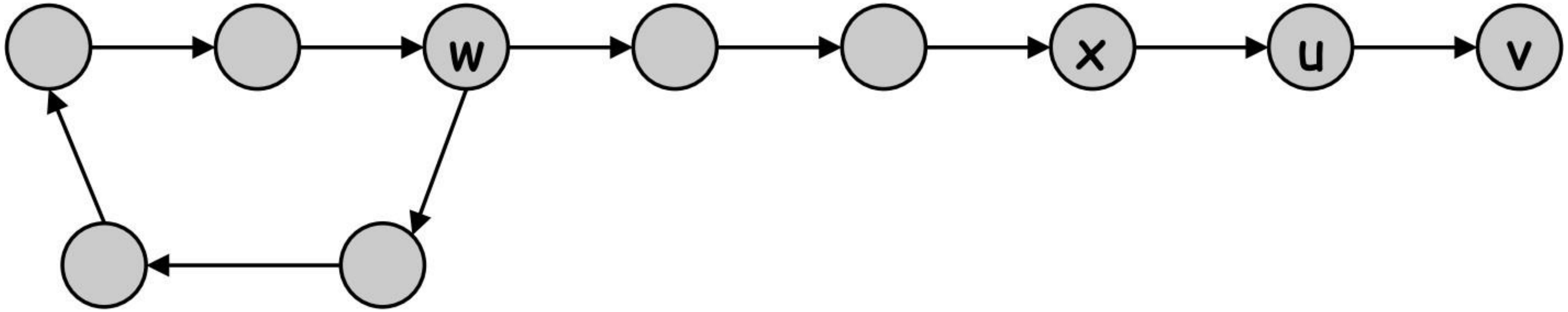
- Suppose G is a DAG and every node has at least one incoming edge.



DAG Properties

- Lemma: If G is a DAG, then G has a node with no incoming edges

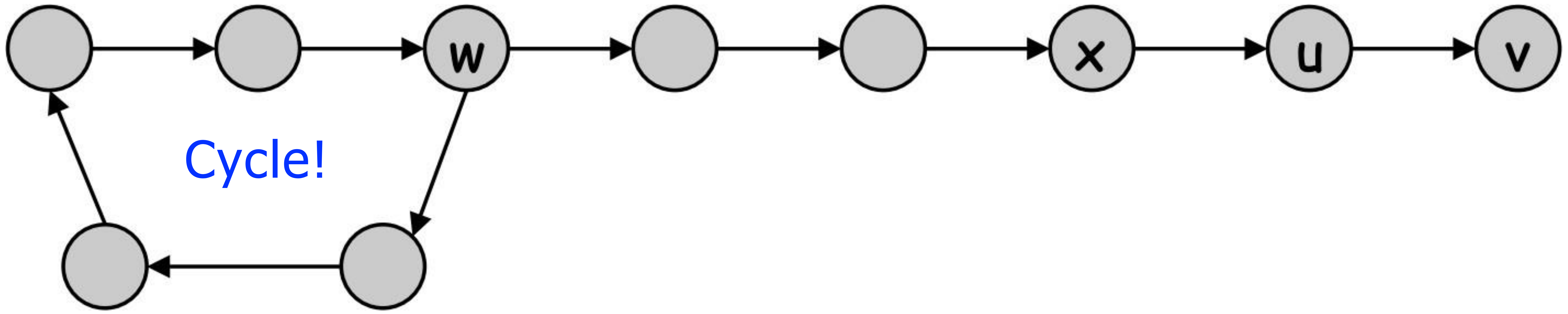
- Suppose G is a DAG and every node has at least one incoming edge.



DAG Properties

- Lemma: If G is a DAG, then G has a node with no incoming edges

- Suppose G is a DAG and every node has at least one incoming edge.



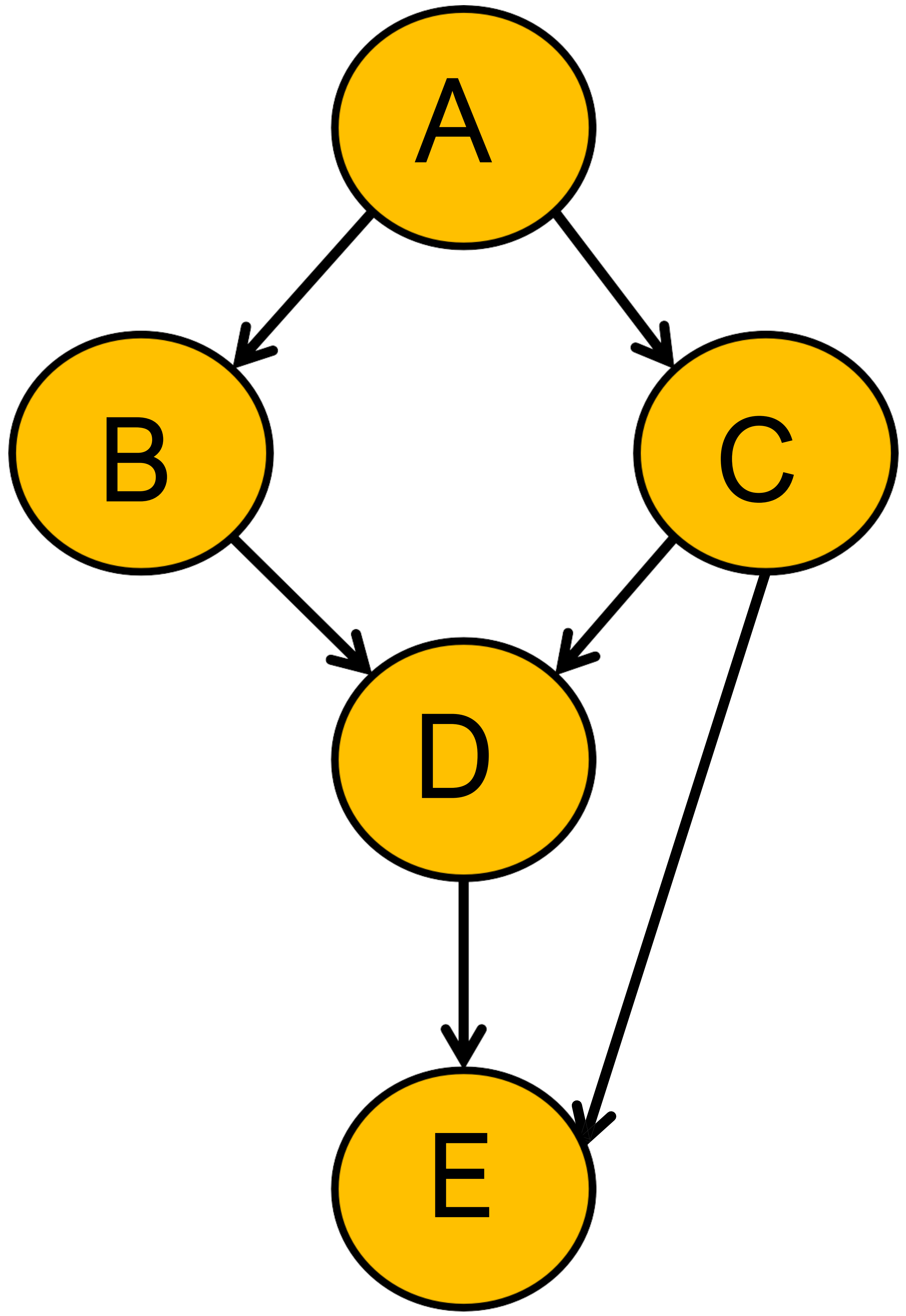
DAG Properties

- Lemma: If G is a DAG, then G has a node with no incoming edges

- Pf. (by **contradiction**): Suppose G is a DAG and every **node has at least one incoming edge.**
- Pick any node v, and **begin following edges backward** from v.
- Since v has at least one incoming edge (u, v) we can walk backward to u.
- Then, since u has at least one incoming edge (x, u), we can walk backward to x.
- Repeat until we visit a node, say w, twice.
- Let C denote the sequence of nodes encountered between successive visits to w. C is a cycle!

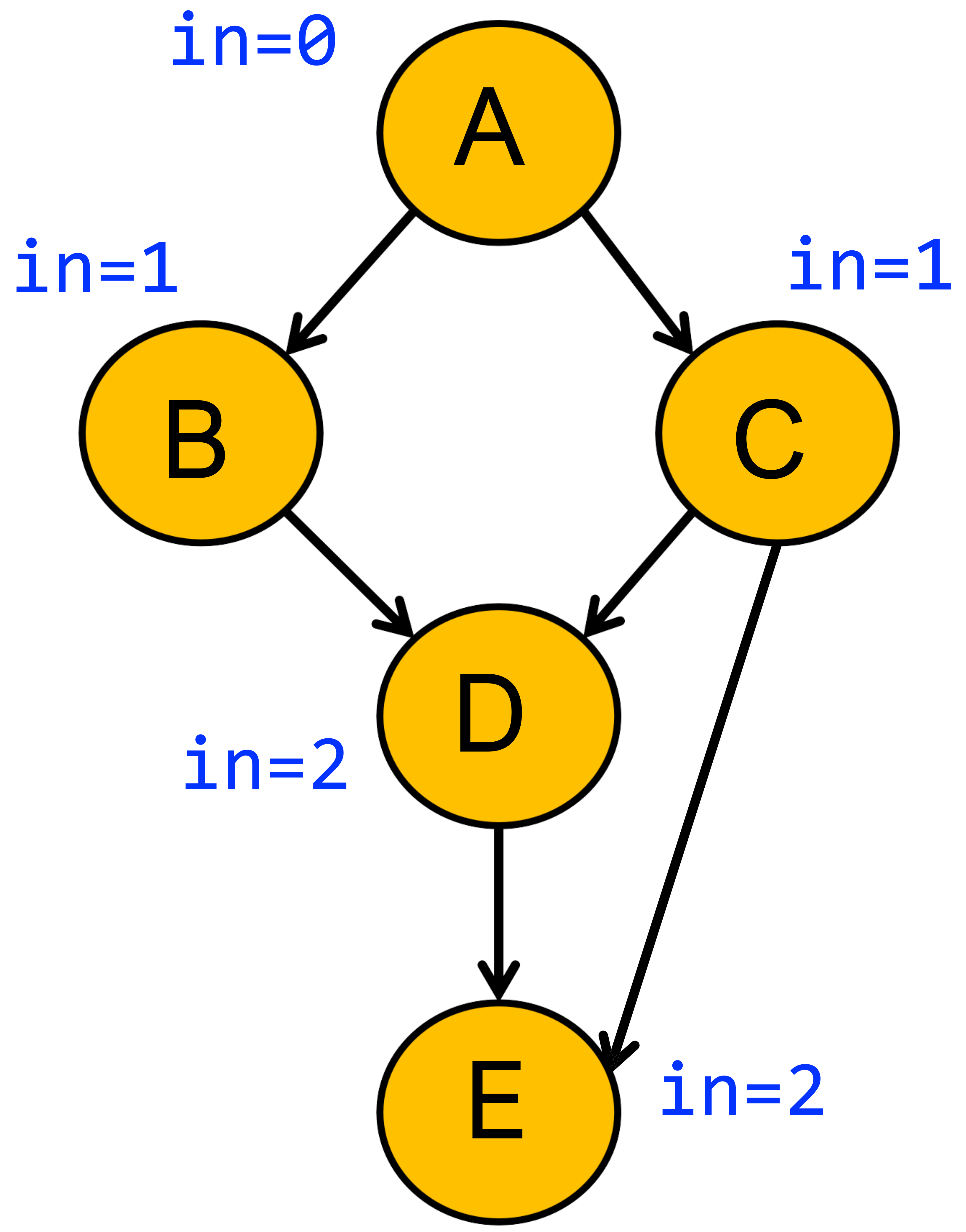
How can we find a valid Topo Sort?

Designing a Topological Sort Algorithm



Designing a Topological Sort Algorithm

- An Idea for topological Sort

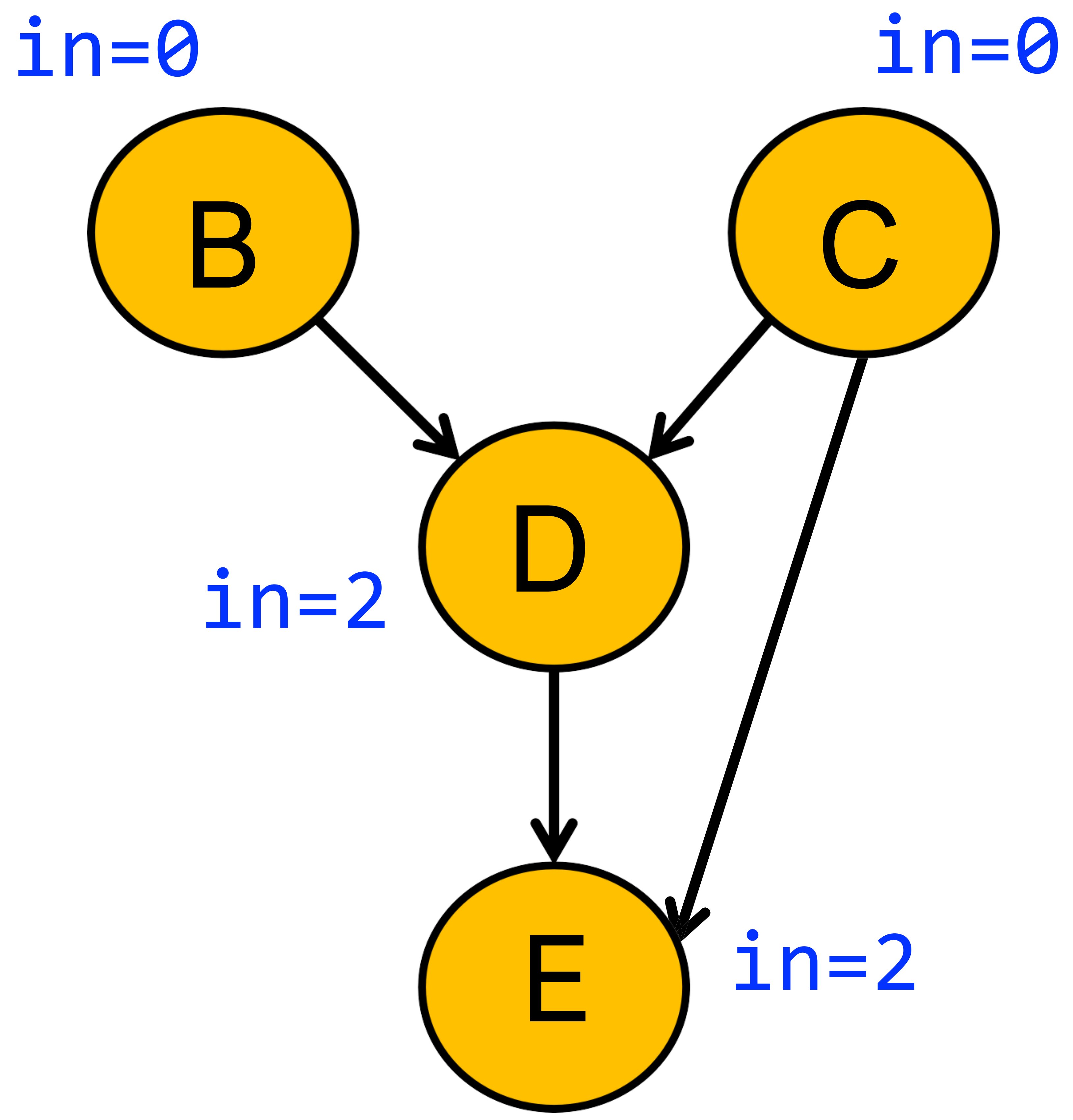


S={}

- Compute the in-degree of each node
- Choose a vertex with in=0 and put in the sorted sequence
- Remove in=0 node from G and recompute in-degree

Designing a Topological Sort Algorithm

- An Idea for topological Sort

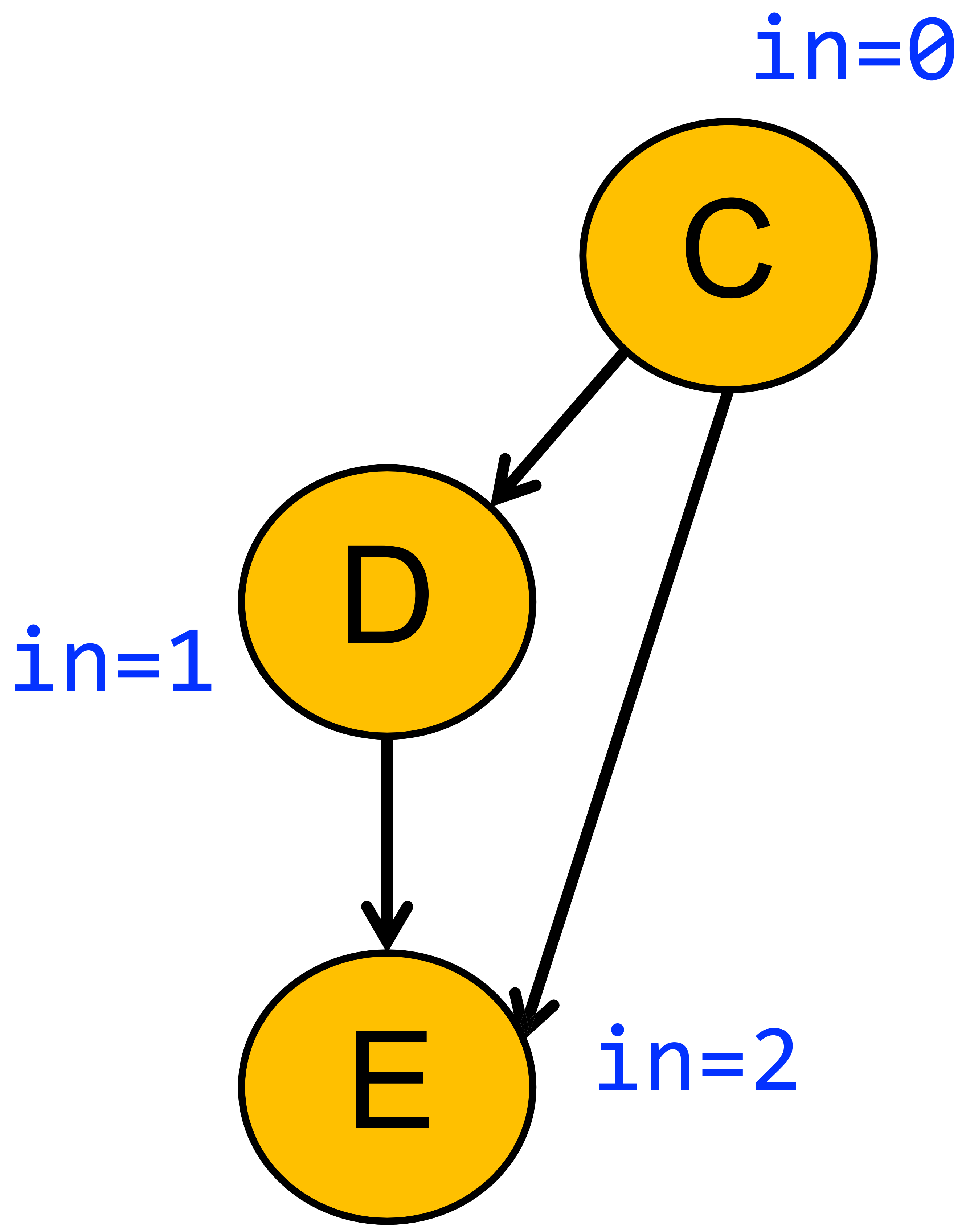


S={A}

- Compute the in- degree of each node
- Choose a vertex with in=0 and put in the sorted sequence
- Remove in=0 node from G and recompute in-degree

Designing a Topological Sort Algorithm

- An Idea for topological Sort

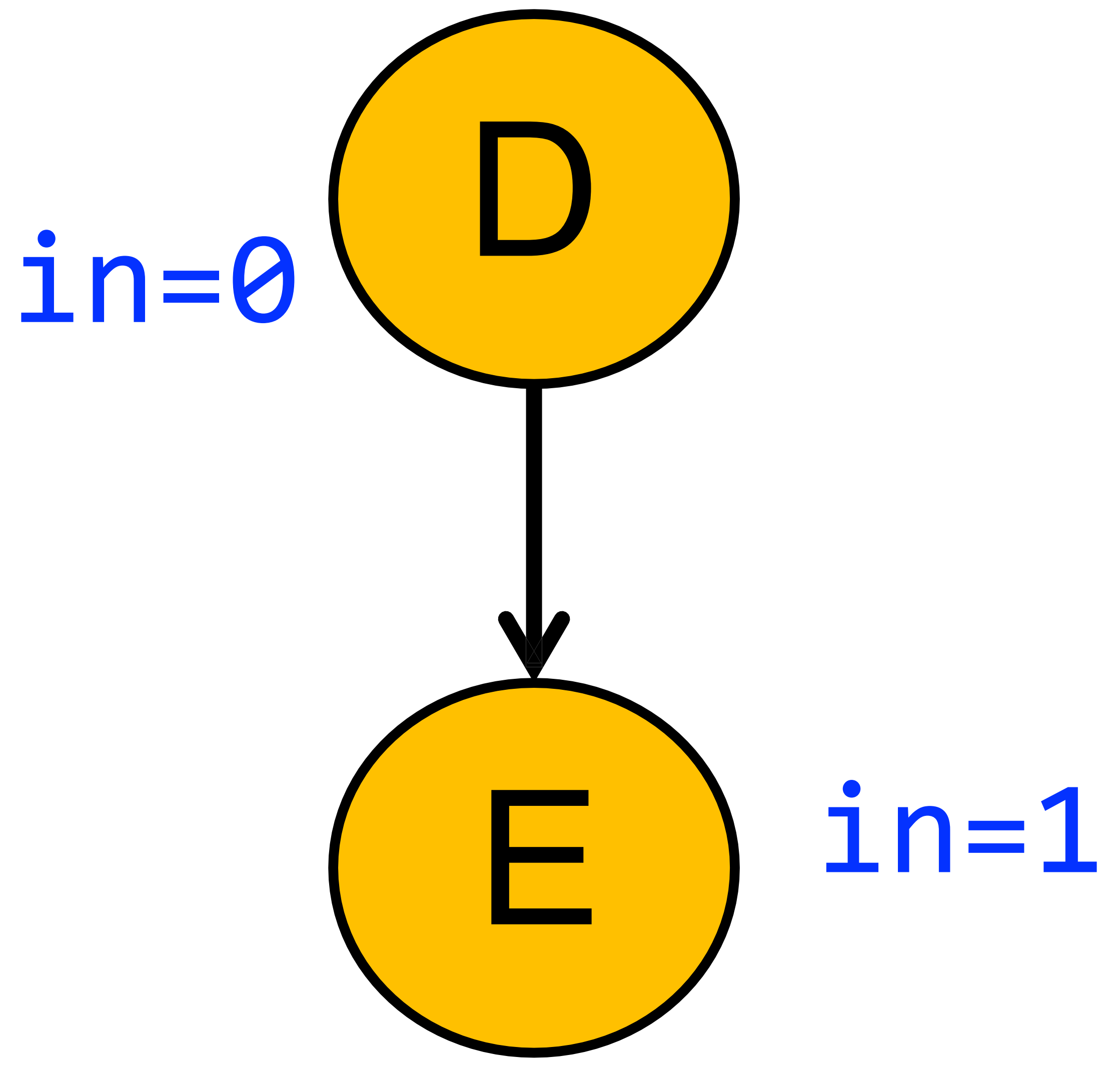


S={A, B}

- Compute the in- degree of each node
- Choose a vertex with in=0 and put in the sorted sequence
- Remove in=0 node from G and recompute in-degree

Designing a Topological Sort Algorithm

- An Idea for topological Sort



S={A, B, C}

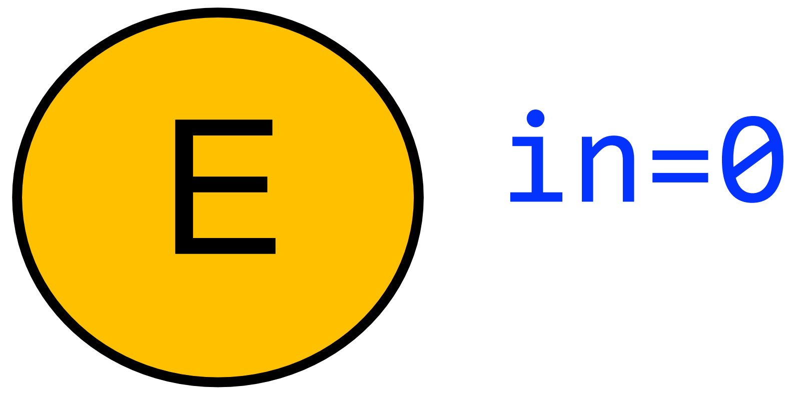
- Compute the in- degree of each node
- Choose a vertex with in=0 and put in the sorted sequence
- Remove in=0 node from G and recompute in-degree

Designing a Topological Sort Algorithm

- An Idea for topological Sort

- Compute the in- degree of each node
- Choose a vertex with in=0 and put in the sorted sequence
- Remove in=0 node from G and recompute in-degree

S={A, B, C, D}



Designing a Topological Sort Algorithm

- An Idea for topological Sort

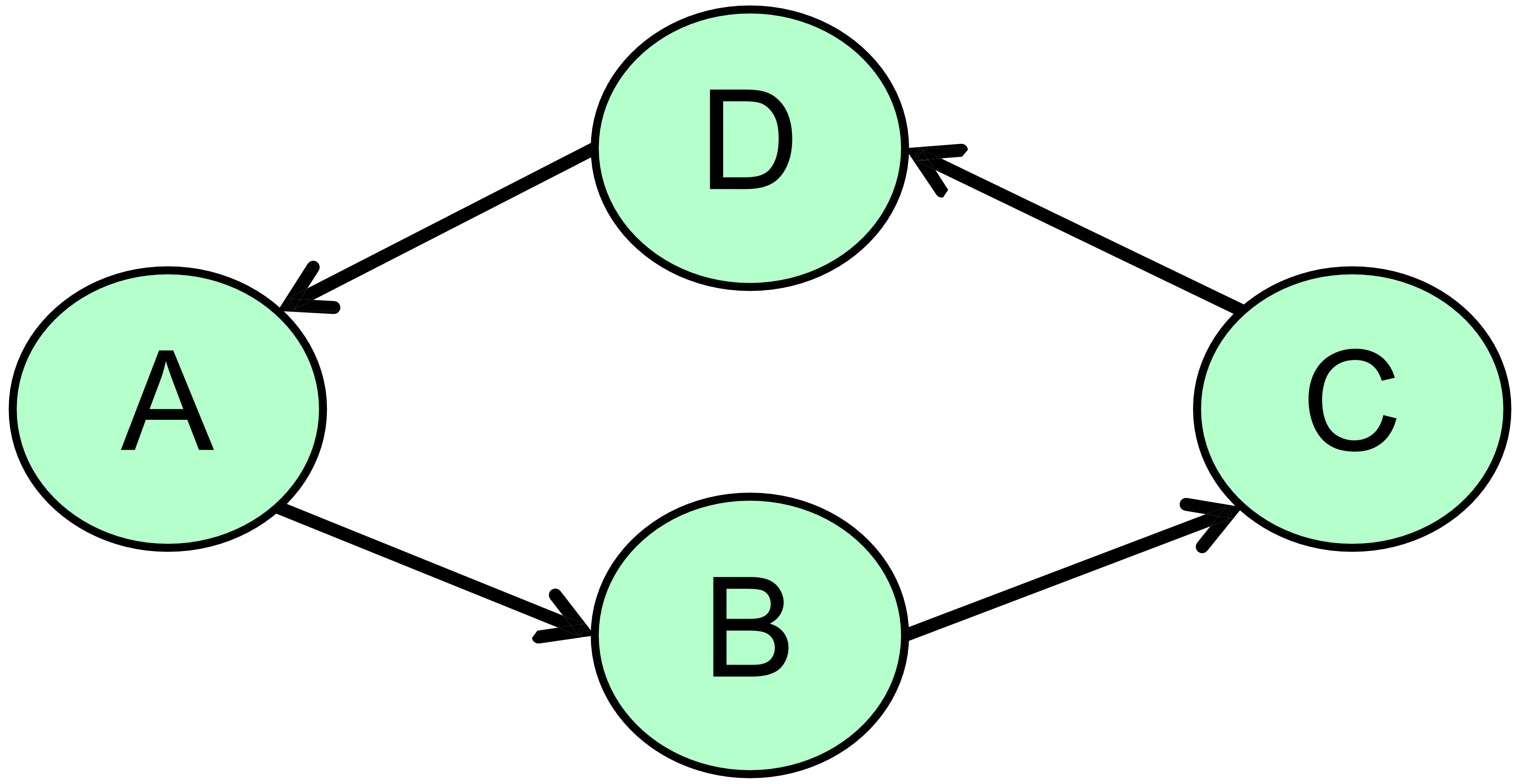
```
Topological Sort

 S={A, B, C, D, E}
```

- Compute the in- degree of each node
- Choose a vertex with in=0 and put in the sorted sequence
- Remove in=0 node from G and recompute in-degree

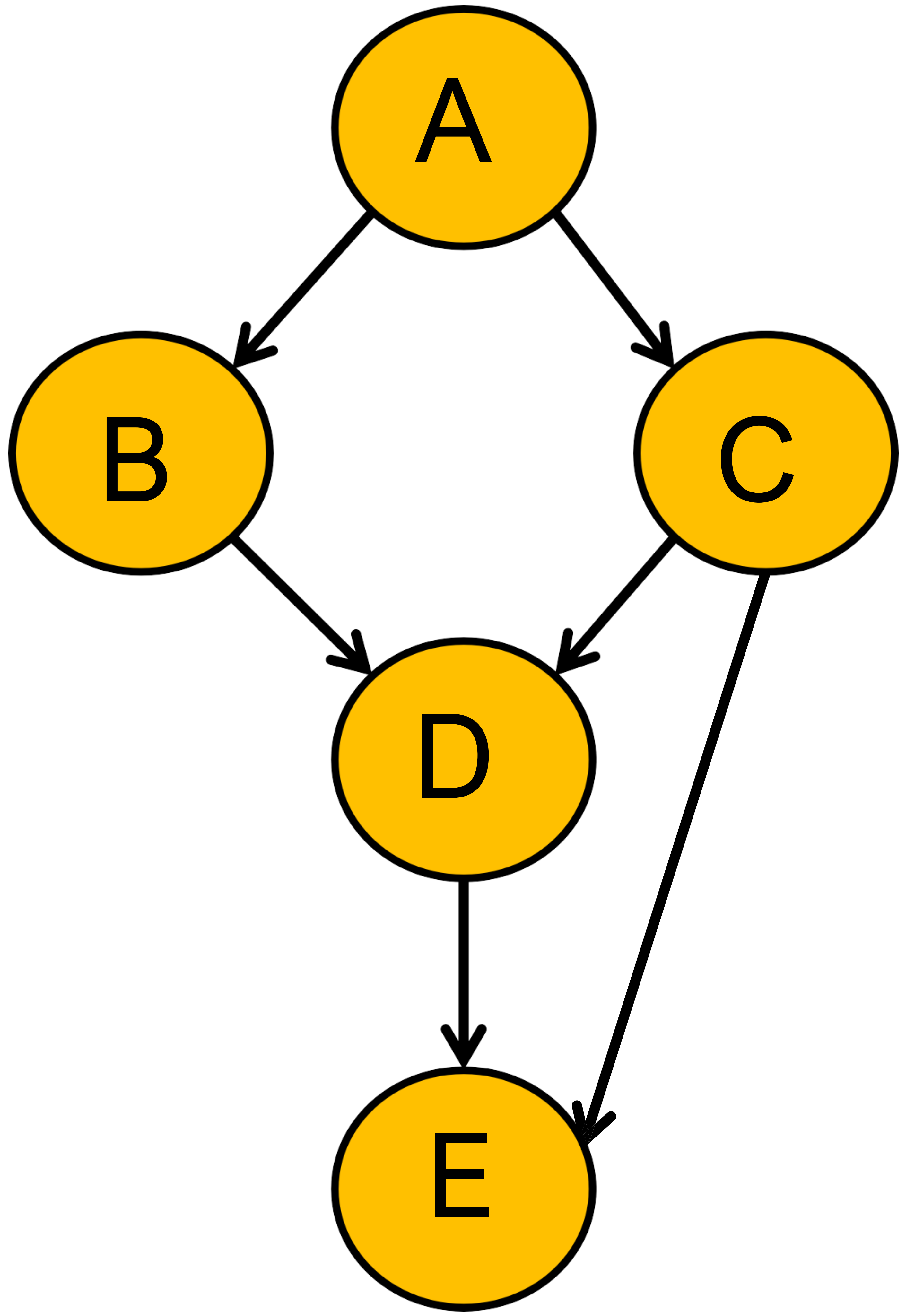
Concept Check!

- What is the topological ordering for the following graph?

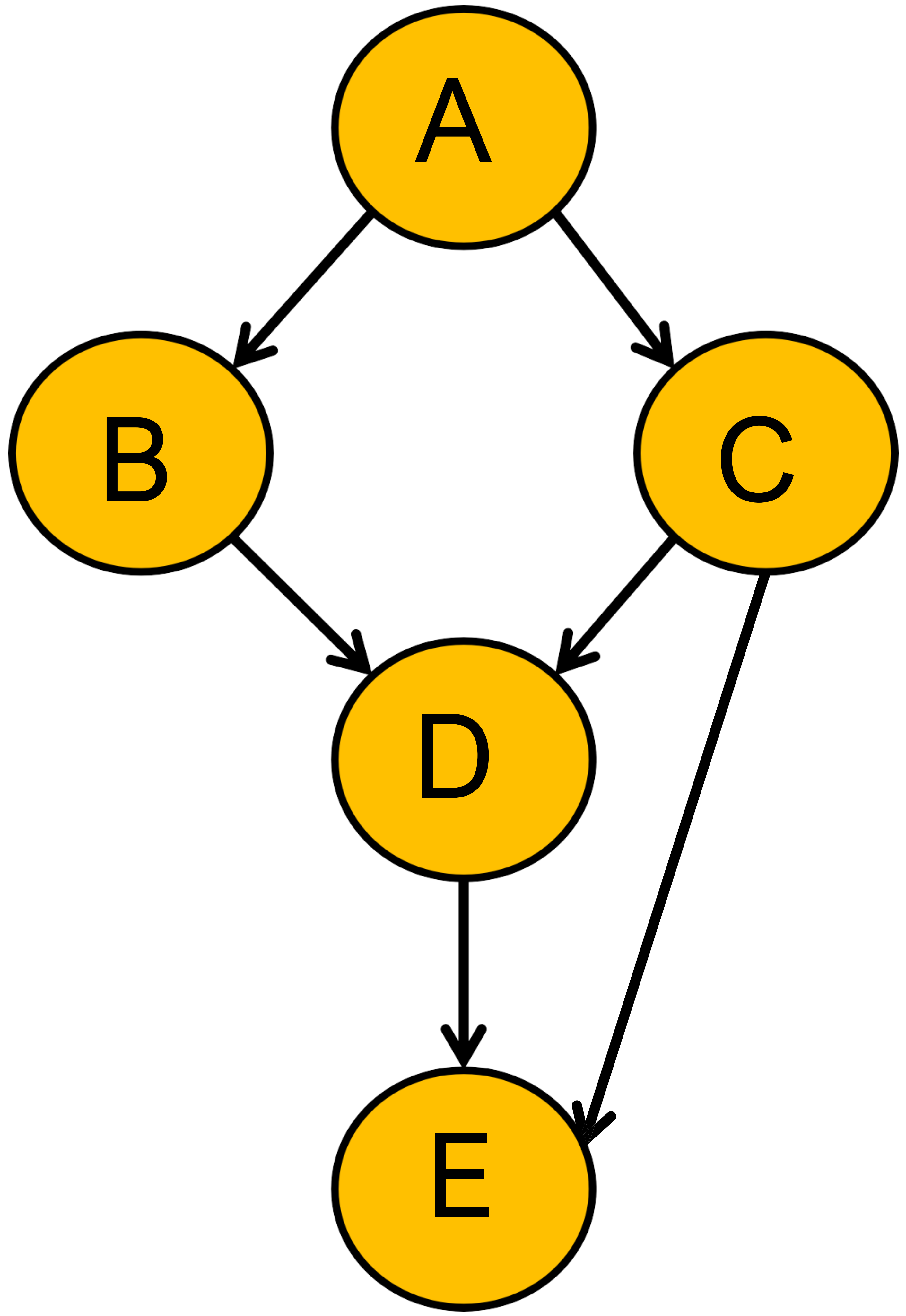


$$\begin{tikzcd}
 & D \arrow[dl] & \\
A \arrow[dr] & & C \arrow[ul] \\
 & B \arrow[ur] &
\end{tikzcd}$$

Can We Use DFS to do Topological Sort?

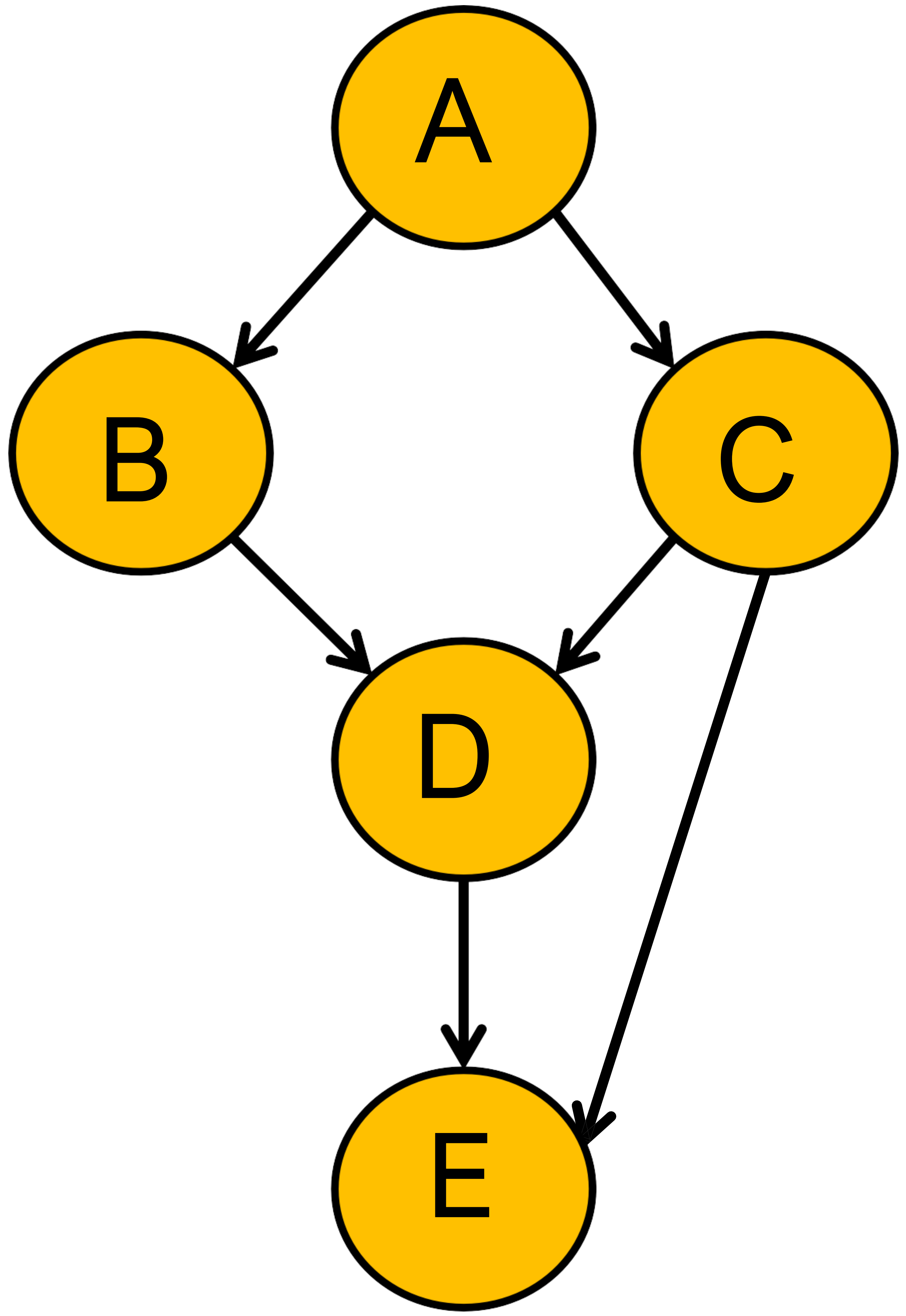


Can We Use DFS to do Topological Sort?



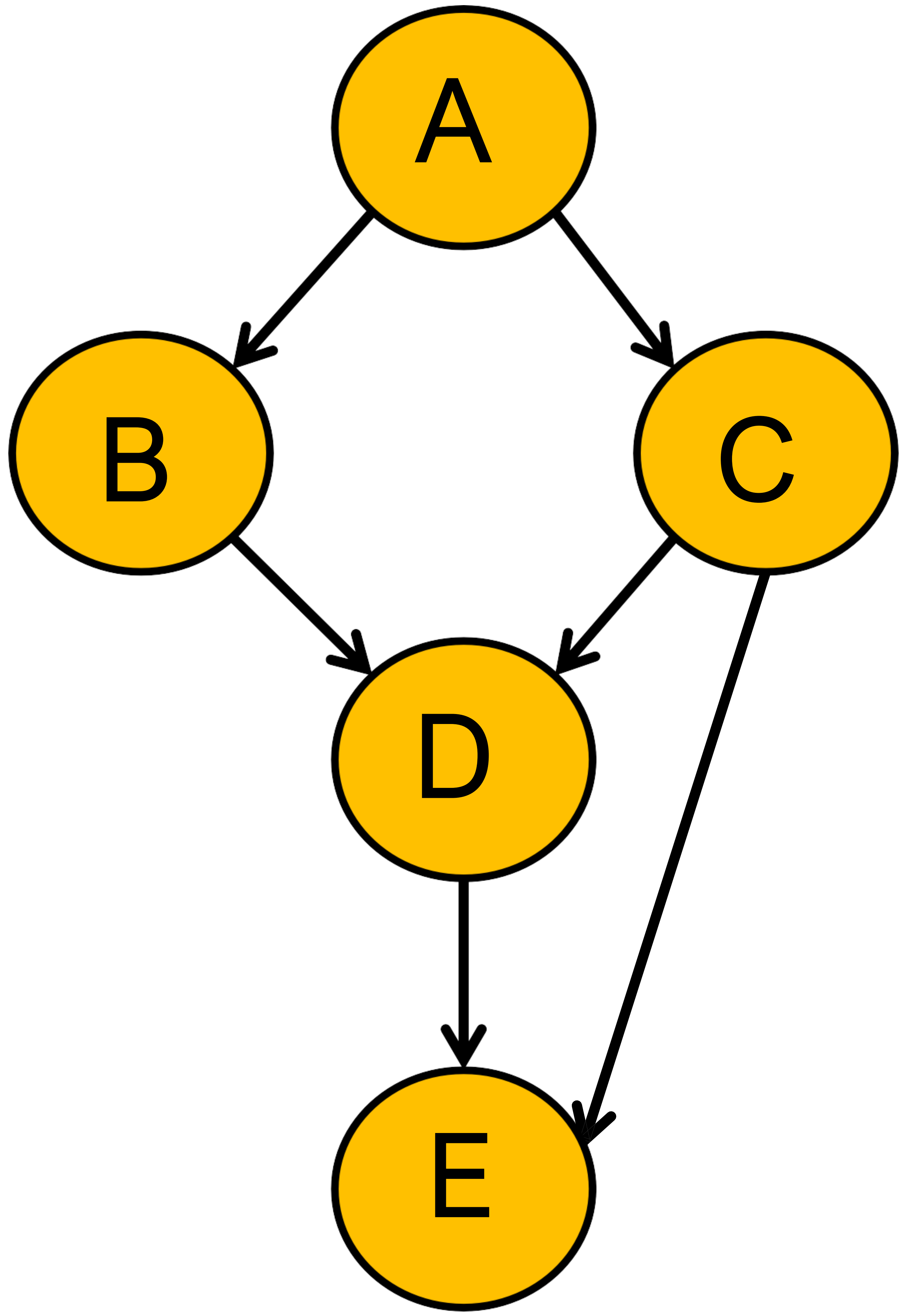
- If we start from any vertex, which node will finish first?

Can We Use DFS to do Topological Sort?



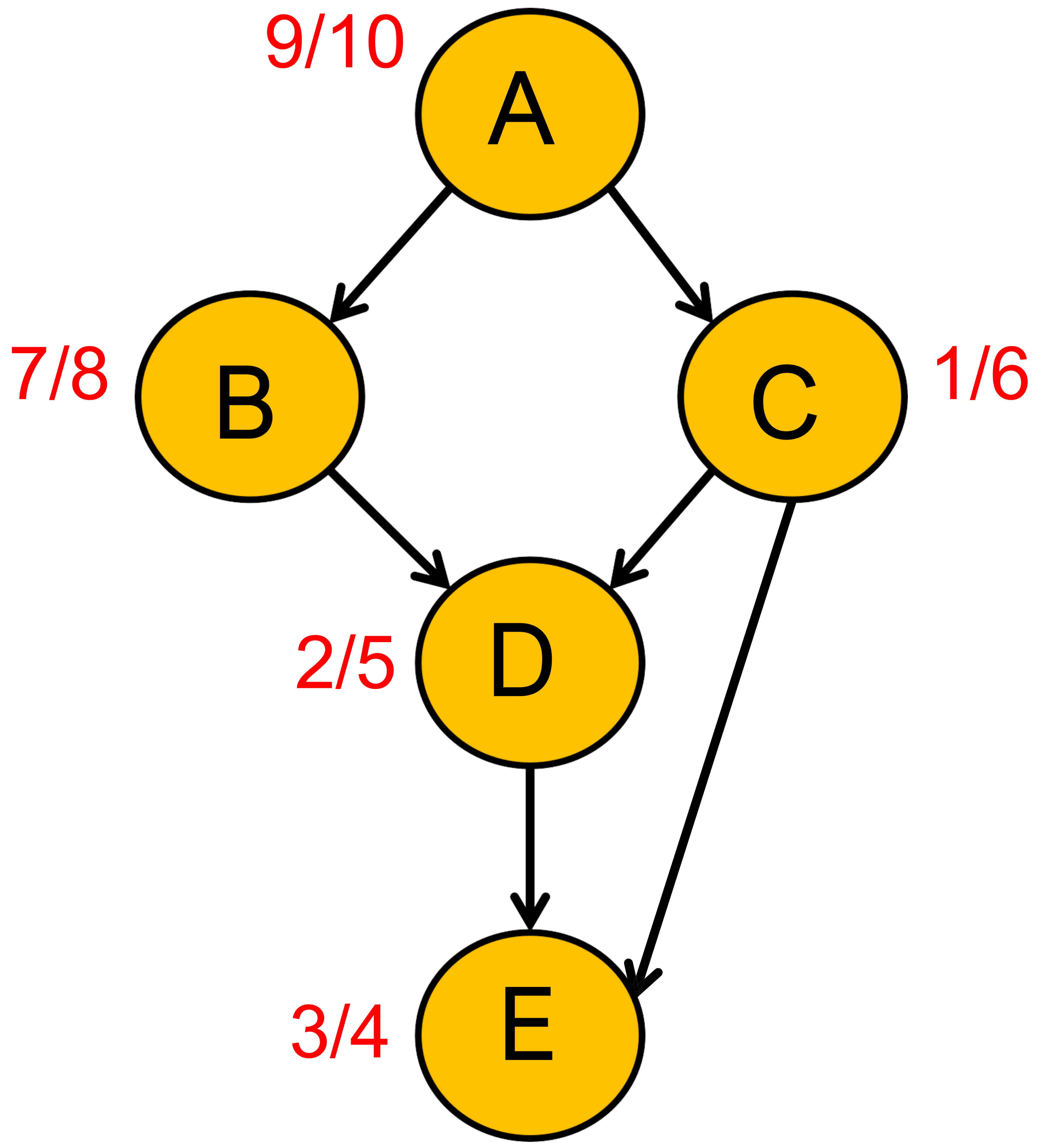
- If we start from any vertex, which node will finish first?
- The node with no outgoing edge

Can We Use DFS to do Topological Sort?



- Algorithm
  - Perform DFS from every vertex in the graph
  - Record DFS finish times along the way without clearing finish times between traversals
  - Topological ordering is the reverse of the finish times

Can We Use DFS to do Topological Sort?

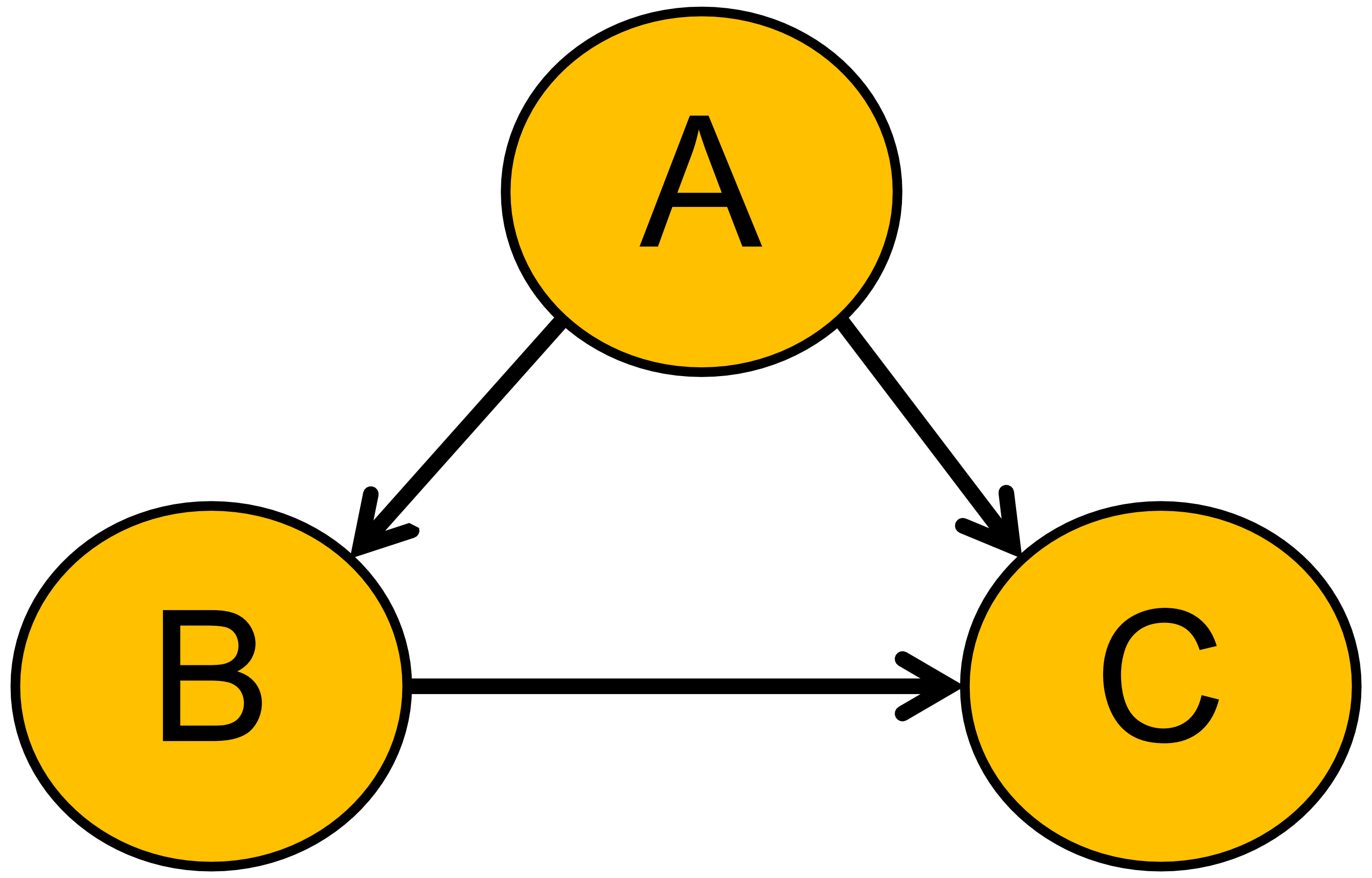


- Algorithm
  - Perform DFS from every vertex in the graph
  - Record DFS finish times along the way without clearing finish times between traversals
  - Topological ordering is the reverse of the finish times

Can We Use BFS to do Topological Sort?

- BFS is not an effective way to implement topological sort since it visits vertices in level order (shortest distance from source order)

Can We Use BFS to do Topological Sort?



BFS(A): A B C ✓

BFS(A): A C B ✗

BFS(B): B C A ✗

BFS(C): C A B ✗

BFS(C): C B A ✗

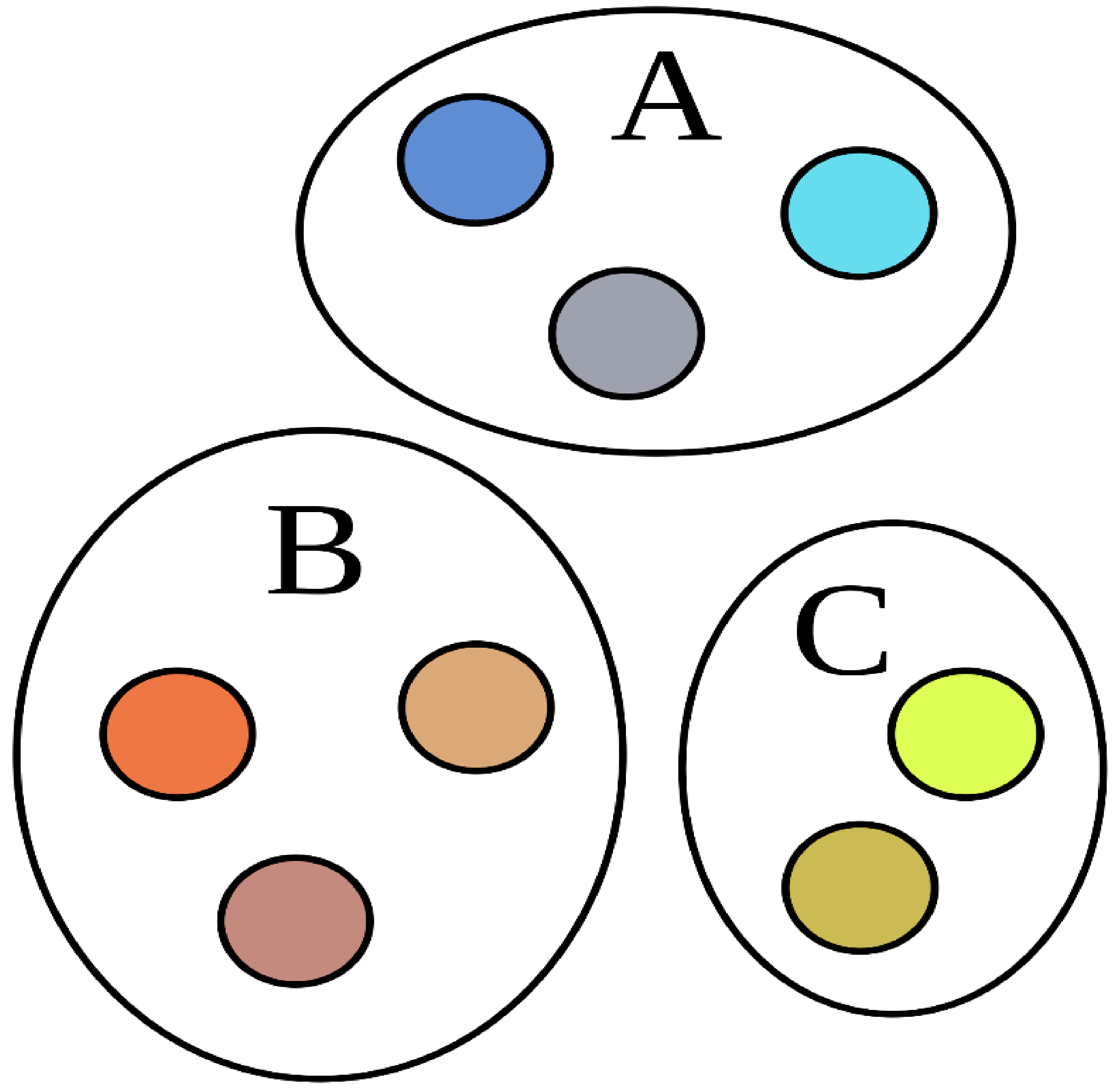
Disjoint Sets (Union–Find Data Structure)

What is a Set?

- A set is an unordered collection of distinct elements
  - If A = {1, 2, 3}, B = {3, 8, 90} then A U B = {1, 2, 3, 8, 90}

- Disjoint sets (aka union-find data structures), are used to keep track of a set of elements partitioned into disjoint (non-overlapping) subsets

- Key operations:
  - union (combining two sets)
  - find



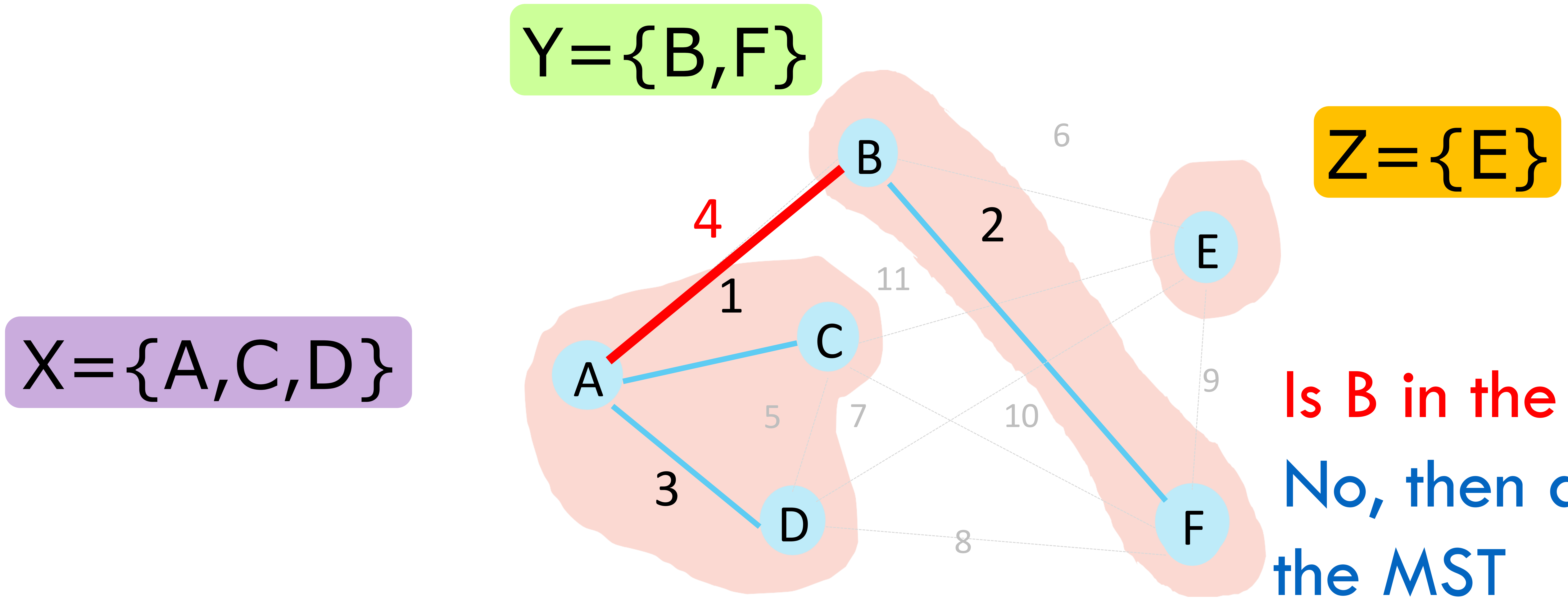
Enter Disjoint Sets ADT

- Attributes
  - Each set has a representative (either a member or a unique ID)

- Methods
  - makeSet(value): Returns a new set with value as only member and ID
  - find(value): Returns ID of the set containing value
  - union(x,y): Combine sets containing x and y into one set with all elements, i.e., choose a common representative ID

Disjoint Sets ADT (aka "Union-Find")

- Kruskal's MST algorithm can use a Disjoint Sets ADT to check whether two vertices are already connected!

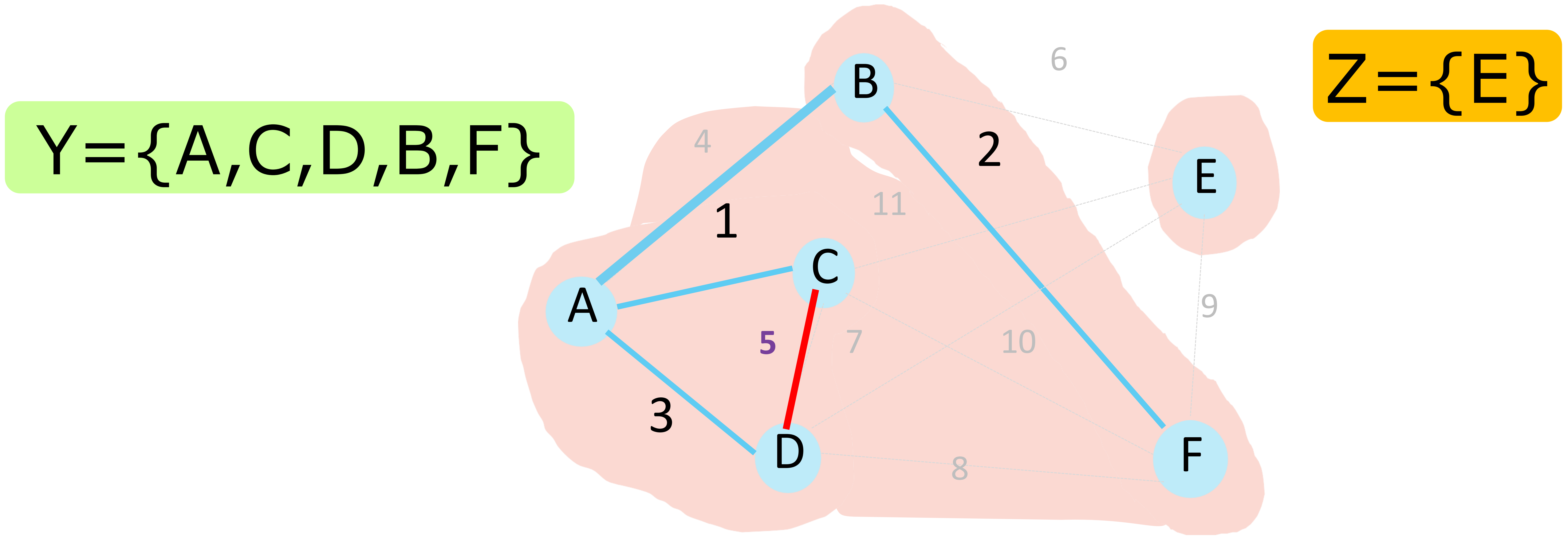


Is B in the same set as A?

No, then add this edge to the MST

Disjoint Sets ADT (aka "Union-Find")

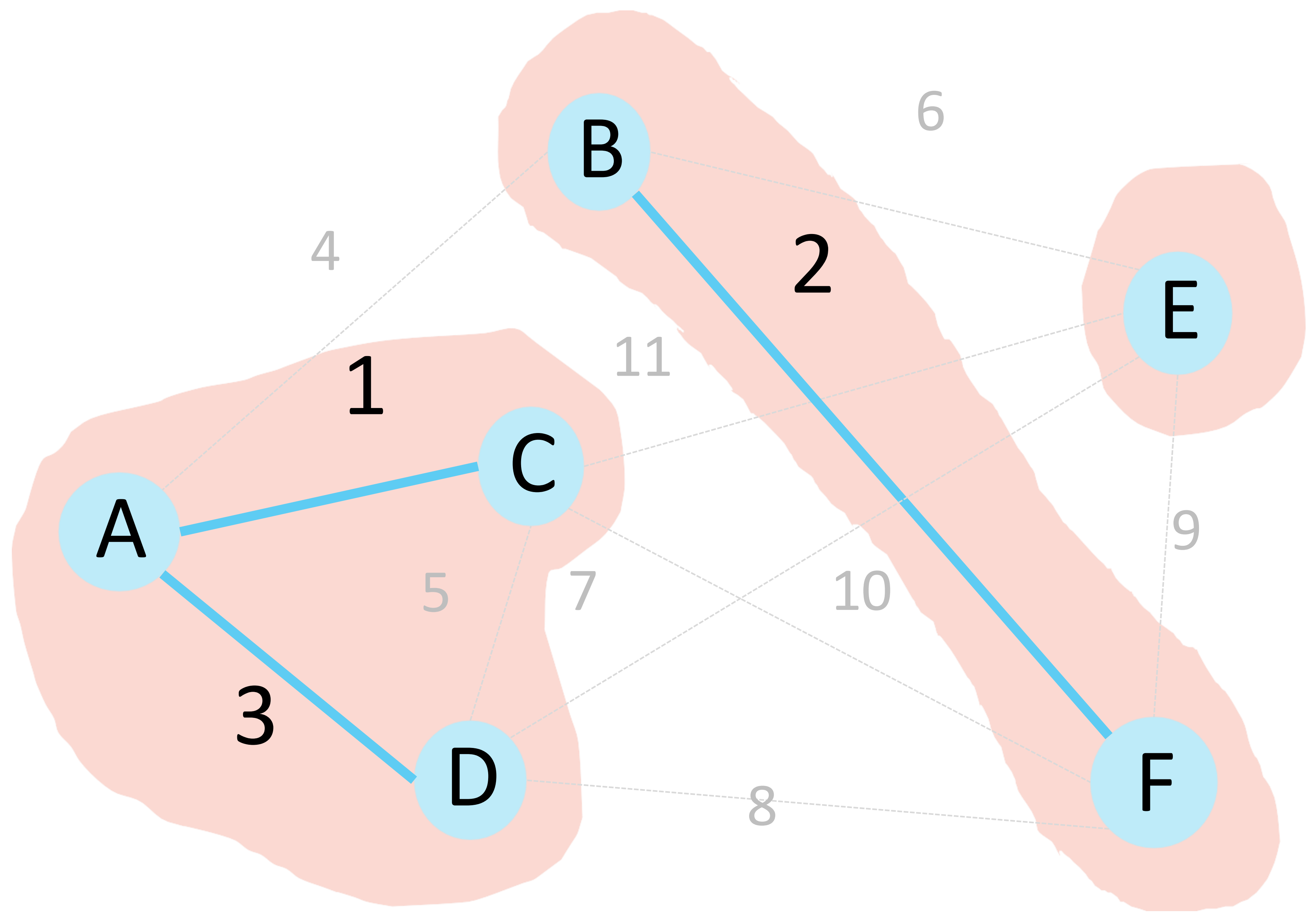
- Kruskal's MST algorithm can use a Disjoint Sets ADT to check whether two vertices are already connected!



Is C in the same set as D? Yes, ignore this edge!

Disjoint Sets ADT (aka "Union-Find")

Kruskal's MST algorithm can use a Disjoint Sets ADT to check whether two vertices are already connected!



```
kruskalMST(G graph)
   (1) DisjointSets<V> msts; Set finalMST;
   (2) initialize msts with each vertex as
       single-element MST
   (3) sort all edges by weight (smallest
       to largest)

   for each edge (u,v) in ascending order:
     uSet = msts.find(u)
     vSet = msts.find(v)
     if(uSet != vSet):
       finalMST.add(edge (u, v))
       msts.union(uSet, vSet);
```

Set Basic Operations

Operations	Complexity
makeSet(value)	Θ(?)
find(value)	Θ(?)
union(x,y)	Θ(?)

How can we implement the Disjoint Sets ADT?

How about using a single linked list?

Linked List to Implement the Disjoint Sets ADT

- Store (set ID, value) in each list node

Example:

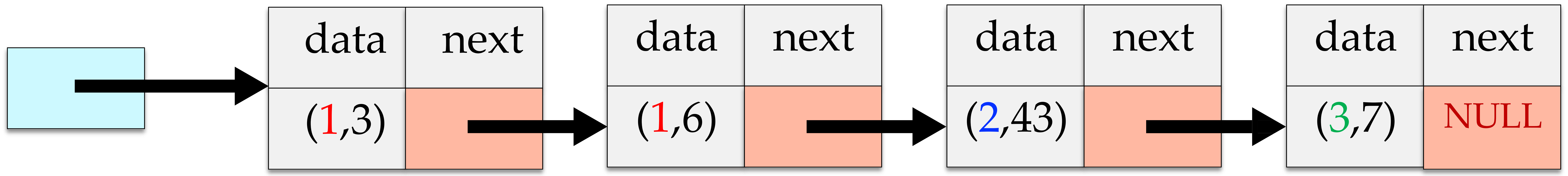
- Set-1={3,6}, Set-2={43}, Set-3={7}

Linked List to Implement the Disjoint Sets ADT

- Store (set ID, value) in each list node

Example:

- Set-1={3,6}, Set-2={43}, Set-3={7}



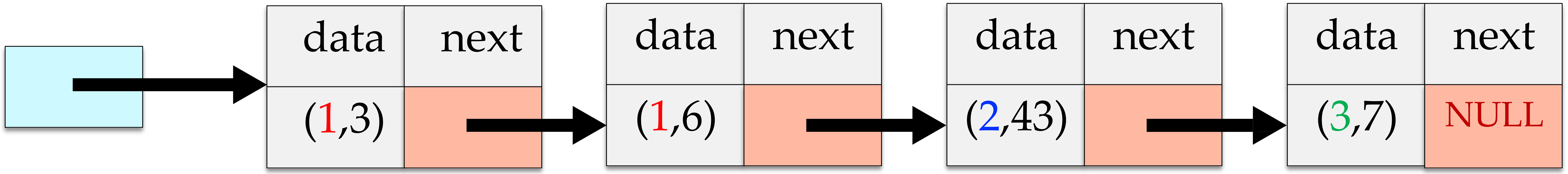
Linked List to Implement the Disjoint Sets ADT

- Store (set ID, value) in each list node

Operations	Complexity
makeSet(value)	Θ(1)
find(value)	Θ(n)
union(x,y)	Θ(n)

Example:

– Set-1={3,6}, Set-2={43}, Set-3={7}

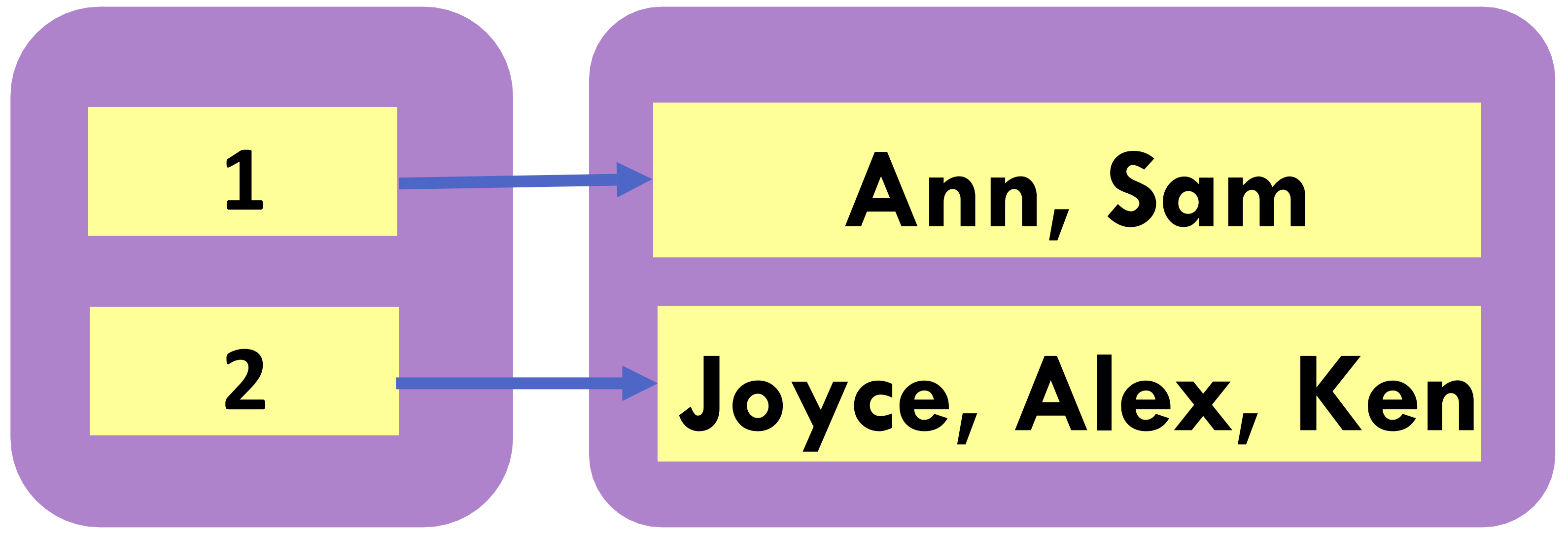


Can we use an existing data structure?

- Hint: Can we use a dictionary?

Can we use an existing data structure?

Dictionary to Sets: map from set IDs (key) to elements in the set (value)



Dictionary to Sets	
makeSet(value)	$\Theta(1)$
find(value)	$\Theta(n)$
union(x, y)	$\Theta(n)$

find(value): scan through every set under every representative

union(x, y): copy all elements from set pointed to by x into set pointed to by y. To union we still need to find x and y!

Can we do better? (e.g., speedup find)

Can We do Better? (e.g., speedup find)

- Hint: Can we swap the set IDs and elements in a dictionary? (values in a set are always distinct)
  - Key = element
  - Value = set ID

Disjoint Sets ADT

QuickFind

Optimizes Find operation

QuickUnion

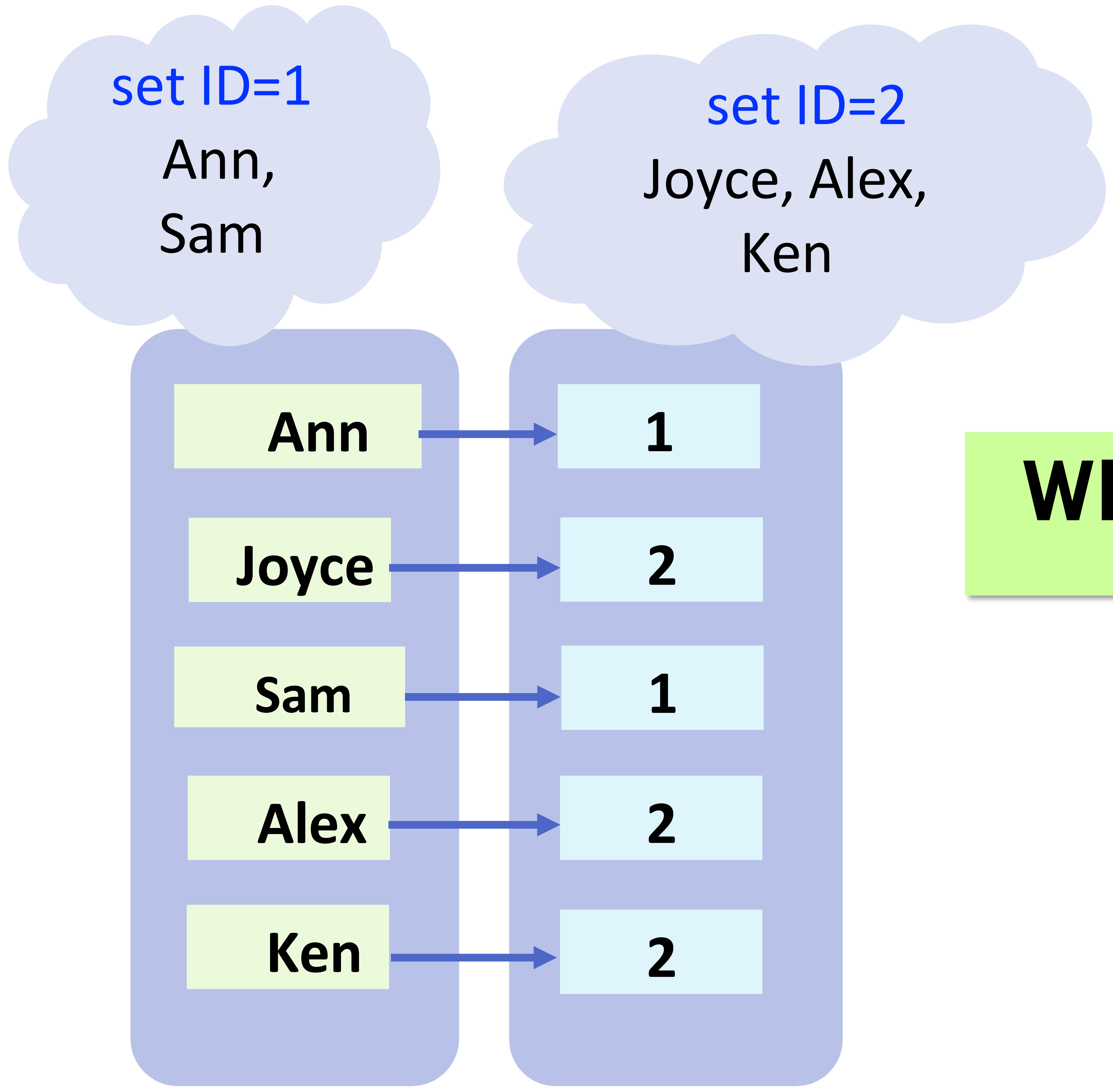
Optimizes union operation

WeightedQuickUnion

Avoids worst case run time for find

QuickFind Implementation

QuickFind: map from value(key) to set ID (value)



```
      find(Sam) = 1
      find(Ken) = 2
find(Sam) != find(Ken)
find(Sam) == find(Ann)
```

What is the time complexity of find and union?

	Dict to Sets	QuickFind
makeSet(value)	$\Theta(1)$	$\Theta(1)$
find(value)	$\Theta(n)$	$\Theta(1)$
union(x, y)	$\Theta(n)$	$\Theta(n)$

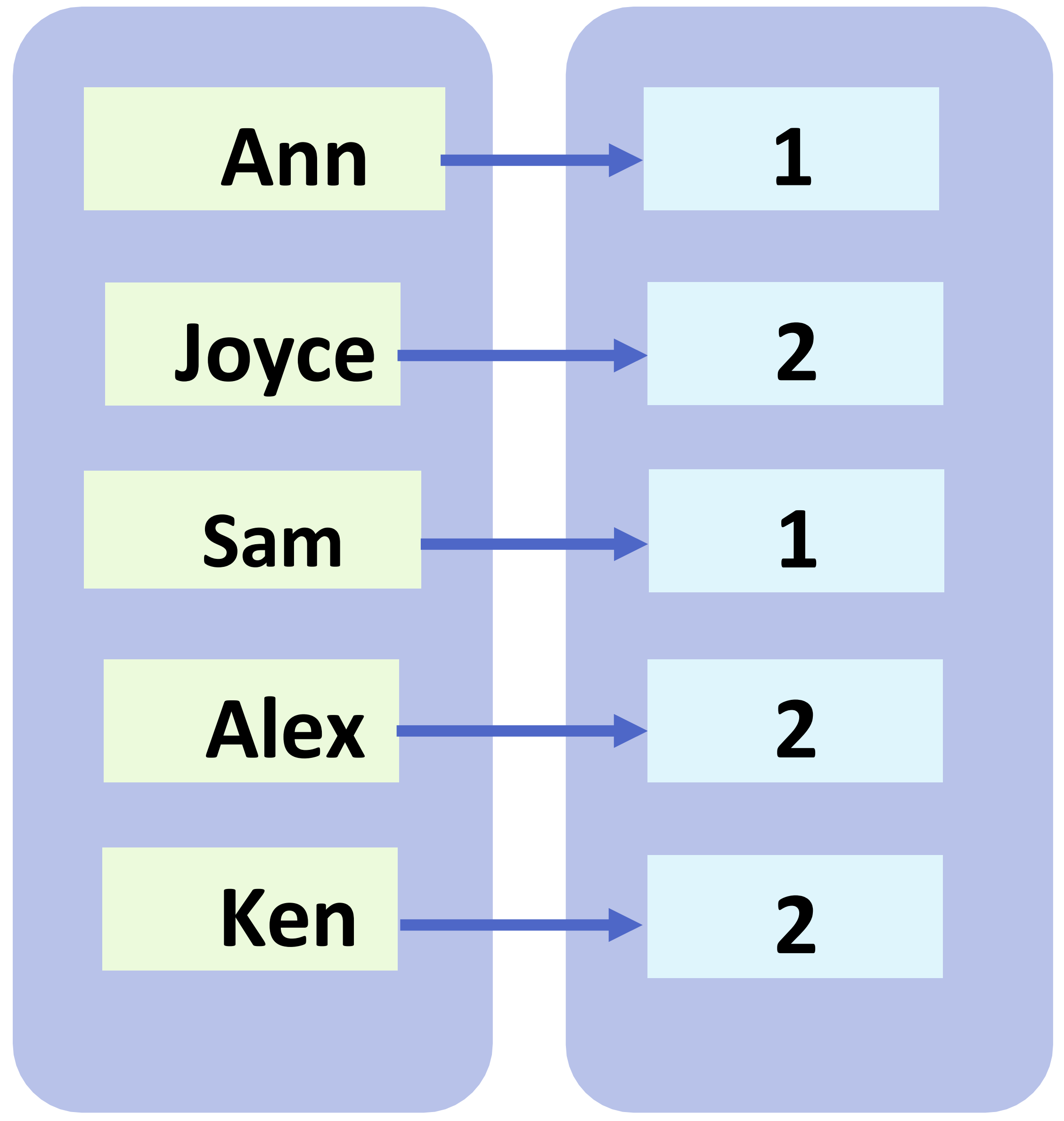
Finds are fast, what about unions?

Can We do Better? (e.g., speedup Union)

- Think about why the Union operation was slow
  - Well, because we had to scan through all elements
  - Ex: union (Ann, Alex)

QuickFind: map from value(key) to set ID (value)

Can we organize elements in a hierarchical structure that won't require us to look at all elements?



Is there a data structure that optimizes the Union operation? If yes, name or describe it.

Nobody has responded yet.

Hang tight! Responses are coming in.

Disjoint Sets ADT

QuickFind

Optimizes Find operation

QuickUnion

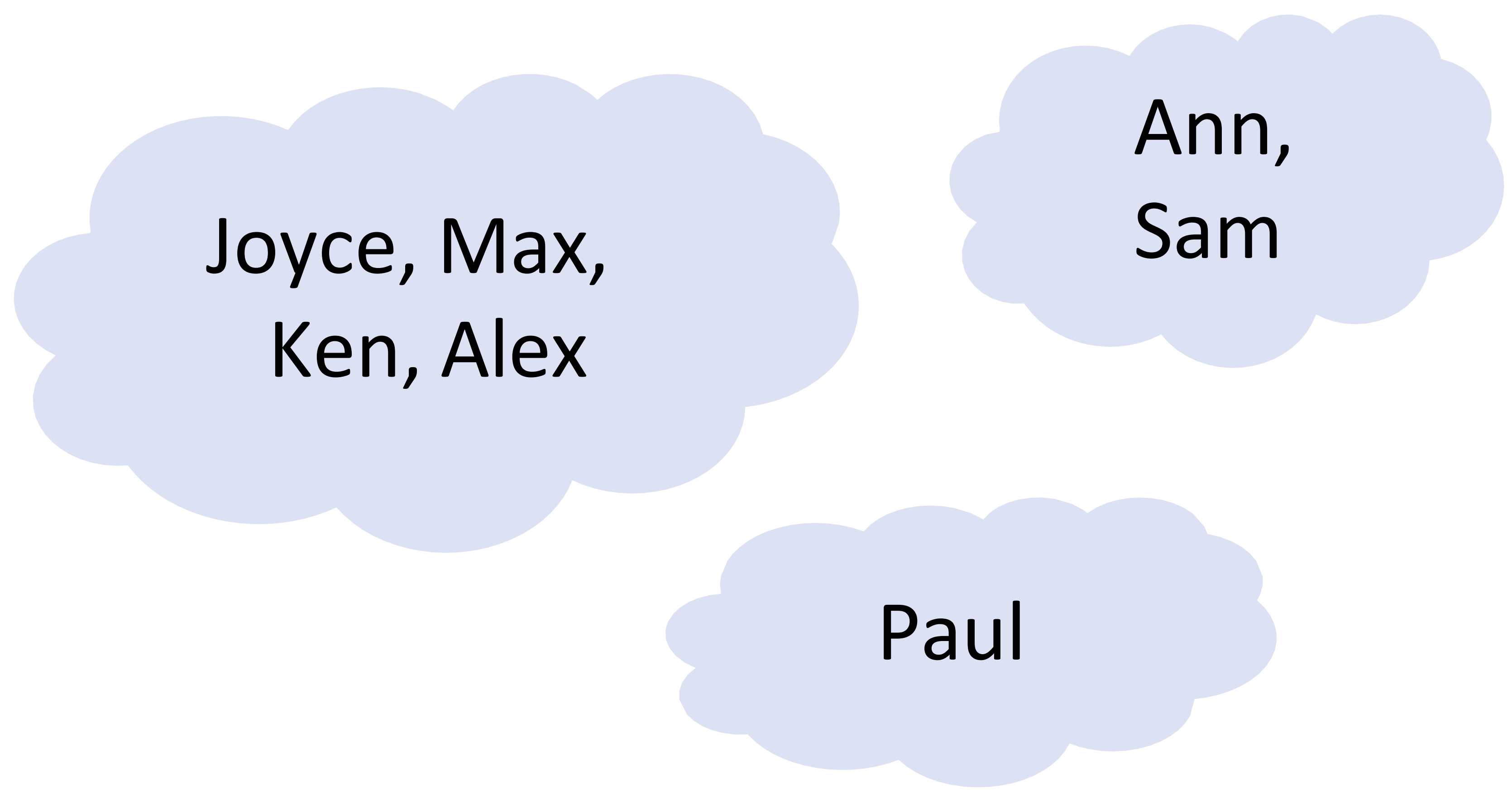
Optimizes union operation

WeightedQuickUnion

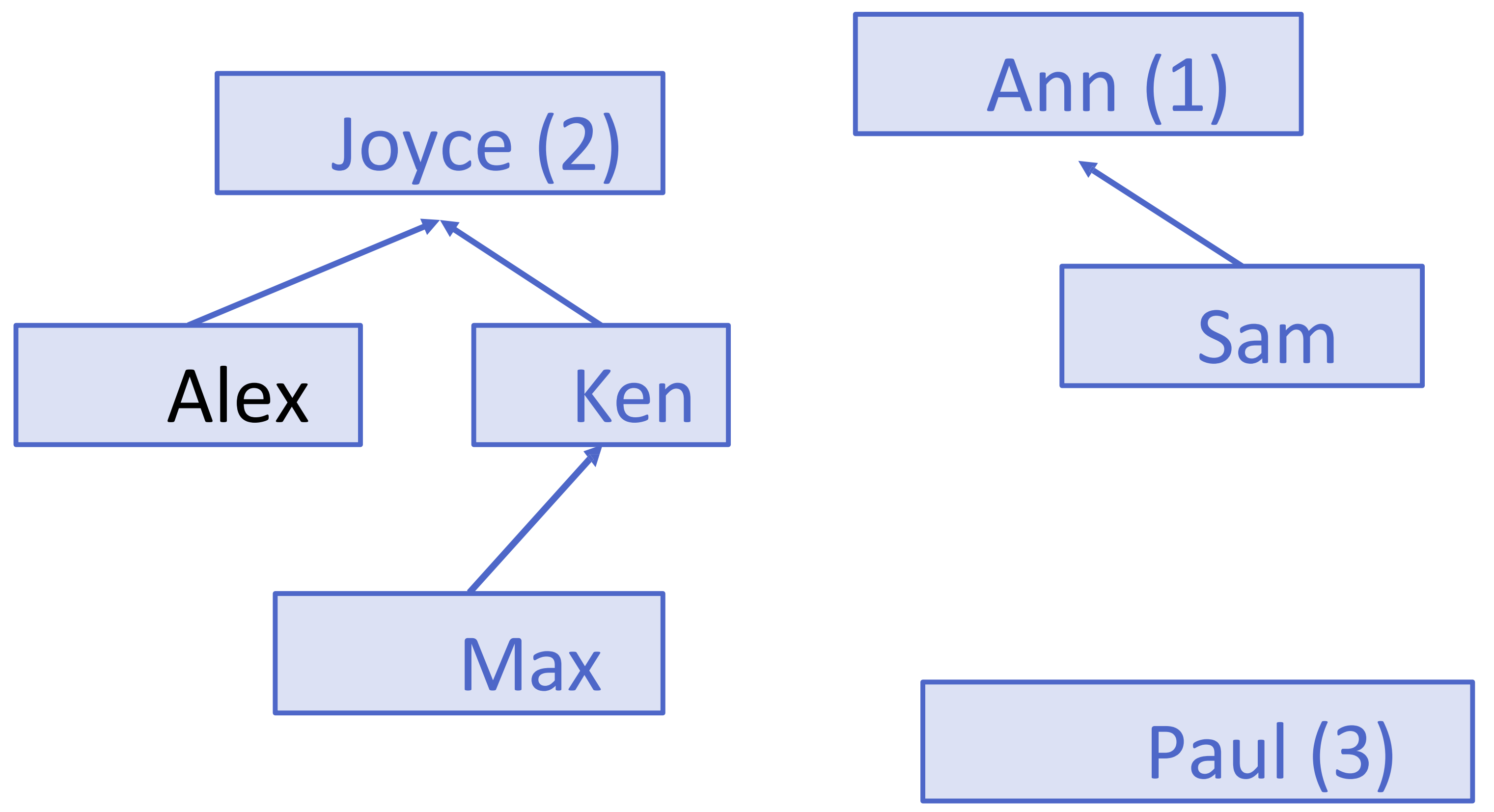
Avoids worst case run time for find

QuickUnion Data Structure – Key Idea

- QuickUnion requires to reset the ID of all elements in one set to the ID of other elements in other set.
- Place each set’s ID at one place (root)
- Each set becomes tree-like, but something slightly different called an **up-tree** (store pointers from children to parents!)



=



Abstract Idea of “Disjoint Sets”

Implementation using QuickUnion

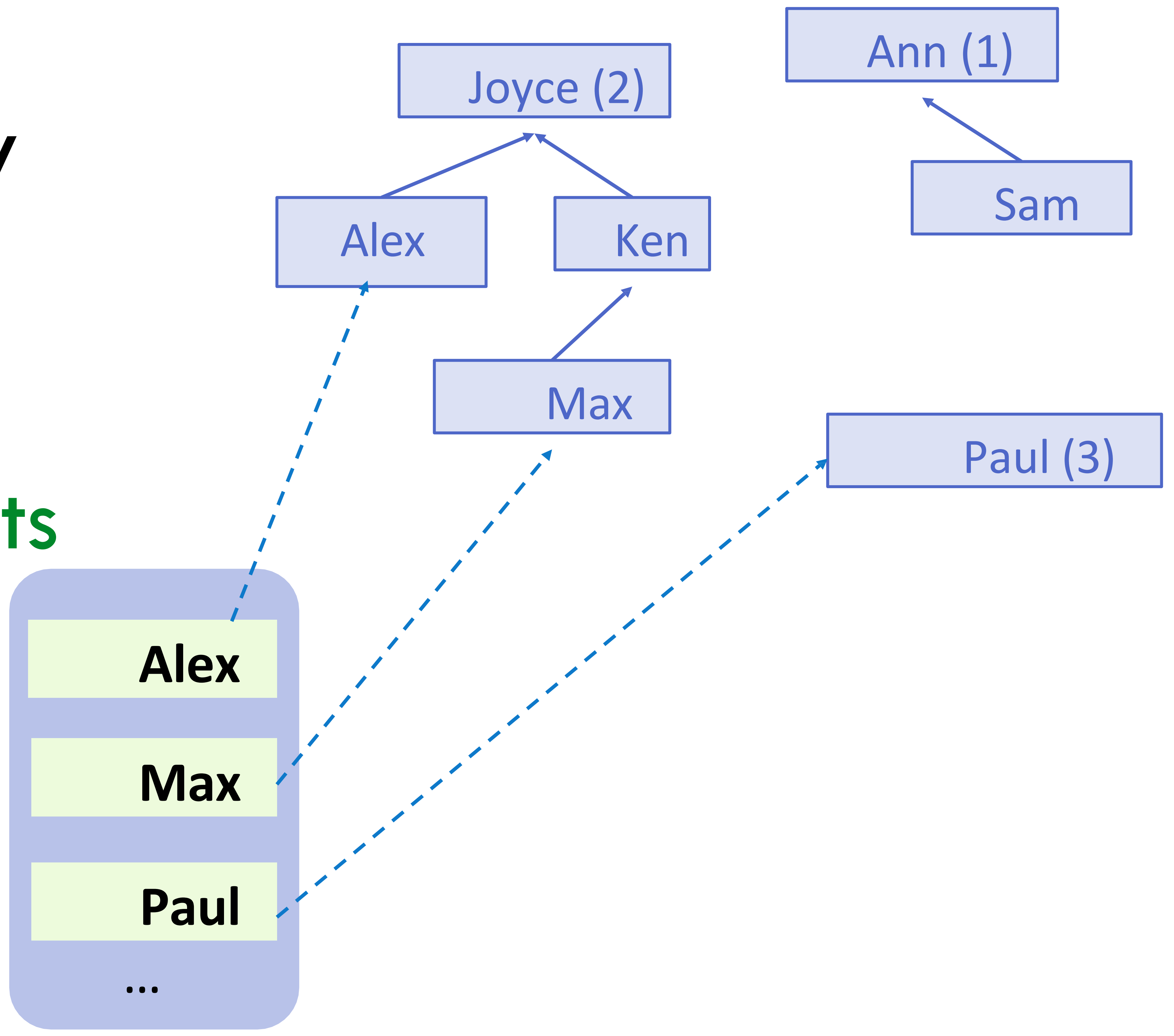
QuickUnion: find(u)

```
find(Ken):
  jump to Ken node
  travel upward until root
  return ID
```

```
        find(Sam) = 1
        find(Ken) = 2
find(Sam) != find(Ken)
find(Sam) == find(Ann)
```

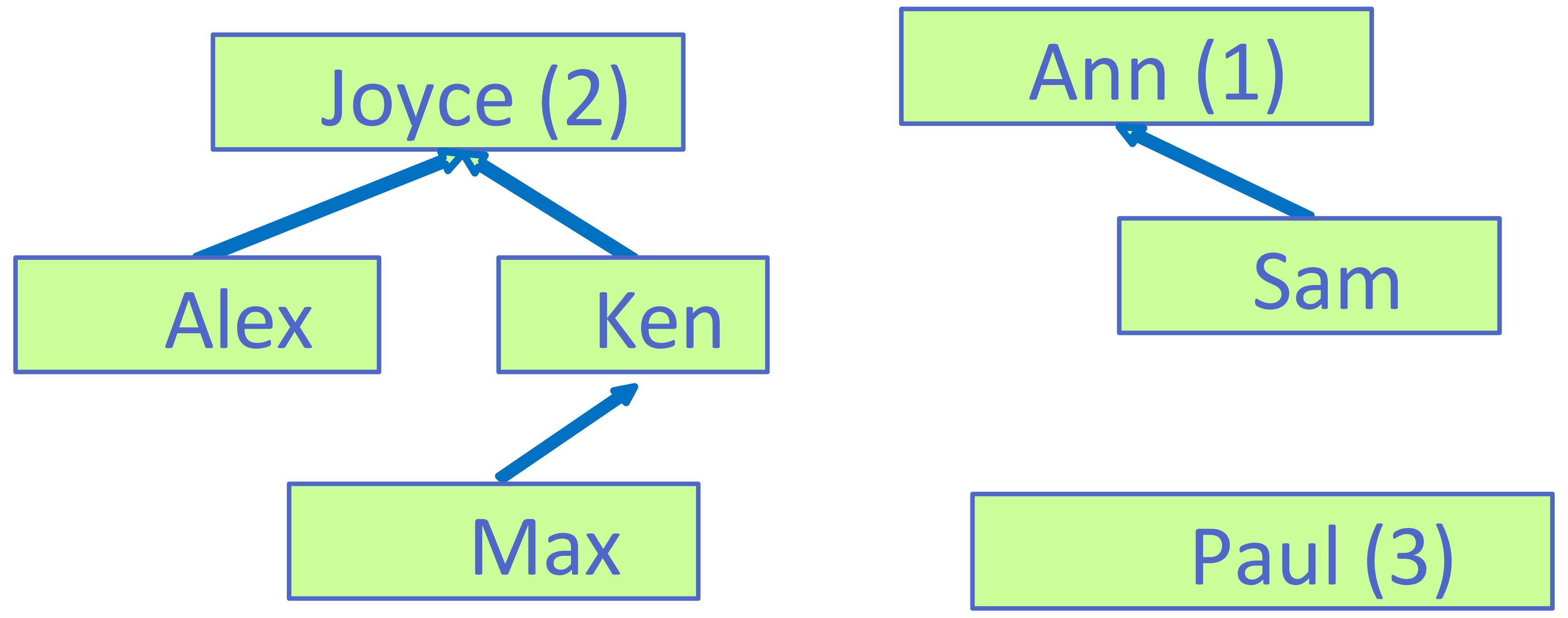
Key idea: can travel upward from any node to find its representative ID

- How do we jump to a node quickly?
  - Also store a dictionary from value to its node

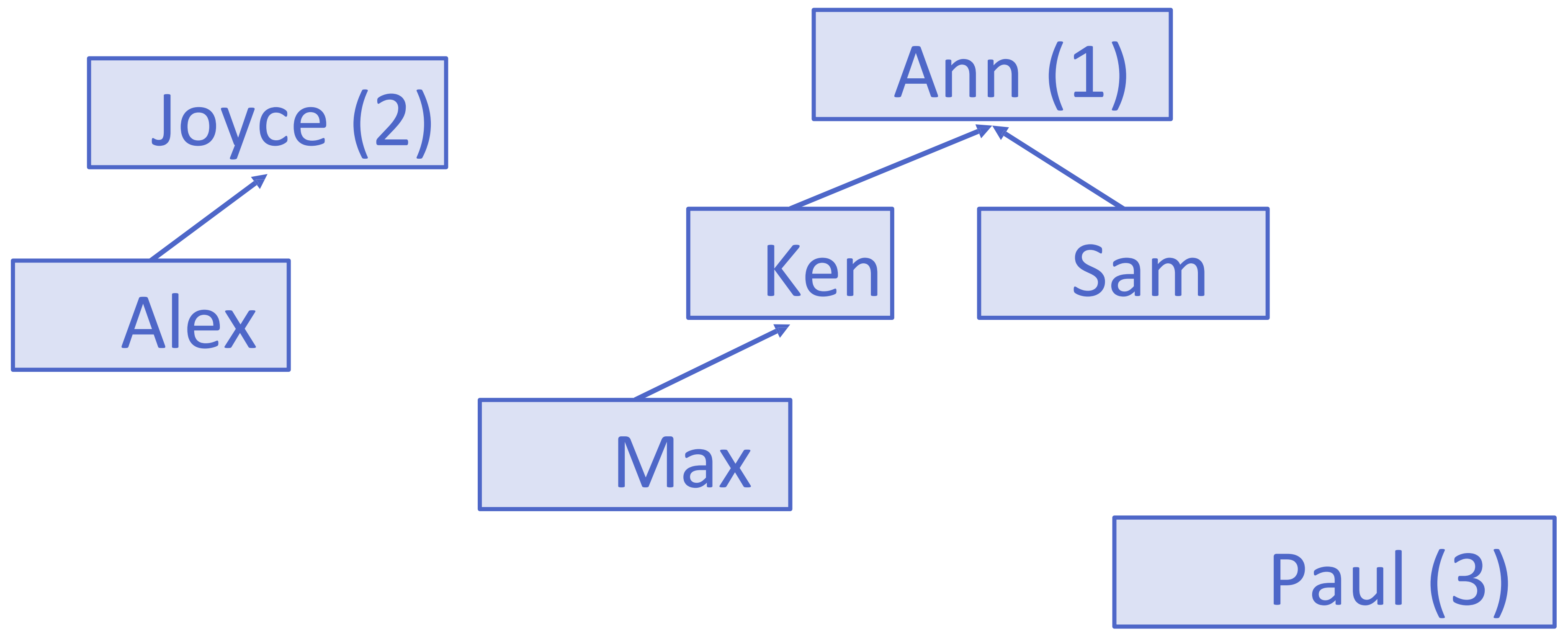


QuickUnion: union(u,v)

- Key idea: easy to simply rearrange pointers to union entire trees together!
- Which of these implementations would you prefer?

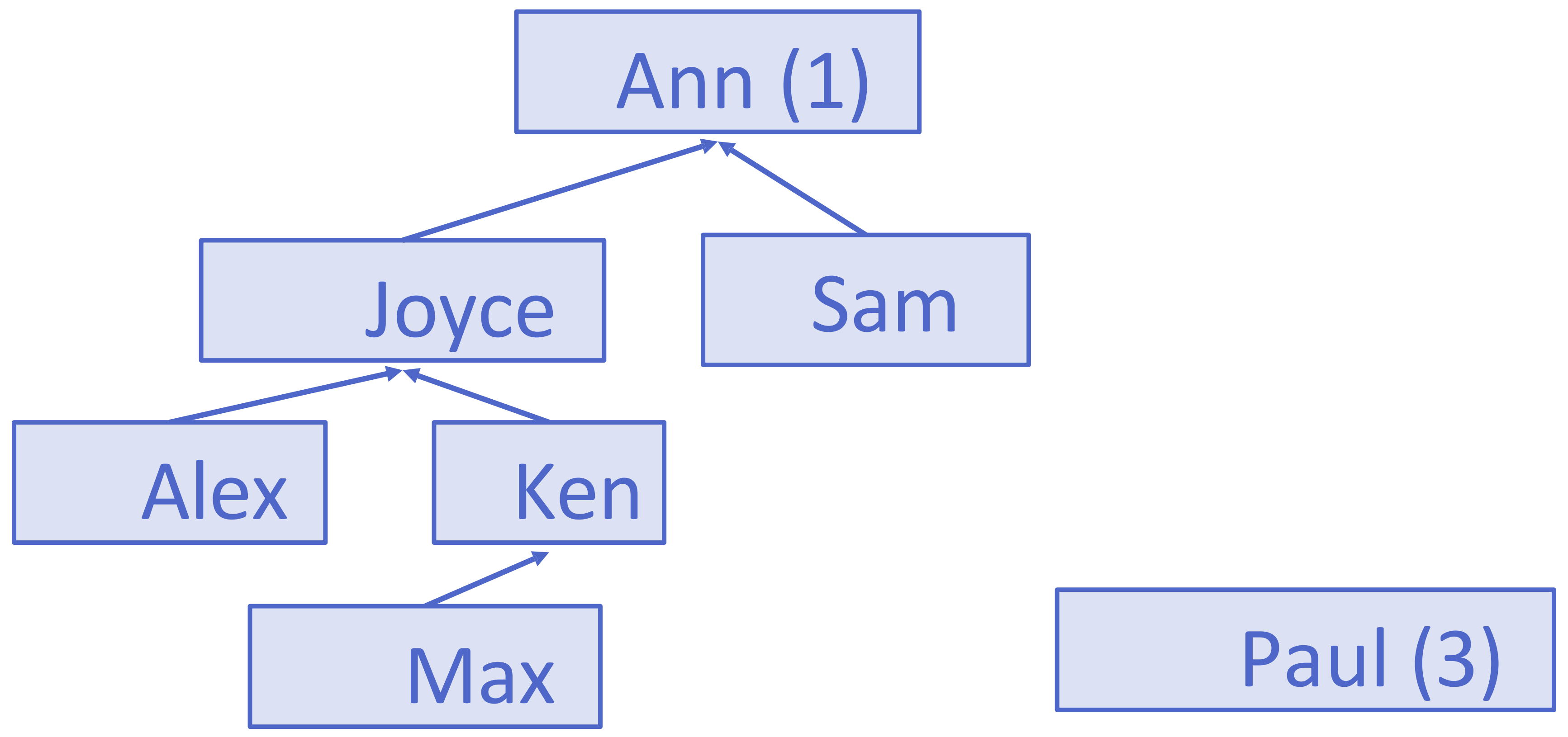


```
union(Ken,Sam):
  rootS = find(Sam)
  set Ken to point to rootS
```



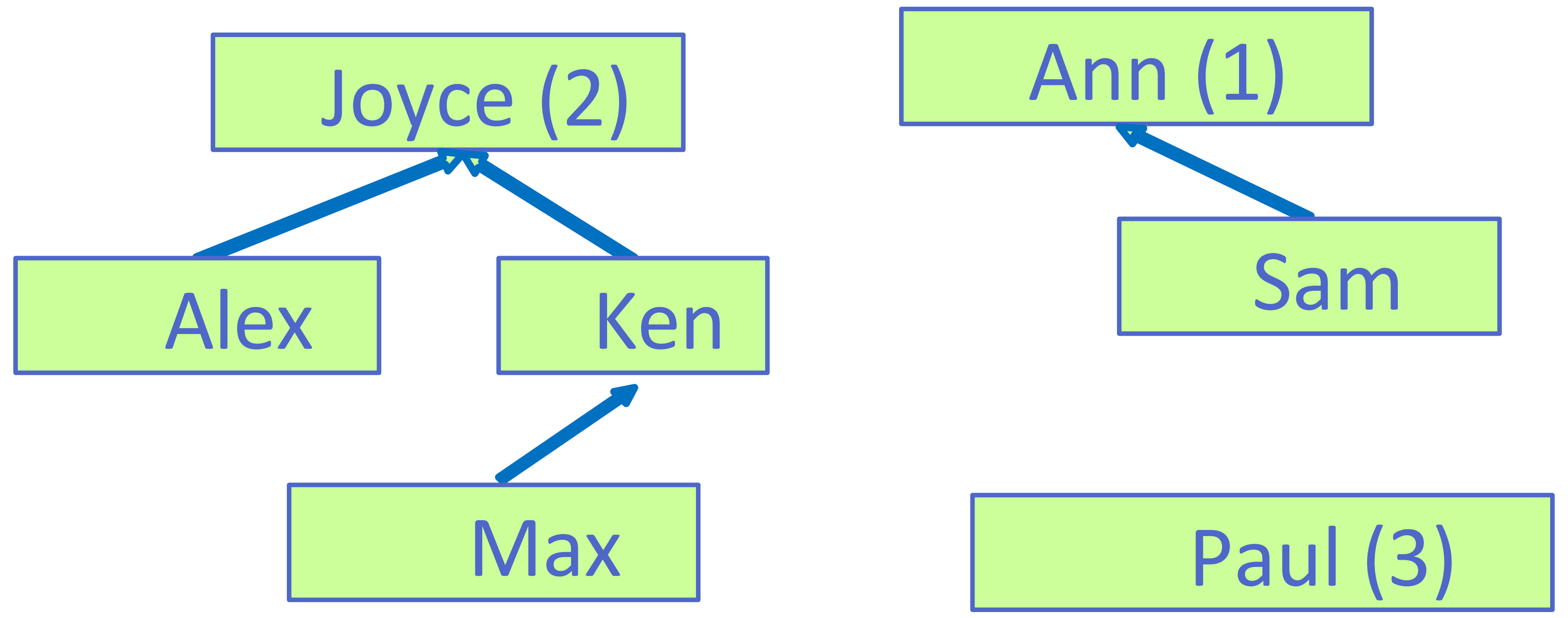
```
union(Ken,Sam):
rootK = find(Ken)
rootS = find(Sam)
set rootK to point to rootS
```

✓



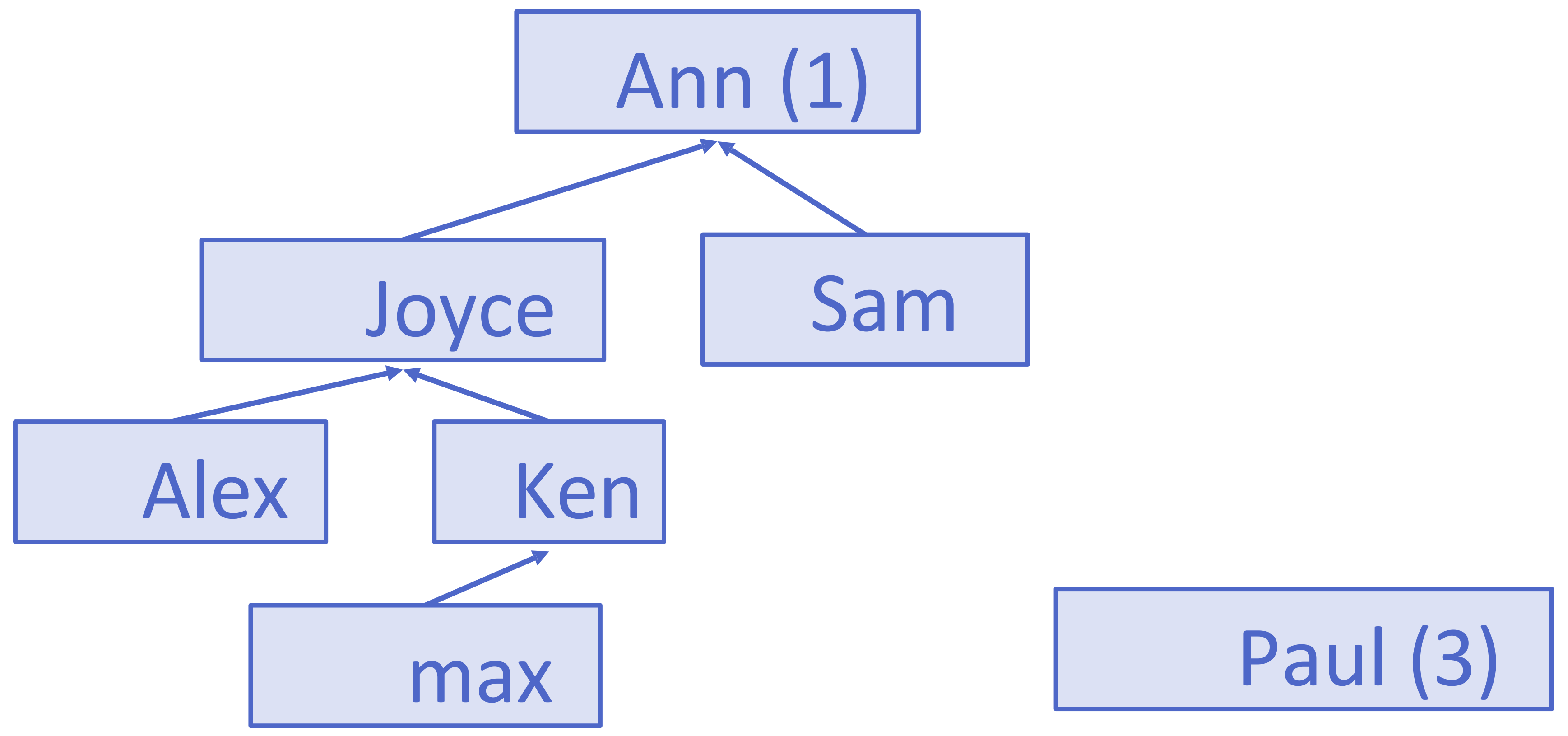
QuickUnion: Why Bother with the Second Root?

- Key idea: will help minimize runtime for future find() calls if we keep the height of the tree short!
  - Pointing directly to the second element would make the tree taller



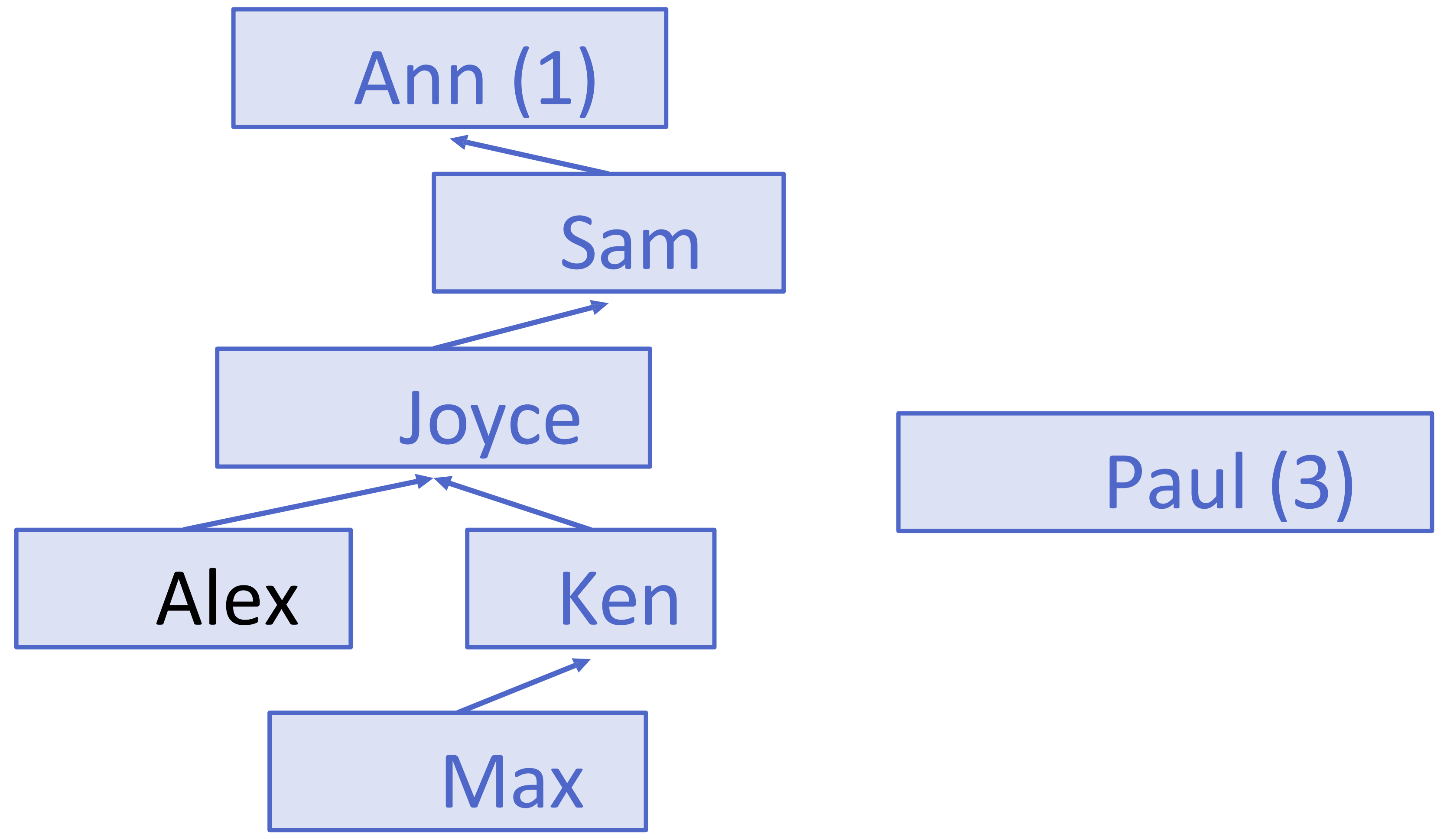
```
union(Ken,Sam):
  rootK = find(Ken)
  rootS = find(Sam)
  set rootK to point to rootS
```

✓



Why not just use:

```
  union(Ken,Sam):
  rootK = find(Ken)
     set rootK to point to Sam
```



QuickUnion: Time Complexity

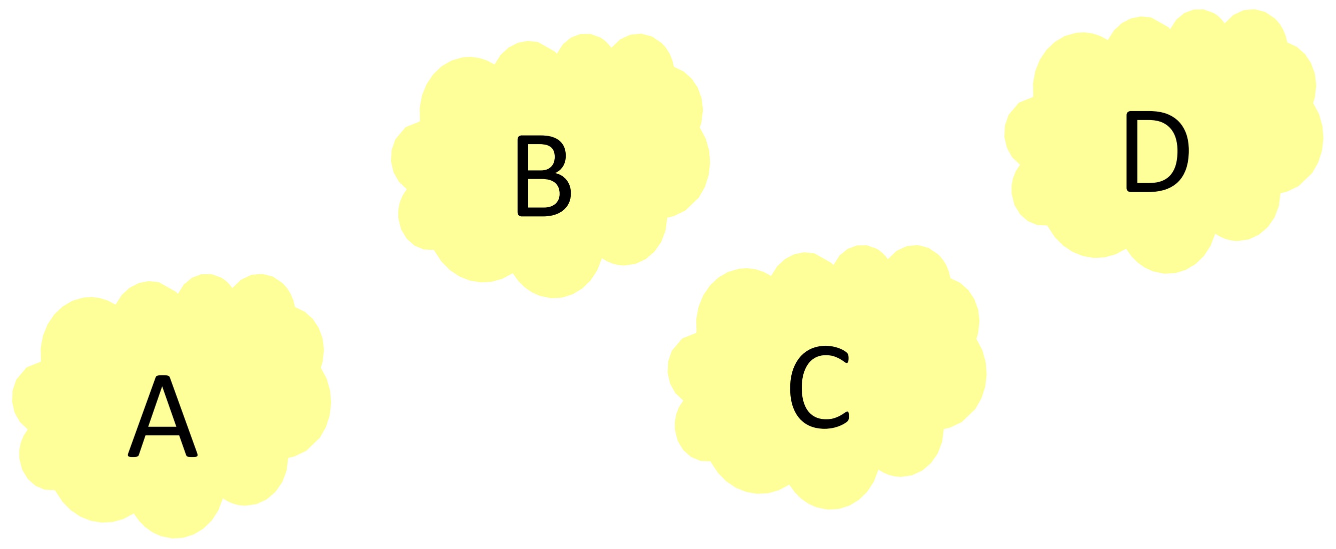
- Only if we discount the runtime from union's calls to find! * Otherwise, $\Theta(n)$

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  set rootA to point to rootB
```

	QuickFind	QuickUnion
makeSet(value)	$\Theta(1)$	$\Theta(1)$
findSet(value)	$\Theta(1)$	$\Theta(n)$
union(x,y)	$\Theta(n)$	$\Theta(1)$*

QuickUnion: Let's Build a Worst Case

- Even with the "use-the-roots" implementation of union, try to come up with a series of calls to union that would create a worst-case running for find on these Disjoint Sets.

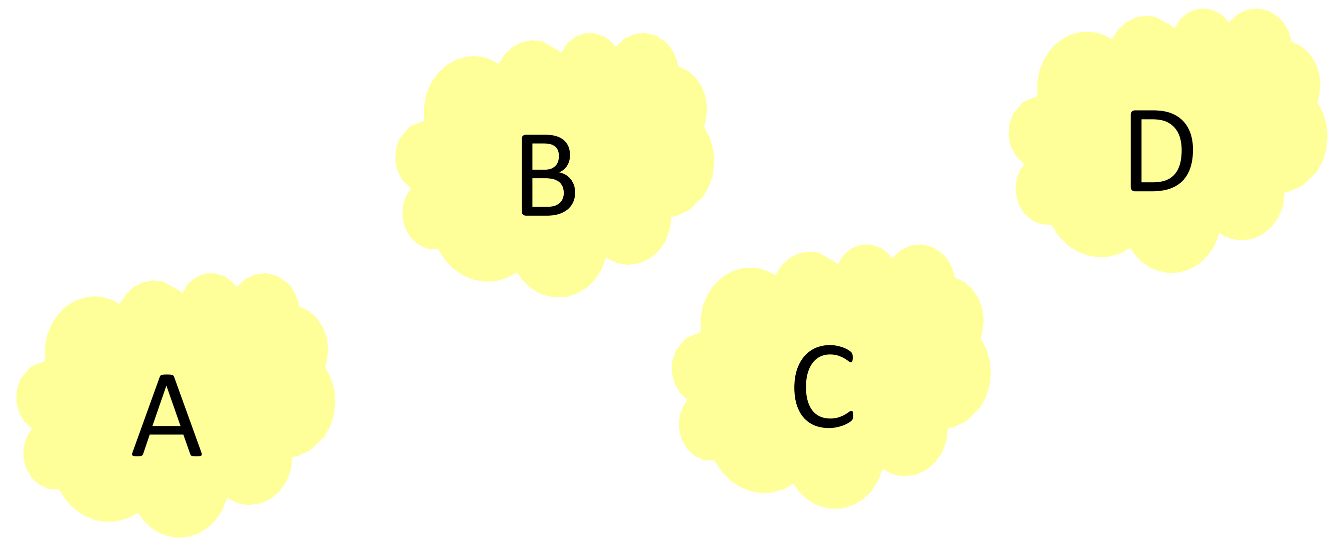


```
find(A):
  jump to A node
  travel upward until root
  return ID
```

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  set rootA to point to rootB
```

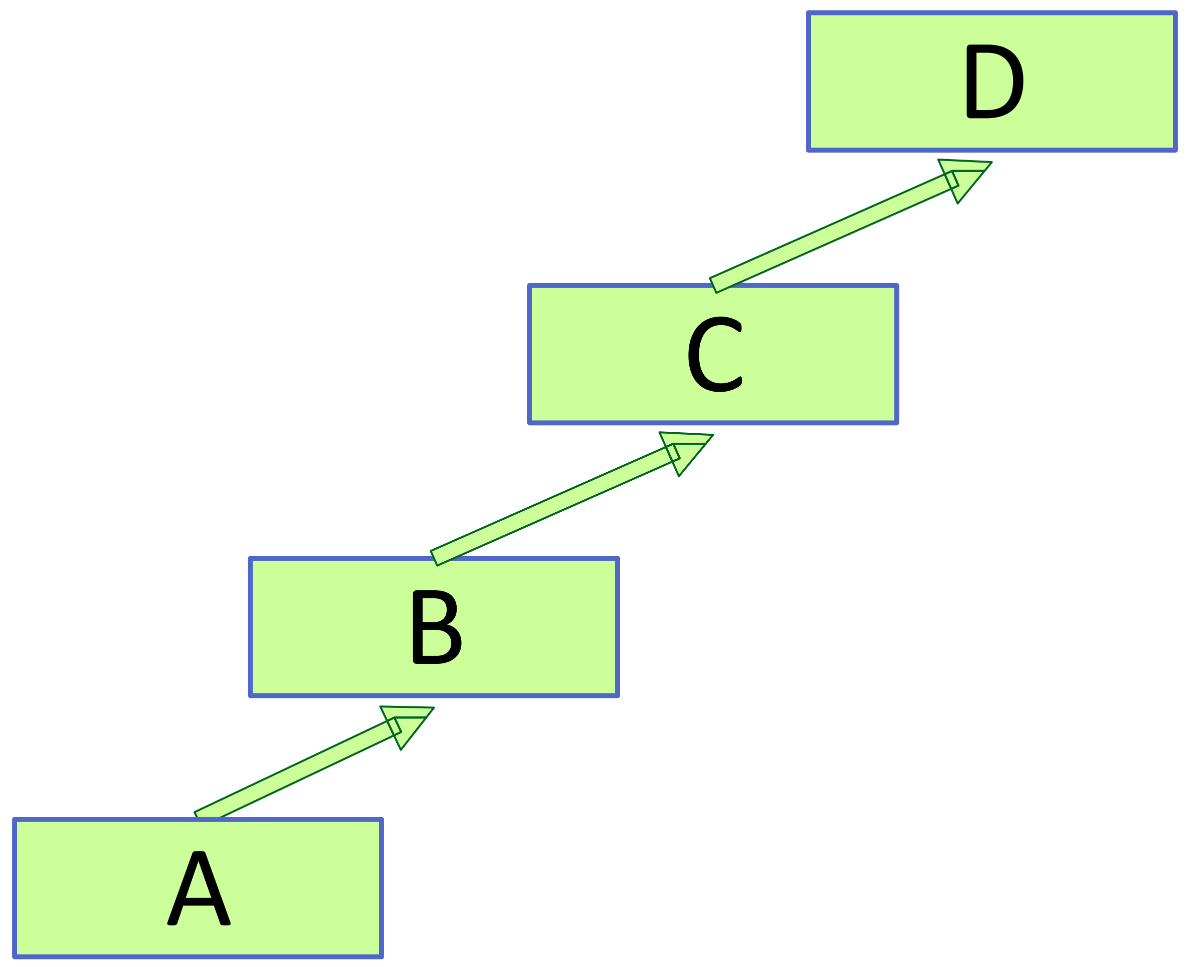
QuickUnion: Let's Build a Worst Case

- Even with the "use-the-roots" implementation of union, try to come up with a series of calls to union that would create a worst-case running for find on these Disjoint Sets.



Operations

```
union(A,B)
union(B,C)
union(C,D)
find(A)
```

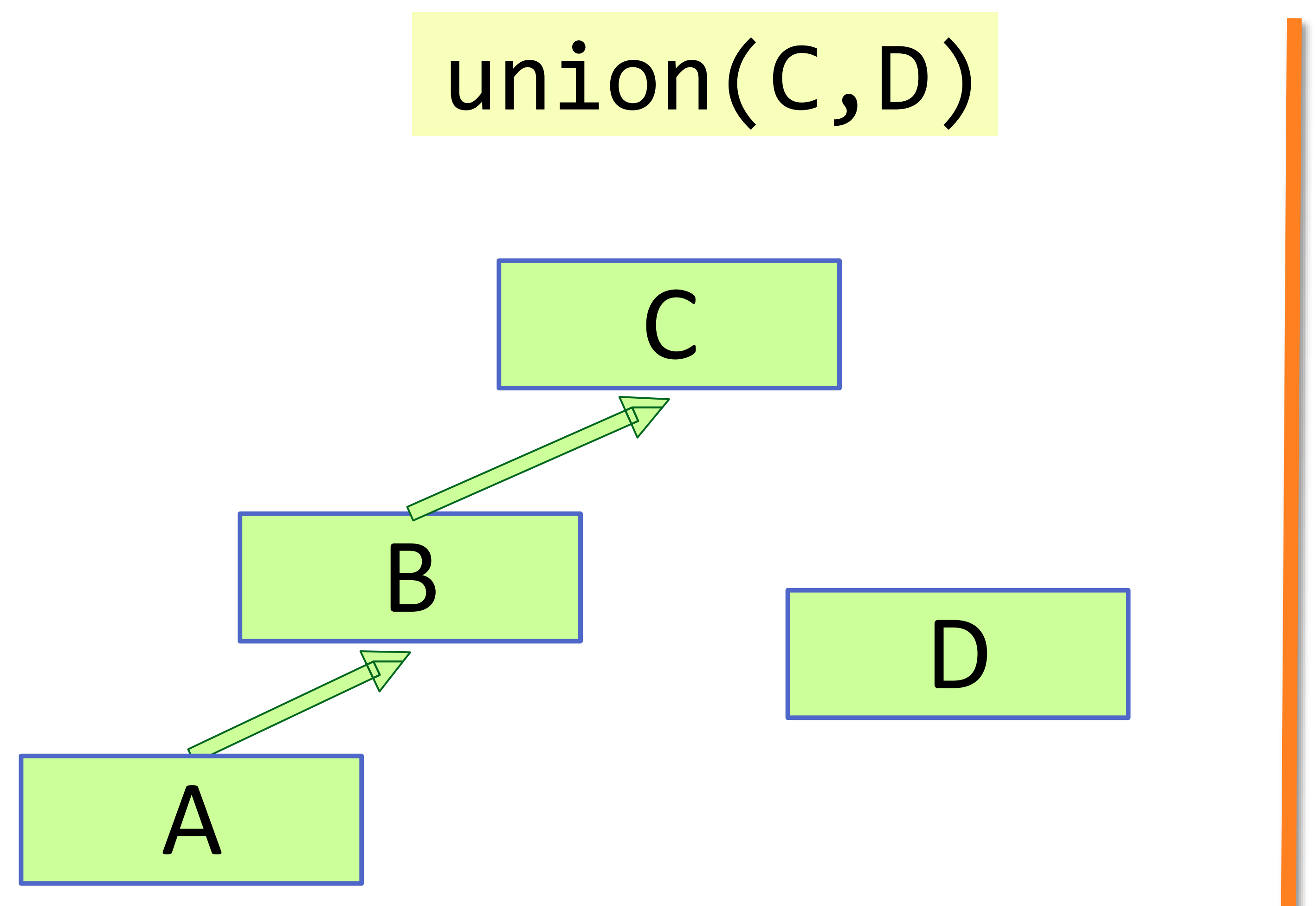


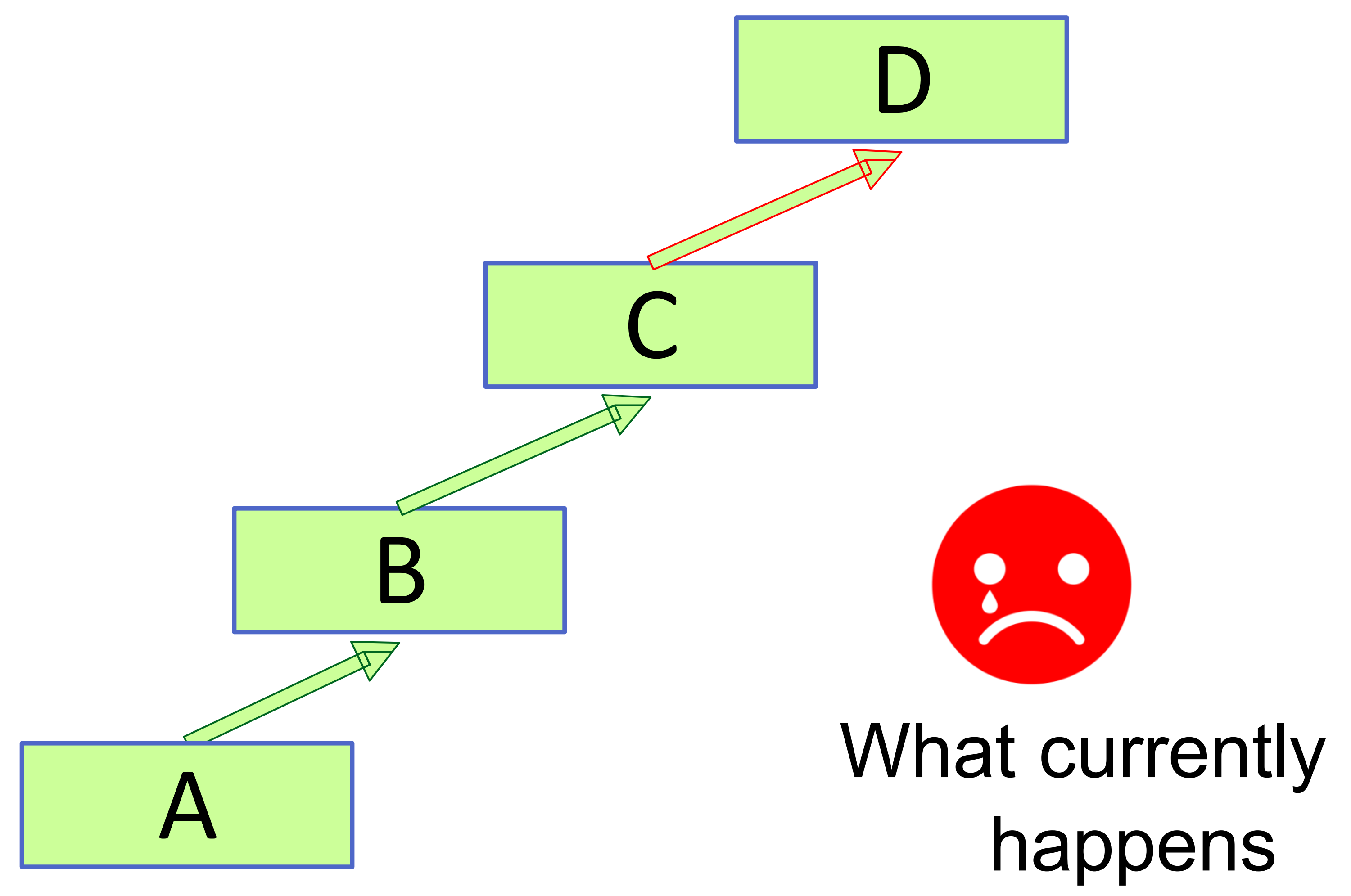
```
find(A):
  jump to A node
  travel upward until root
  return ID
```

```
union(A, B):
  rootA = find(A)
  rootB = find(B)
  set rootA to point to rootB
```

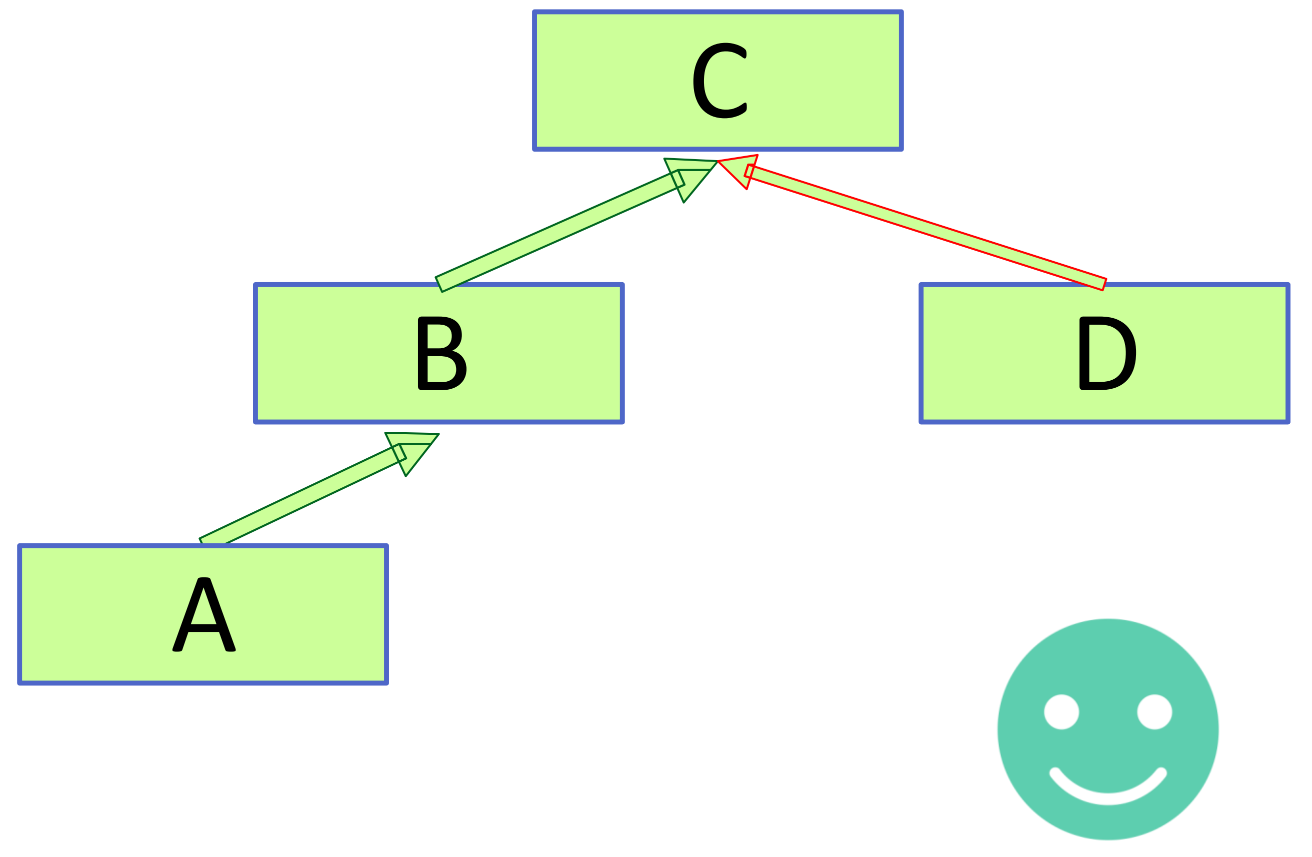
Analyzing the QuickUnion Worst Case

- How did we get a degenerate tree?
  - We can get a degenerate tree if we put the root of a large tree under the root of a small tree
  - In QuickUnion, rootA always goes under rootB
    - But what if we could ensure the smaller tree went under the larger tree?





What currently happens



What would help avoid degenerate tree

Disjoint Sets ADT

QuickFind

Optimizes Find operation

QuickUnion

Optimizes union operation

WeightedQuickUnion

Avoids worst case run time for find

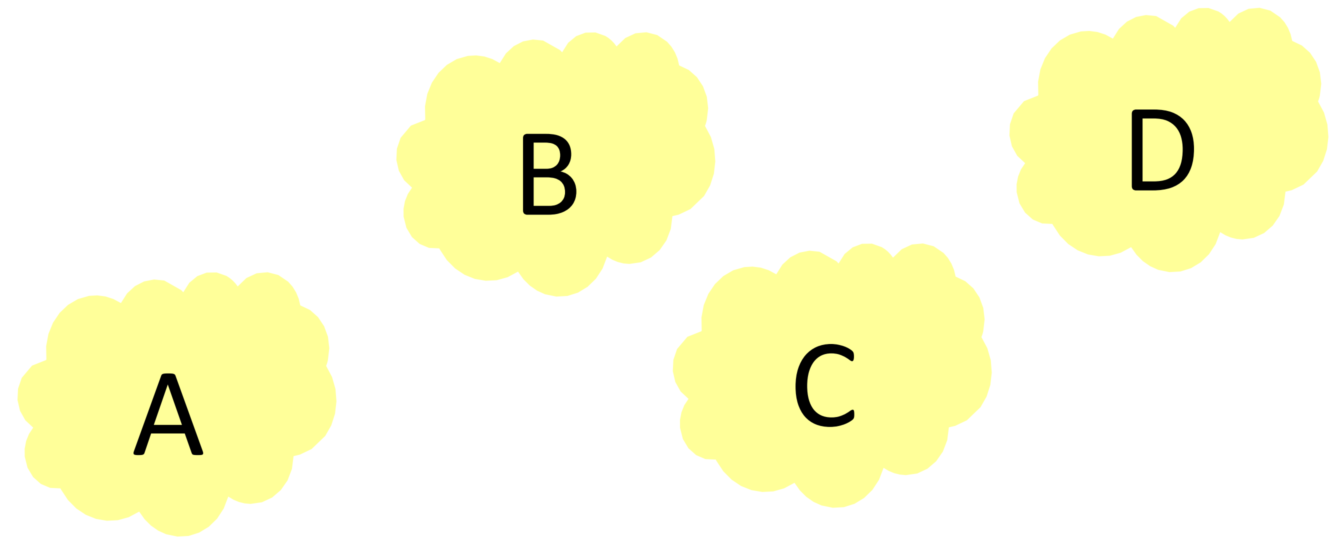
WeightedQuickUnion

- Goal: Always pick the smaller tree to go under the larger tree
- Implementation: Store the number of nodes (or "weight") of each tree in the root
  - Constant-time lookup instead of having to traverse the entire tree to count

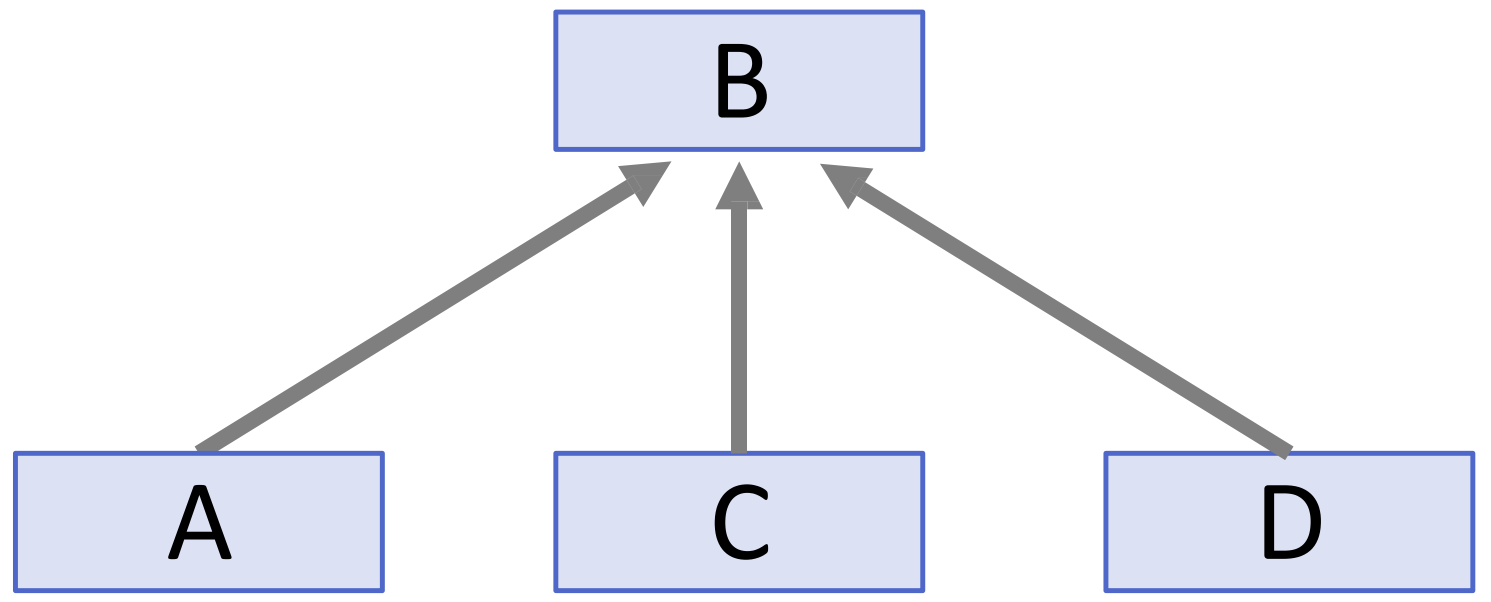
```
union(A,B):
  rootA = find(A)
  rootB = find(B)

  put lighter root under heavier root
```

```
union(A,B)
union(B,C)
union(C,D)
find(A)
```



Now what happens?



Perfect! Best runtime we can get.

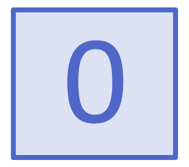
WeightedQuickUnion: Performance

- union()'s runtime is still dependent on find()'s runtime, which is a function of the tree's height
- What's the worst-case height for Weighted QuickUnion?

```
union(A,B):
  rootA = find(A)
  rootB = find(B)
  put lighter root under heavier root
```

WeightedQuickUnion: Performance

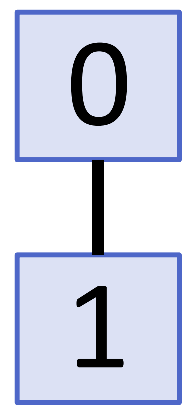
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0

WeightedQuickUnion: Performance

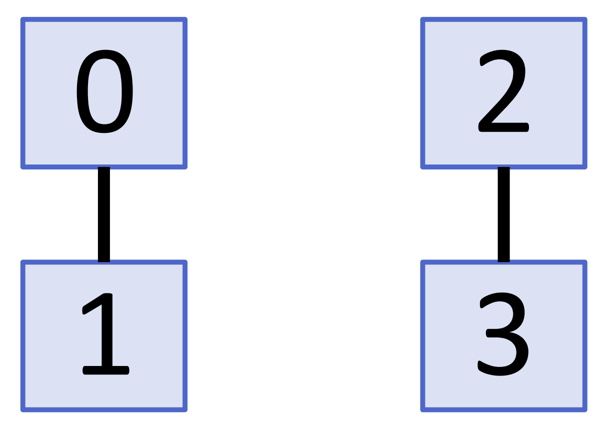
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1

WeightedQuickUnion: Performance

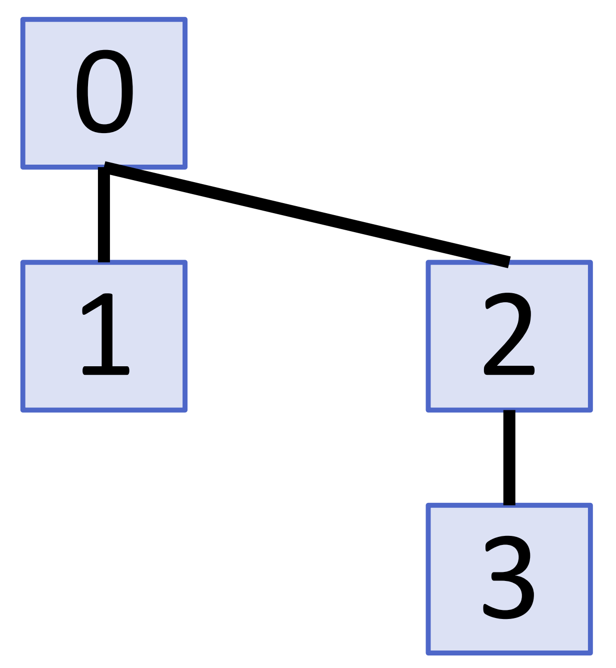
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	?

WeightedQuickUnion: Performance

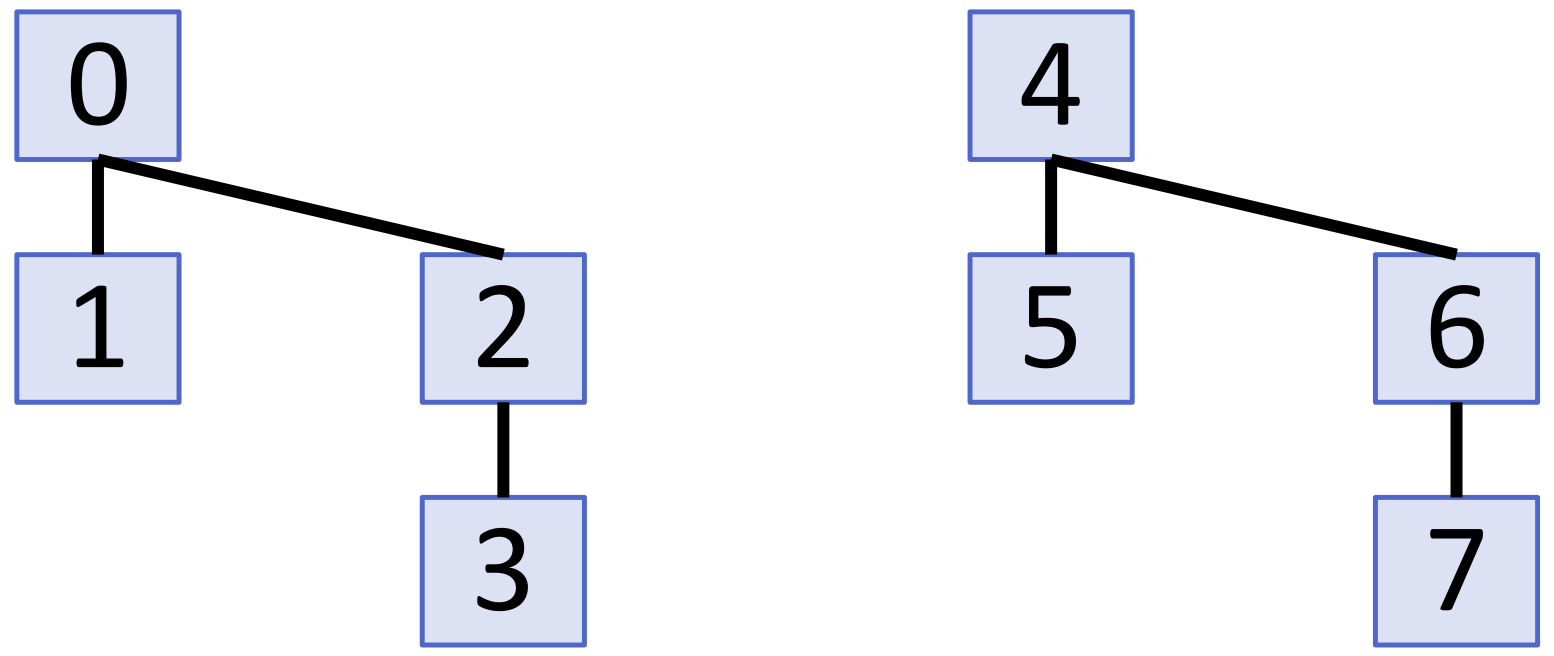
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	2

WeightedQuickUnion: Performance

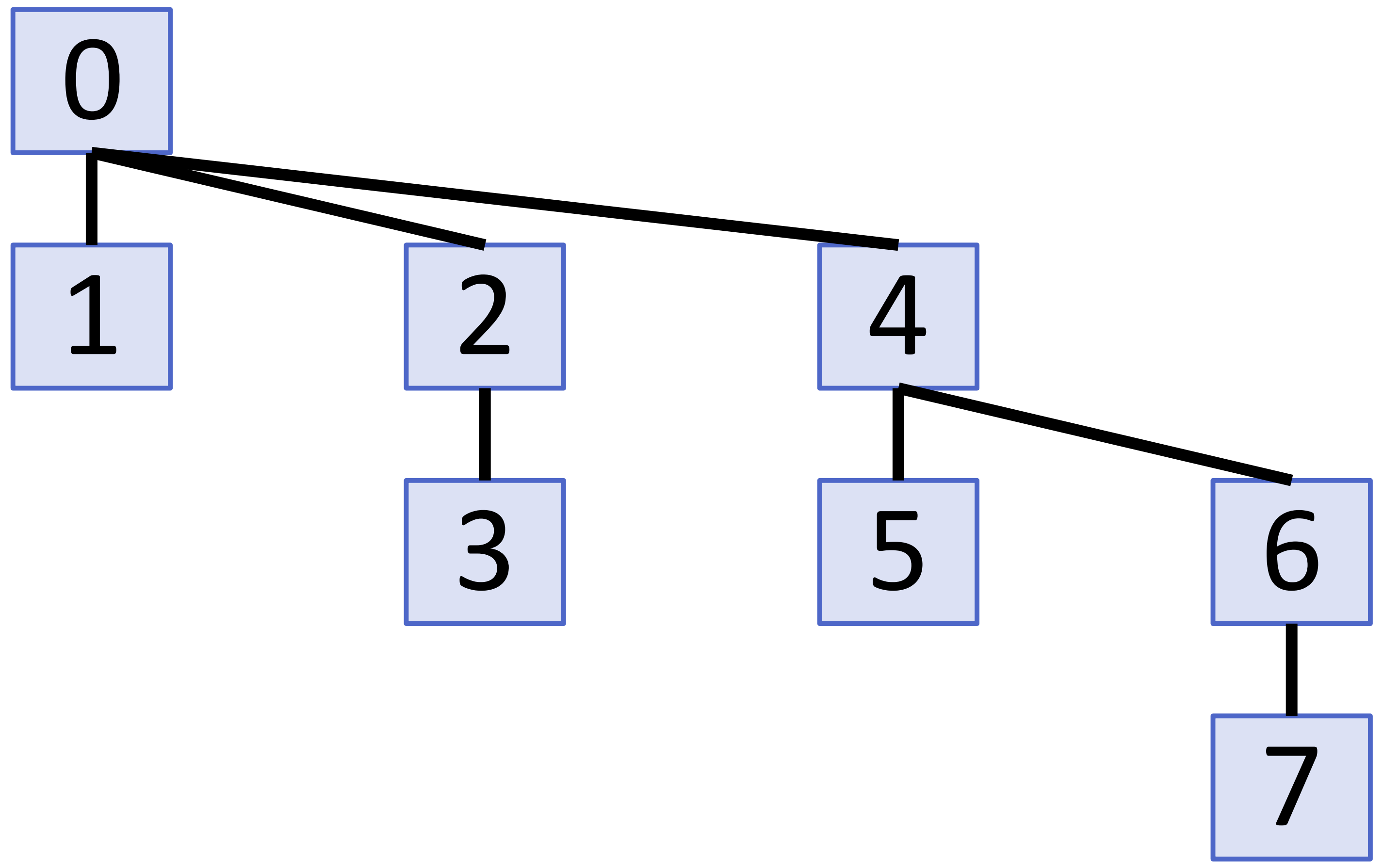
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	2
8	?

WeightedQuickUnion: Performance

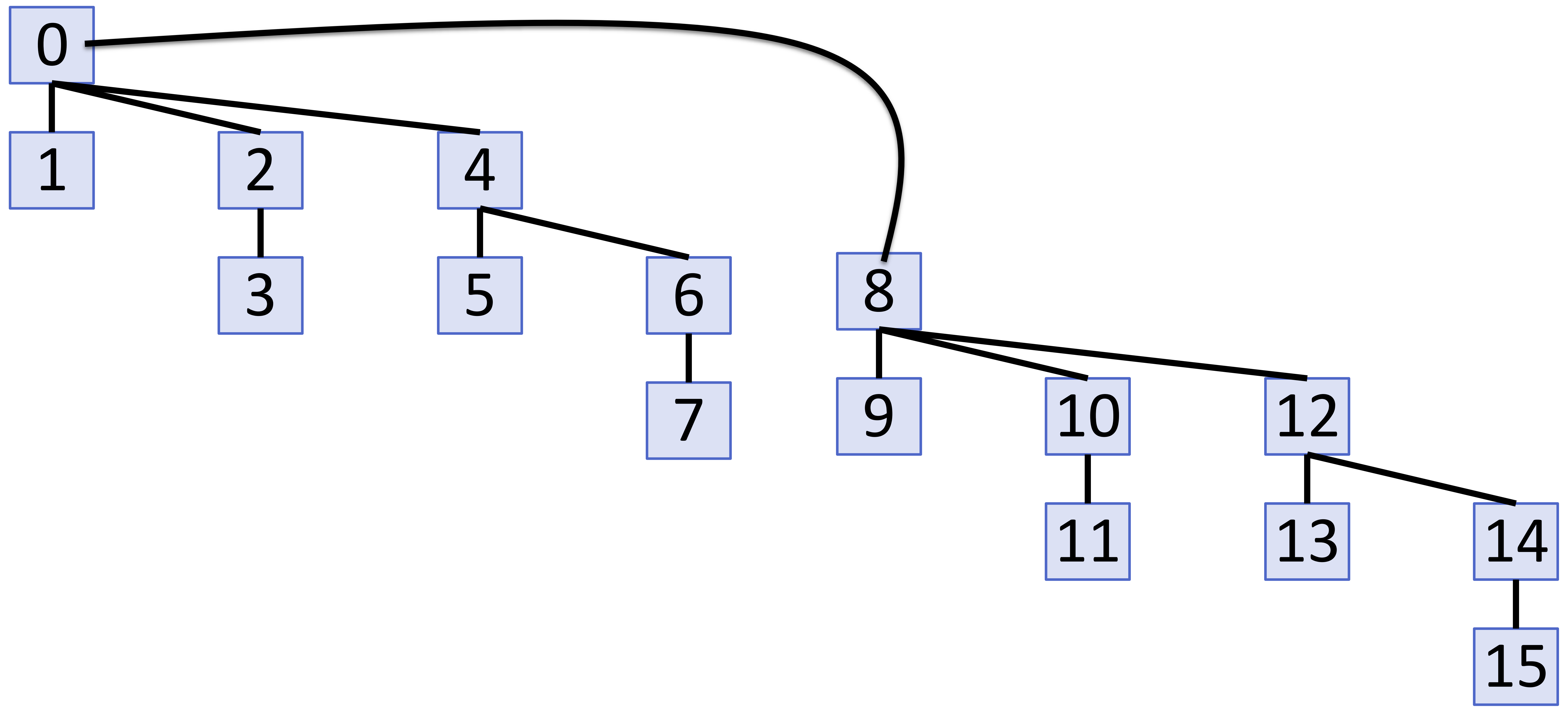
- Consider the worst case where the tree height grows as fast as possible



N	H
1	0
2	1
4	2
8	3

WeightedQuickUnion: Performance

- Consider the worst case where the tree height grows as fast as possible



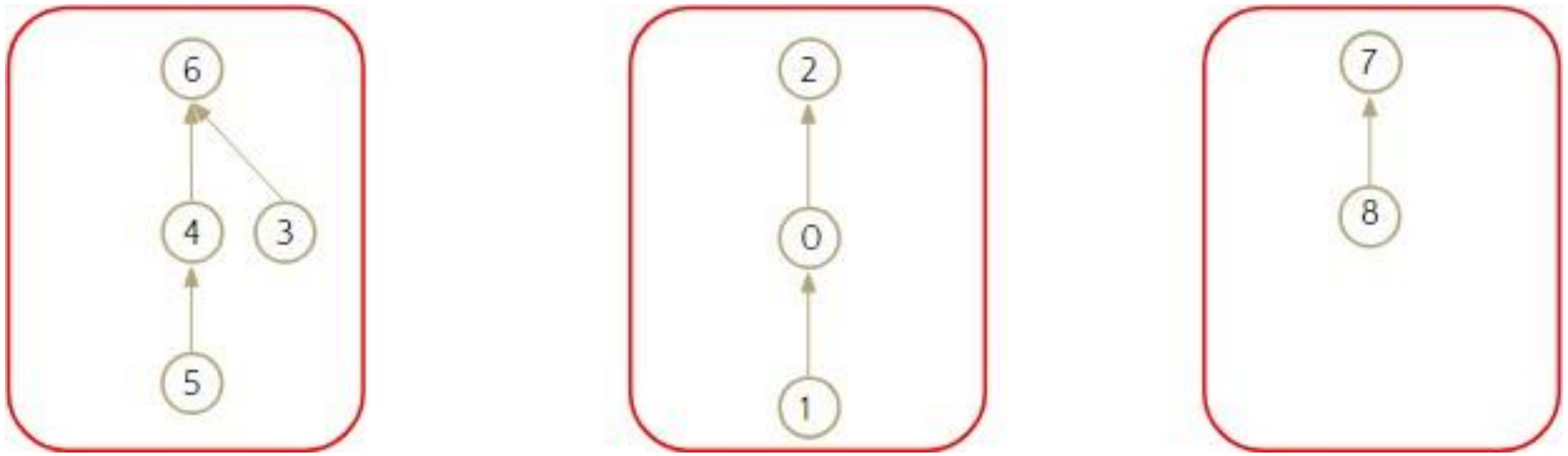
N	H
1	0
2	1
4	2
8	3
16	4

Why Weights Instead of Heights?

- We used the number of items in a tree to decide upon the root. Why not use the height of the tree?
  - HeightedQuickUnion's runtime is asymptotically the same: $\Theta(\log(n))$
  - It's usually easier to track weights than heights

Concept Check

- Draw the resulting state of the forest for the following Disjoint Set after completing the given method calls.
  - makeSet(9)
  - union(1, 9)
  - union(0, 7)
  - union(8, 5)



WeightedQuickUnion: Runtime

	QuickFind	QuickUnion	Weighted QuickUnion
`makeSet(value)`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`find(value)`	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
`union(x,y)` `assuming root args`	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
`union(x,y)`	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$

Questions