CS2040S
Data Structures and Algorithms

Union-Find

Today

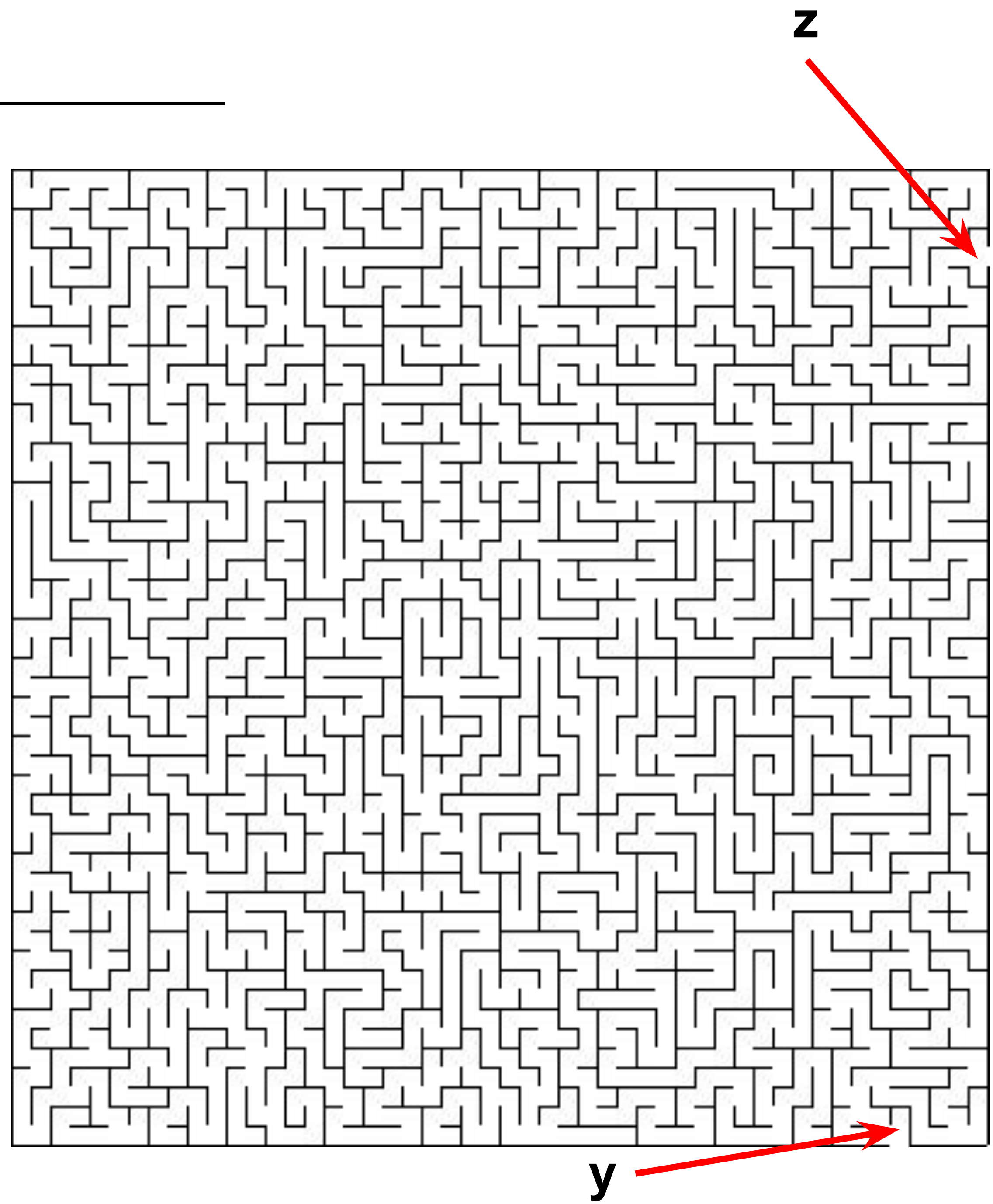
Disjoint Set Data Structure

- Problem: Dynamic Connectivity
- Algorithm: Union-Find
- Applications

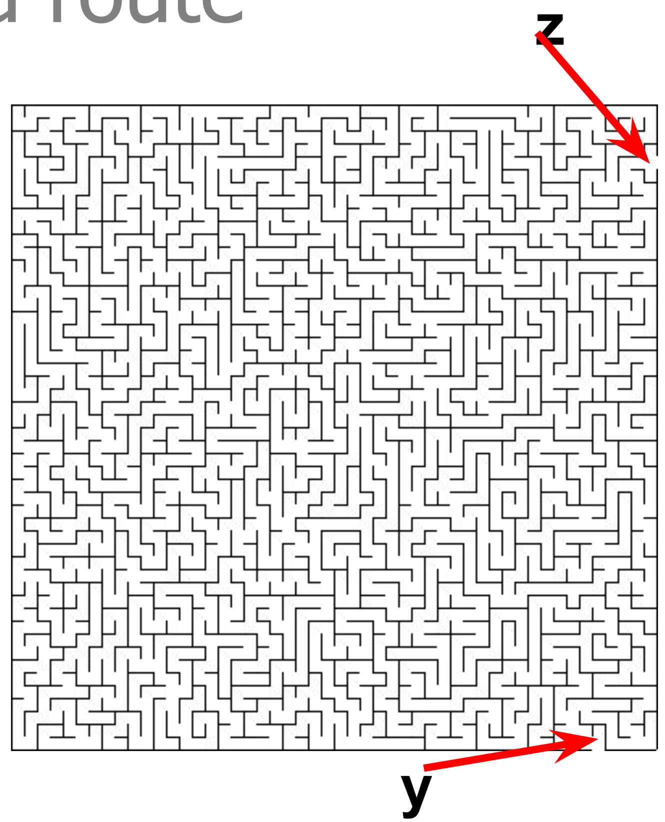
- Revisiting Kruskals

Mazes

Is there any route from y to z?



Best way to find if there is a route from Y to Z?



Breadth-first search

Depth-first search

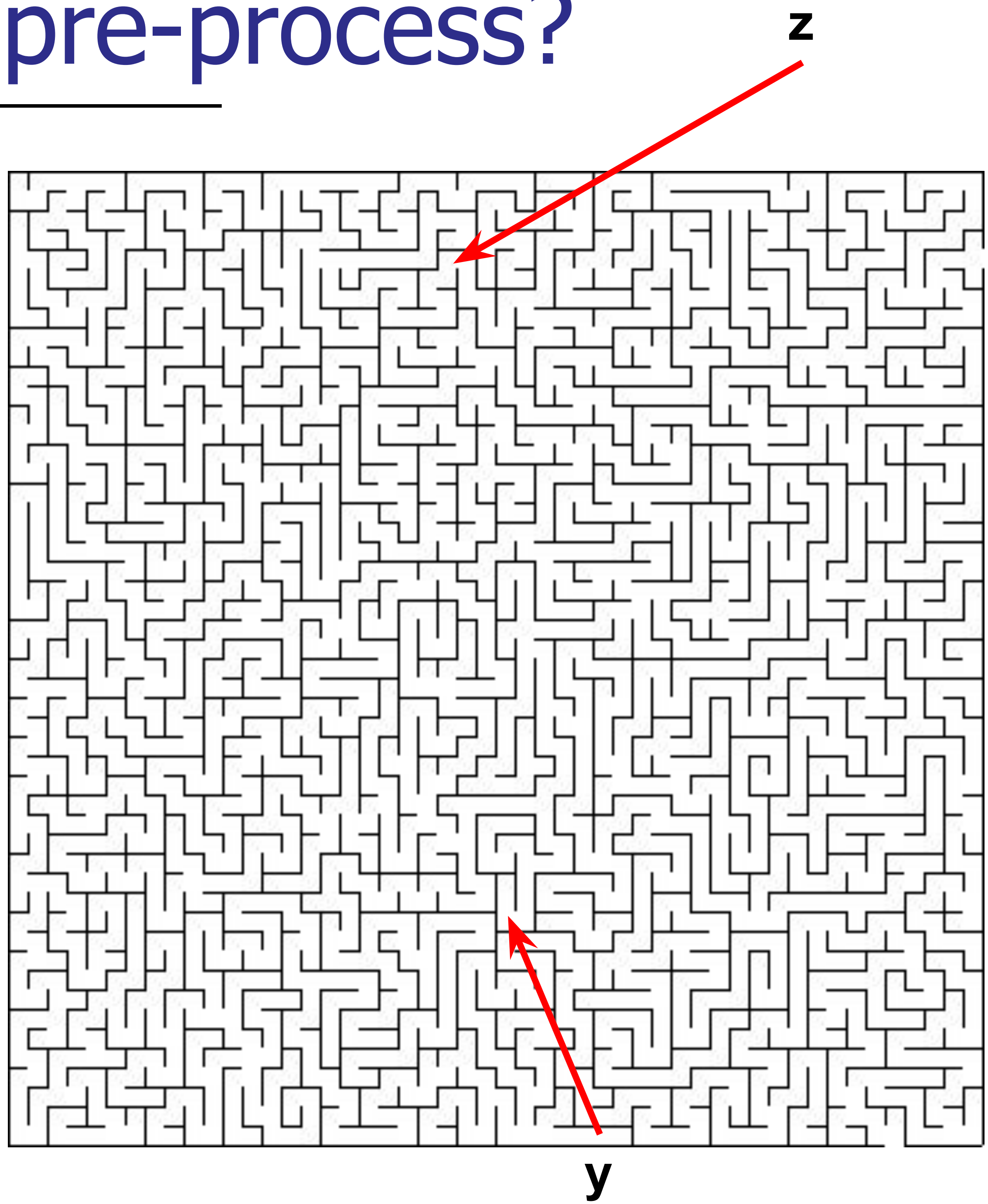
How do you pre-process?

Two steps:

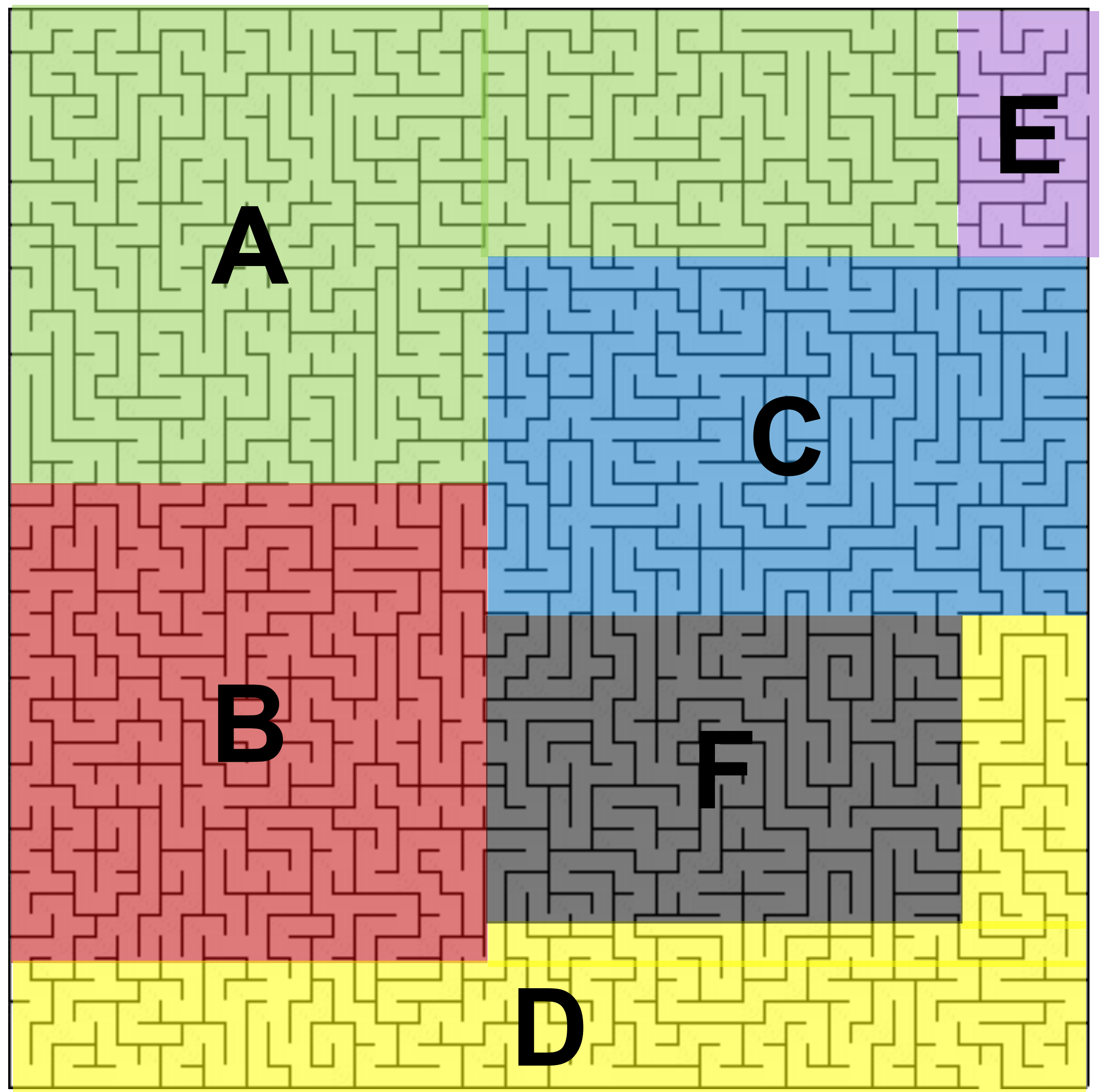
1. Pre-process maze
2. Answer queries

isConnected(y,z) :

Returns true if there is a path from A to B, and false otherwise.



Mazes



Preprocess:

Identify connected components. Label each location with its component number.

isConnected(y,z) :

Returns true if A and B are in the same connected component.

Mazes

Preprocess:

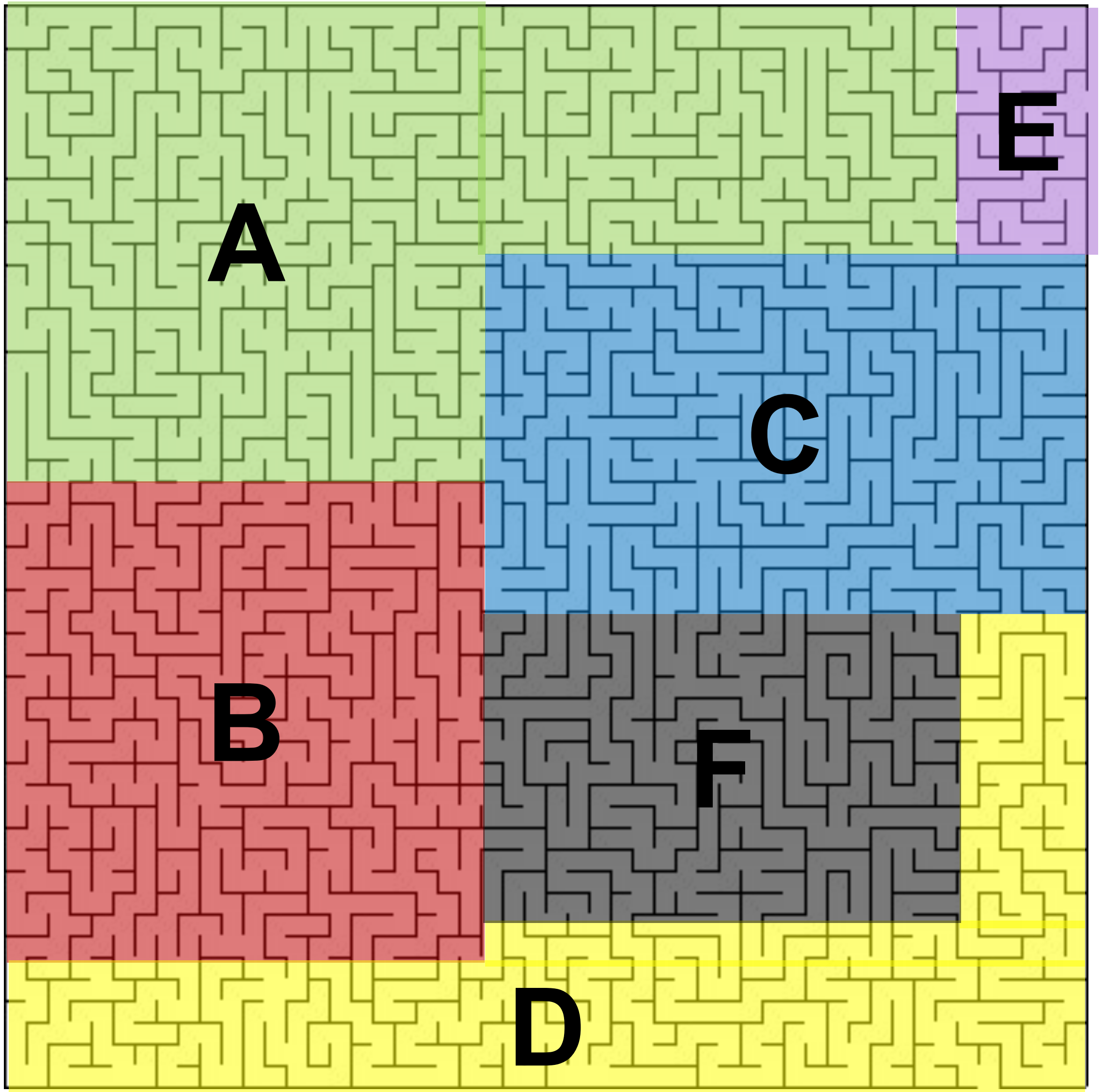
Prepare to answer queries.

destroyWall(x):

Remove walls from the maze using your superpowers.

isConnected(y, z):

Answer connectivity queries.

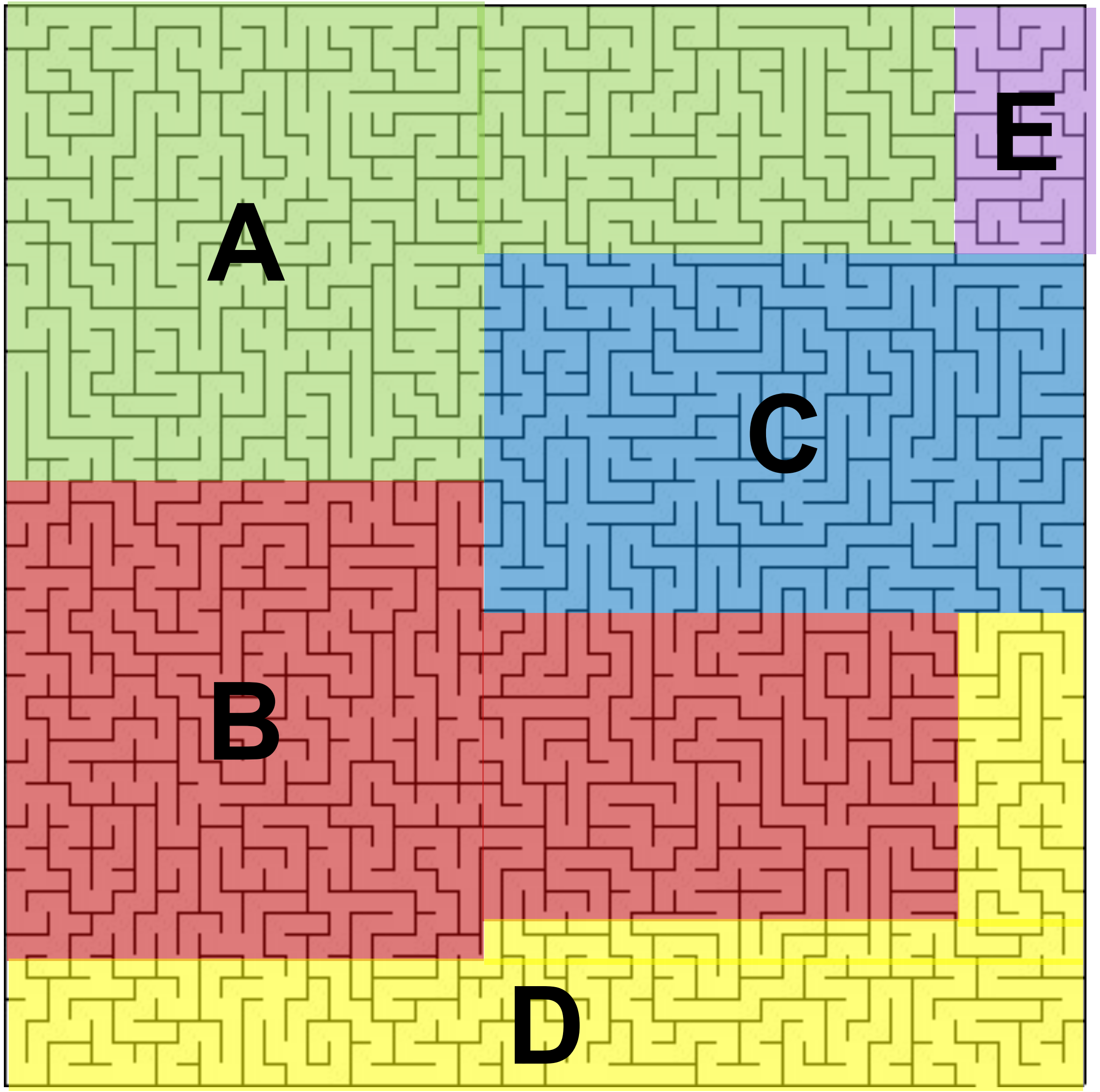


Mazes

Preprocess:
Prepare to answer queries.

destroyWall(x):
Remove walls from the maze using your superpowers.

isConnected(y, z):
Answer connectivity queries.

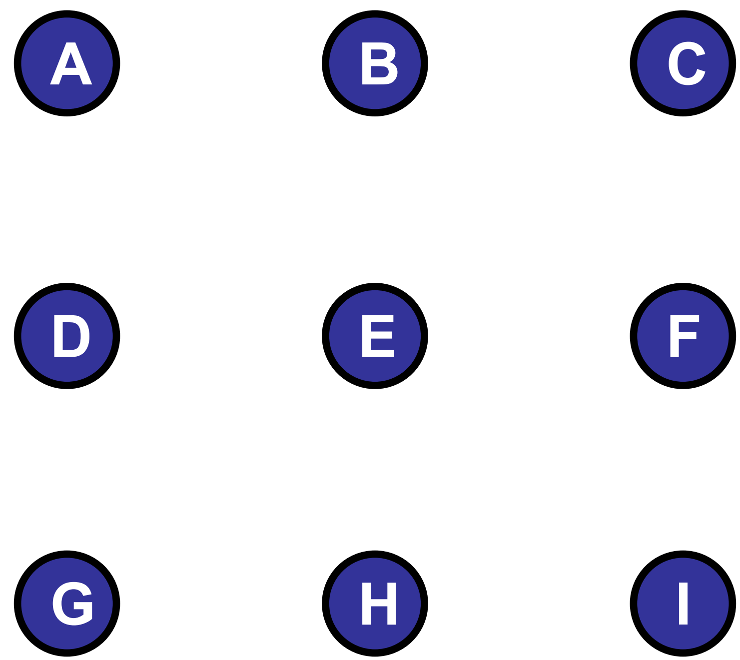


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **???**
find(D, F) = **???**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

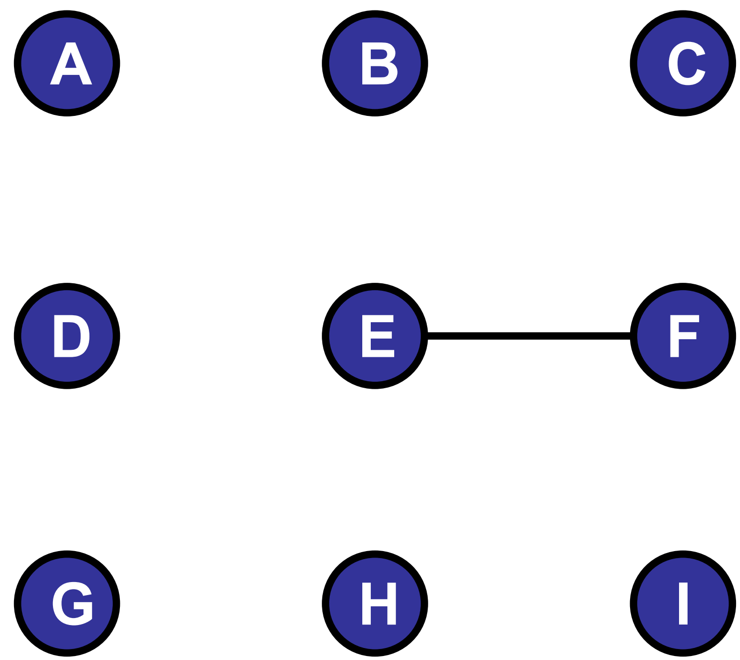


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **???**
find(D, F) = **???**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

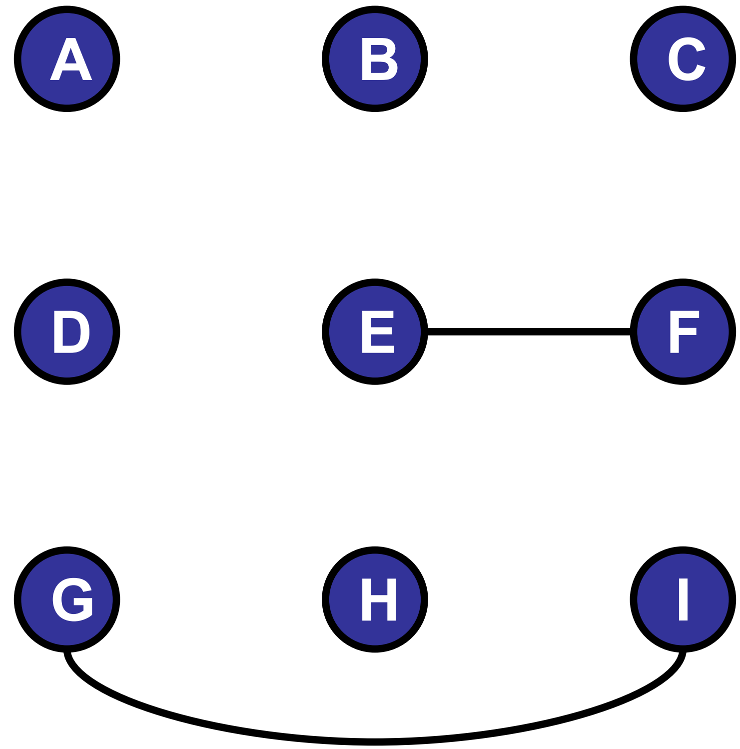


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **???**
find(D, F) = **???**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

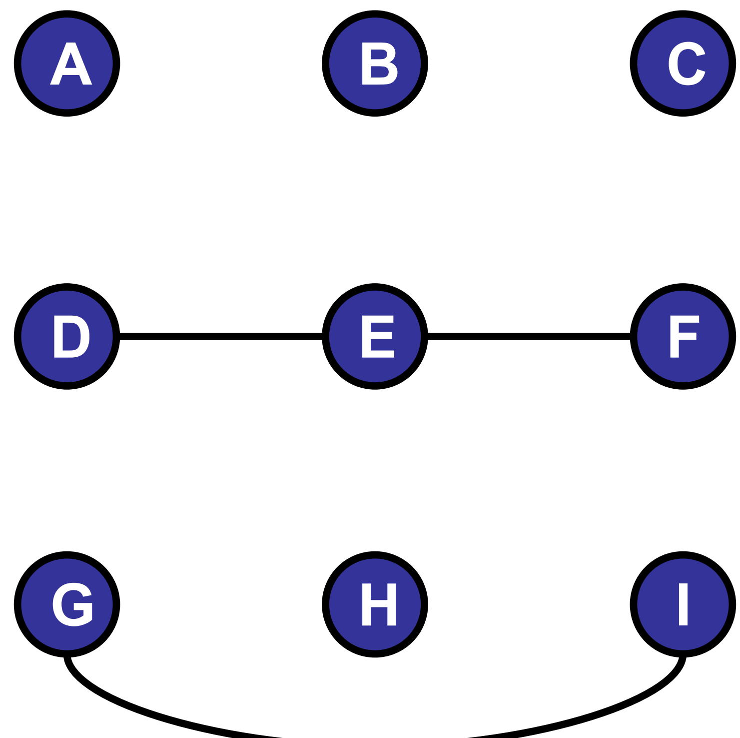


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **???**
find(D, F) = **???**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

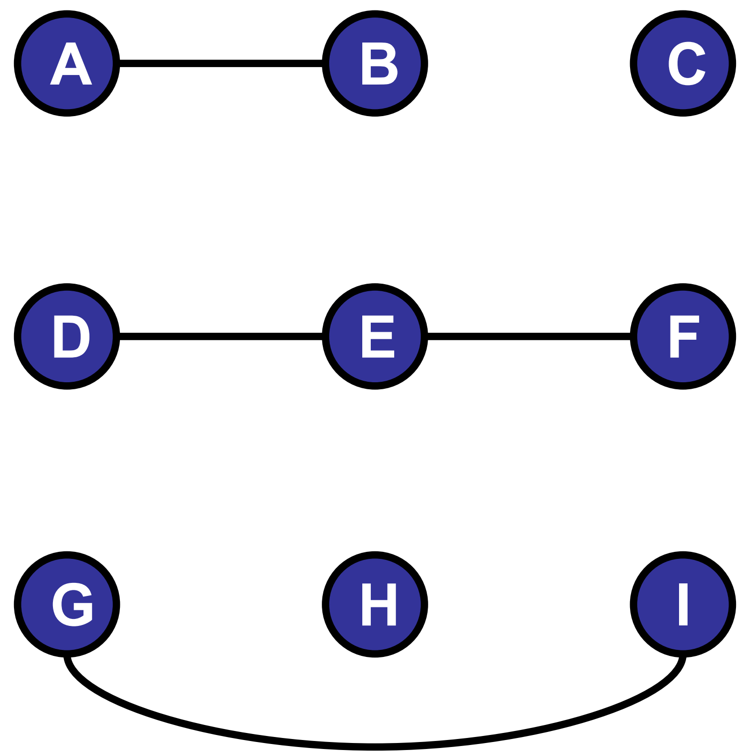


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **???**
find(D, F) = **???**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**



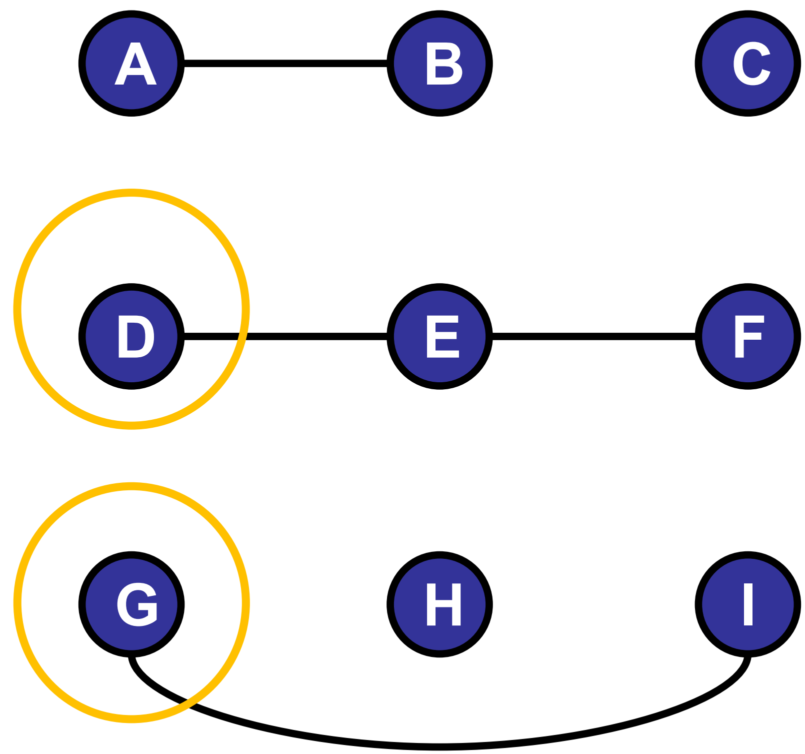
Dynamic Connectivity

Given a set of objects:

– Union: connect two objects

– Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **???**
find(D, F) = **???**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

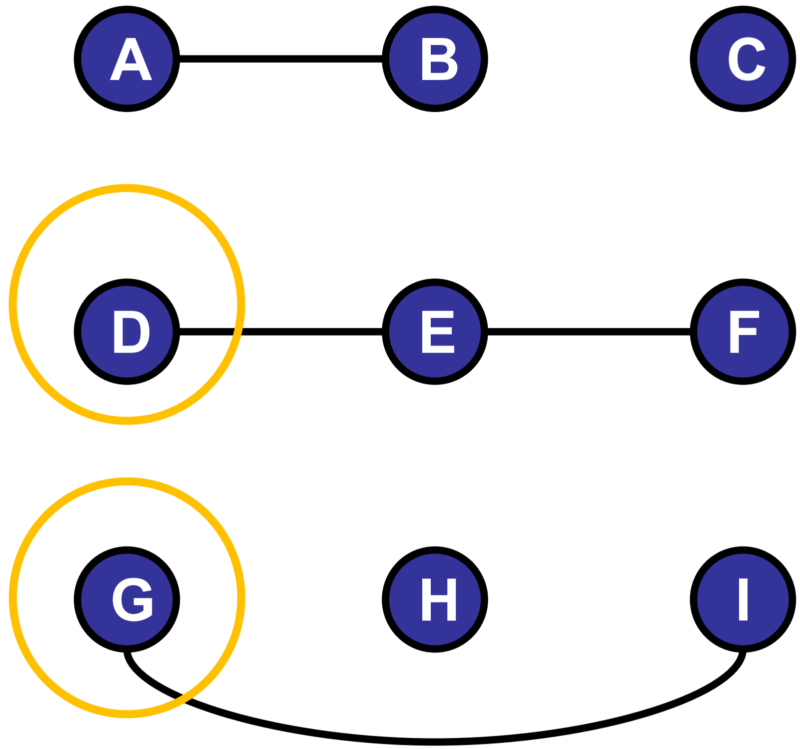


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **???**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

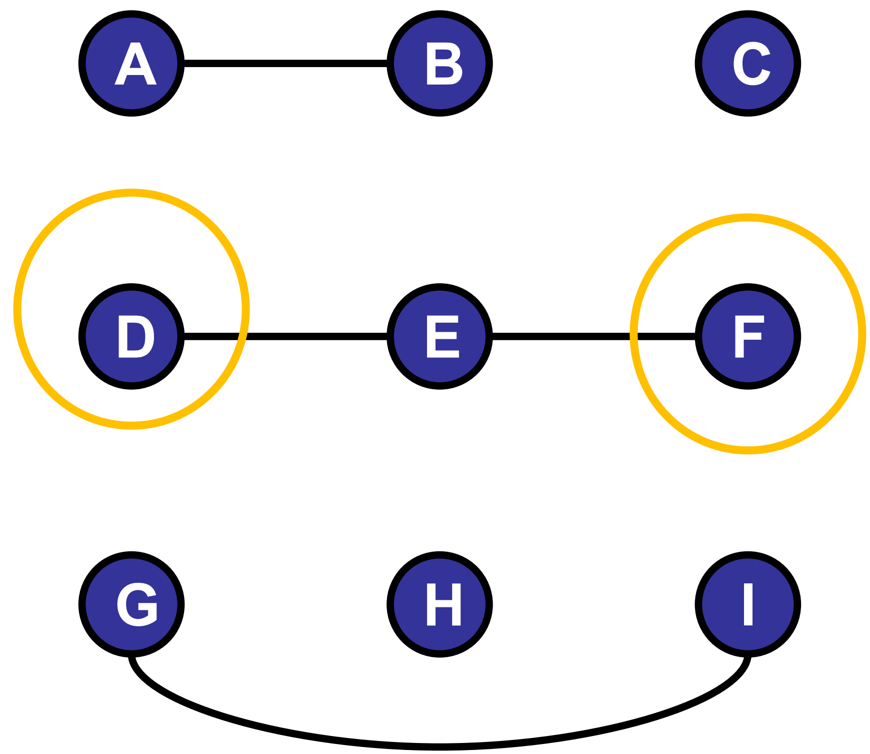


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **???**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

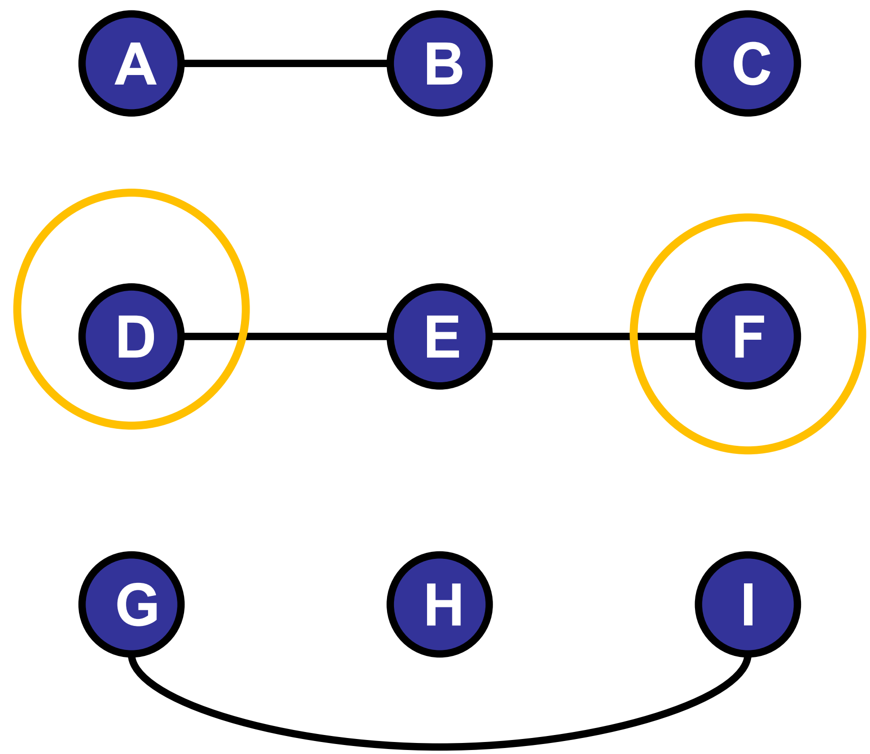


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **true**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

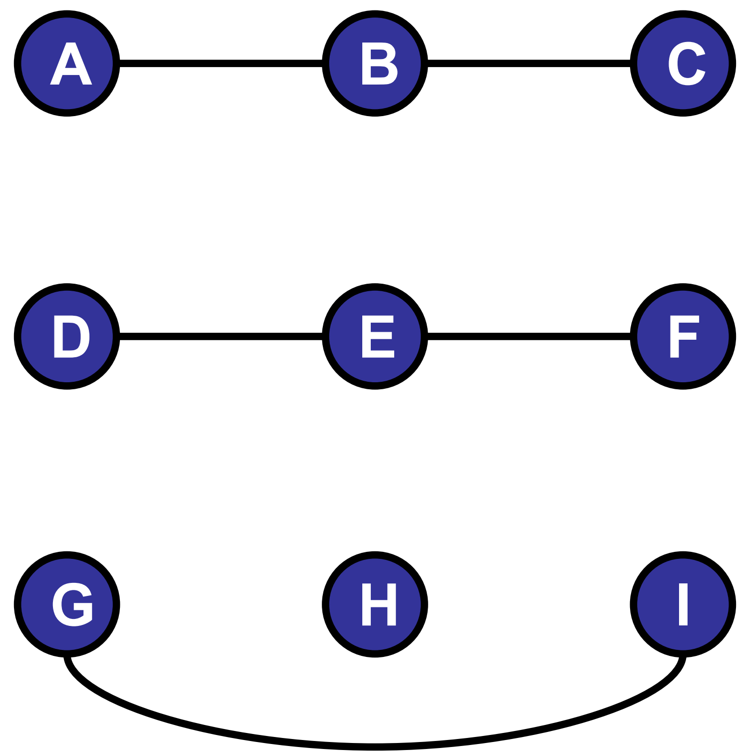


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **true**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

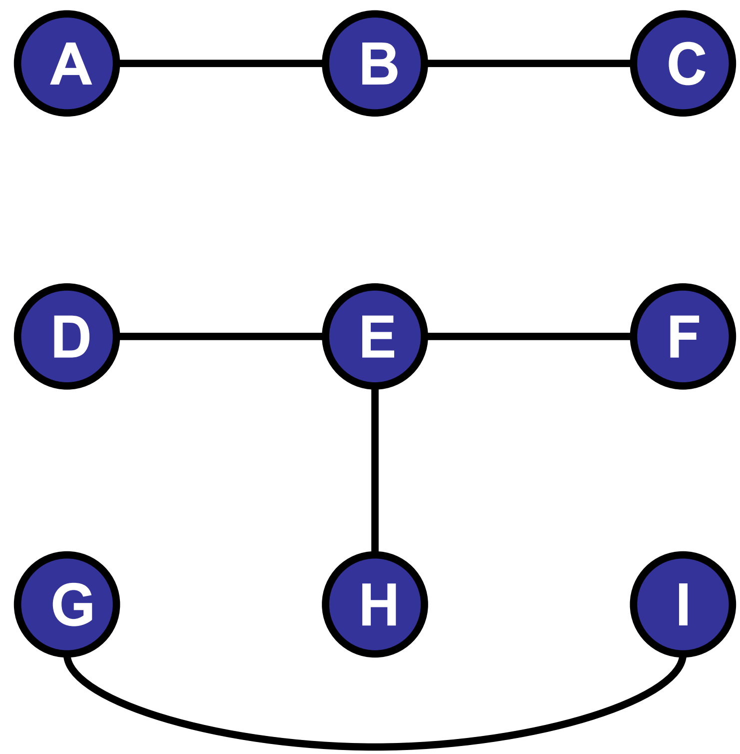


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **true**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

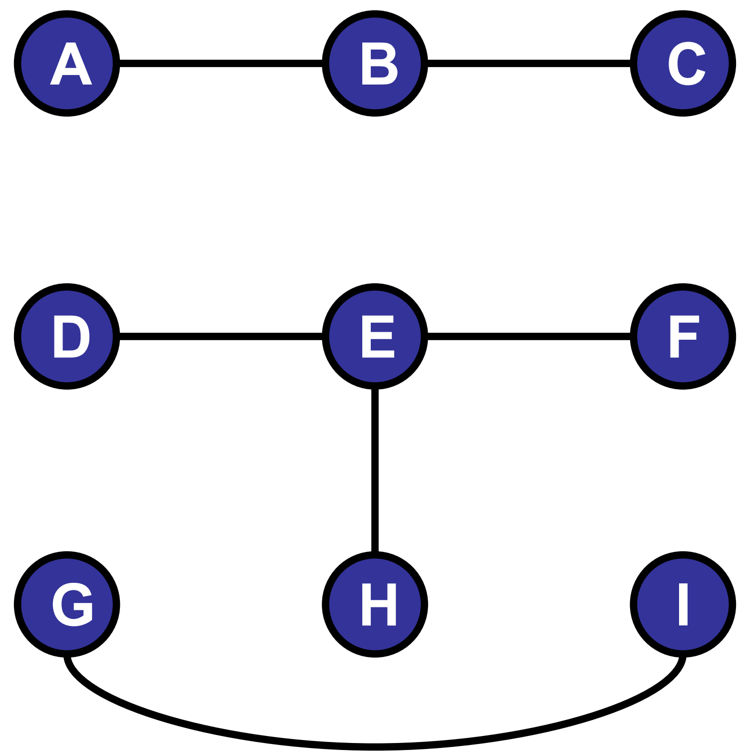


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **true**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

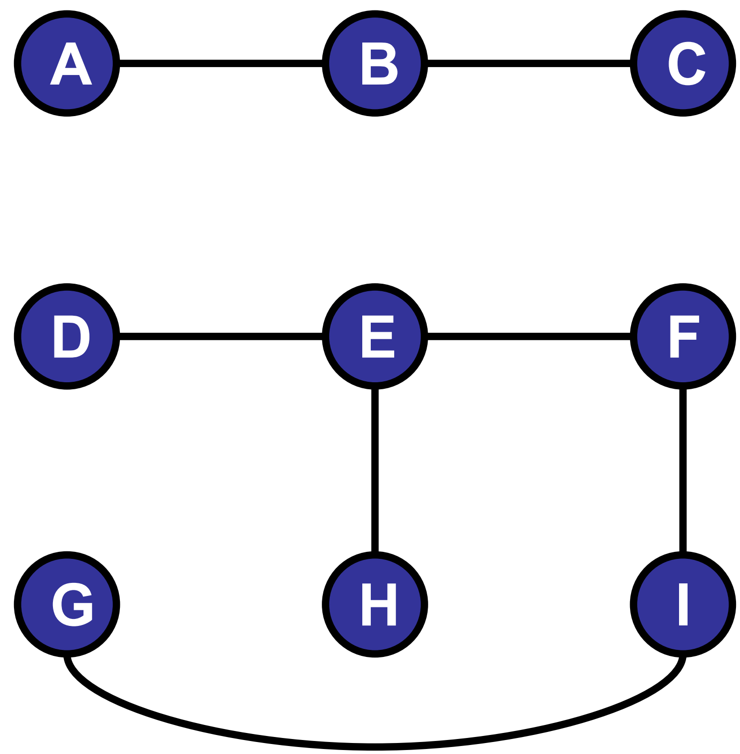


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **true**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

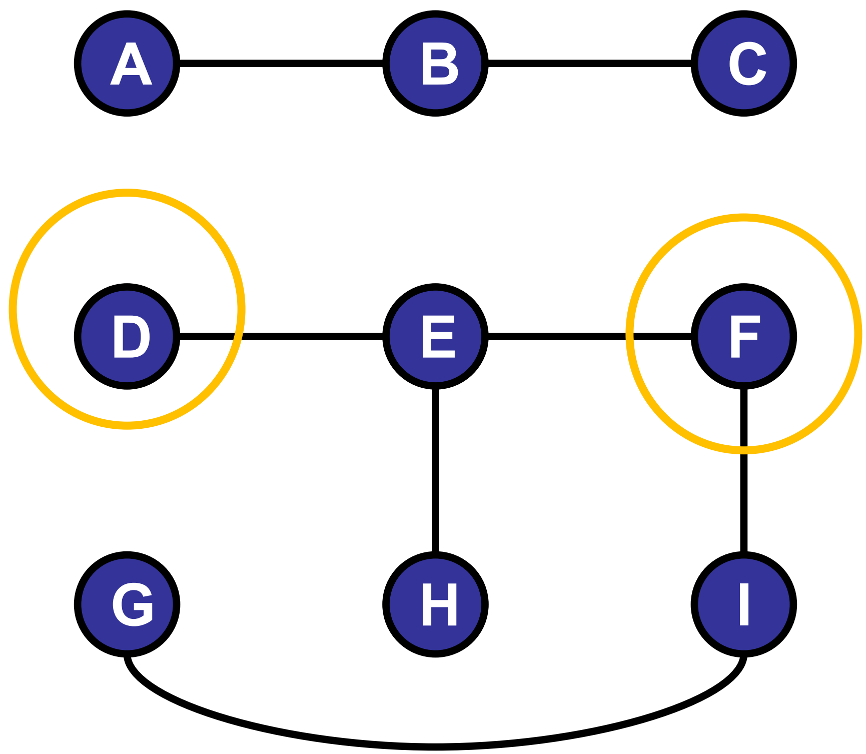


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **true**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **???**

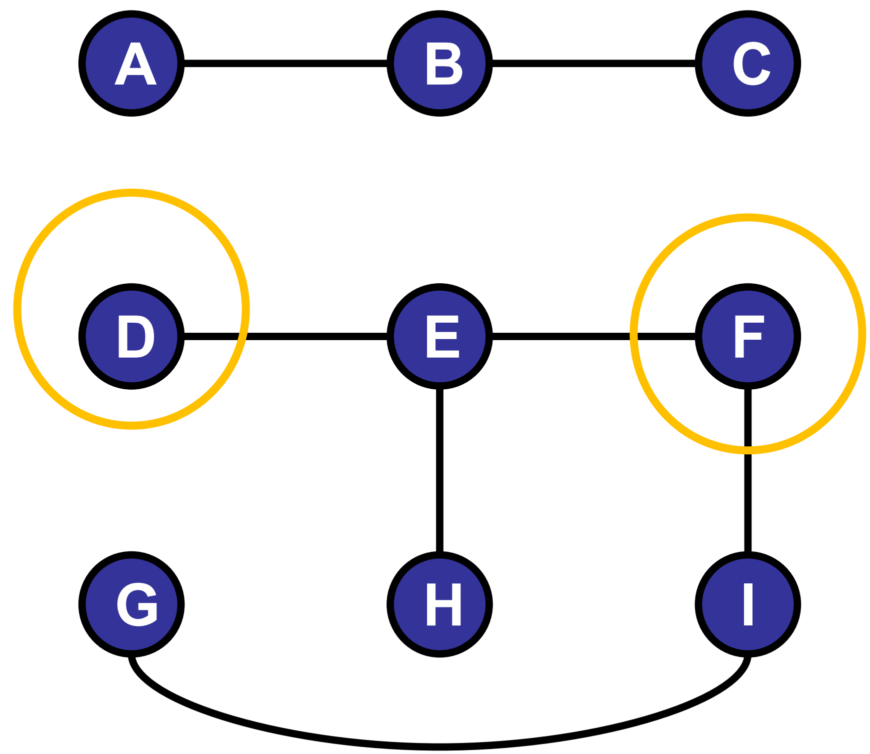


Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

union(E, F)
union(I, G)
union(D, E)
union(B, A)
find(G, D) = **false**
find(D, F) = **true**
union(B, C)
union(H, E)
union(A, C)
union(F, I)
find(G, D) = **true**



Dynamic Connectivity

Given a set of objects:

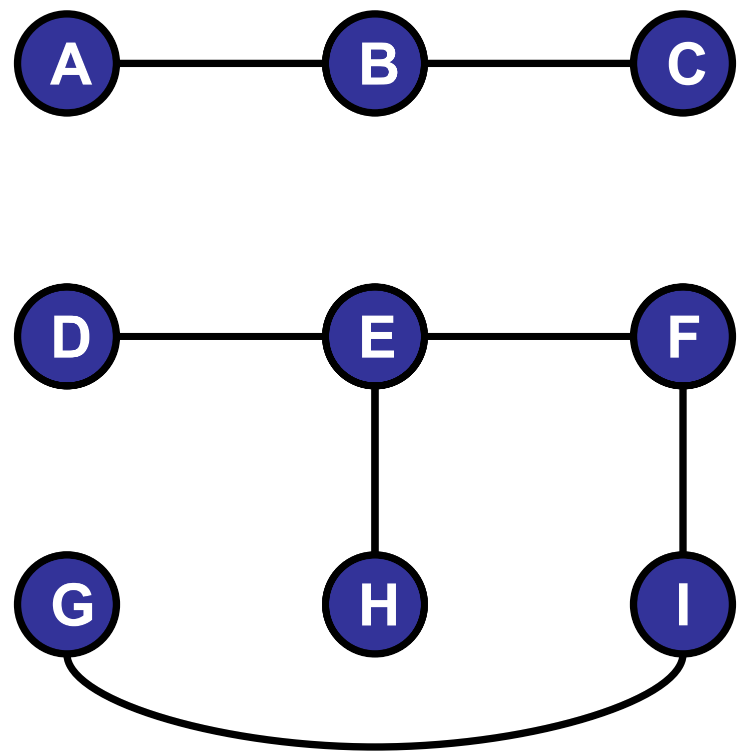
- Union: connect two objects
- Find: is there a path connecting the two objects?

Transitivity

- If **p** is connected to **q** and
  if **q** is connected to **r**,
  then **p** is connected to **r**.

Connected components:

- Maximal set of mutually connected objects.



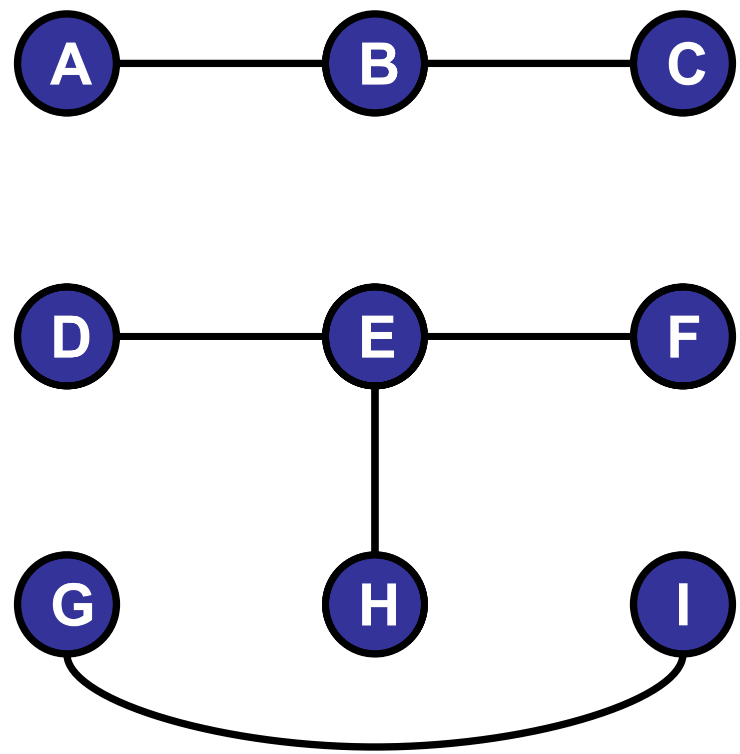
Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

Maintain **disjoint** sets of nodes:

{A, B, C}
{D, E, F, H}
{G, I}



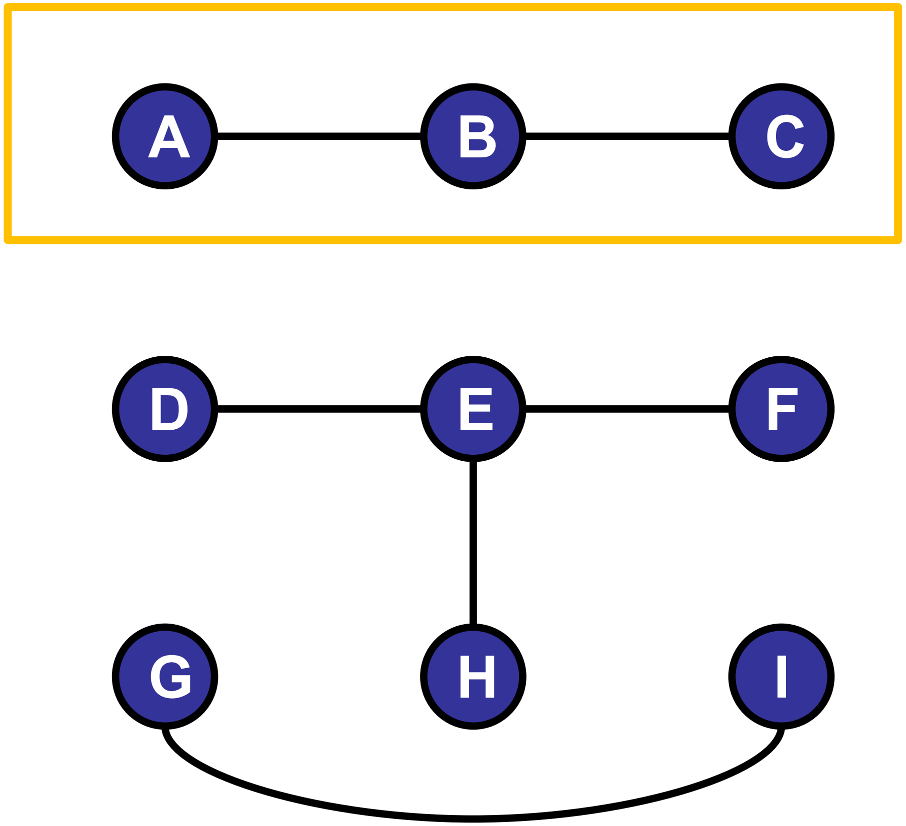
Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

Maintain **disjoint** sets of nodes:

{A, B, C}
{D, E, F, H}
{G, I}



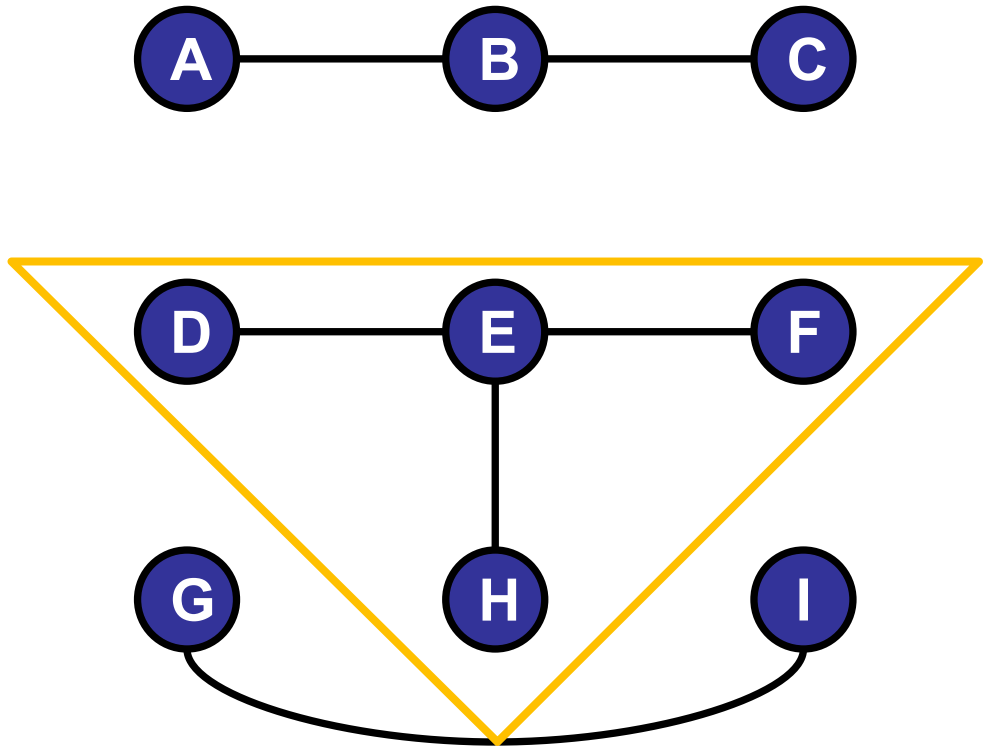
Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

Maintain **disjoint** sets of nodes:

{A, B, C}
{D, E, F, H}
{G, I}



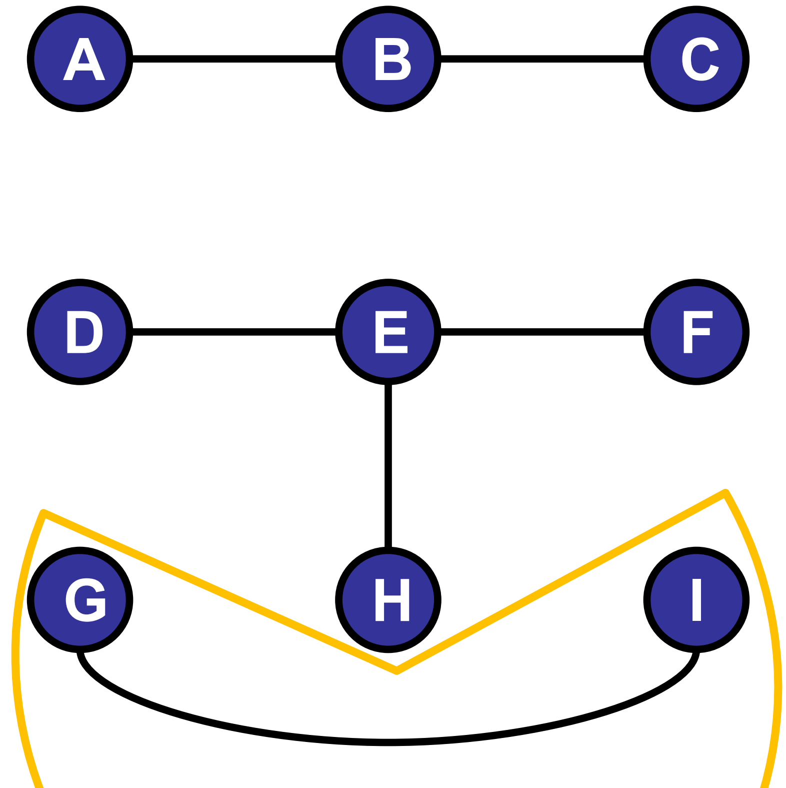
Dynamic Connectivity

Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

Maintain **disjoint** sets of nodes:

{A, B, C}
{D, E, F, H}
{G, I}



Dynamic Connectivity

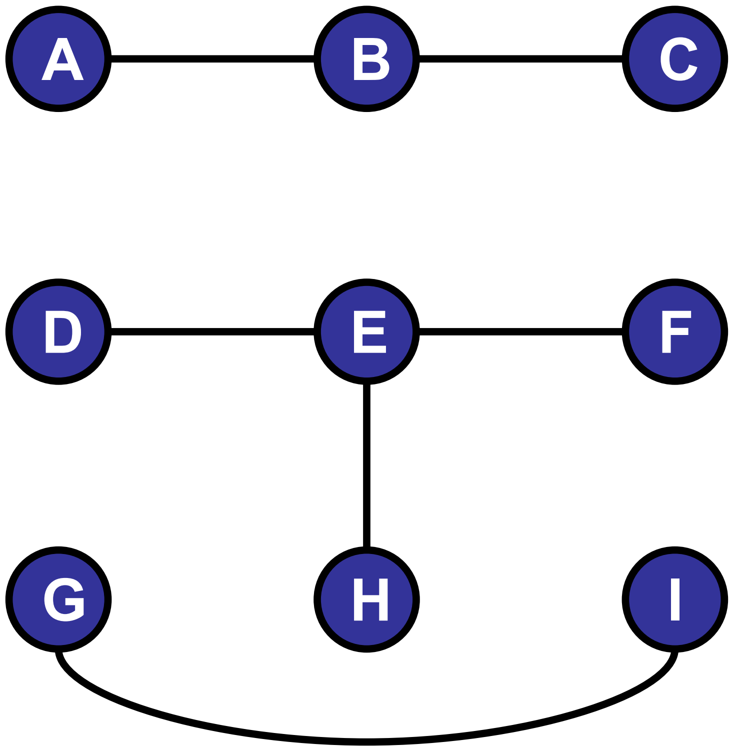
Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

Maintain **disjoint** sets of nodes:

{A, B, C}
{D, E, F, H}
{G, I}

Union



Dynamic Connectivity

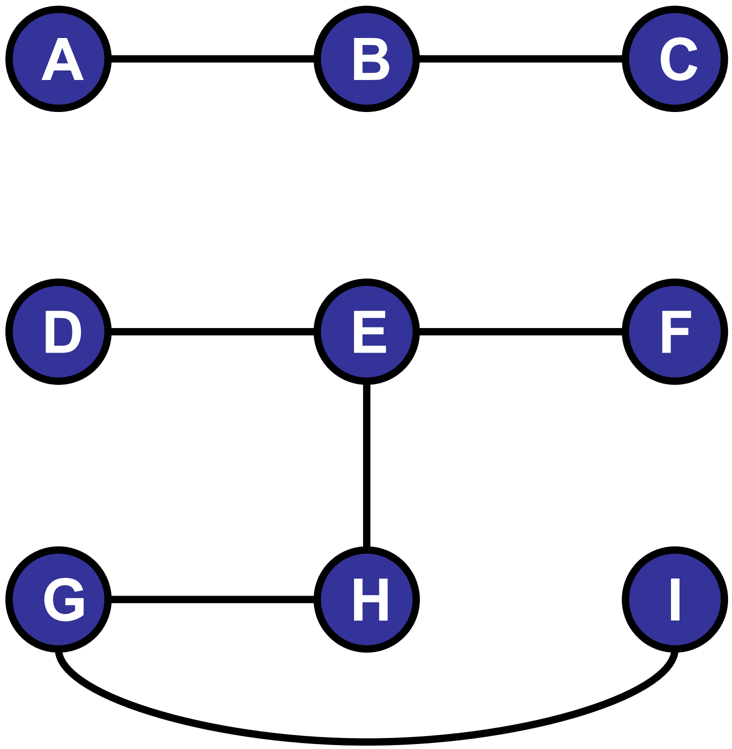
Given a set of objects:

- Union: connect two objects
- Find: is there a path connecting the two objects?

Maintain **disjoint** sets of nodes:

{A, B, C}

{D, E, F, H, G, I}



Abstract Data Type

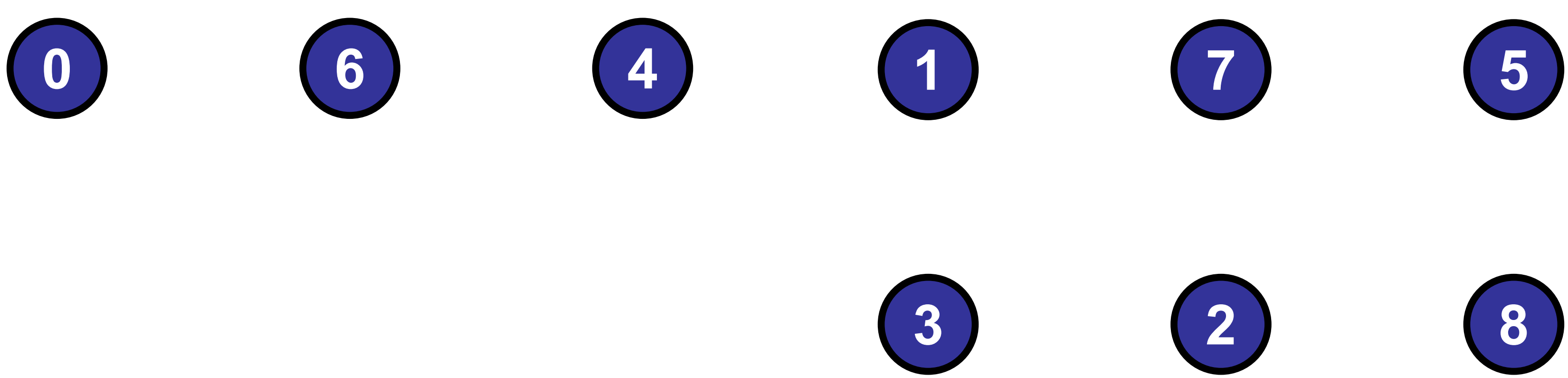
Disjoint Set (Union-Find)

public interface	DisjointSet<Key>	
	DisjointSet(int N)	*constructor: N objects*
boolean	find(Key p, Key q)	*are p and q in the same set?*
void	union(Key p, Key q)	*replace sets containing p and q with their union*

Version 1

`Initial state of data structure:`

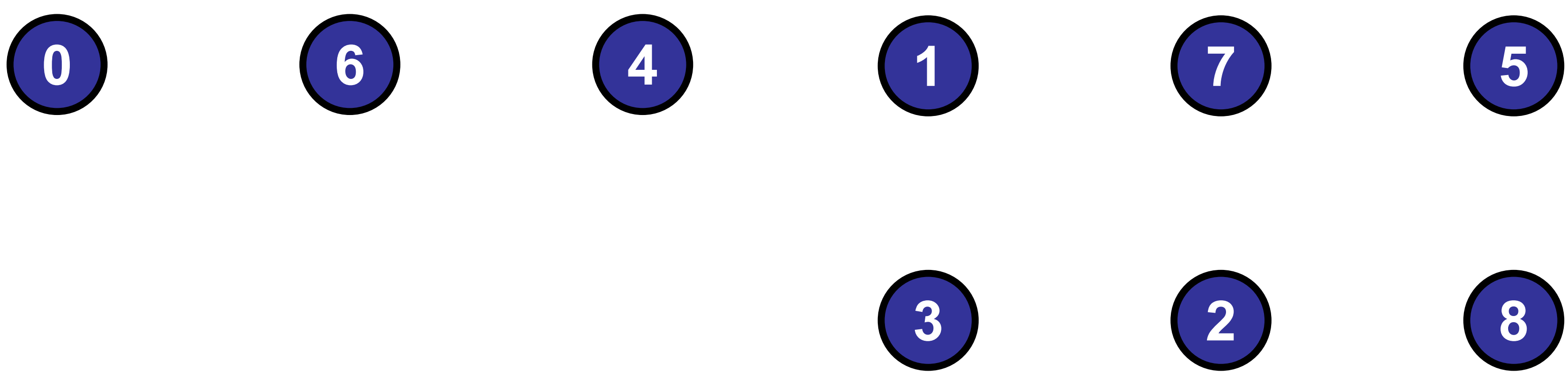
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
component identifier	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**



Version 1

Initially, every object is its own component.

object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	4	5	6	7	8

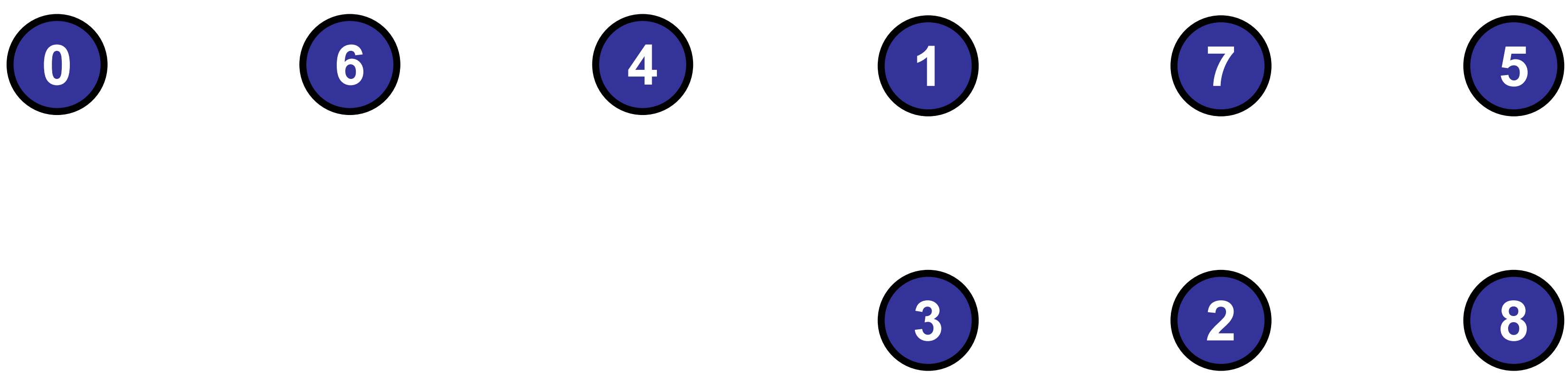


Version 1

Initially, every object is its own component.

Component identifier tells us which component it belongs to.

object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
component identifier	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**

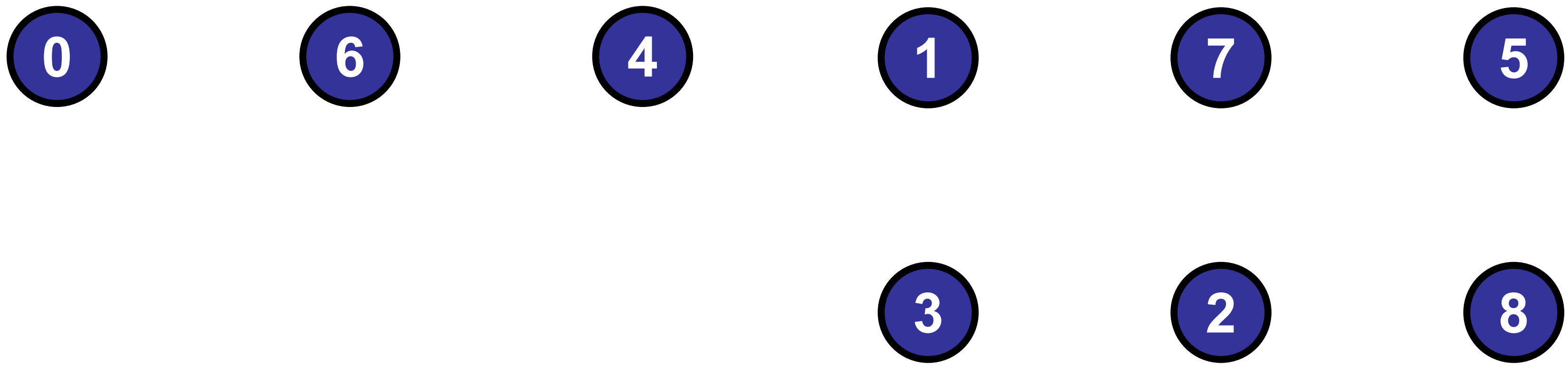


Initial Idea:

How about, to union two components a, b:

Run through all objects, if their identifier is b: set it to a.

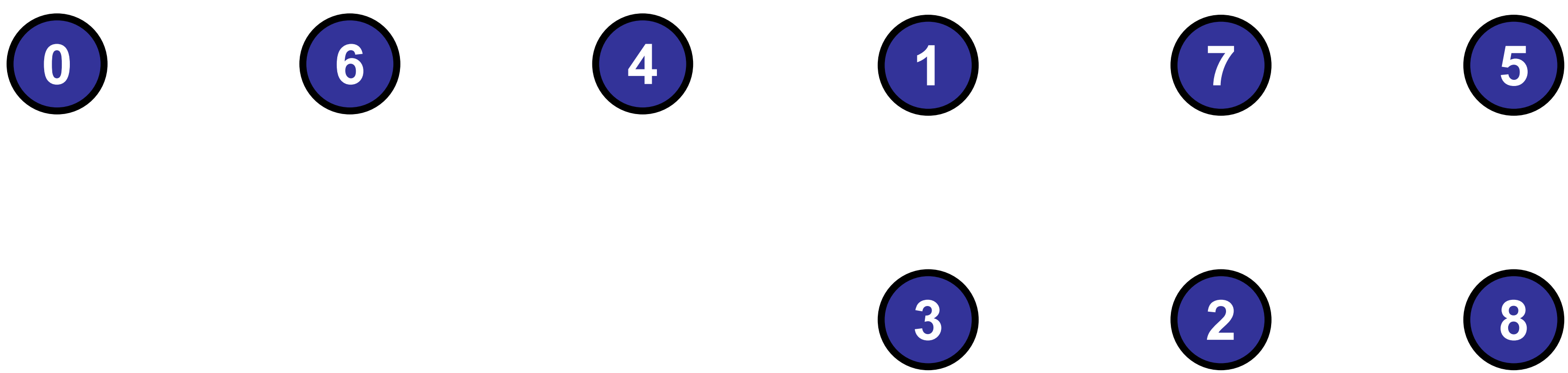
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	4	5	6	7	8



Version 1

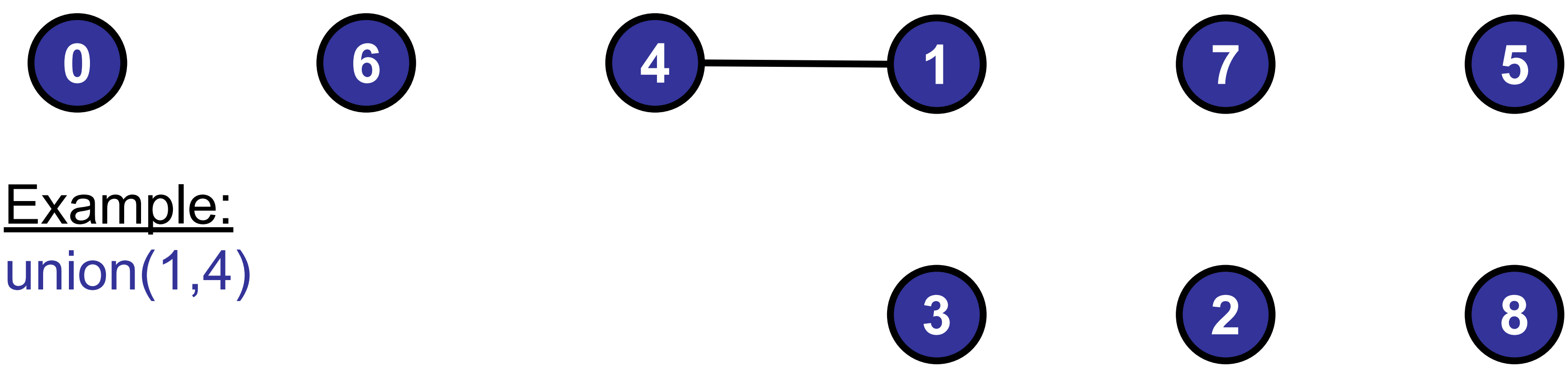
```
union(int p, int q)
   for (int i=0; i<componentId.length; i++)
        if (componentId[i] == componentId[q])
           componentId[i] = componentId[p];
```

object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	4	5	6	7	8



Version 1

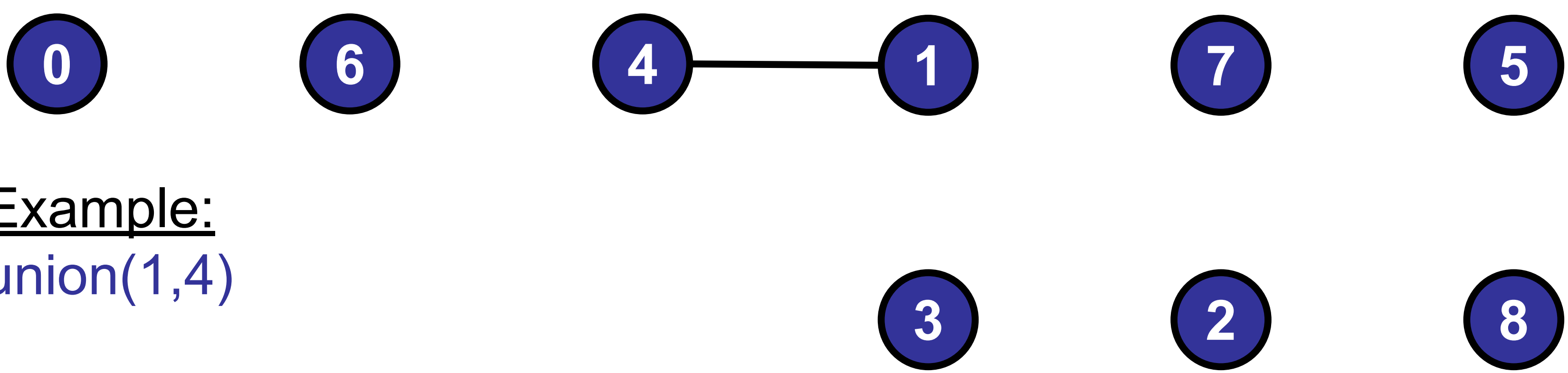
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	4	5	6	7	8



Example:
union(1,4)

Version 1

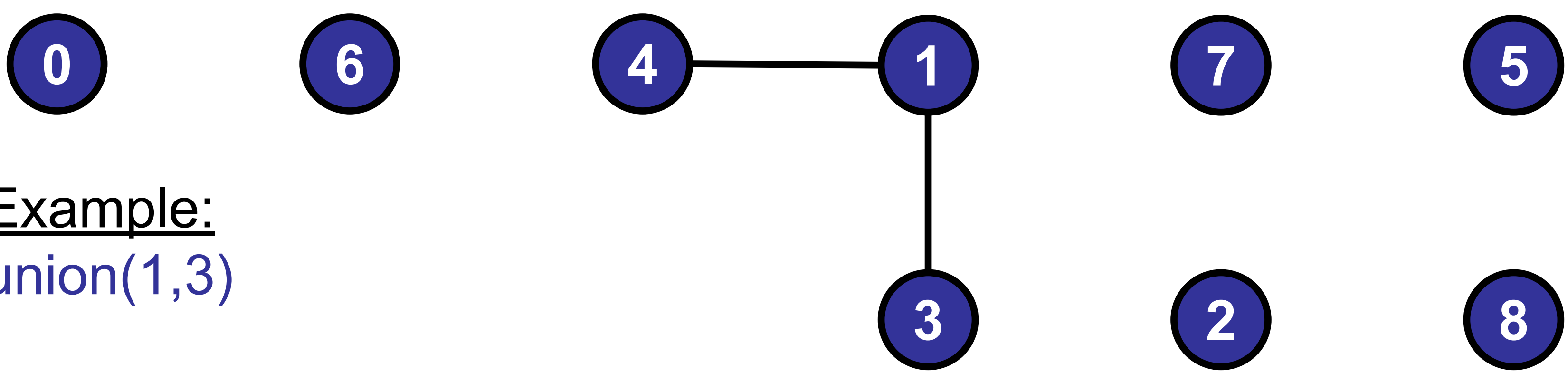
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	1	5	6	7	8



Example:
union(1,4)

Version 1

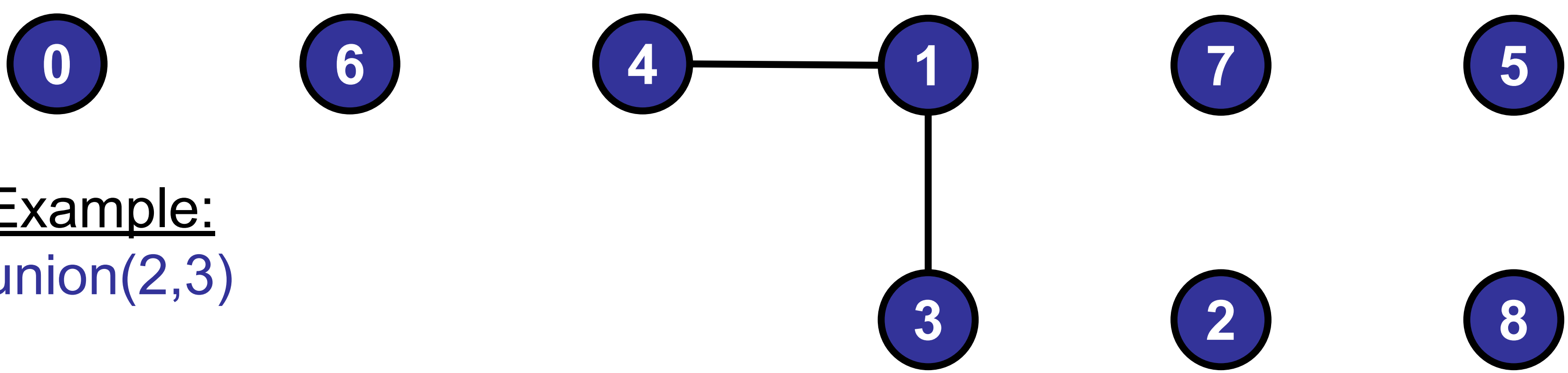
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	1	1	5	6	7	8



Example:
union(1,3)

Version 1

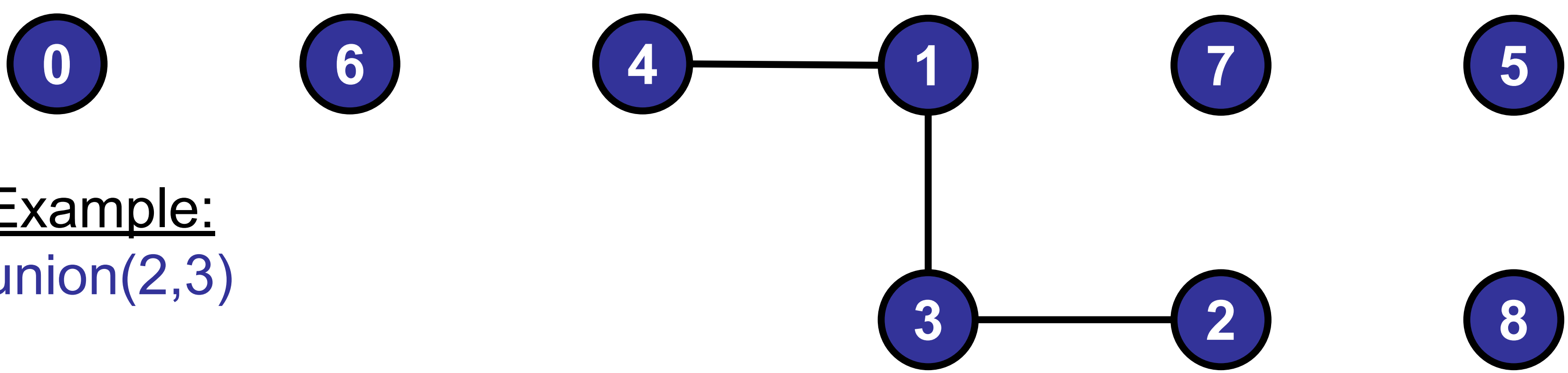
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	1	1	5	6	7	8



Example:
union(2,3)

Version 1

object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	6	7	8

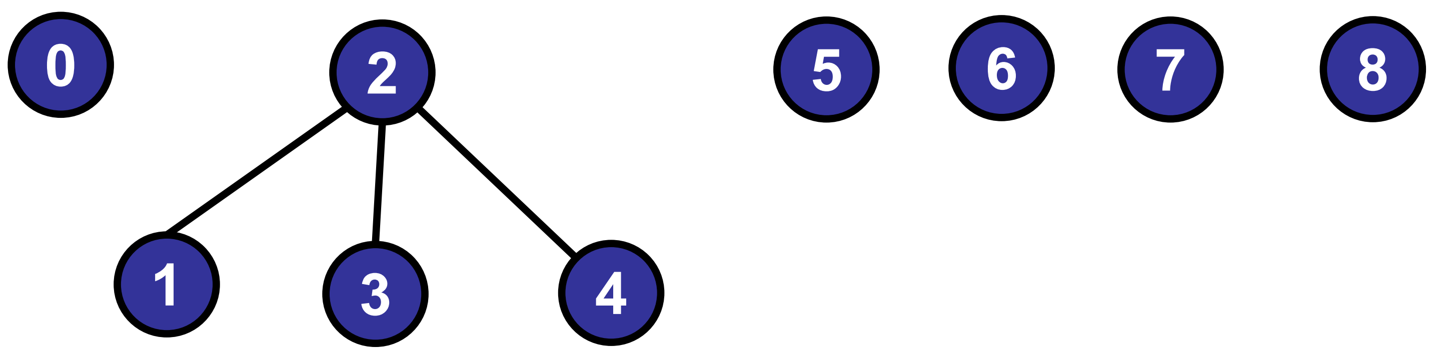


Example:
union(2,3)

Version 1

`Flat trees:`

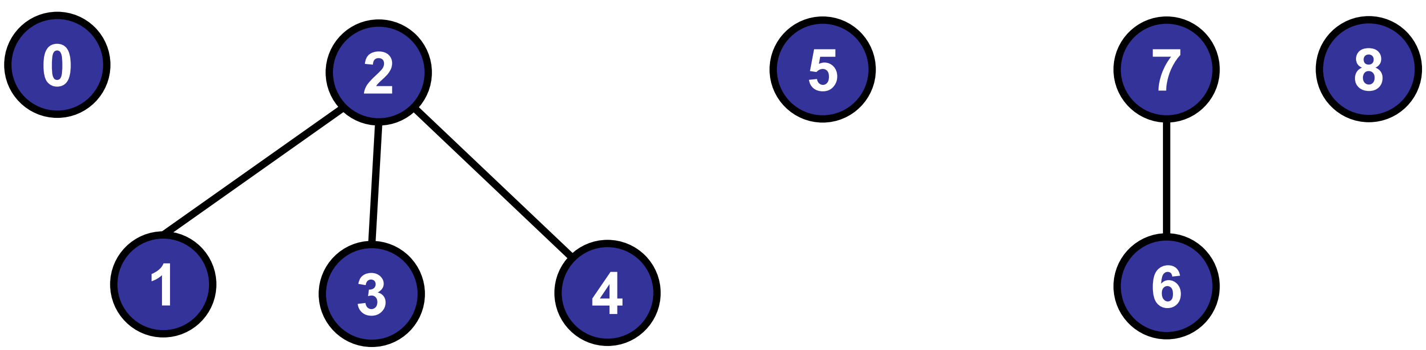
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	6	7	8



Version 1

Flat trees:

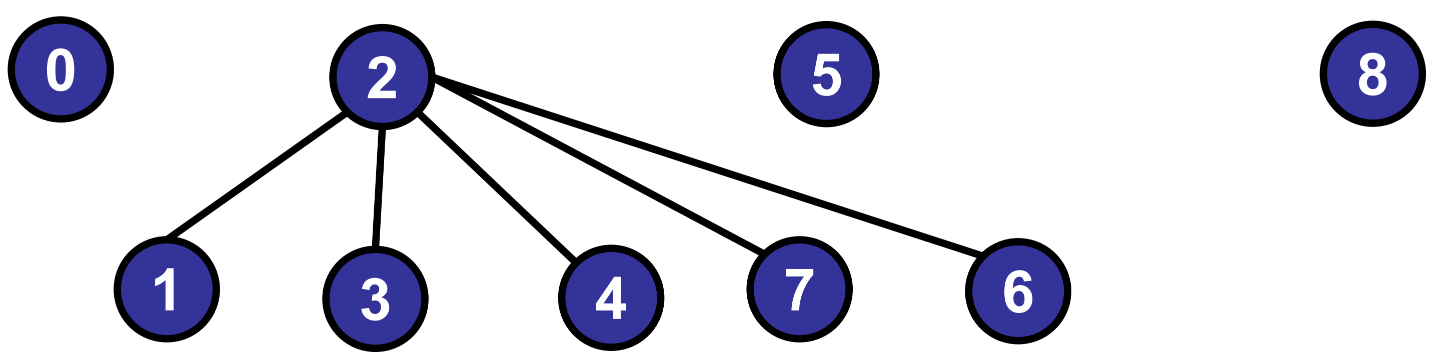
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	7	7	8



Version 1

Flat trees:

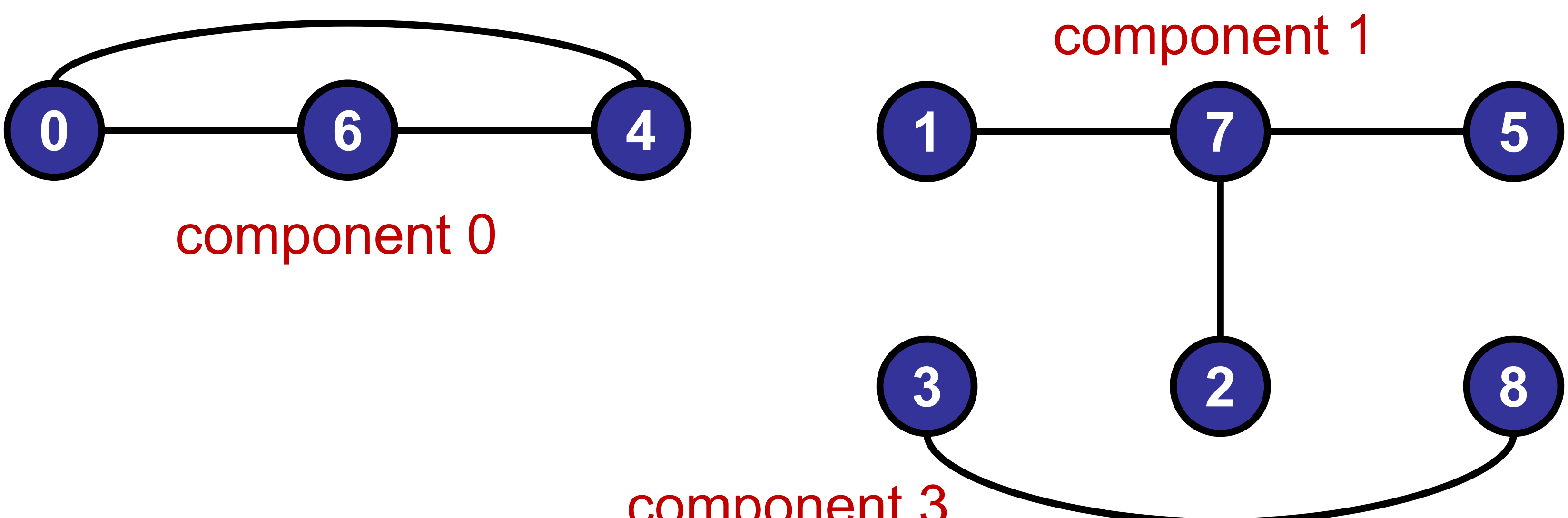
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	2	2	8



Version 1

```
find(int p, int q)
  return(componentId[p] == componentId[q]);
```

object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



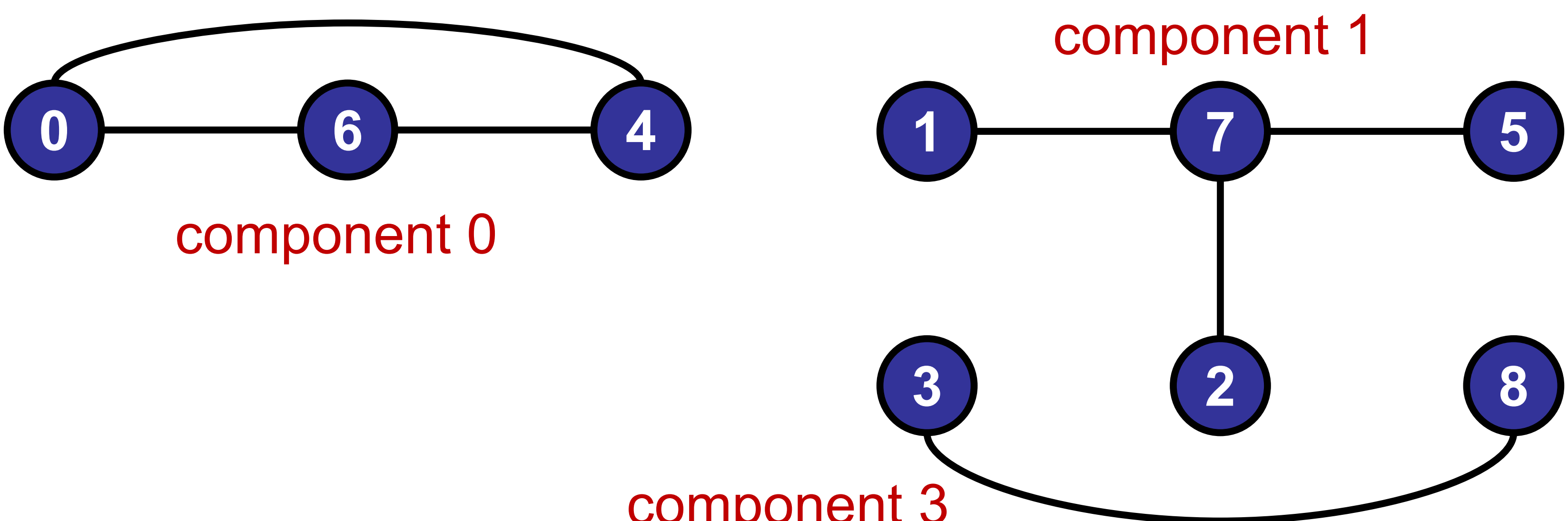
Running time of (Find, Union):

1. O(1), O(1)
2. O(1), O(n)
3. O(n), O(1)
4. O(n), O(n)
5. O(log n), O(log n)
6. None of the above.

Doing Better:

Union takes too long. Reason being that we are too aggressively updating the component identifier.

object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3

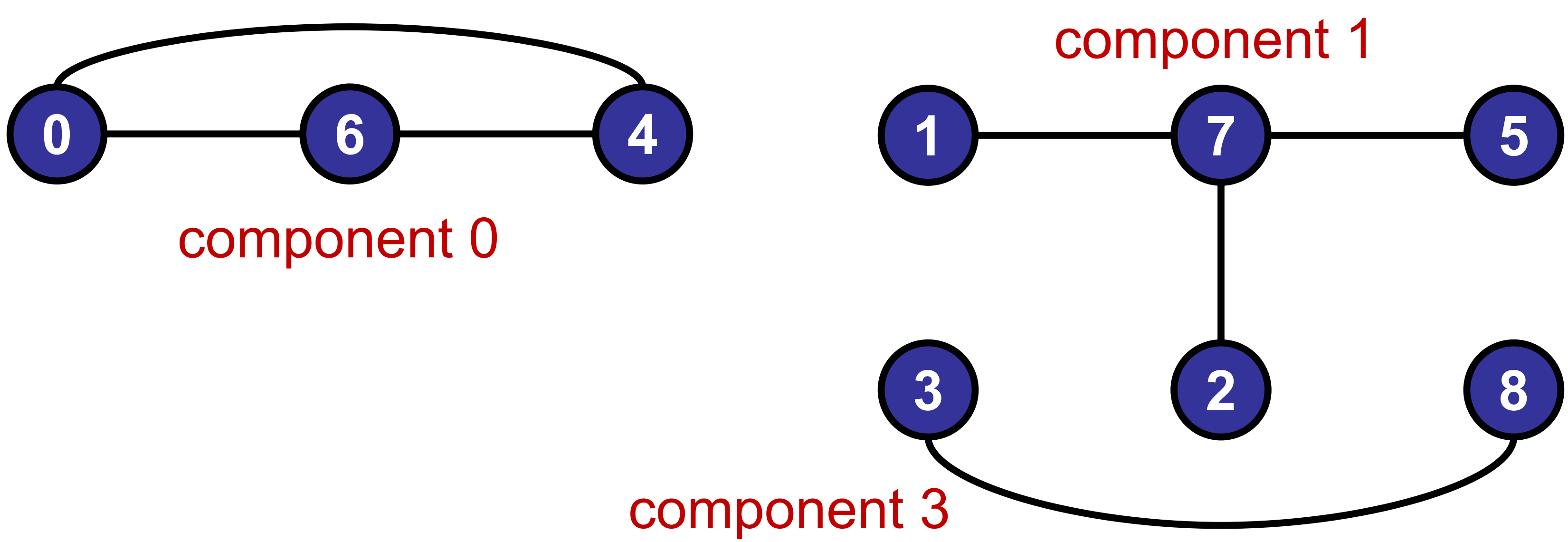


Doing Better:

Union takes too long. Reason being that we are too aggressively updating the component identifier.

Let's try to see what happens if we were a little lazier.

object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
component identifier	**0**	**1**	**1**	**3**	**0**	**1**	**0**	**1**	**3**

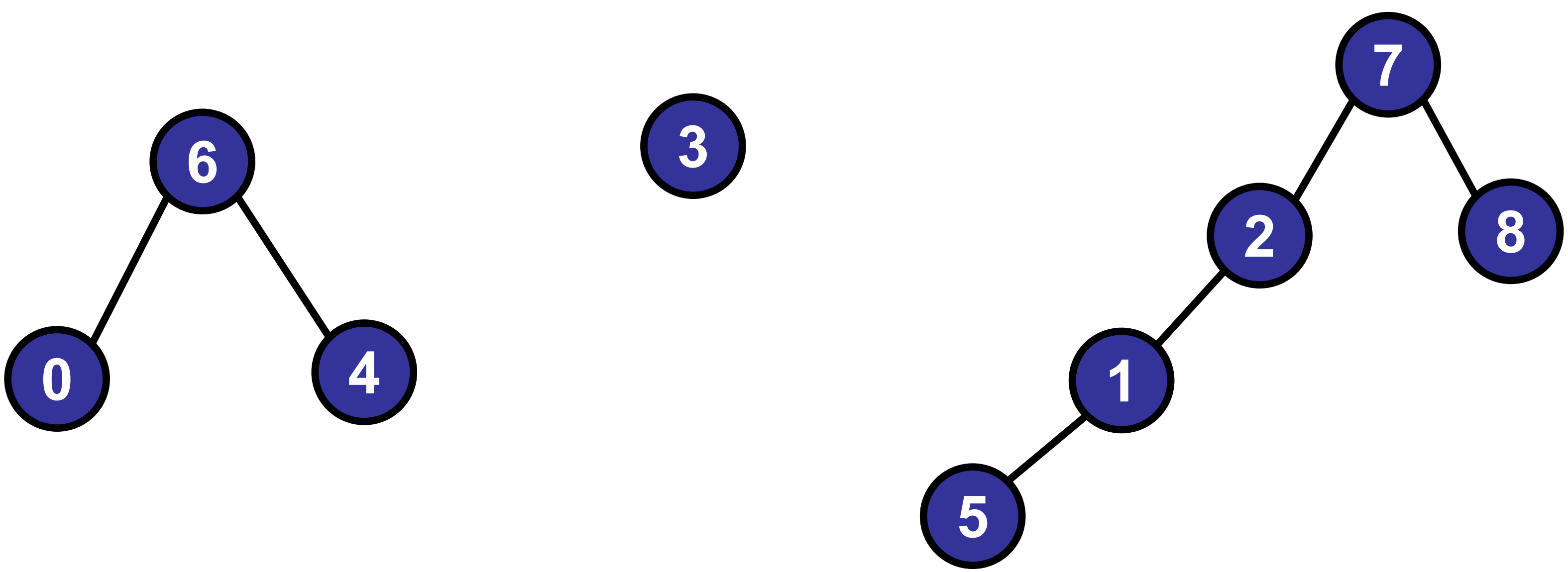


Using Parent Pointers Instead

Data structure:

- Integer array: int[] parent
- Two objects are connected if they are part of the same tree.

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7

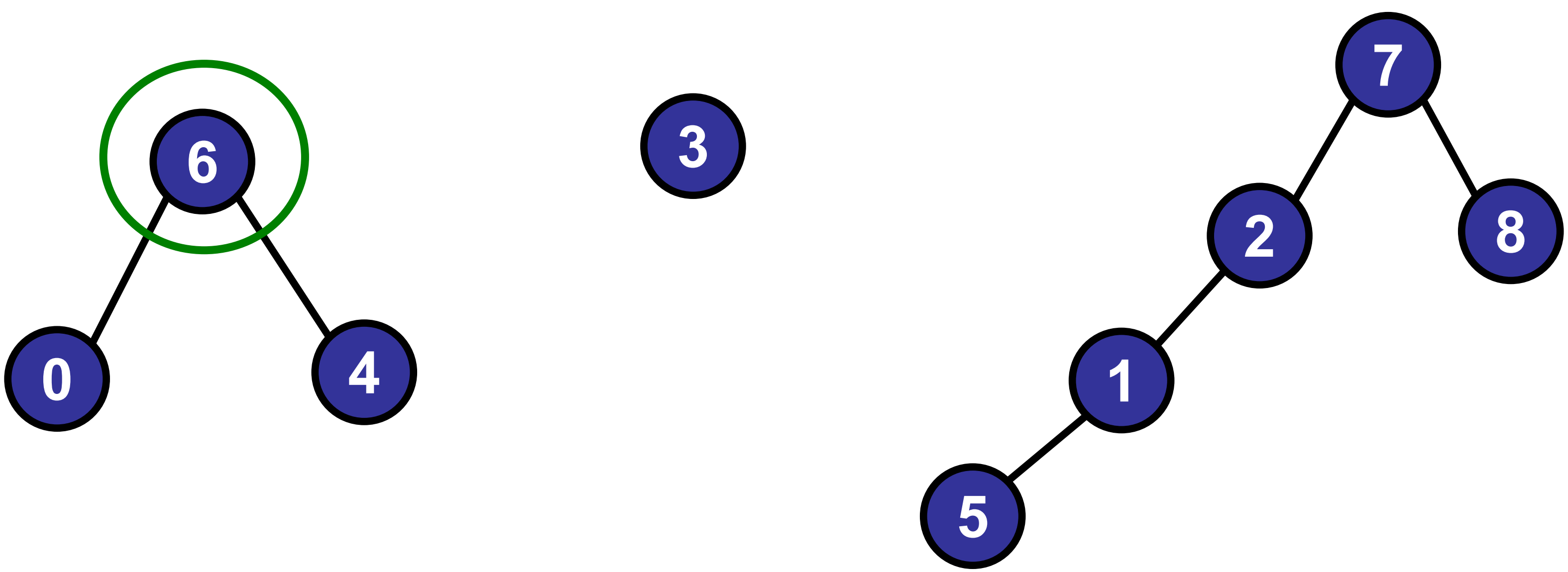


Ver 2

Data structure:

- Integer array: int[] parent
- Two objects are connected if they are part of the same tree.

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7

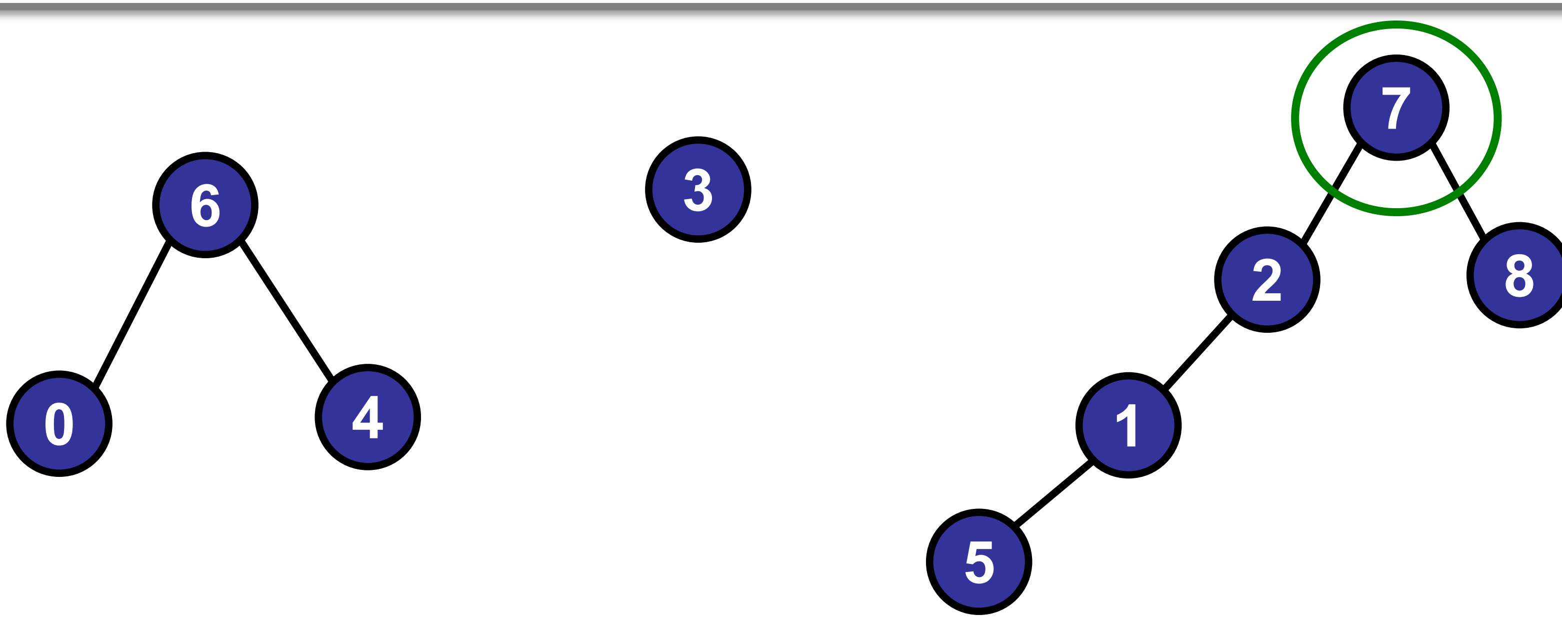


Ver 2

Data structure:

- Integer array: int[] parent
- Two objects are connected if they are part of the same tree.

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7

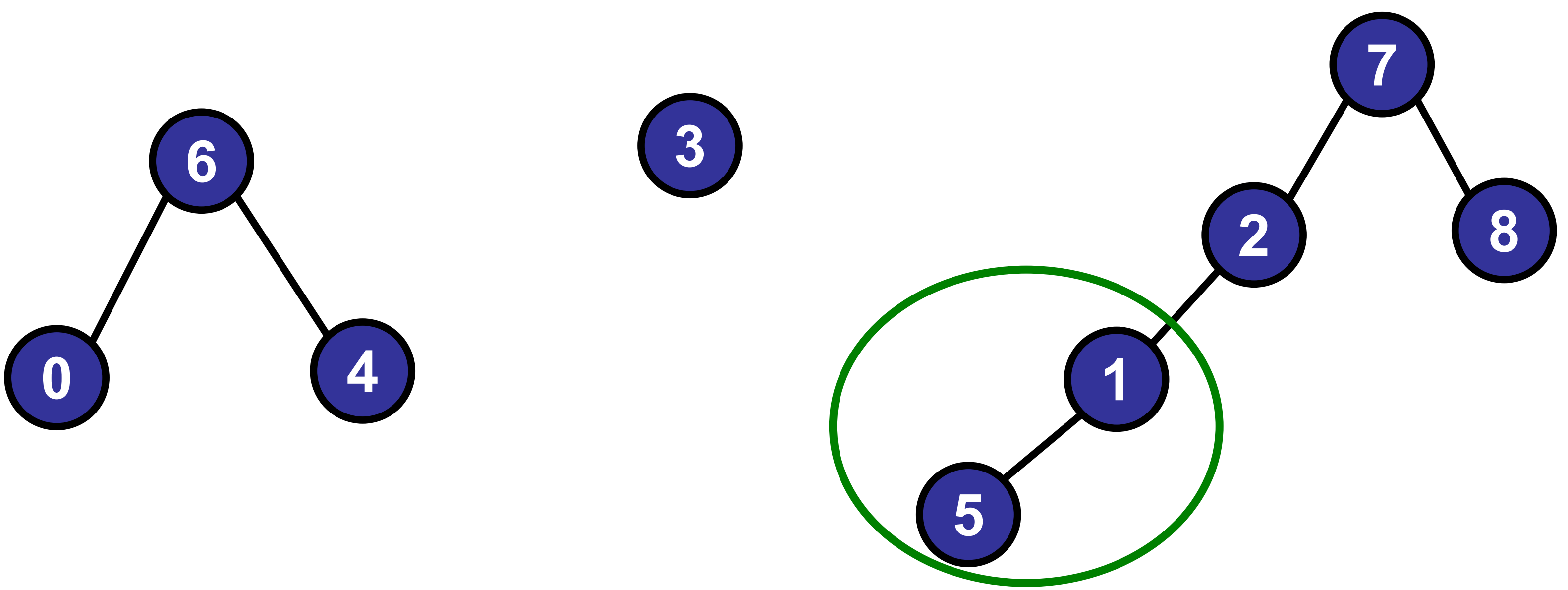


Ver 2

Data structure:

- Integer array: int[] parent
- Two objects are connected if they are part of the same tree.

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7

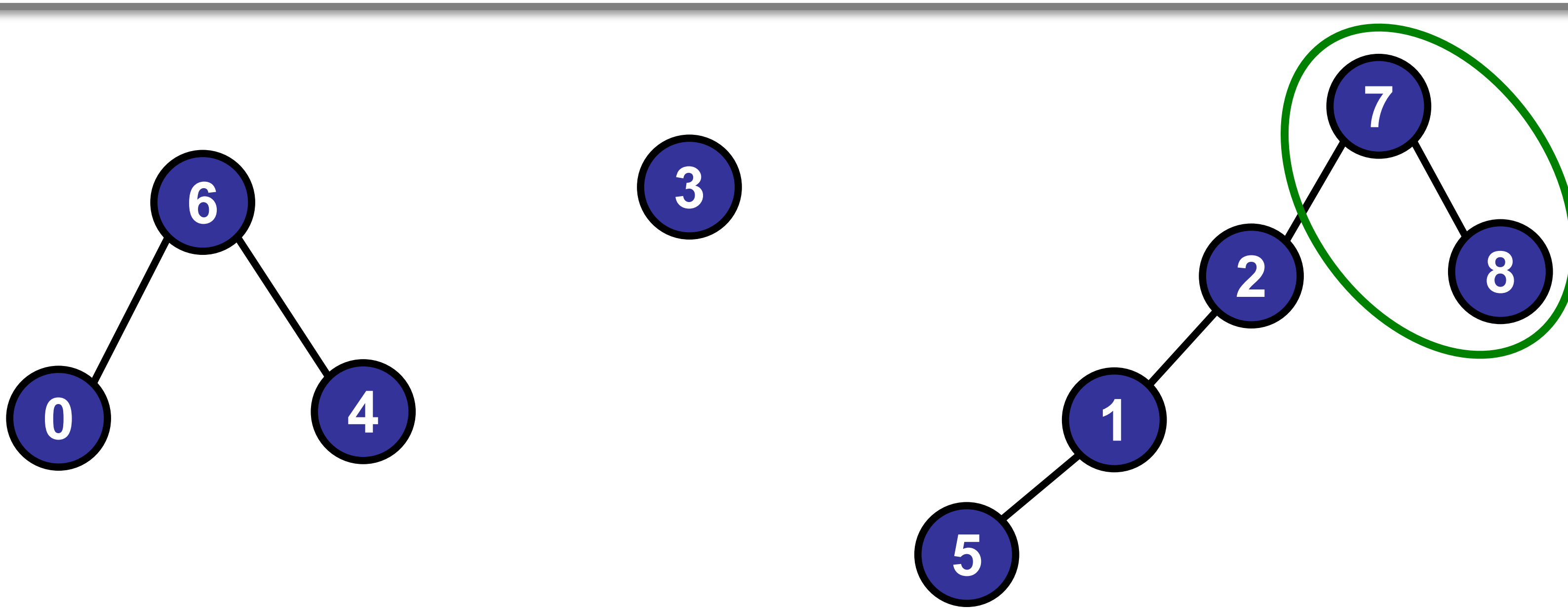


Ver 2

Data structure:

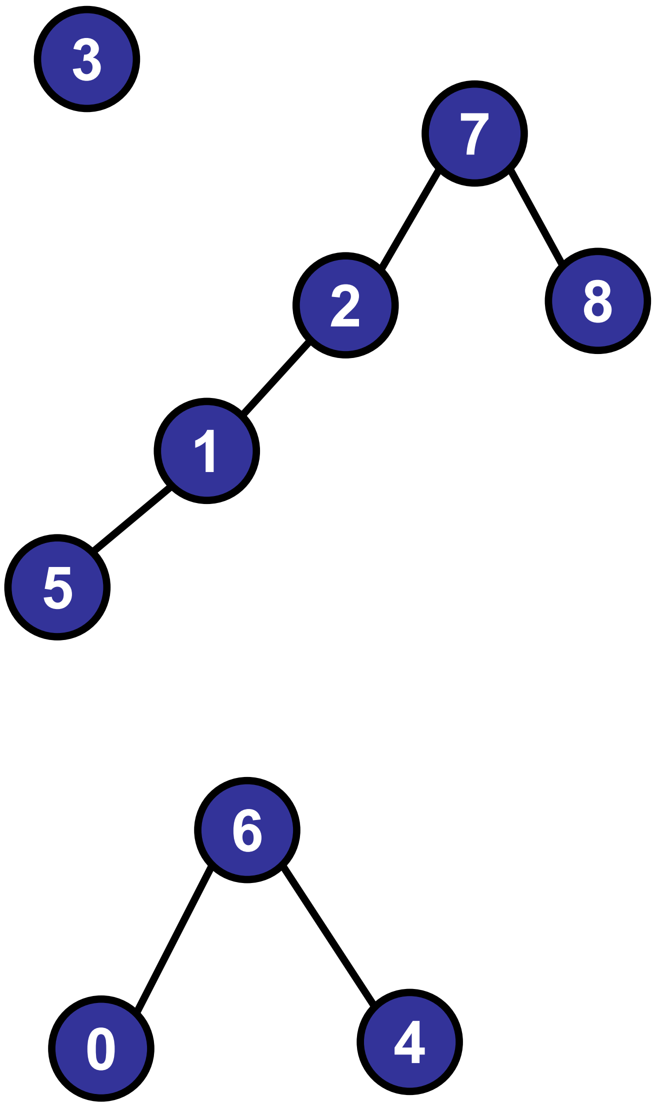
- Integer array: int[] parent
- Two objects are connected if they are part of the same tree.

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



How do we tell if two objects are in the same component?

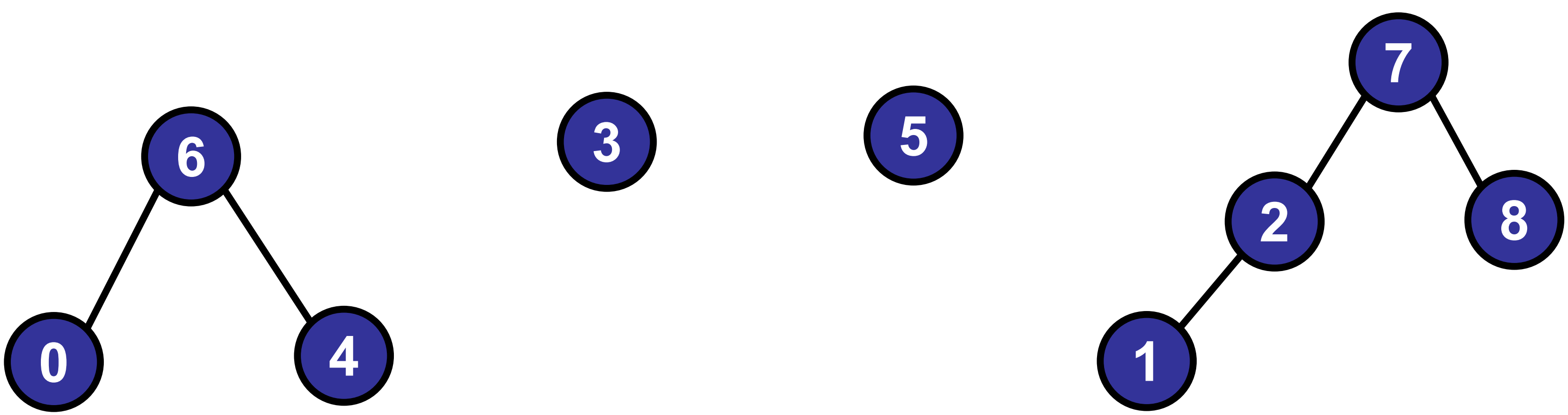
1. When they have the same component identifier.
2. When they have the same parent.
3. When they have the same root



Ver 2

```
find(int p, int q)
   traverse up the tree to obtain p's root p_root
   traverse up the tree to obtain q's root q_root
   return p_root == q_root
```

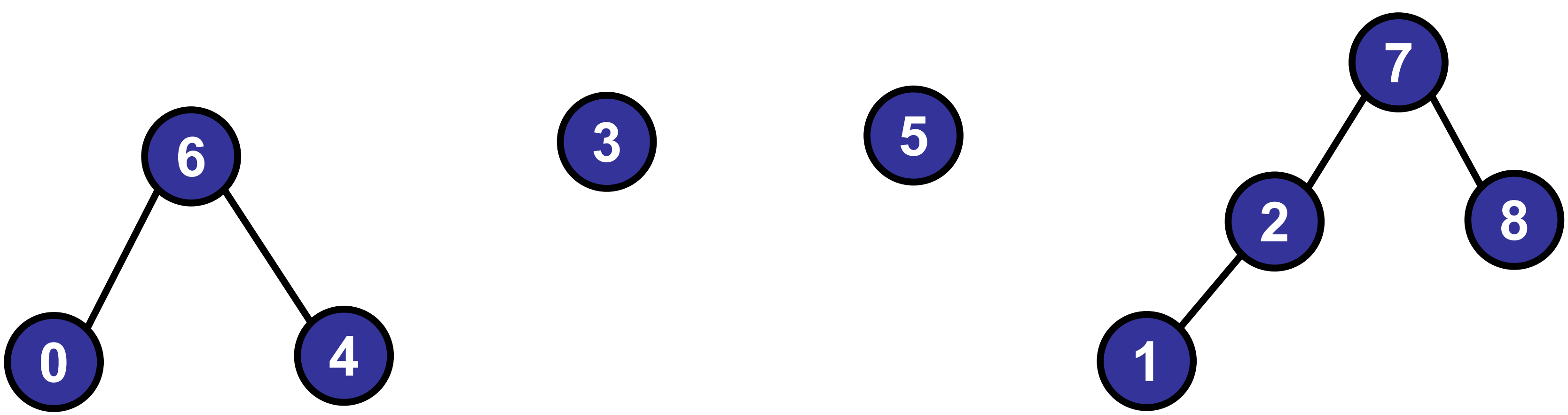
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Ver 2

```
find(int p, int q)
  while (parent[p] != p) p = parent[p];
  while (parent[q] != q) q = parent[q];
  return (p == q);
```

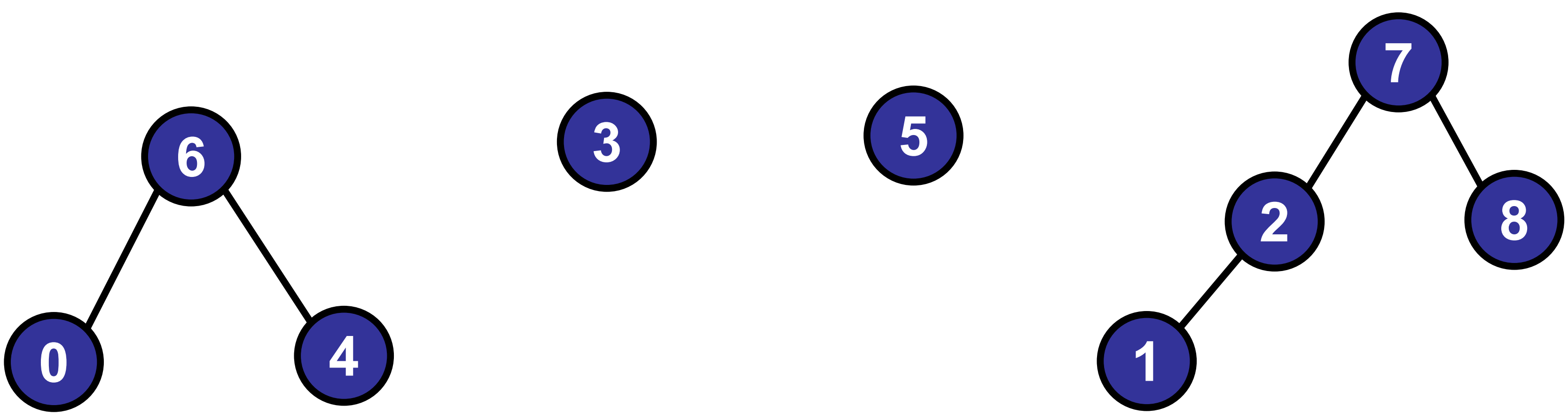
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Ver 2

```
find(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q = parent[q];
   return (p == q);
```

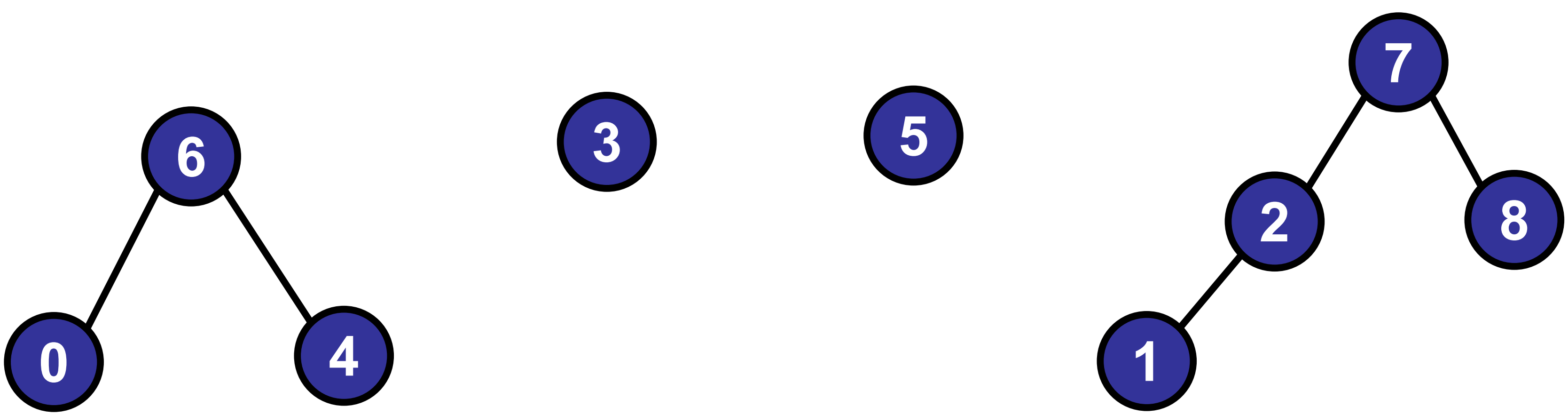
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Ver 2

```
find(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q = parent[q];
   return (p == q);
```

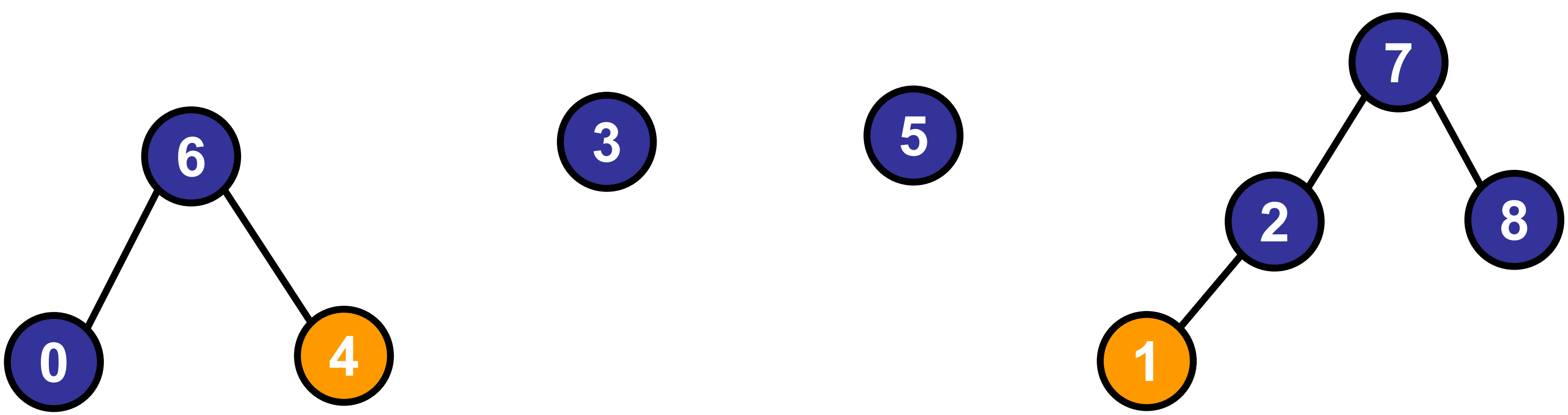
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Ver 2

`Example: find(4, 1)`

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7

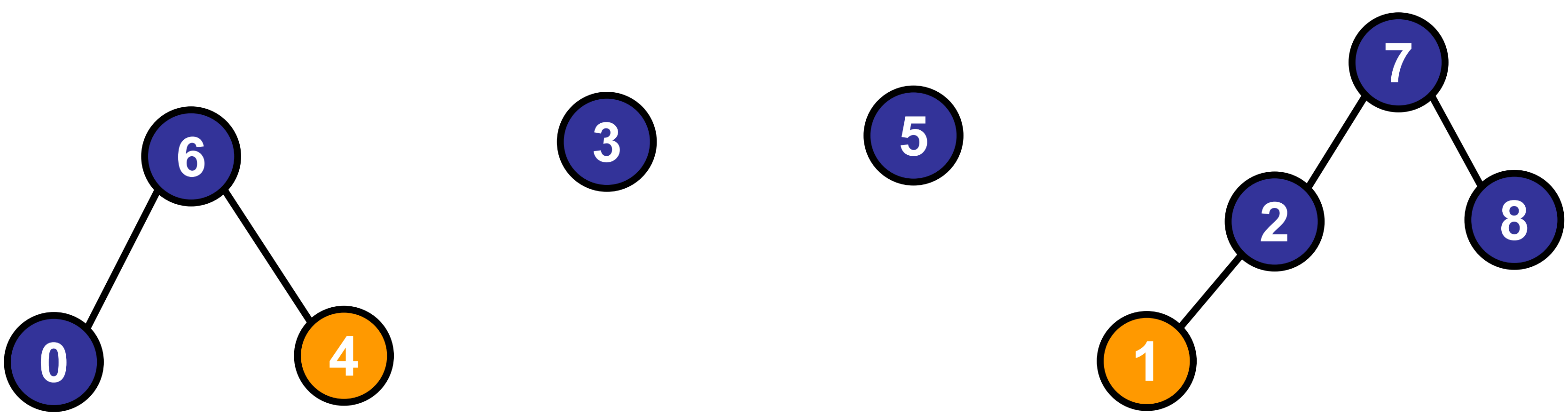


Ver 2

Example: find(4, 1)

4 □ 6 □ 6;

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



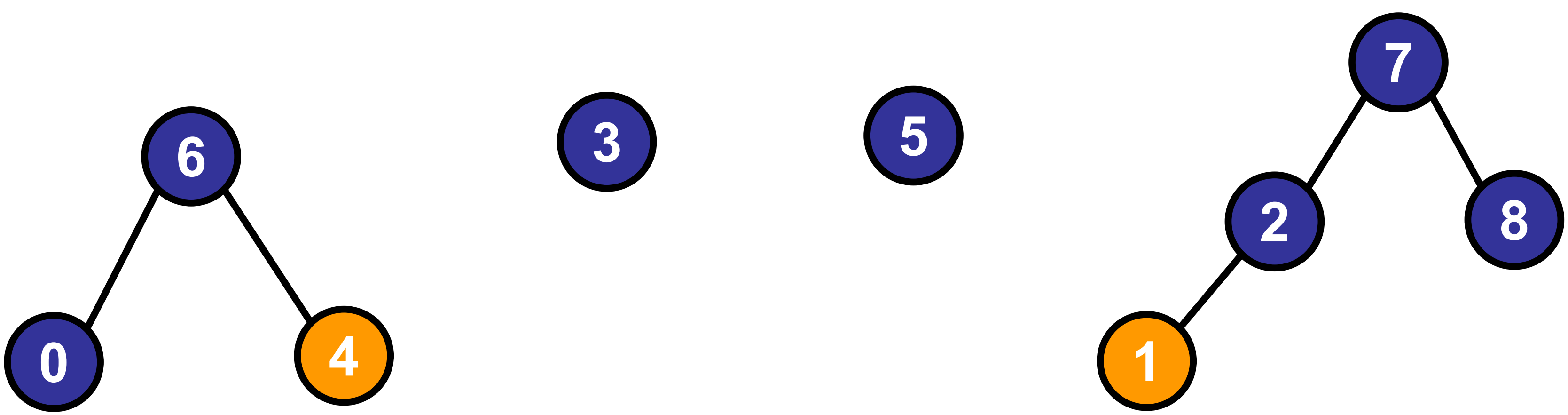
Ver 2

Example: find(4, 1)

4 □ 6 □ 6

1 □ 2 □ 7 □ 7

object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**

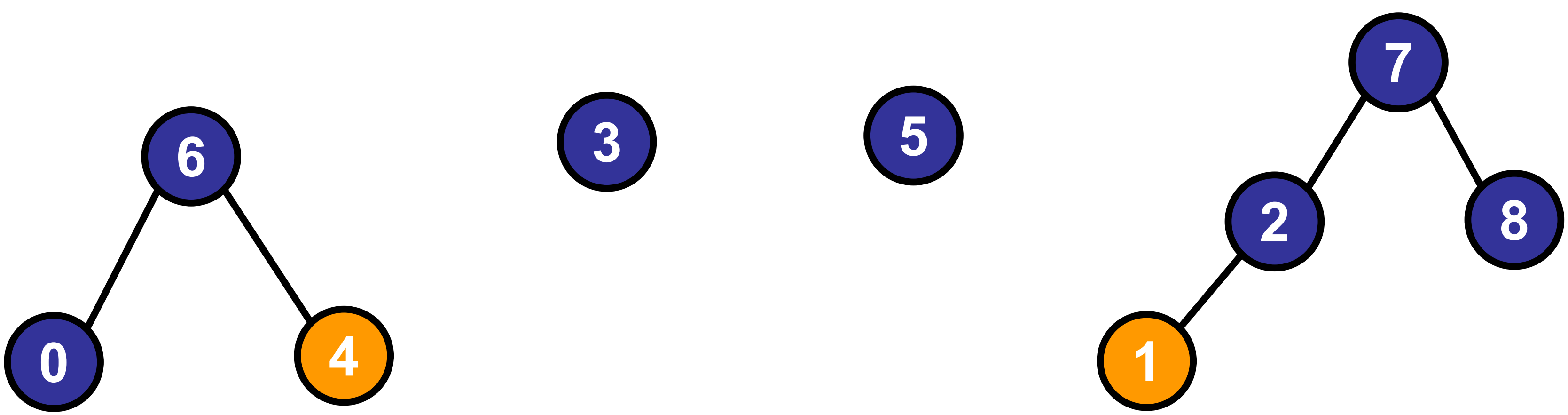


Ver 2

Example: find(4, 1)

```
4 □ 6 □ 6

1 □ 2 □ 7 □ 7

return (6 == 7) □ false
```

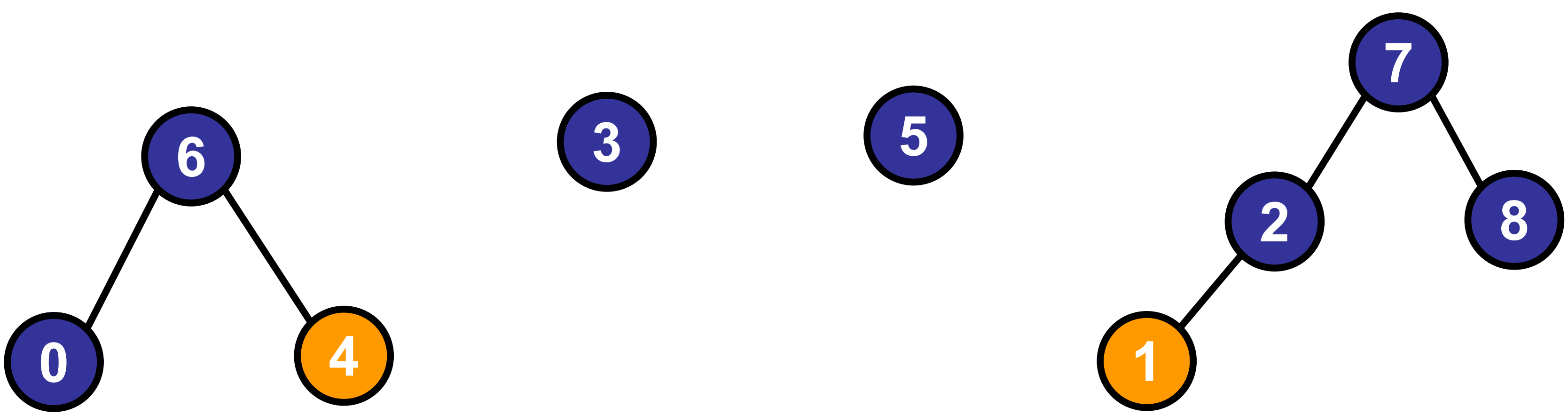
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Ver 2

But what about union?

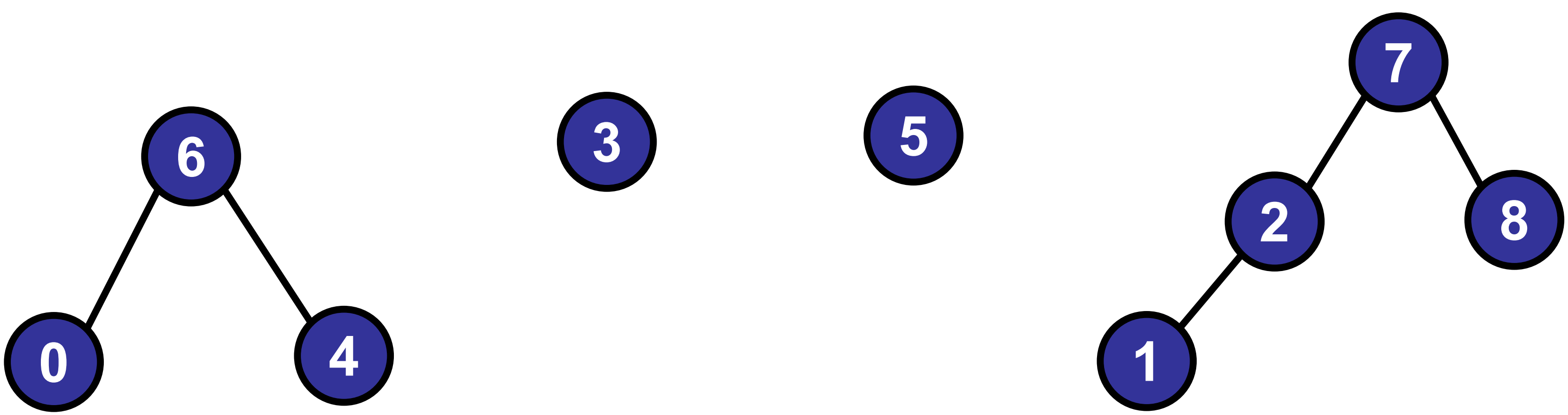
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Ver 2

```
union(int p, int q)
   traverse up the tree to obtain p's root p_root
   traverse up the tree to obtain q's root q_root
   set p_root's parent == q_root
```

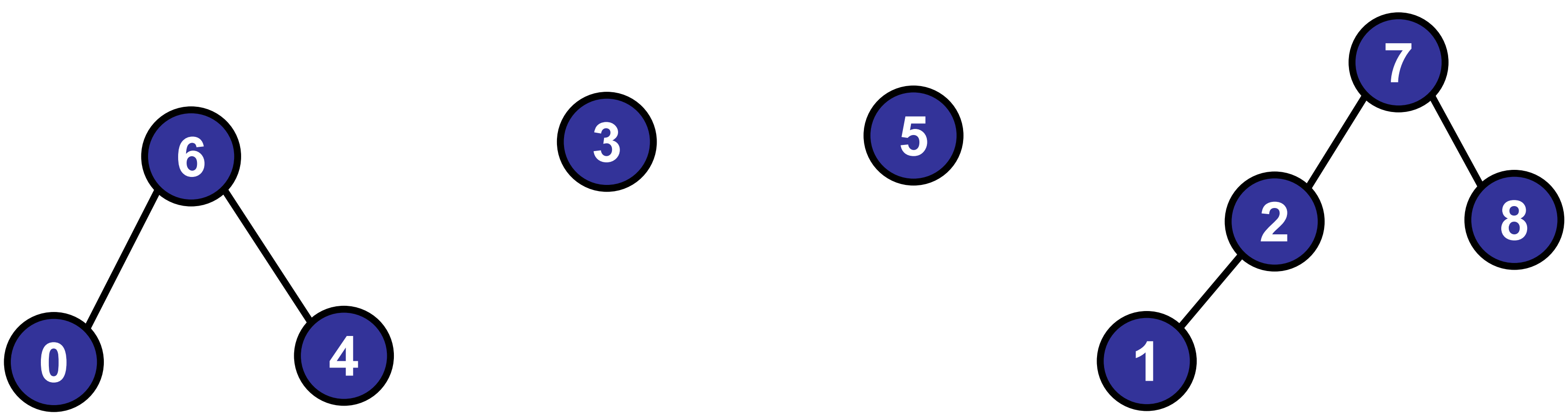
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Ver 2

```
union(int p, int q)
  while (parent[p] != p) p = parent[p];
  while (parent[q] != q) q = parent[q];
  parent[p] = q;
```

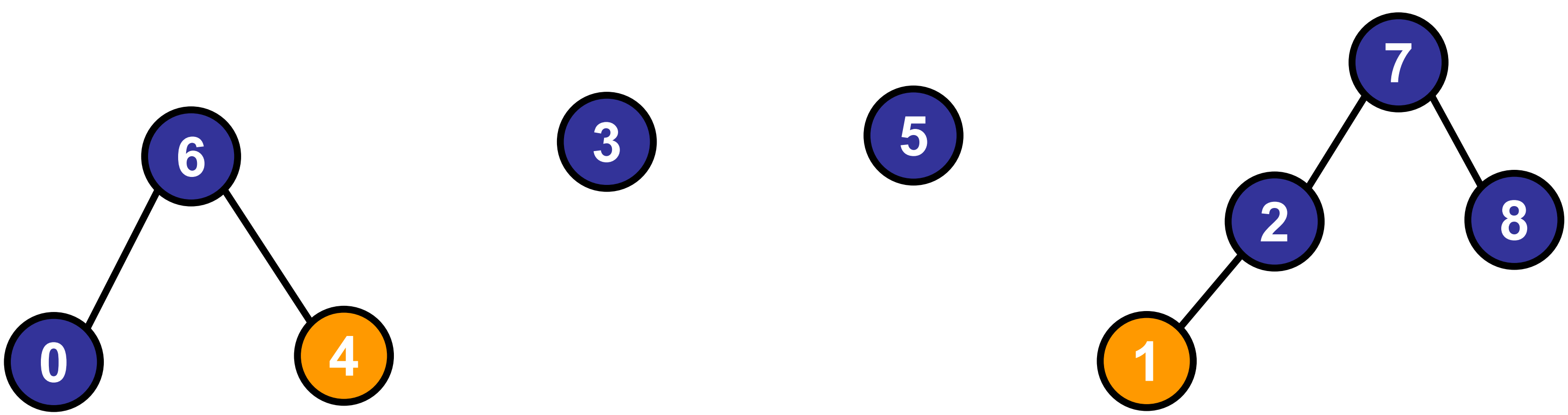
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Ver 2

Example: union(1, 4)

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



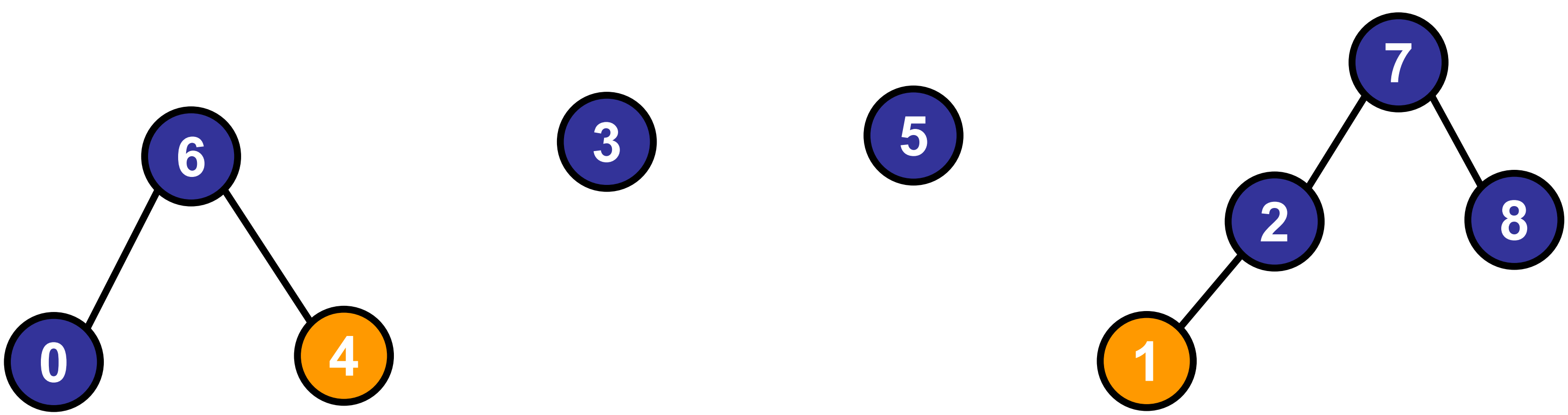
Ver 2

Example: union(1, 4)

4 □ 6 □ 6

1 □ 2 □ 7 □ 7

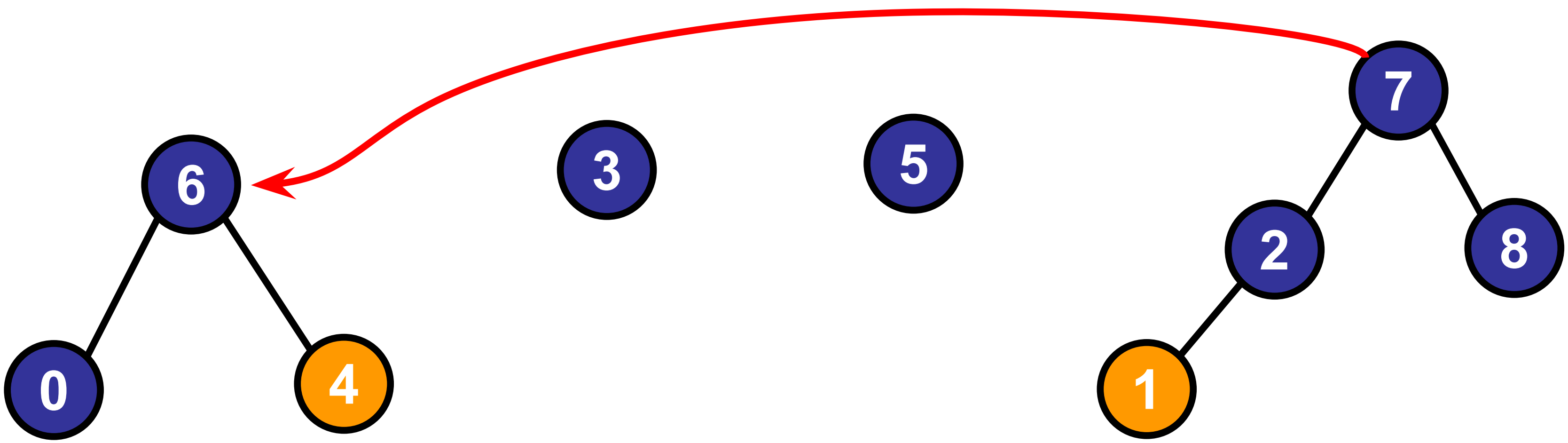
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Ver 2

```
Example: union(1, 4)

4 □ 6 □ 6

1 □ 2 □ 7 □ 7

parent[7] = 6;
```

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	6	7

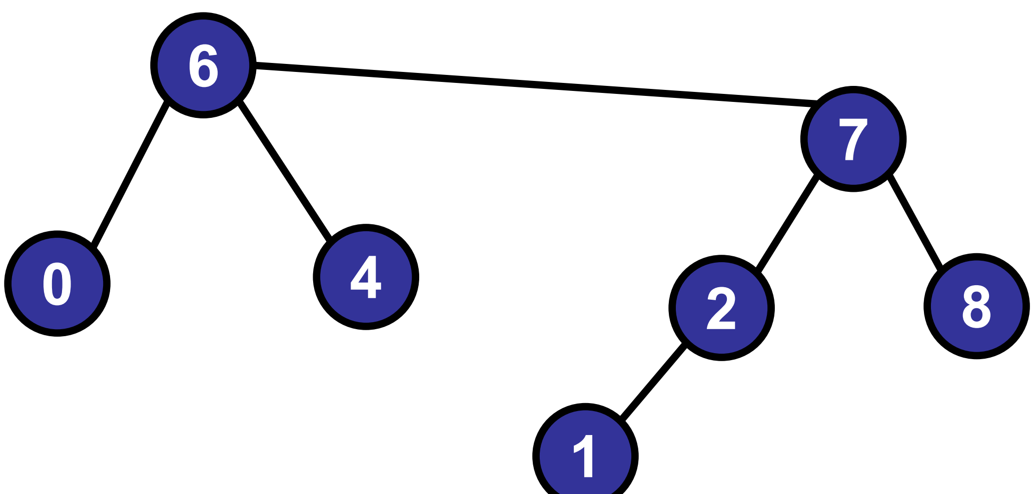


Ver 2

Example: union(1, 4)

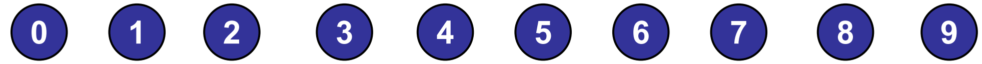
```
4 □ 6 □ 6

1 □ 2 □ 7 □ 7

parent[7] = 6;
```

object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**6**	**7**



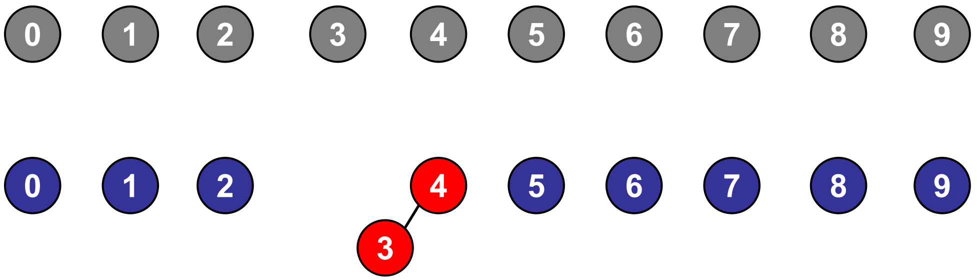
Example:

P	0	1	2	3	4	5	6	7	8	9



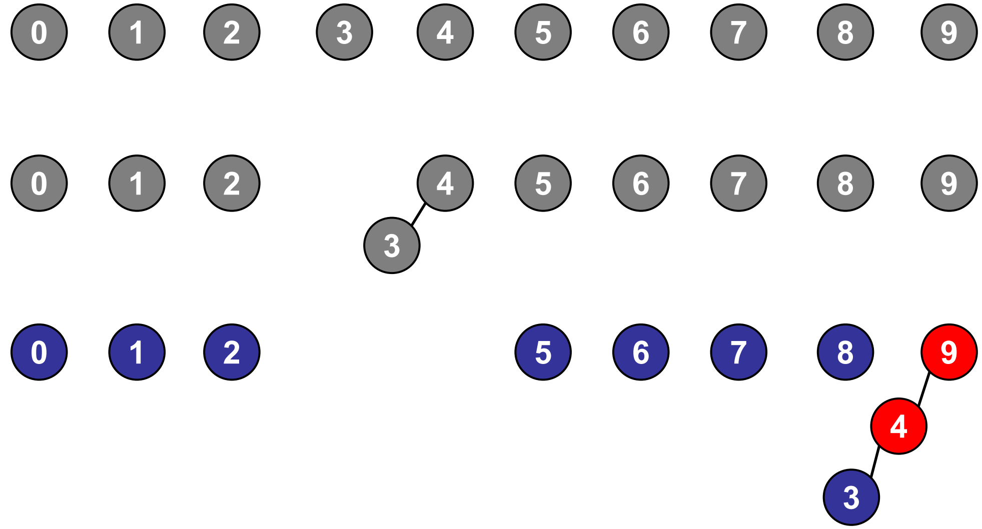
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9



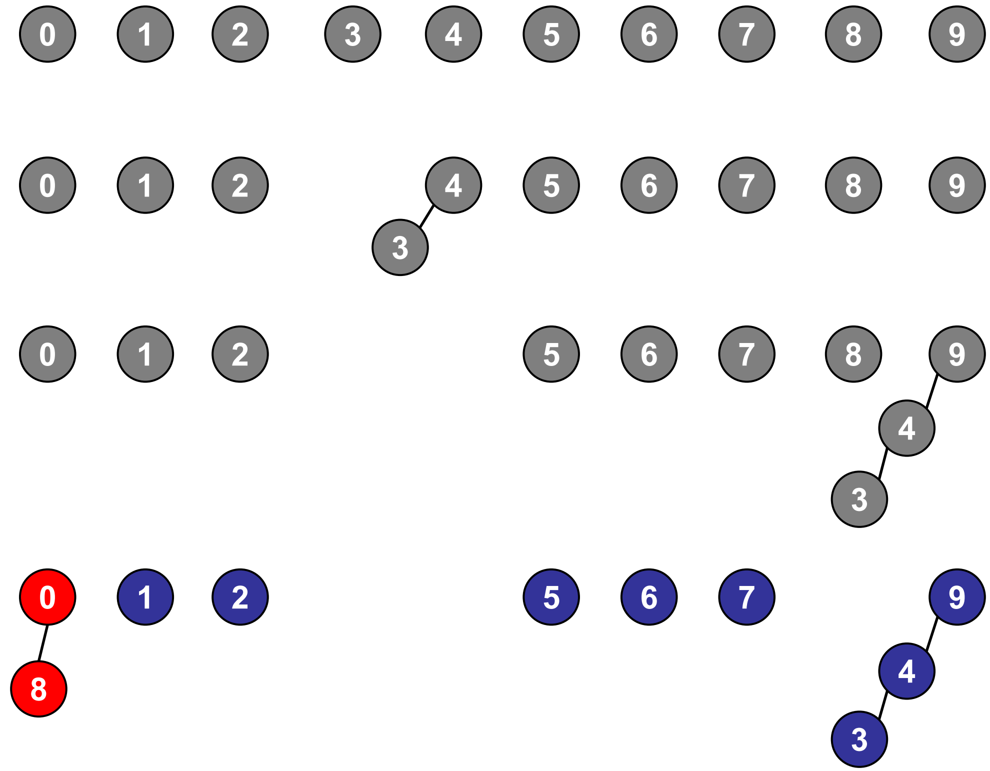
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9



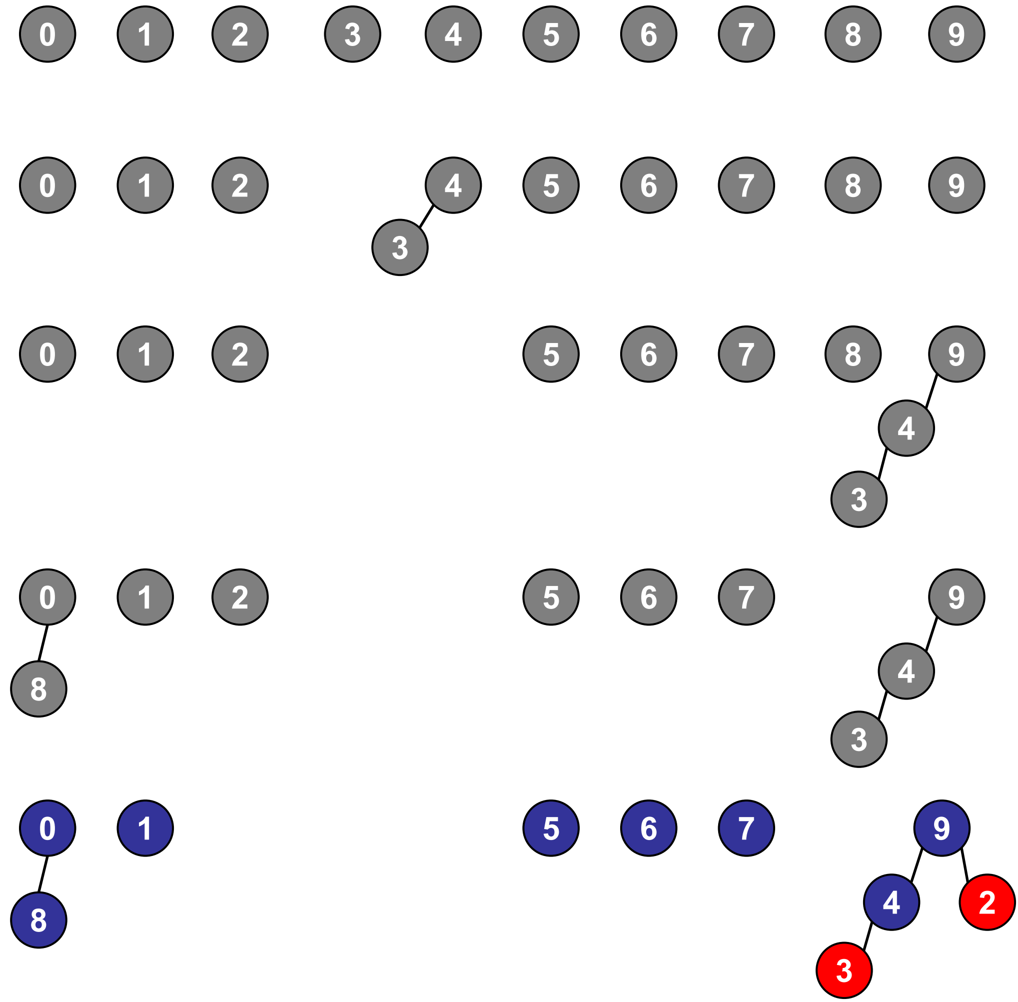
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9



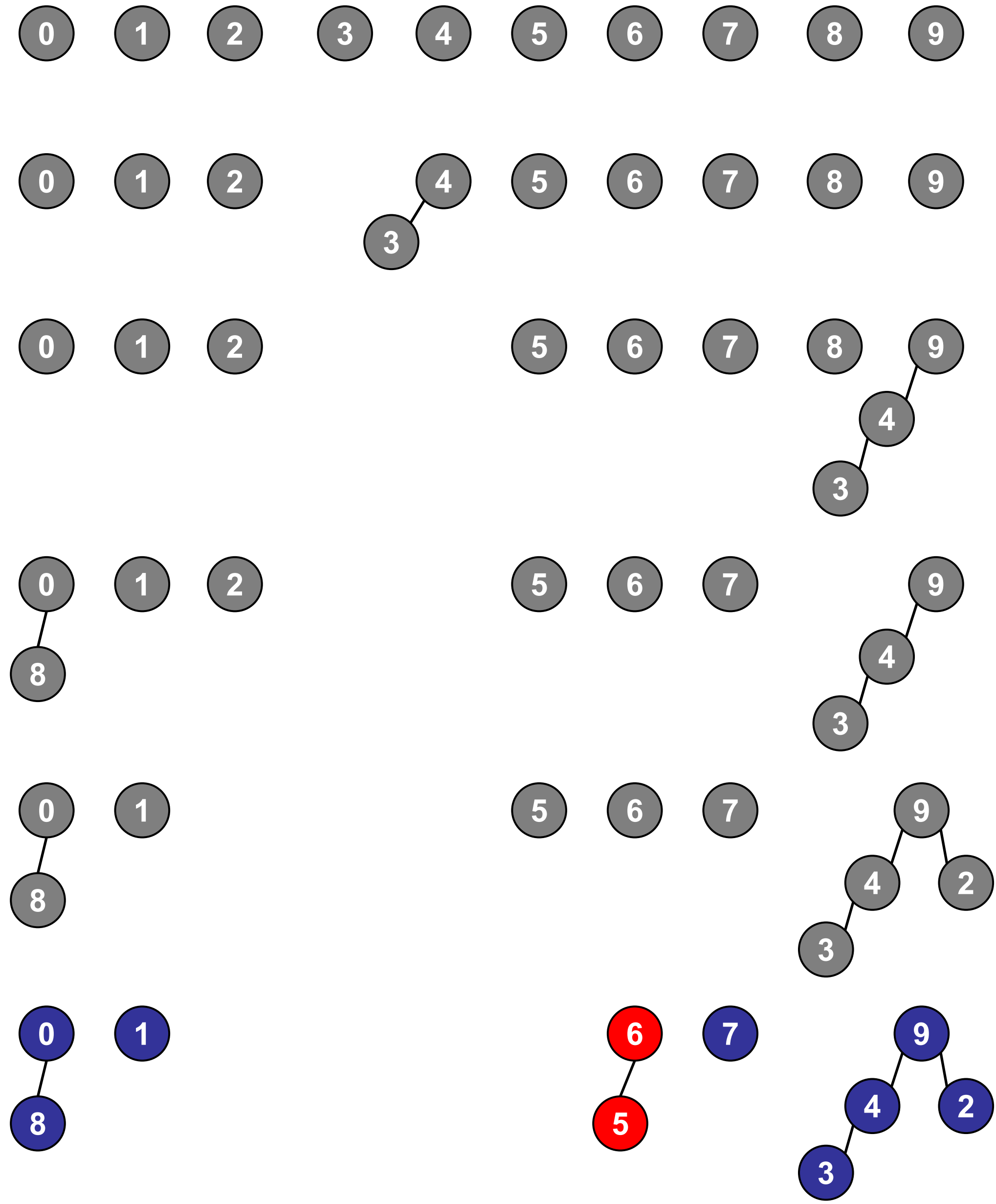
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9



Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9

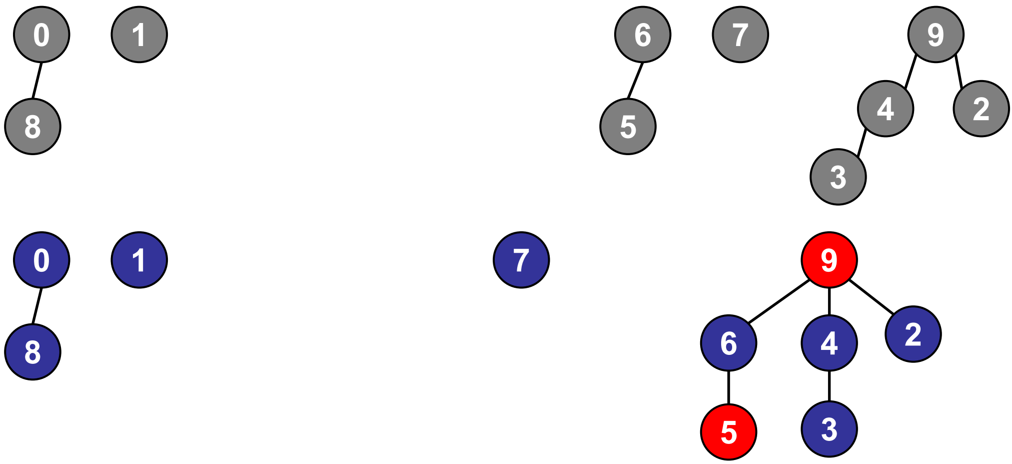


Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9

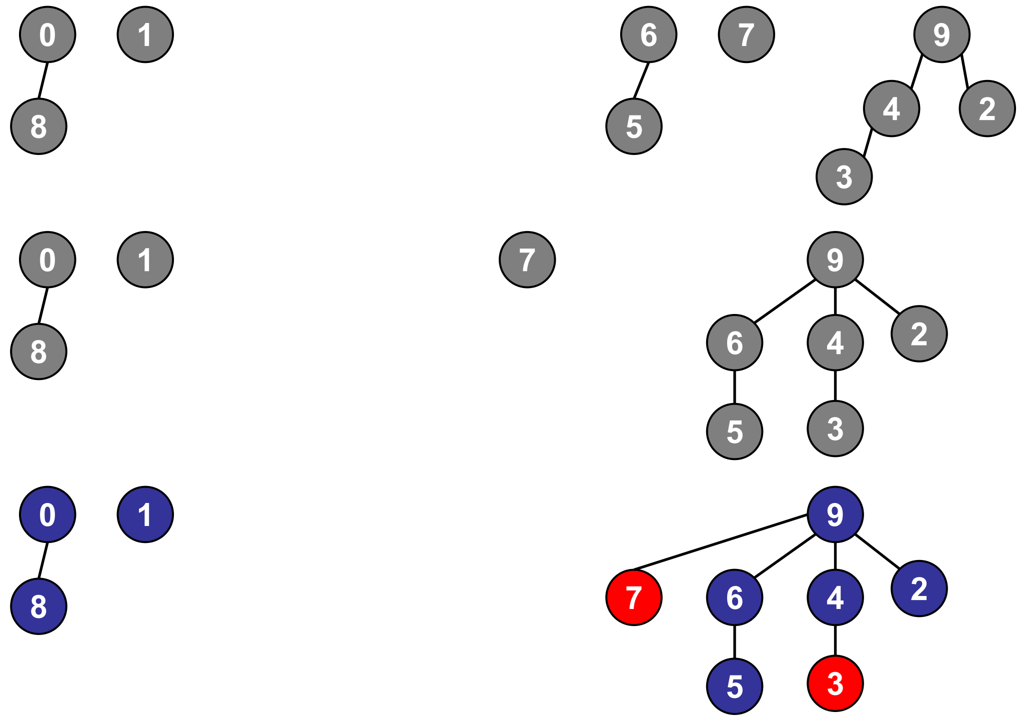
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9



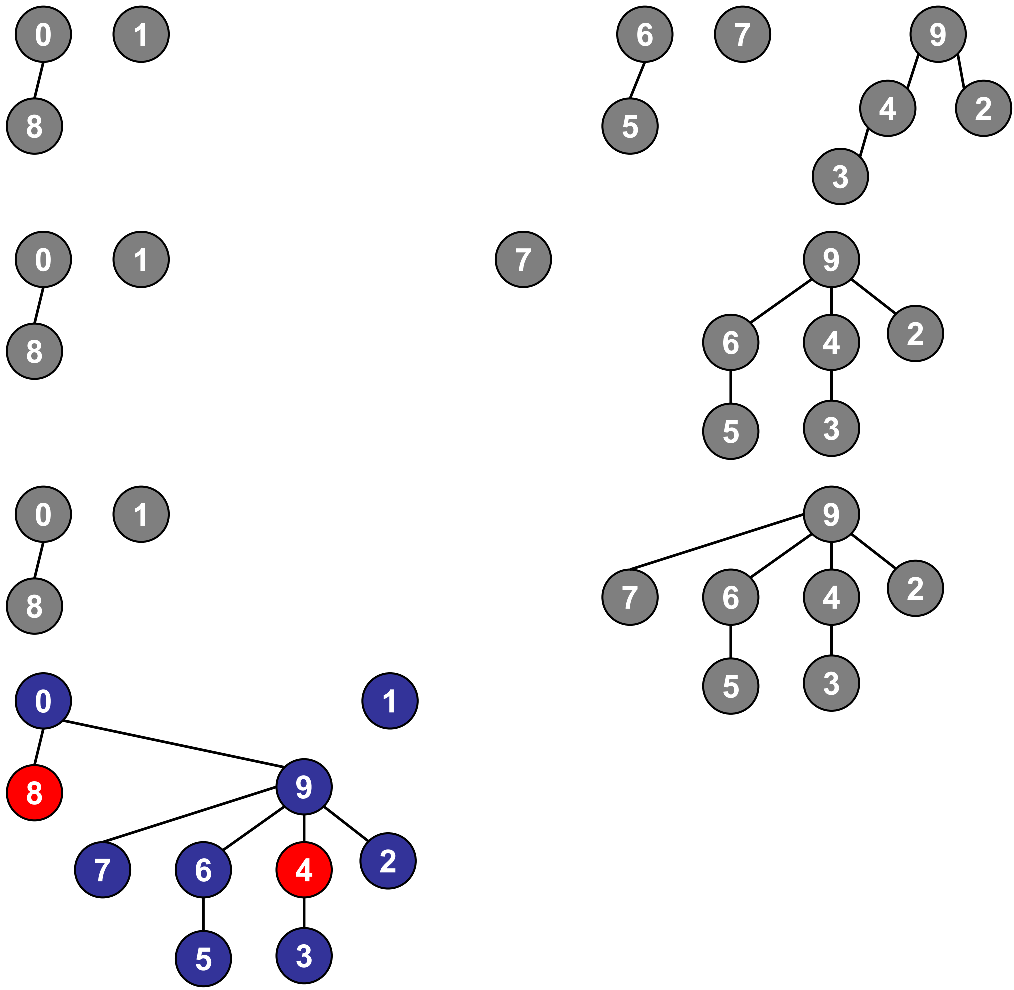
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9



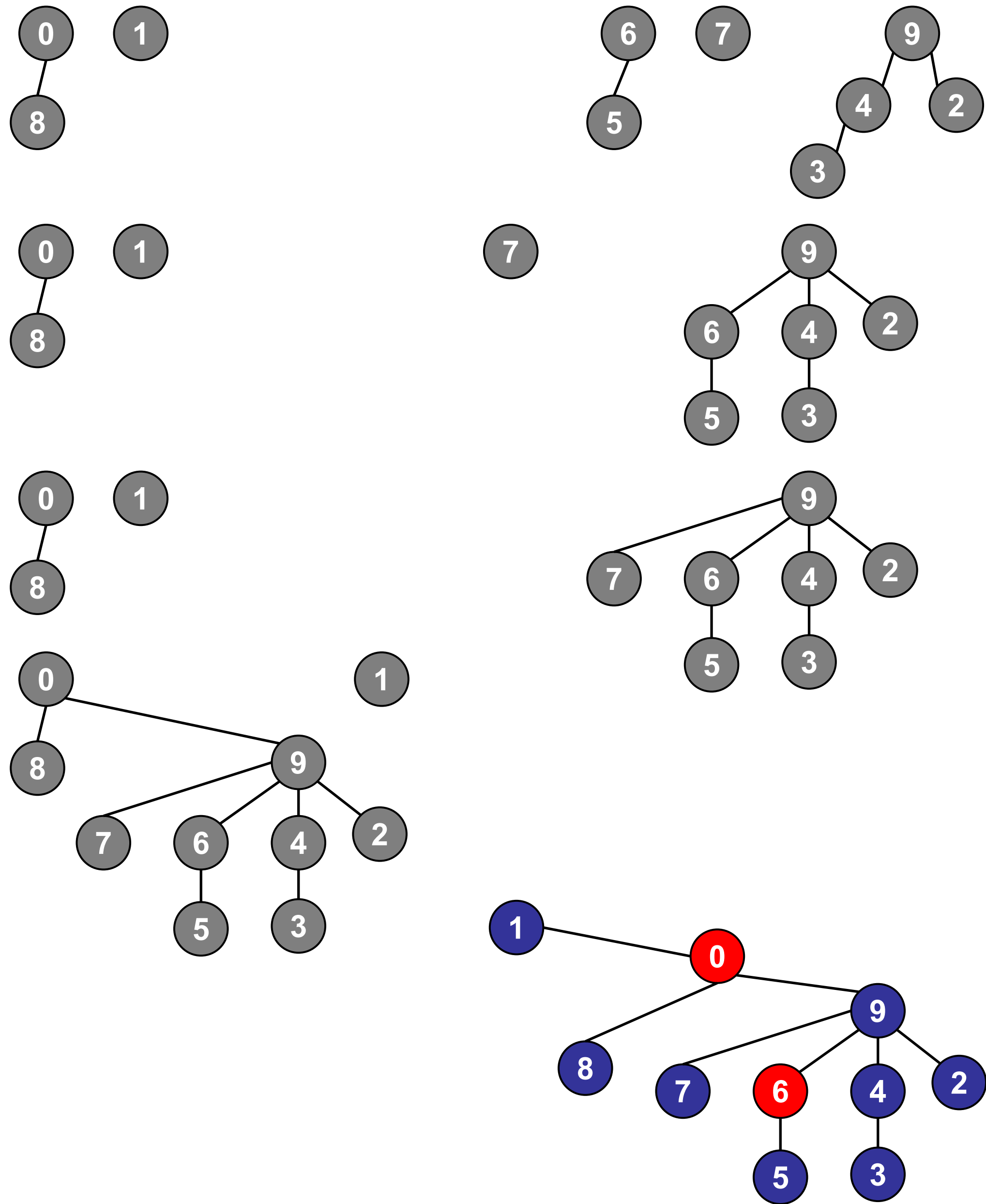
Example:

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0



Example:

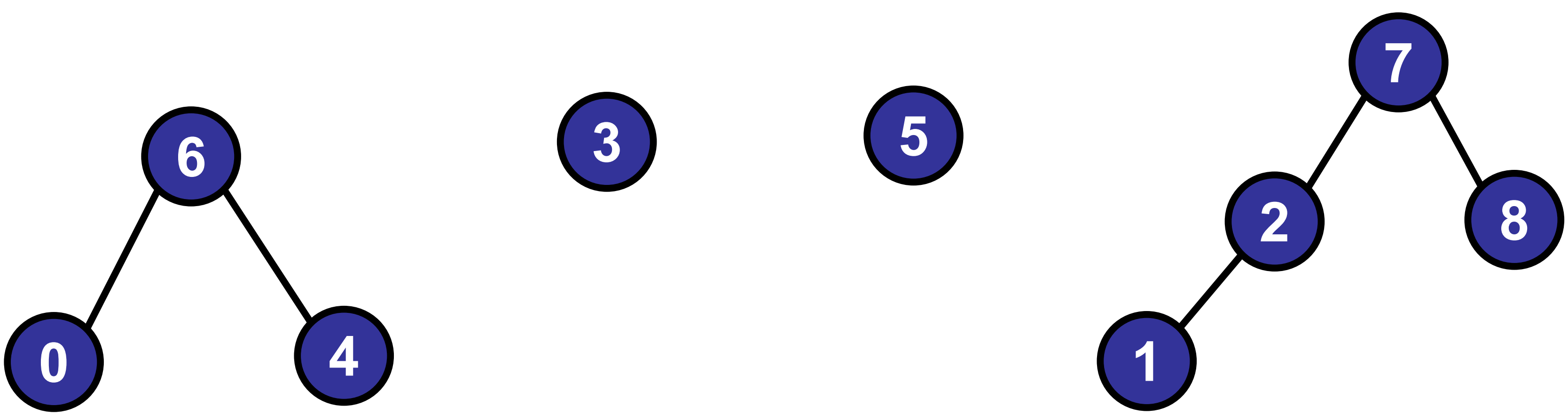
P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



Ver 2

```
union(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q = parent[q];
   parent[p] = q;
```

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Running time of (Find, Union):

1. O(1), O(1)
2. O(1), O(n)
3. O(n), O(1)
4. O(n), O(n)
5. O(log n), O(log n)
6. None of the above.

Running time of (Find, Union):

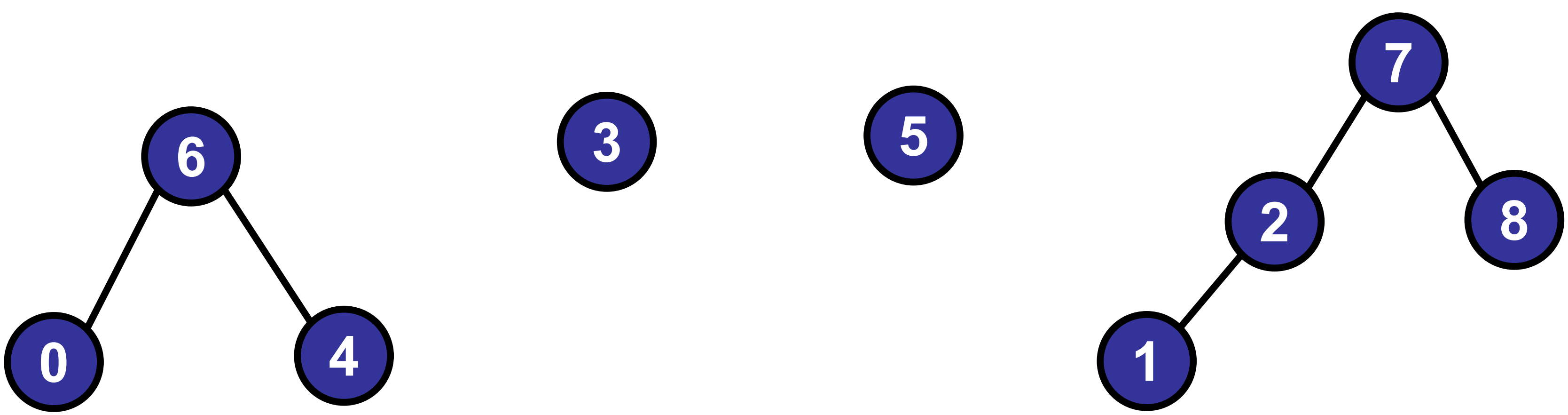
1. O(1), O(1)
2. O(1), O(n)
3. O(n), O(1)
4. O(n), O(n)
5. O(log n), O(log n)
6. None of the above.

Somehow even slower than before!

Ver 2

```
find(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q = parent[q];
   return (p == q);
```

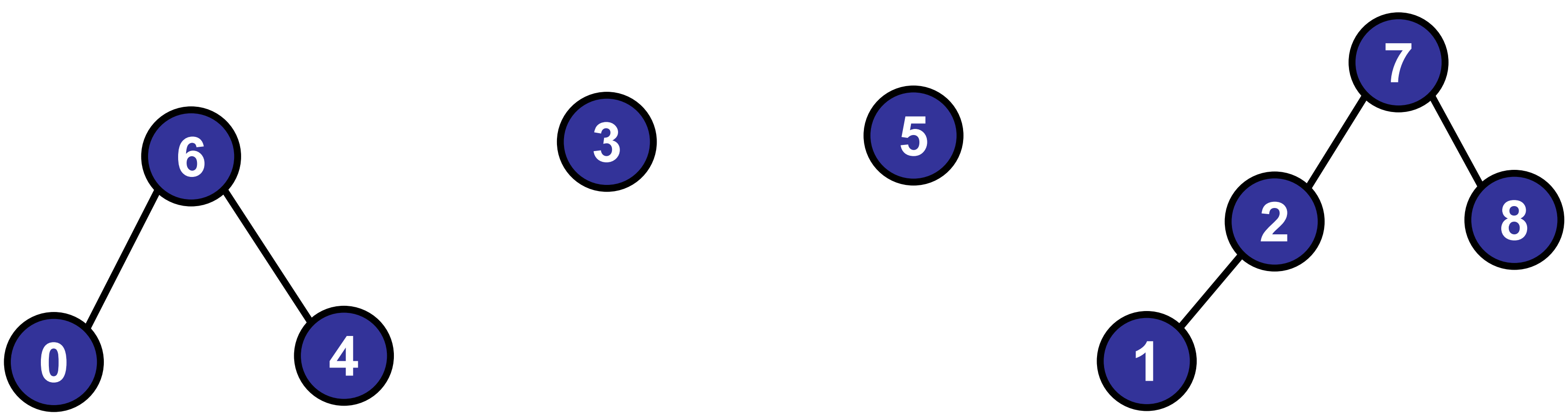
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Ver 2

```
union(int p, int q)
   while (parent[p] != p) p = parent[p];
   while (parent[q] != q) q = parent[q];
   parent[p] = q;
```

object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Union-Find Summary

Ver 1 is slow:

- Union is expensive
- Tree is flat

Ver 2 is slow:

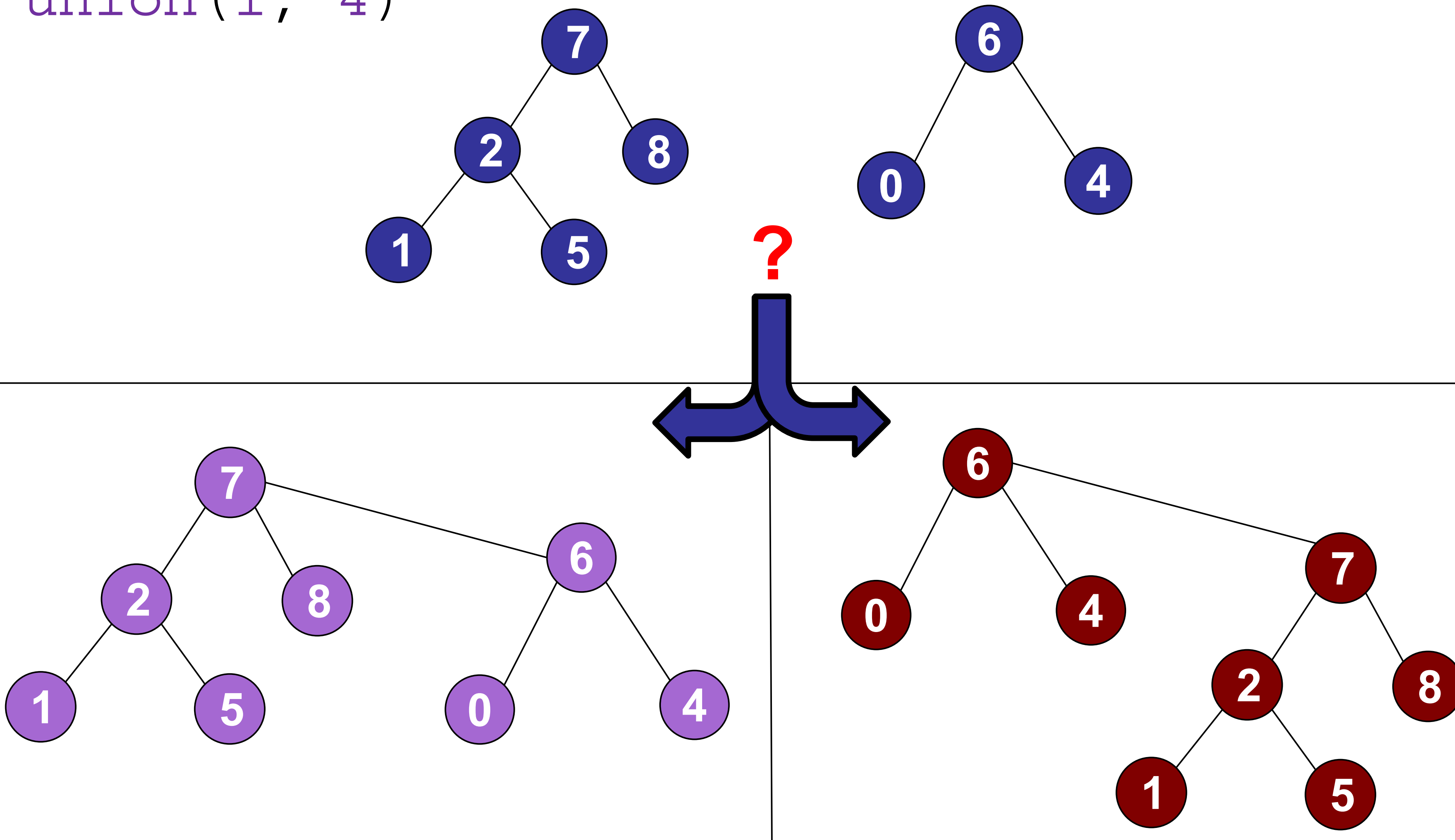
- Trees too tall (i.e., unbalanced)
- Union *and* find are expensive.

	find	union
Ver 1	O(1)	O(n)
Ver 2	O(n)	O(n)

Weighted Union

Question: which tree should you make the root?

`union(1, 4)`



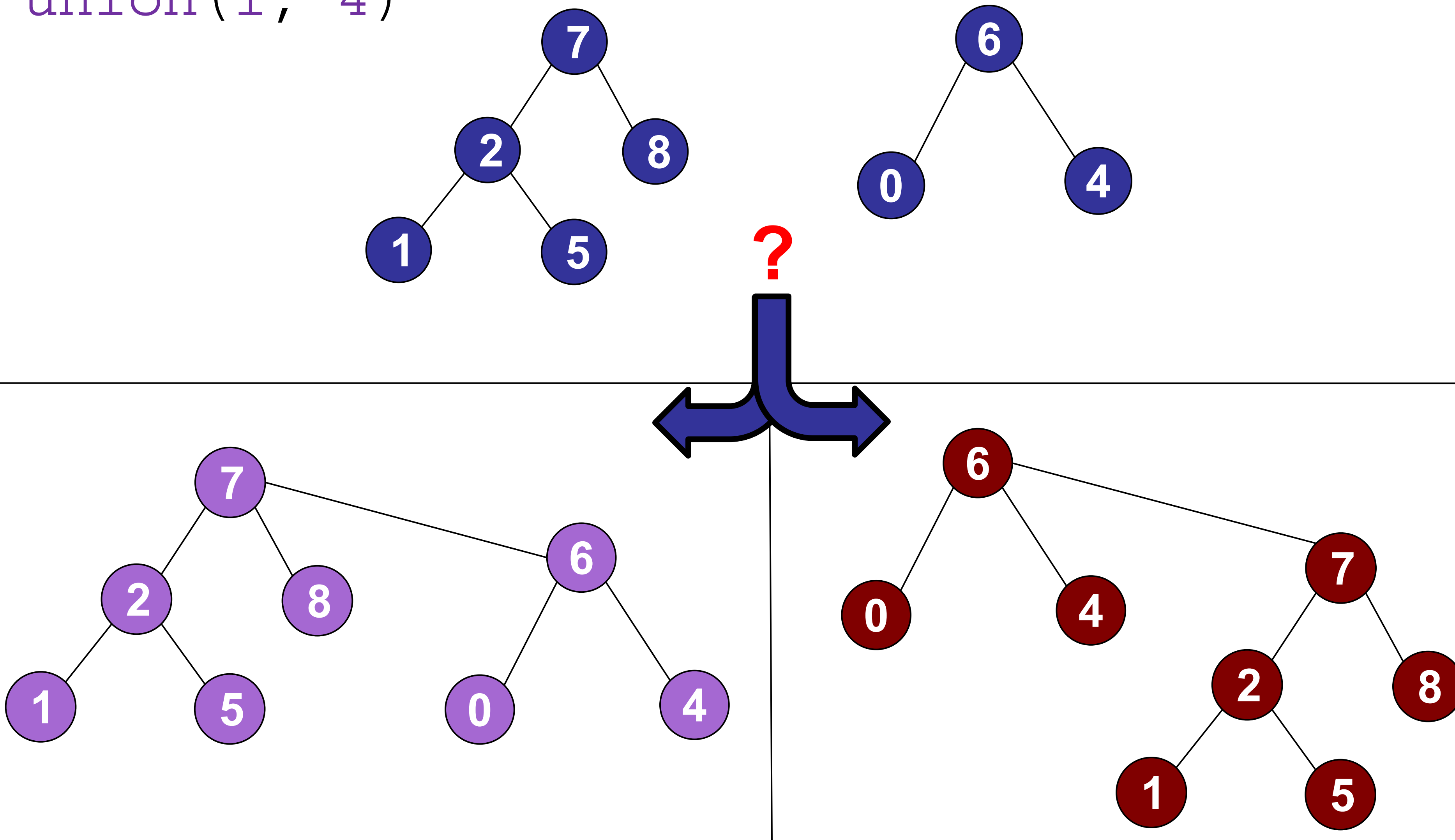
Weighted Union

Two possible alternatives:

1. The one that has larger size
2. The one that has larger height

Question: which tree should you make the root?

`union(1, 4)`



Weighted Union

```
union(int p, int q)
   while (parent[p] !=p) p = parent[p];
   while (parent[q] !=q) q = parent[q];
   if (size[p] > size[q] {
        parent[q] = p;    // Link q to p
        size[p] = size[p] + size[q];
   }
   else {
        parent[p] = q; // Link p to q
        size[q] = size[p] + size[q];
   }
```

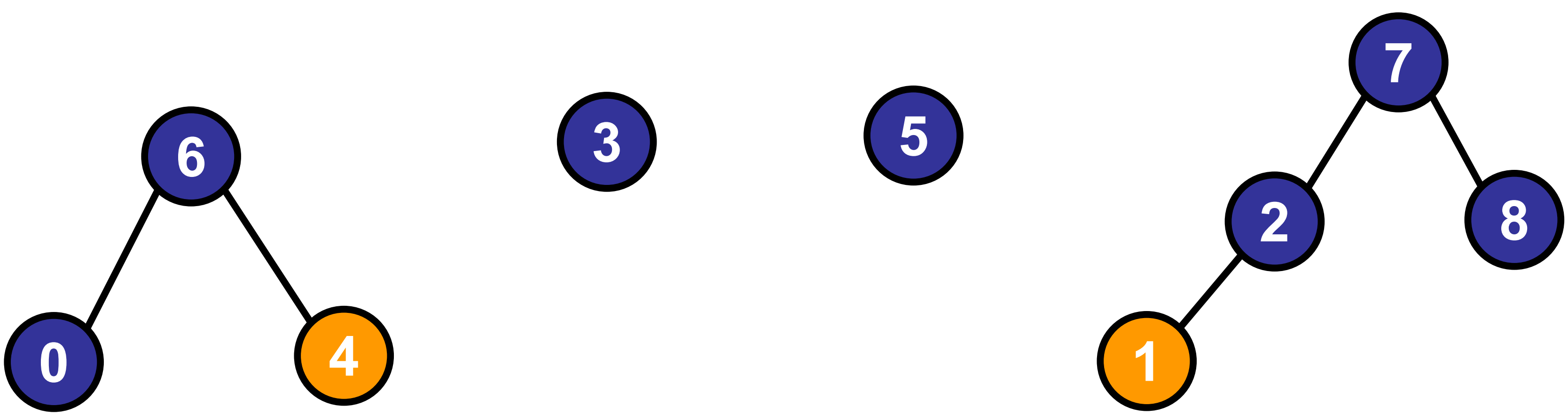
Weighted Union

```
union(int p, int q)
   while (parent[p] !=p) p = parent[p];
   while (parent[q] !=q) q = parent[q];
   if (size[p] > size[q] {
        parent[q] = p;    // Link q to p
        size[p] = size[p] + size[q];
   }
   else {
        parent[p] = q; // Link p to q
        size[q] = size[p] + size[q];
   }
```

Weighted Union

```
union(1, 4)
```

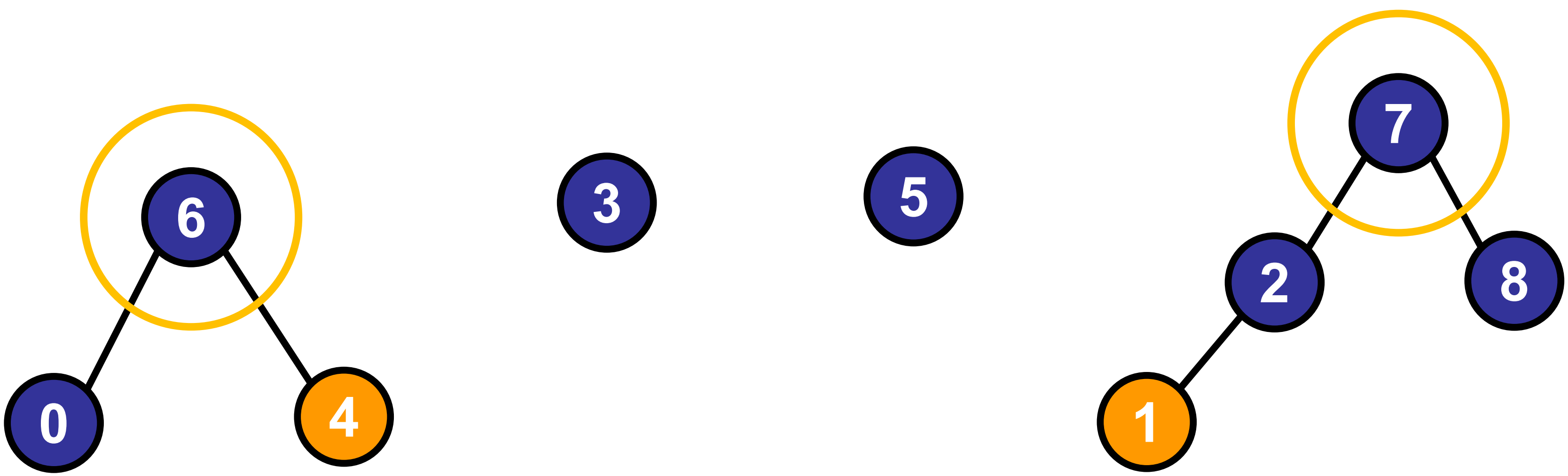
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

```
union(1, 4)
```

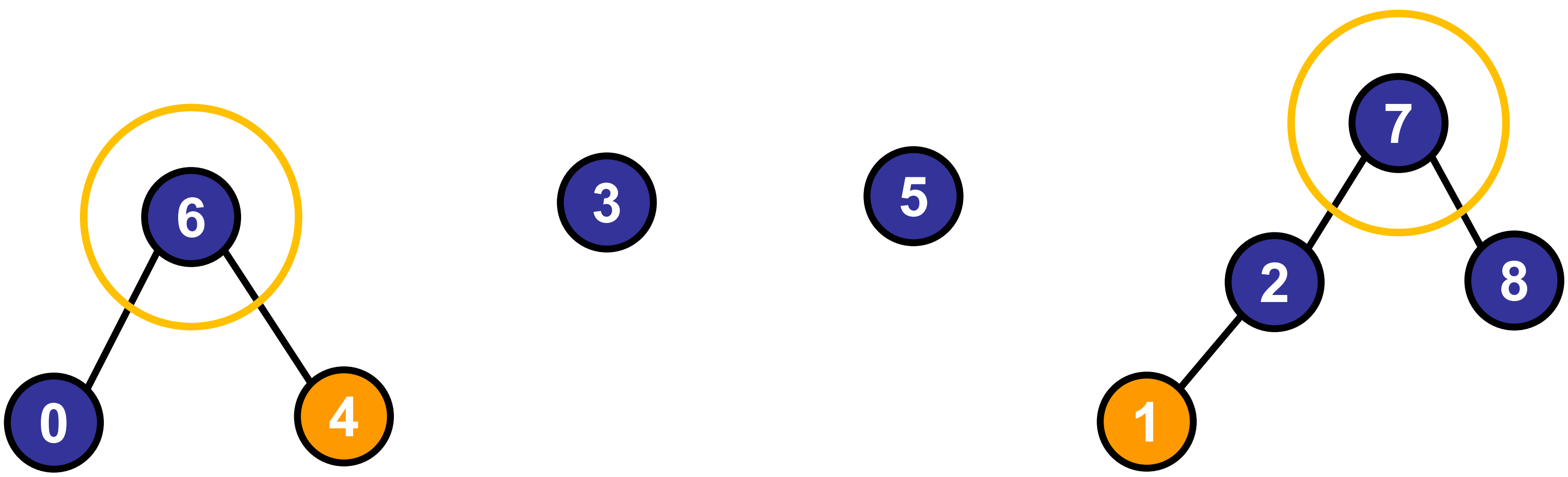
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Which one should be the new root?

```
union(1, 4)
```

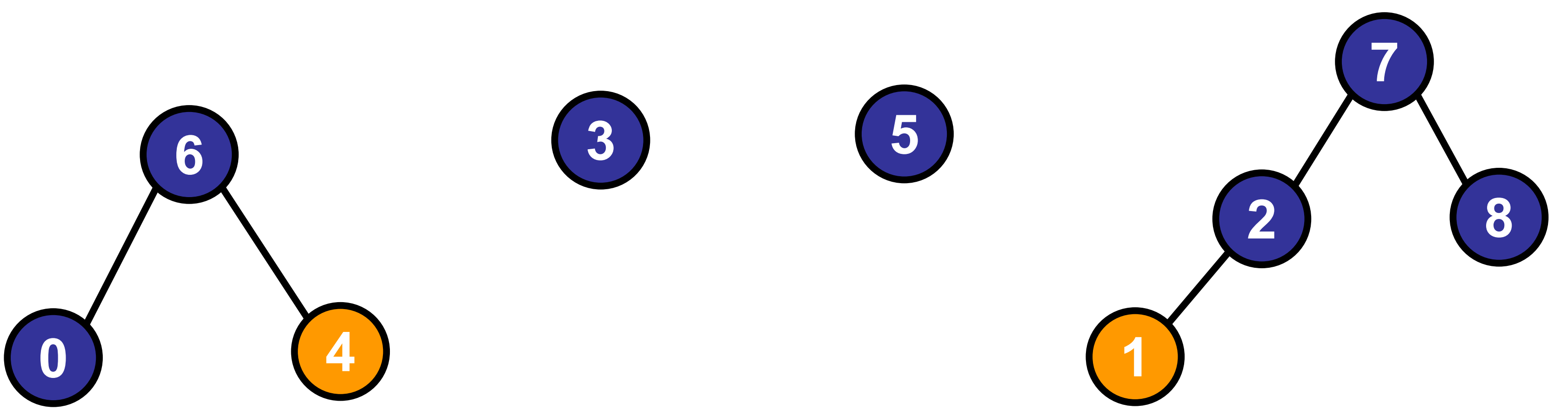
1. 1
2. 4
3. 7
4. 6



Weighted Union

`union(1, 4)`

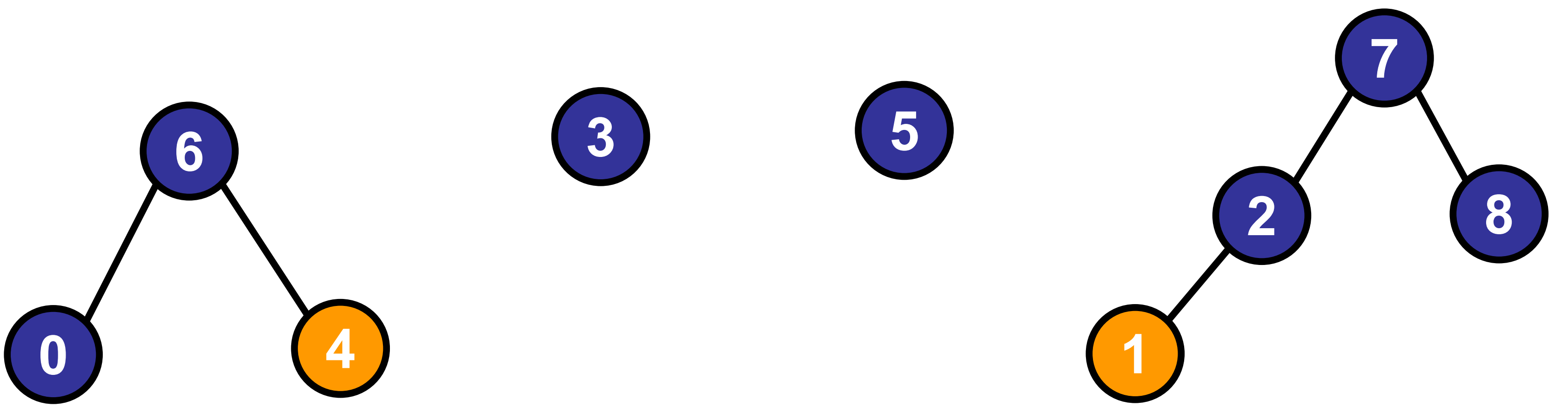
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

```
union(1, 4)
```

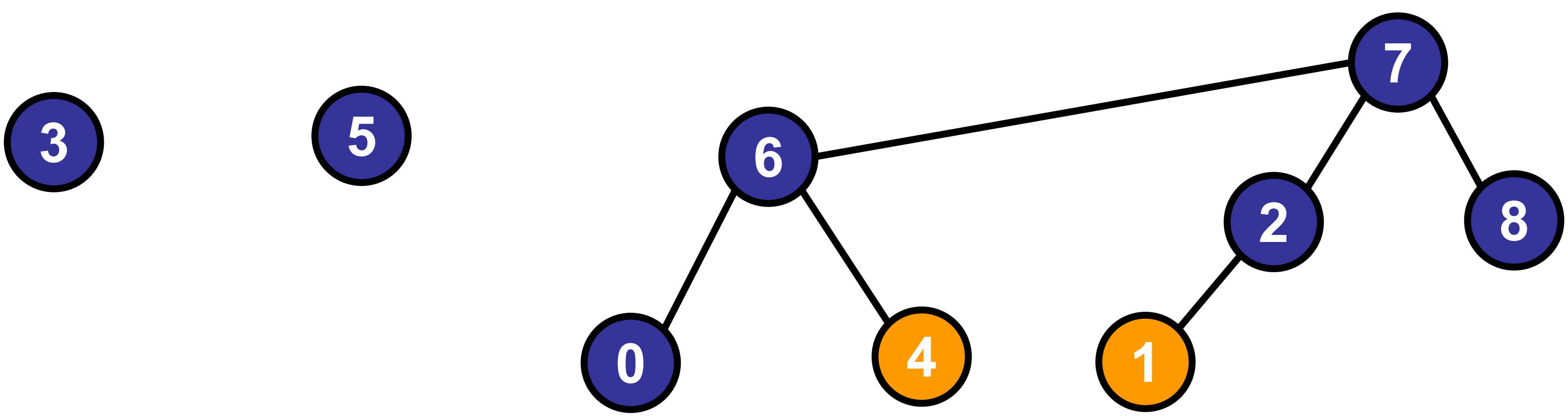
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

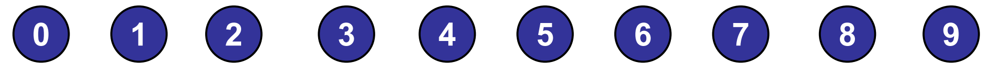
`union(1, 4)`

object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	7	1
parent	6	2	7	3	6	1	6	7	7



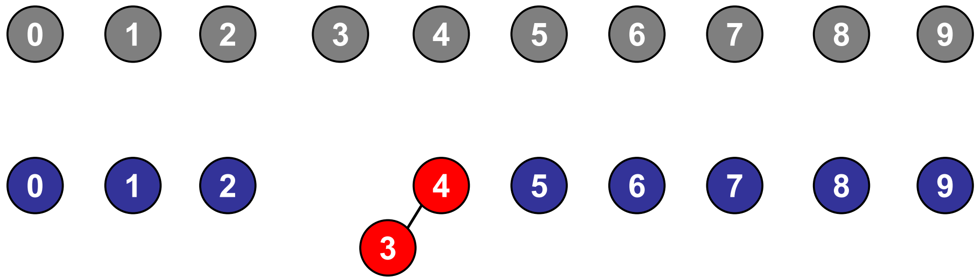
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9



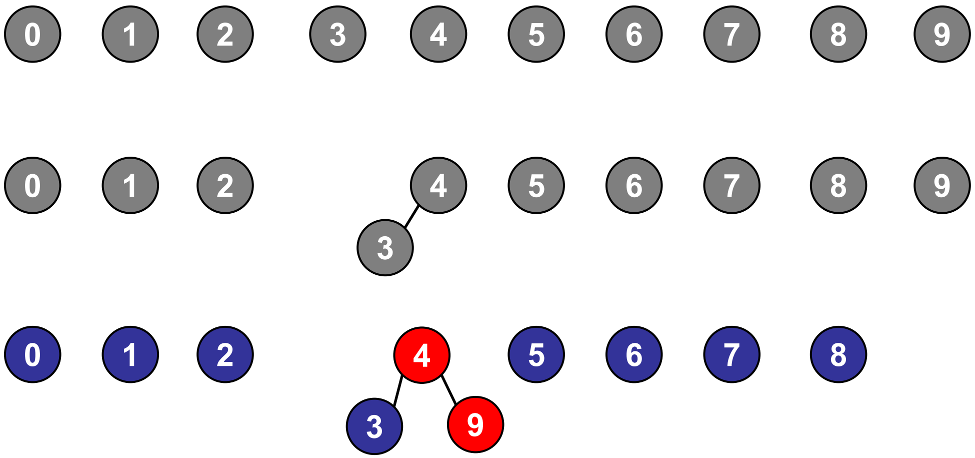
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9



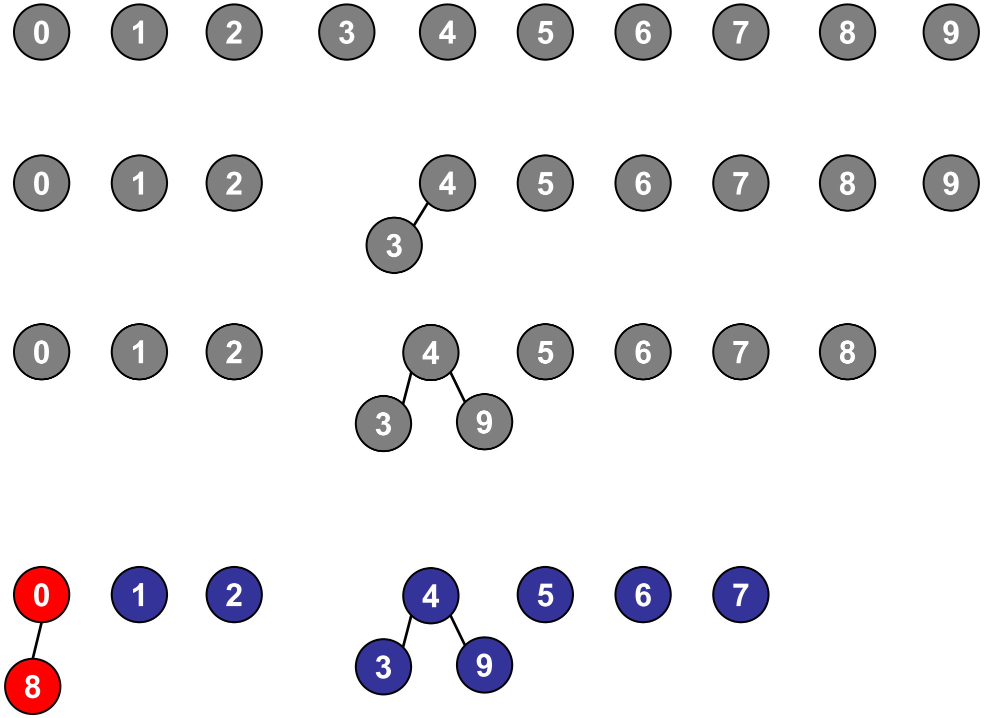
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4



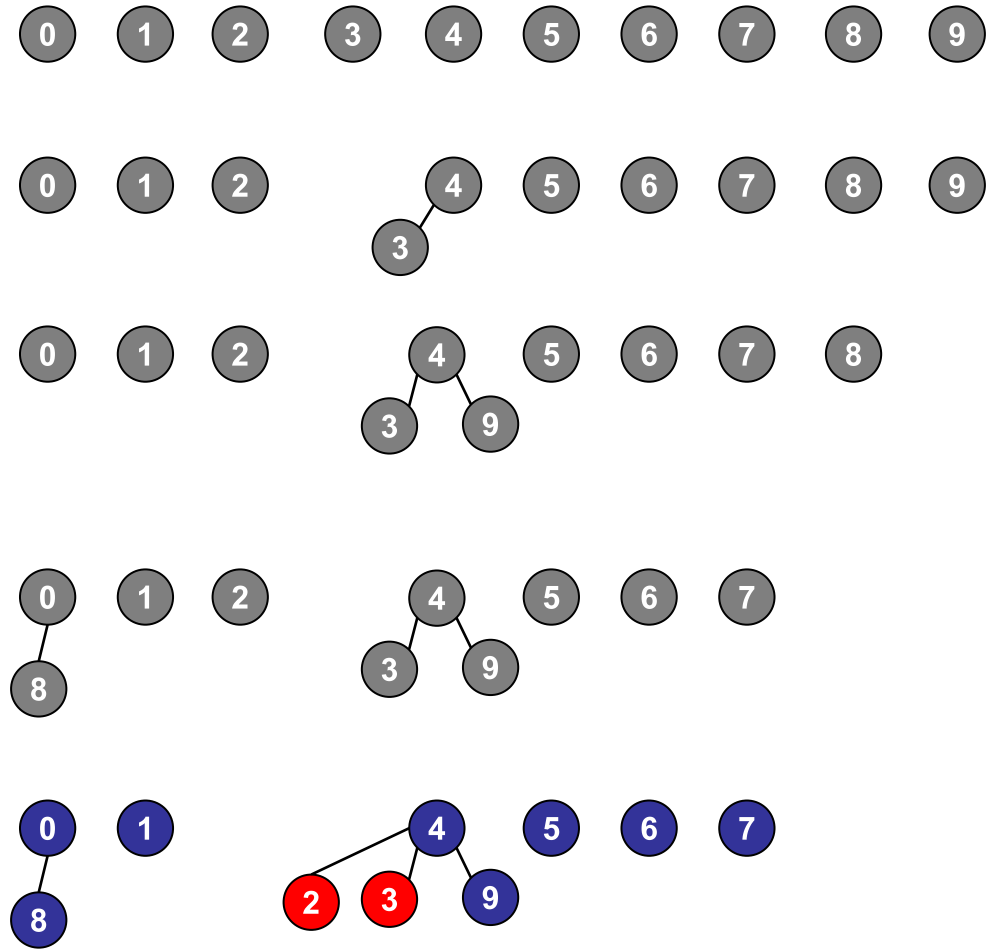
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4



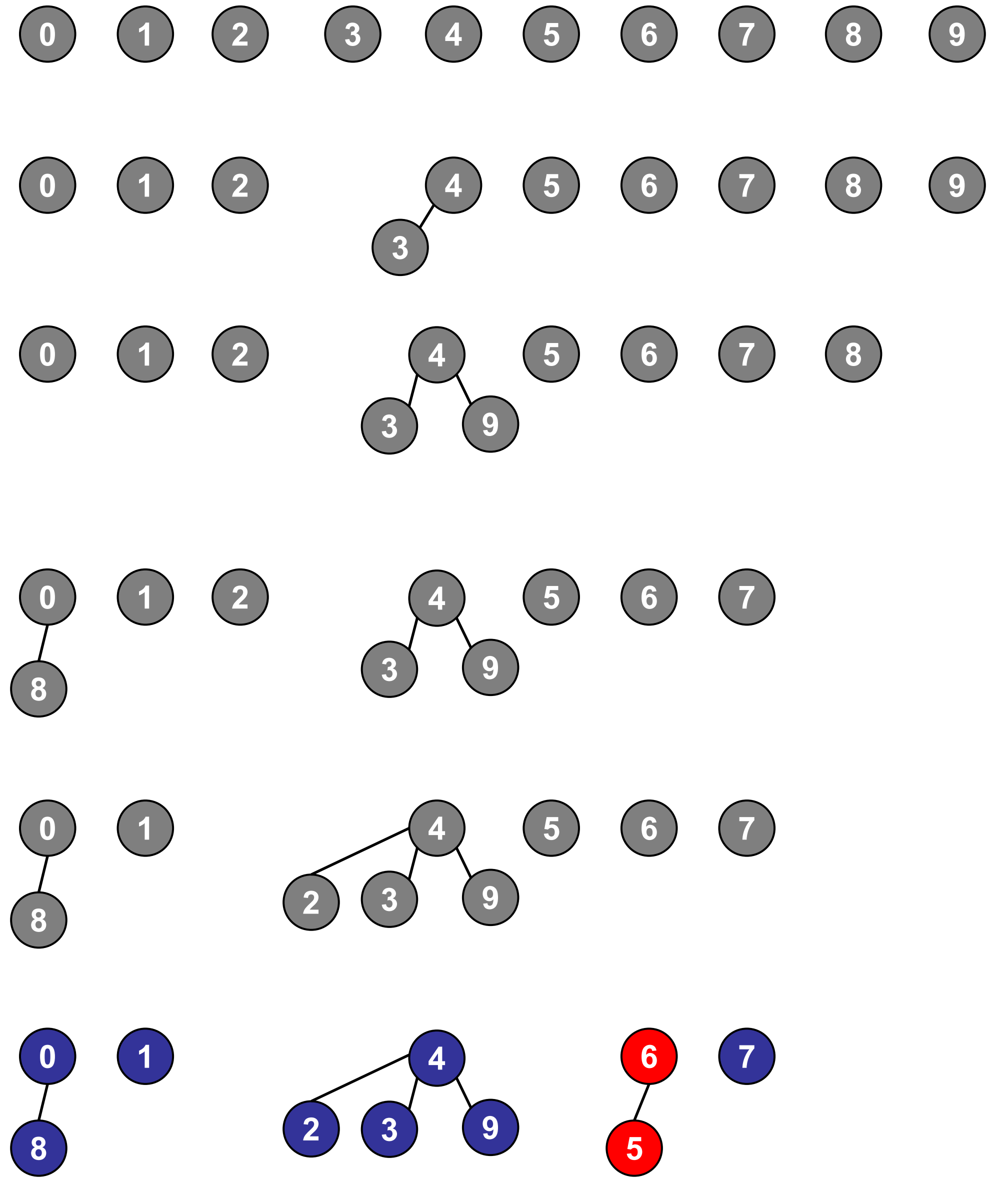
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4



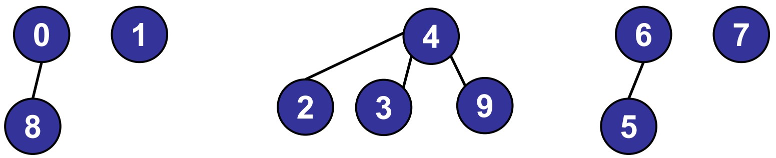
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4



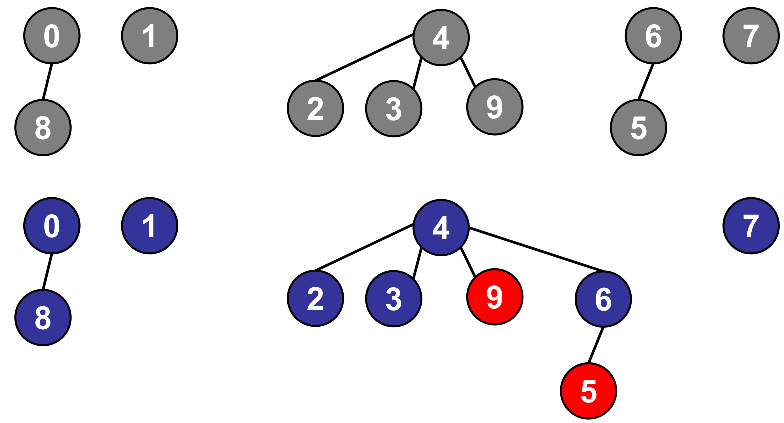
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4



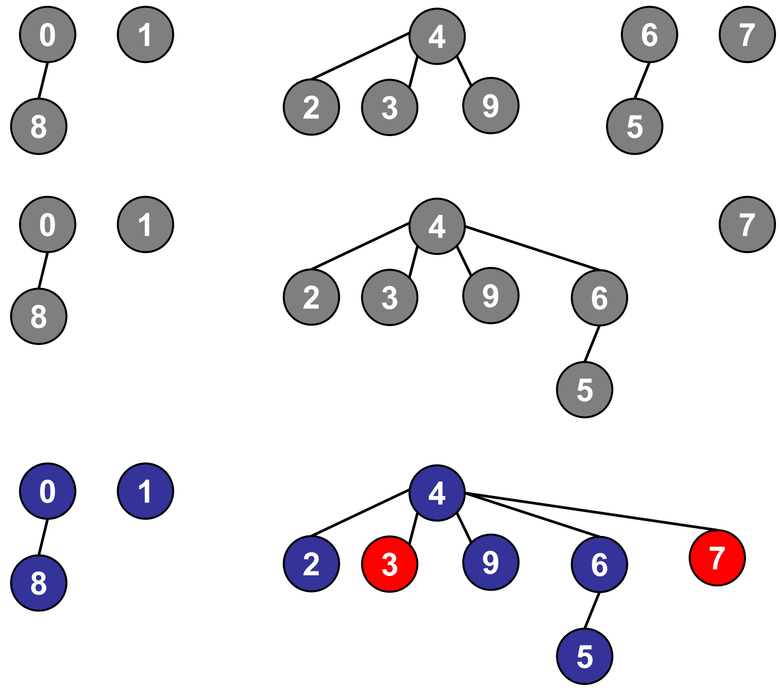
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4



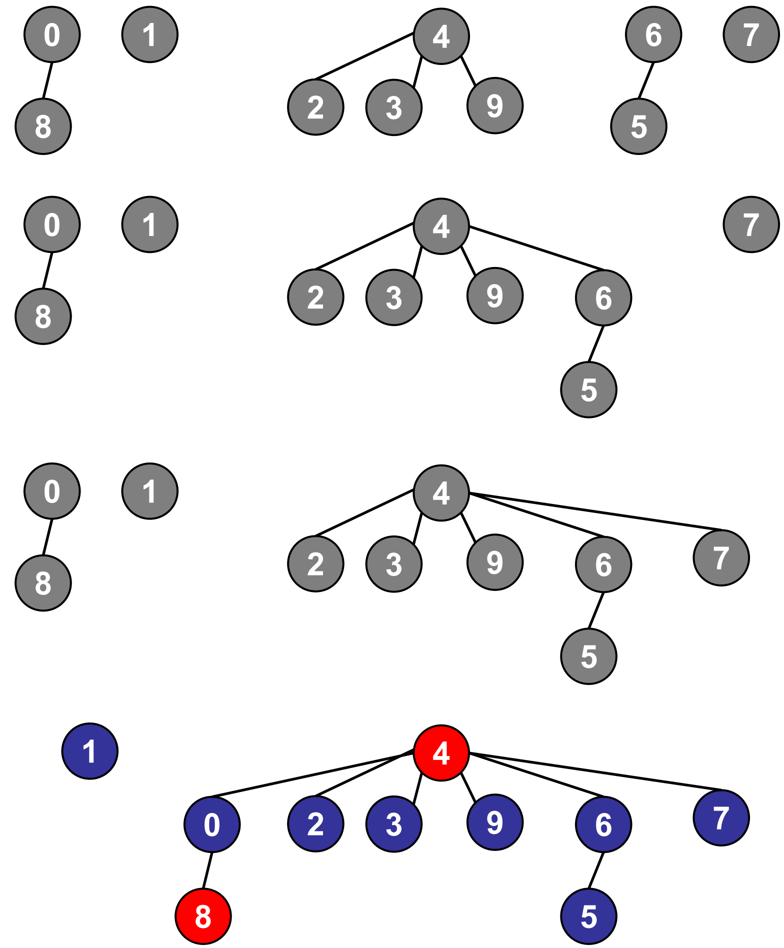
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4
7-3	0	1	4	4	4	6	4	4	0	4



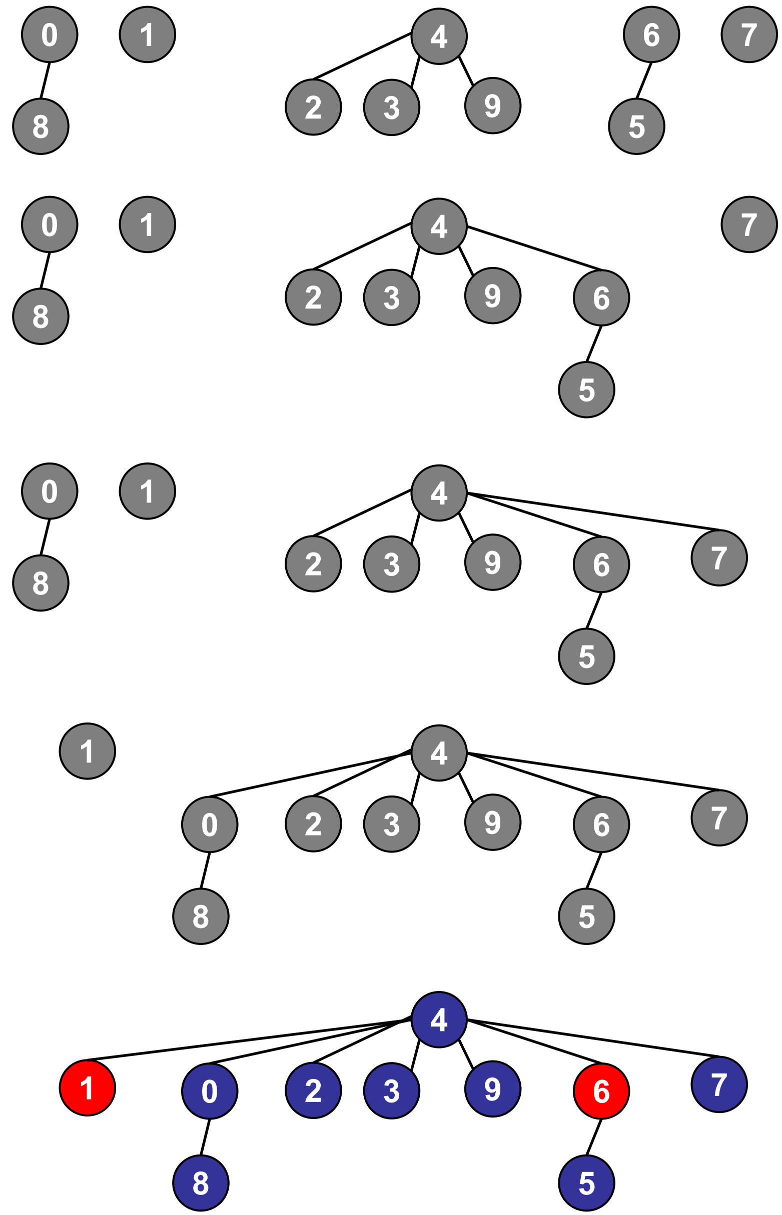
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4
7-3	0	1	4	4	4	6	4	4	0	4
4-8	4	1	4	4	4	6	4	4	0	4



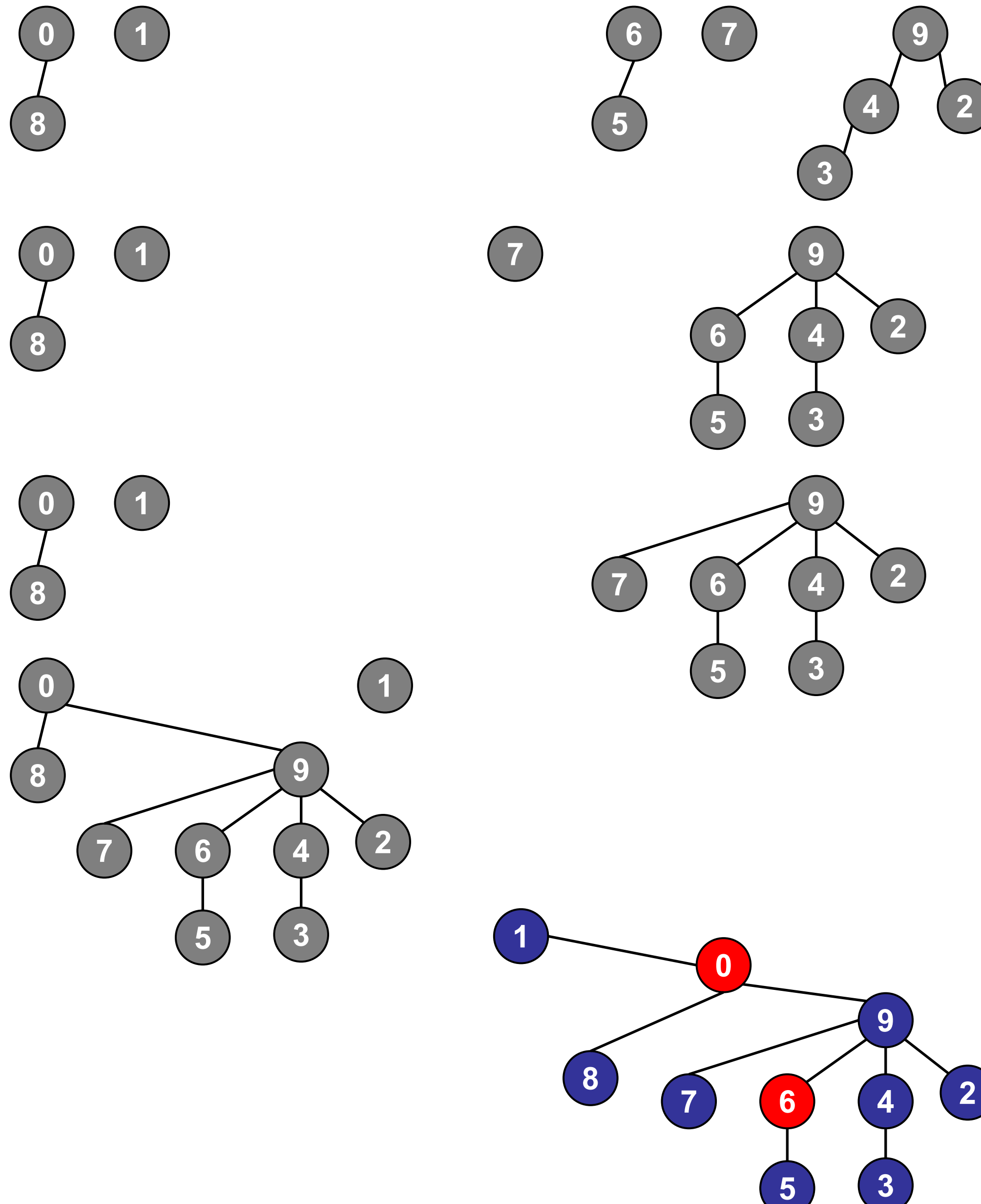
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4
7-3	0	1	4	4	4	6	4	4	0	4
4-8	4	1	4	4	4	6	4	4	0	4
6-1	4	4	4	4	4	6	4	4	0	4



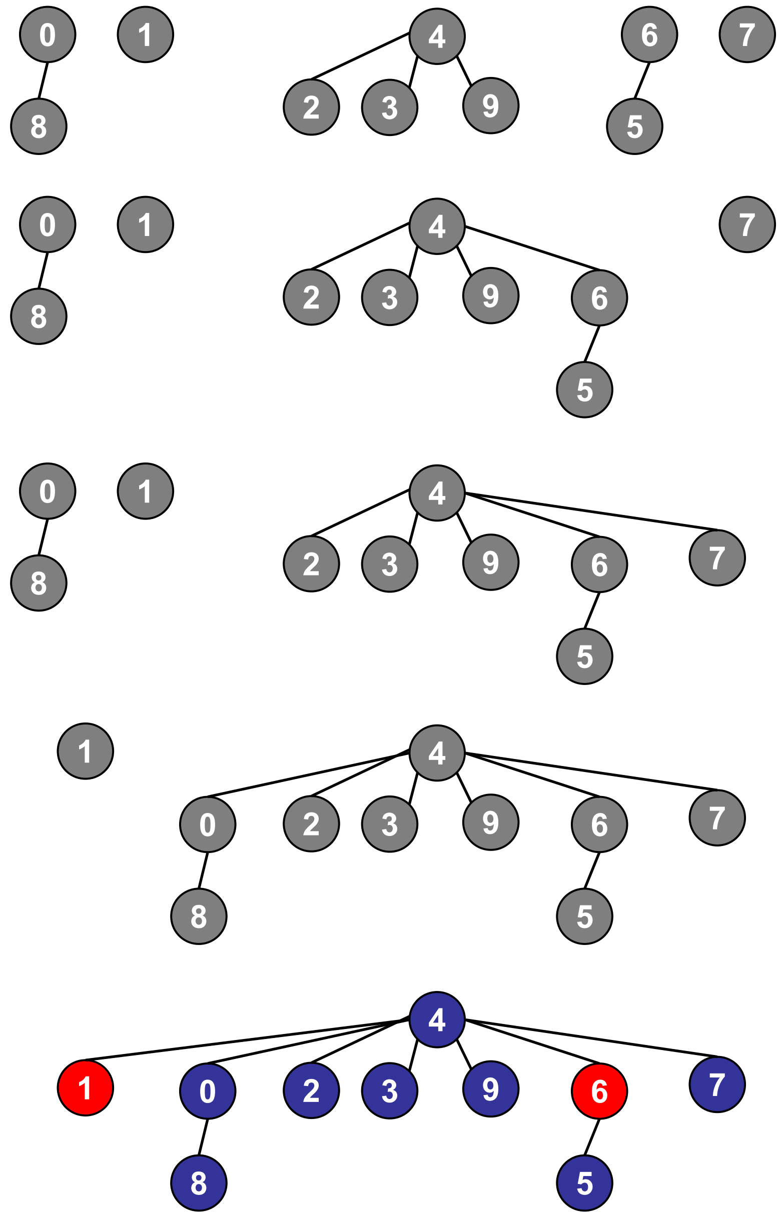
Example: (Unweighted) Quick Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	4	5	6	7	8	4
8-0	0	1	2	4	4	5	6	7	0	4
2-3	0	1	4	4	4	5	6	7	0	4
5-6	0	1	4	4	4	6	6	7	0	4
5-9	0	1	4	4	4	6	4	7	0	4
7-3	0	1	4	4	4	6	4	4	0	4
4-8	4	1	4	4	4	6	4	4	0	4
6-1	4	4	4	4	4	6	4	4	0	4



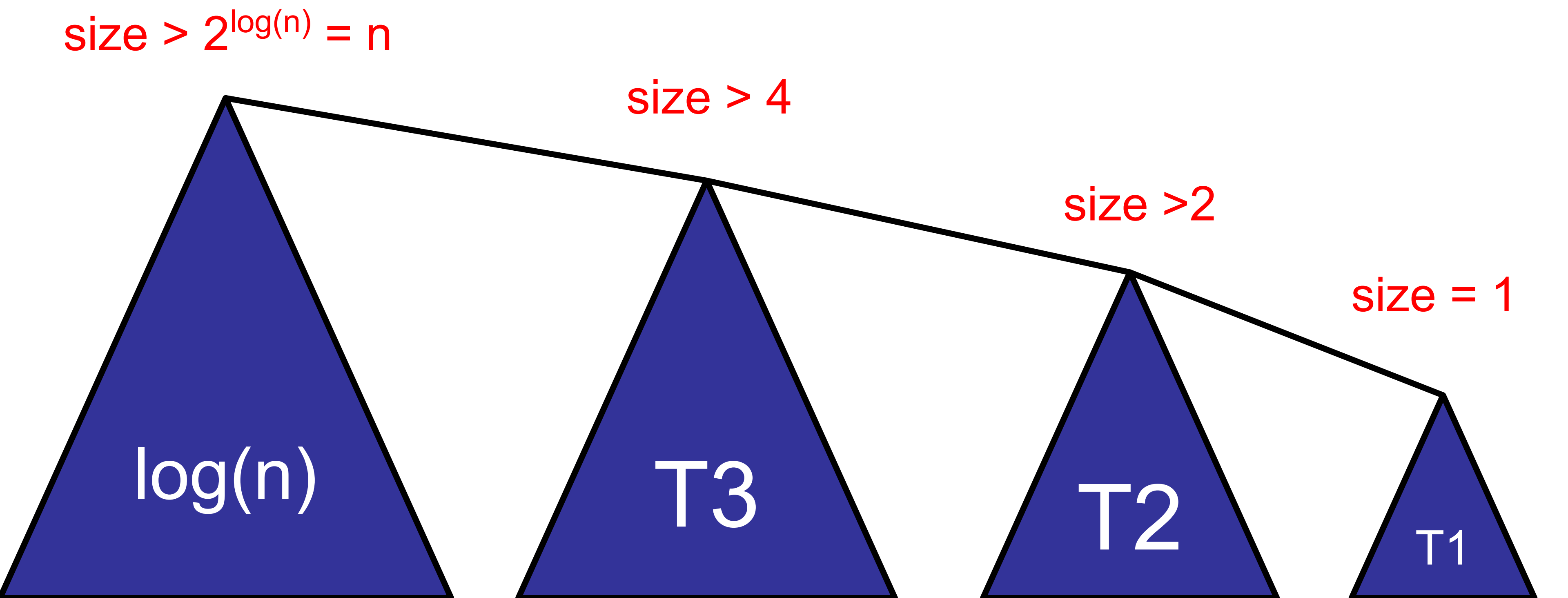
Maximum depth of tree?

1. $O(1)$
2. $O(\log n)$
3. $O(n)$
4. $O(n \log n)$
5. $O(n^2)$
6. None of the above.

Weighted Union

Key idea:

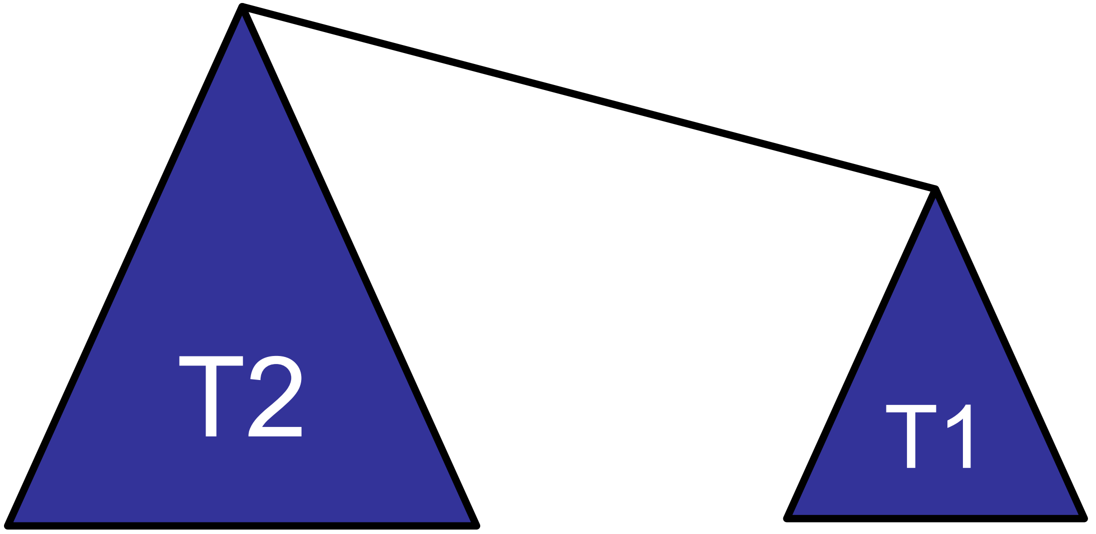
height only increases when total size doubles



Weighted Union

Analysis:

– Base case: tree of height 0 contains 1 object.

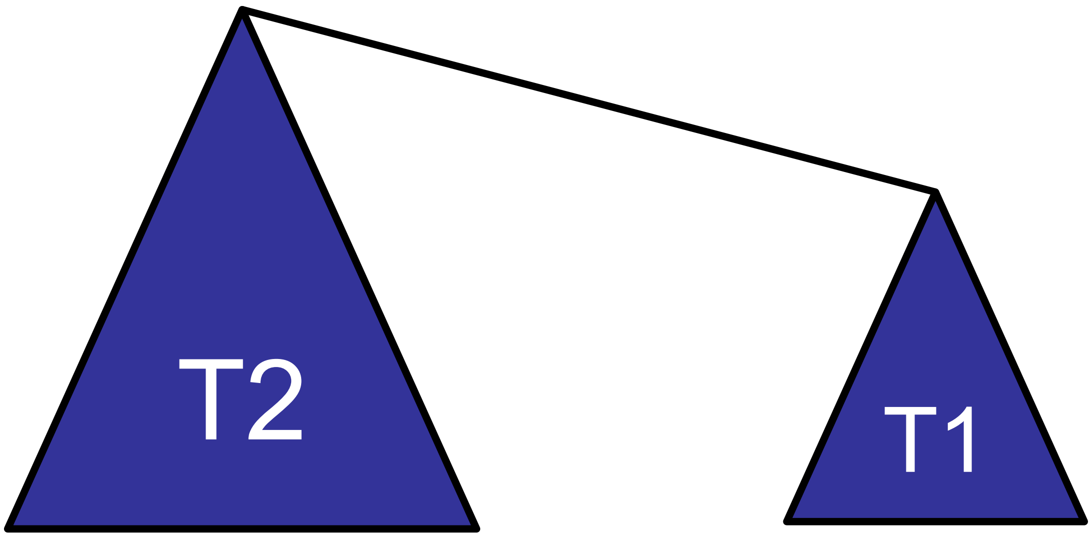


Weighted Union

Claim:

A tree of height k has size at least 2^k.

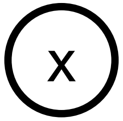
☐ height of tree of size n is at most log(n)



Weighted Union

Analysis:

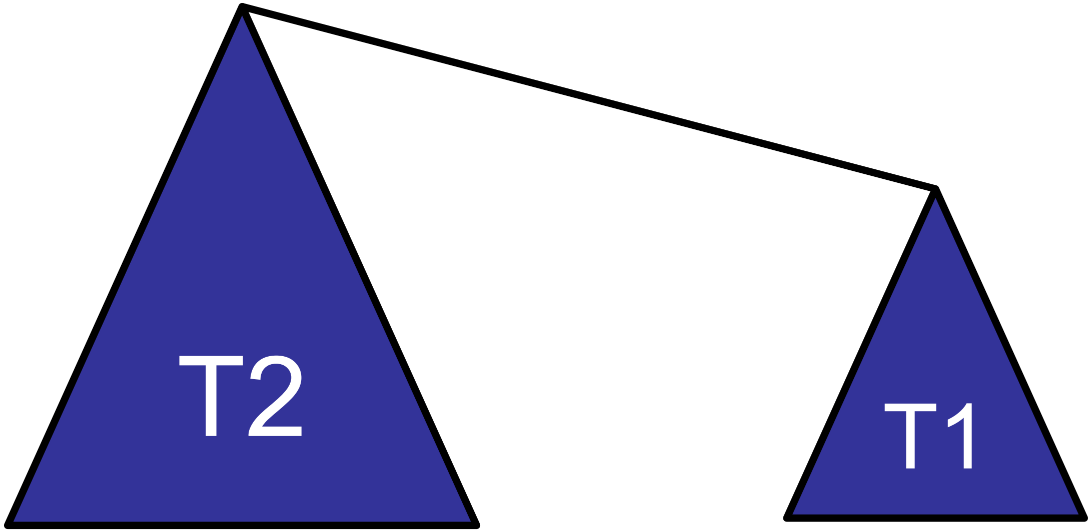
– Base case: tree of height 0 contains 1 object.



Weighted Union

Induction:

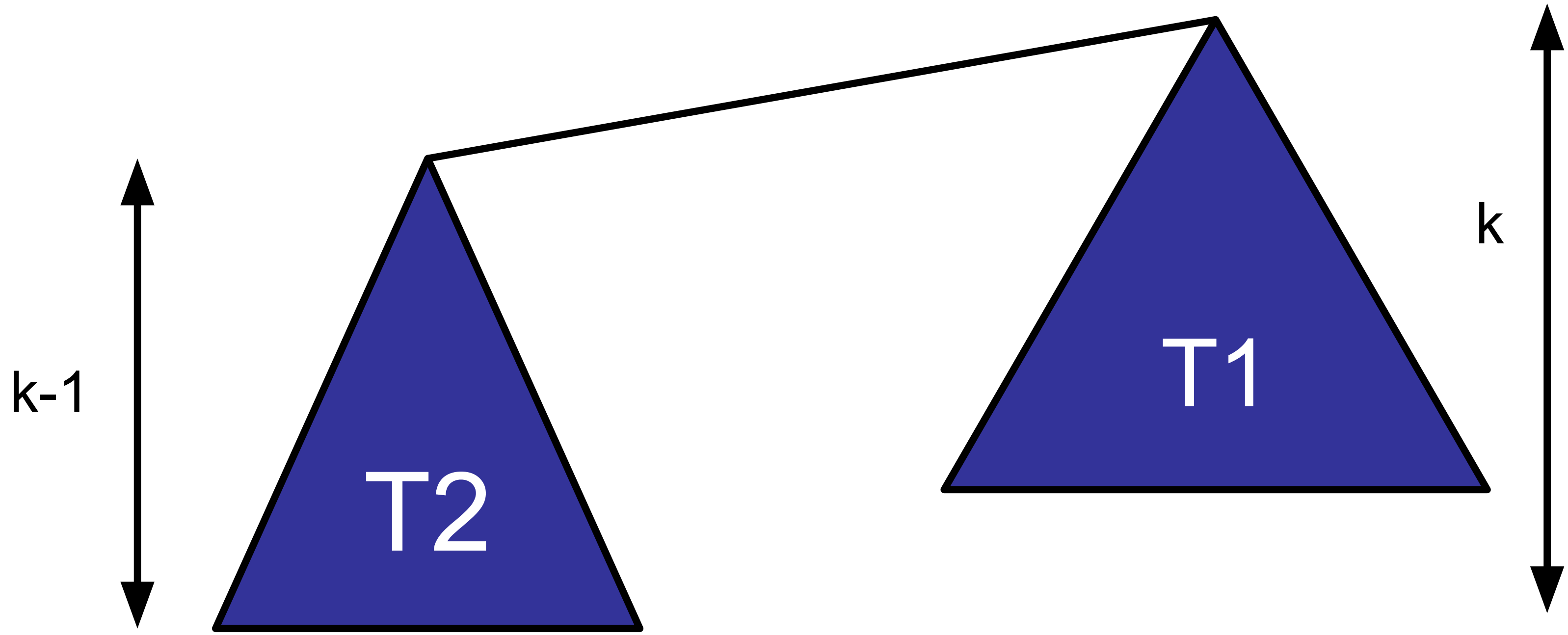
- Assume: A tree of height k-1 has size at least 2^{k-1}.
- Show: A tree of height k has size at least 2^k.



Weighted Union

How do you get a tree of height k?

Make tree of height (k-1) the child of another tree.

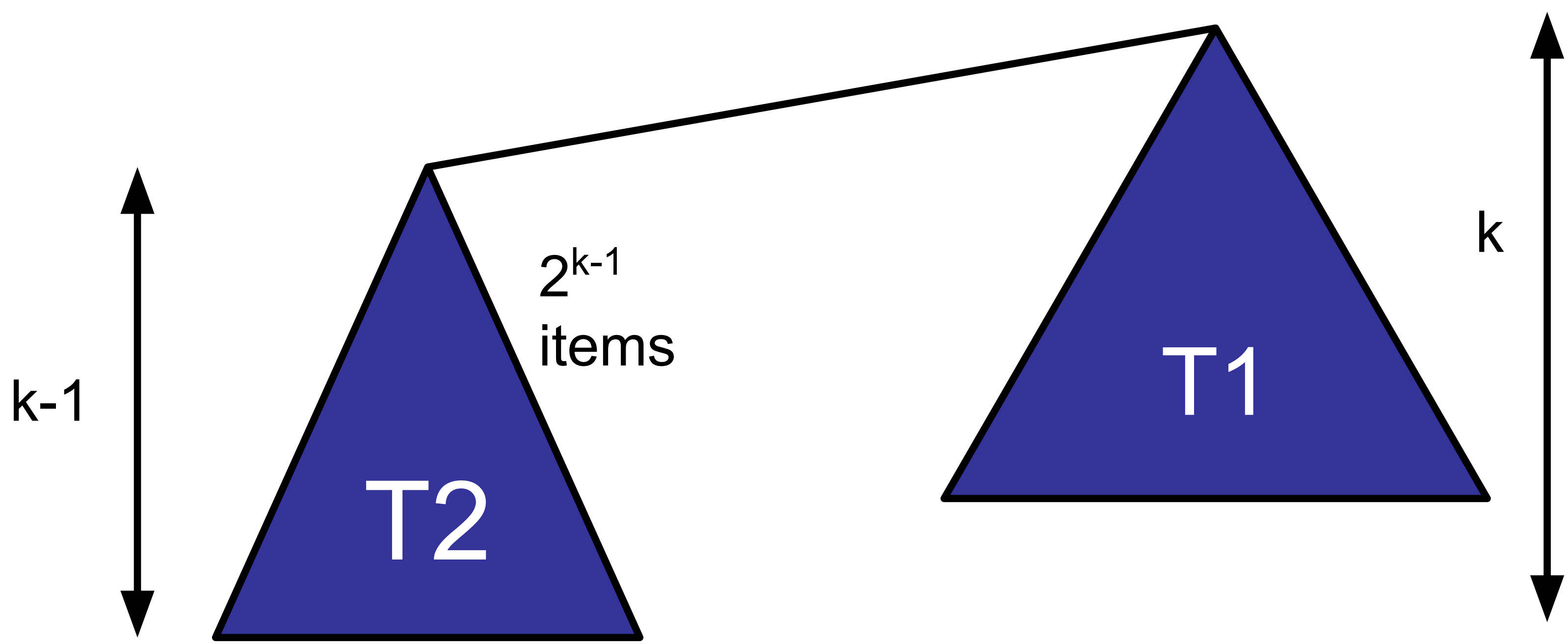


Weighted Union

How do you get a tree of height k?

Make tree of height (k-1) the child of another tree.

Tree T2 has size at least 2^{k-1} by induction.

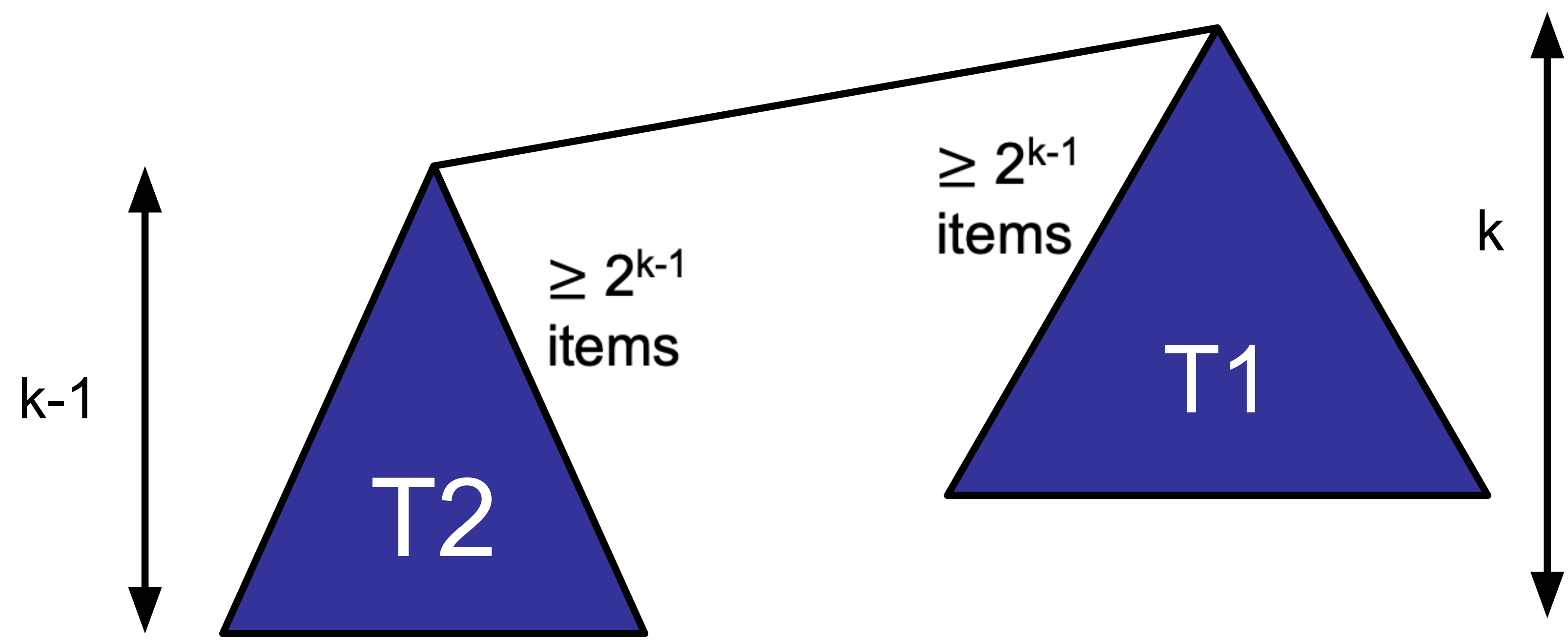


Weighted Union

- How do you get a tree of height k?

Tree T2 has size at least 2^{k-1} by induction.

➔ $\text{size}[T1] \geq \text{size}[T2] \geq 2^{k-1}$ by union-by-weight-rule



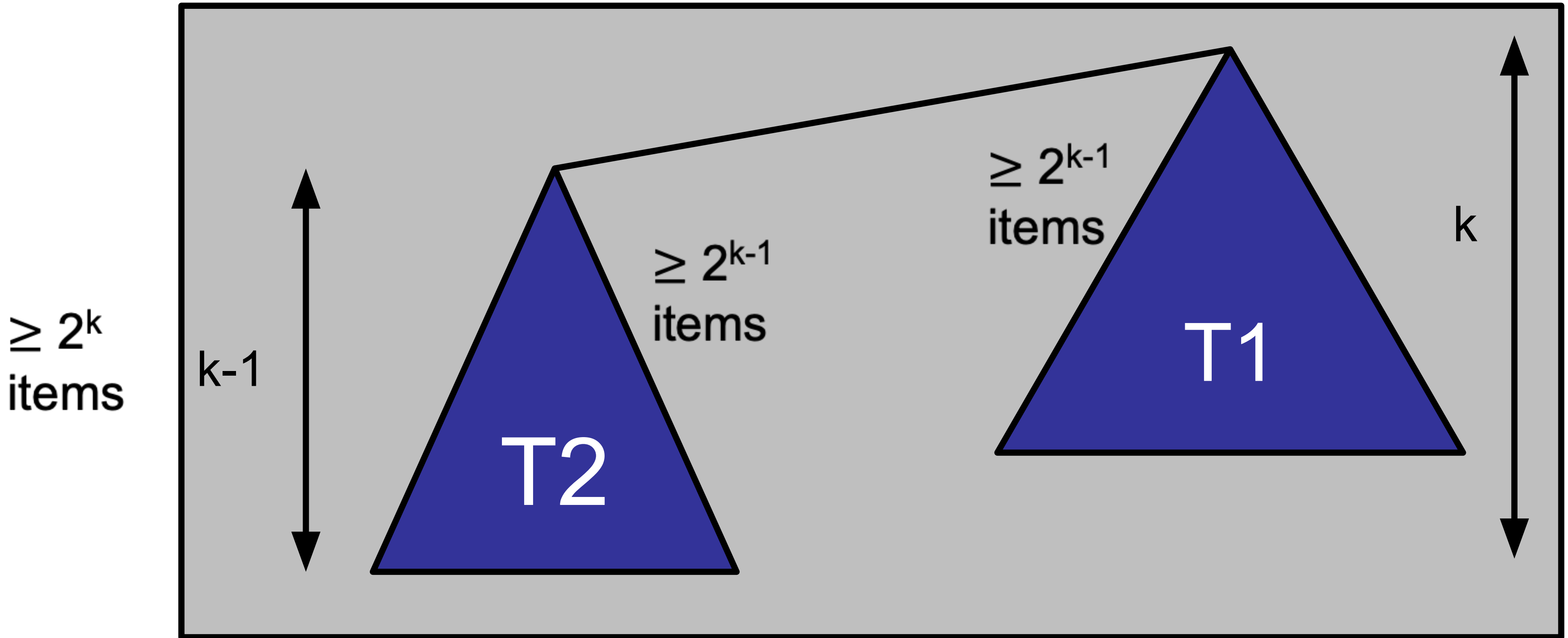
Weighted Union

- How do you get a tree of height k?

Tree T2 has size at least 2^{k-1} by induction.

➔ $\text{size}[T1] \geq \text{size}[T2] \geq 2^{k-1}$ by union-by-weight-rule

➔ $\text{size}[T1 + T2] \geq 2^{k-1} + 2^{k-1} \geq 2^k$

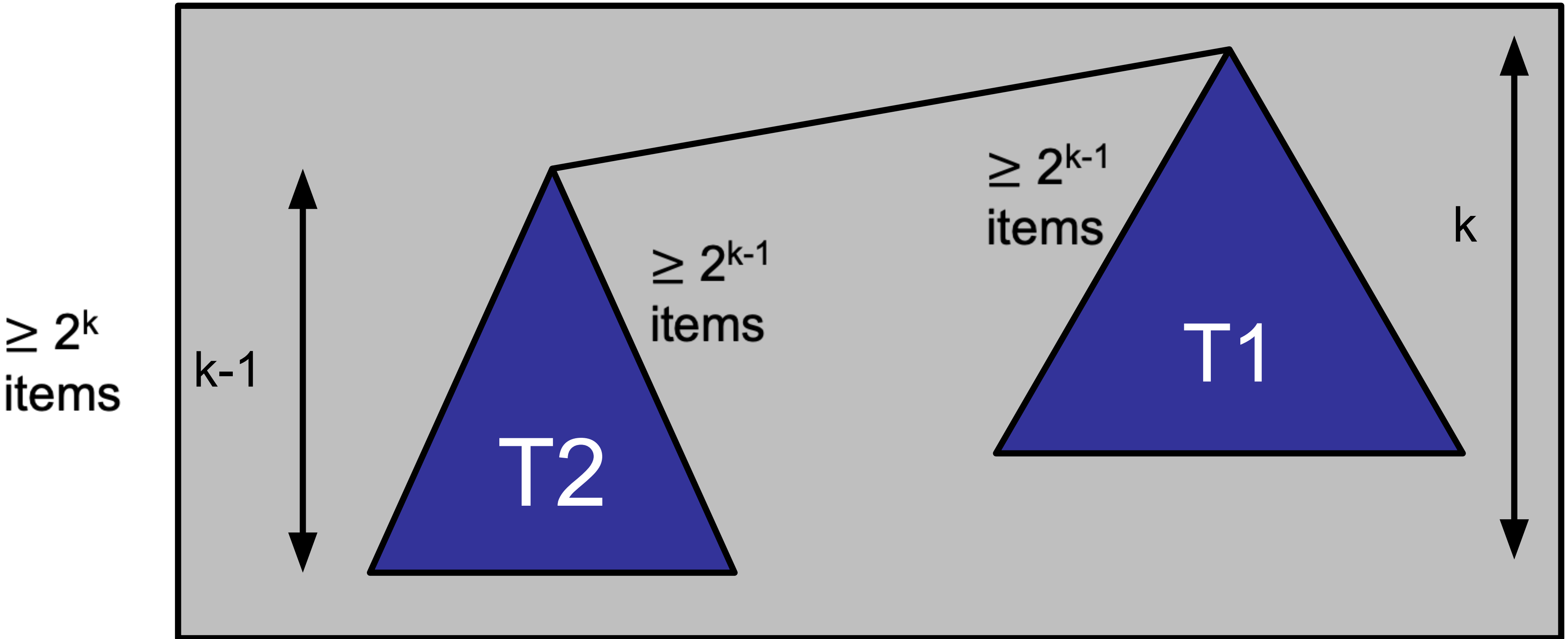


Weighted Union

Claim:

A tree of height k has size at least 2^k.

□ height of tree of size n is at most log(n)



Running time of (Find, Union):

1. O(1), O(1)
2. O(1), O(n)
3. O(n), O(1)
4. O(n), O(n)
5. O(log n), O(log n)
6. None of the above.

Weighted Union

```
union(int p, int q) {
  while (parent[p] !=p) p = parent[p];
  while (parent[q] !=q) q = parent[q];
  if (size[p] > size[q] {
      parent[q] = p;   // Link q to p
      size[p] = size[p] + size[q];
  }
  else {
      parent[p] = q; // Link p to q
      size[q] = size[p] + size[q];
  }
}
```

Union-Find Summary

Ver 1 and Ver 2 are slow:

- Union and/or find is expensive
- Quick-union: tree is too deep

Weighted-union is faster:

- Trees too balanced: O(log n)
- Union *and* find are O(log n)

	find	union
Ver 1	O(1)	O(n)
Ver 2	O(n)	O(n)
Ver 2	O(log n)	O(log n)

Union-Find Summary

Notes:

- Some prefer union-by-rank (where rank = log(size))
- Some prefer union-by-height (same idea)

Important property:

- weight/rank/size/height of subtree does not change except at root (so only update root on union).
- weight/rank/size/height only increases when tree size doubles.

Union-Find Summary

Notes

- Son… un… og(size))
- Some…

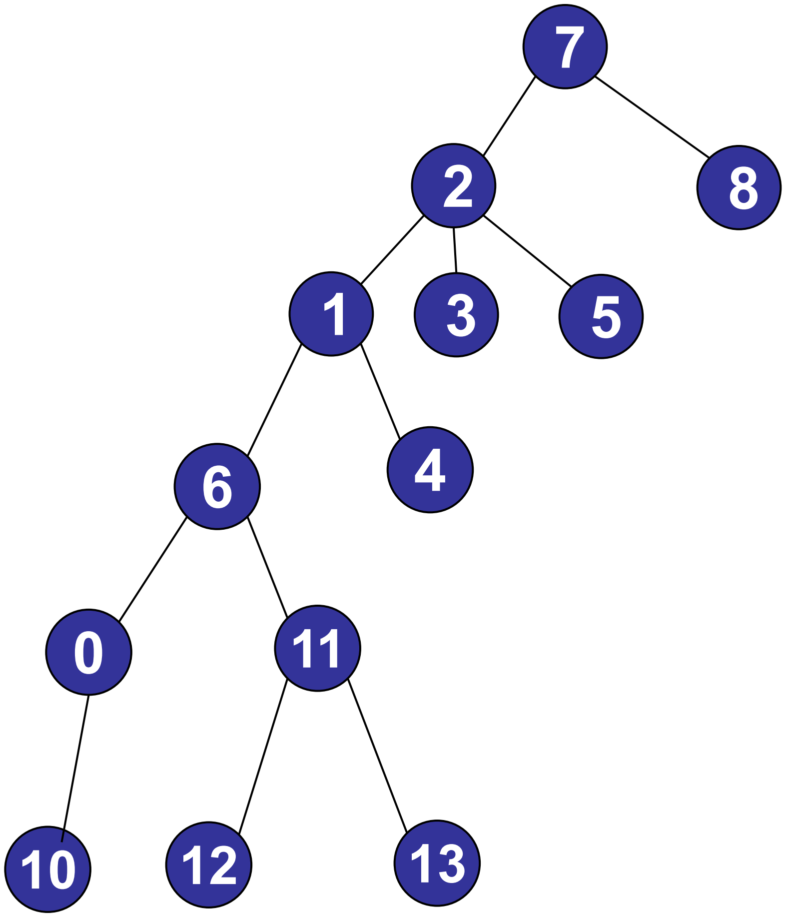
Importa…

- w… t chang… xcept at root (s…
- weight/ra… size… t of… eases w… tree size doubles.

But wait!
We can go even faster!

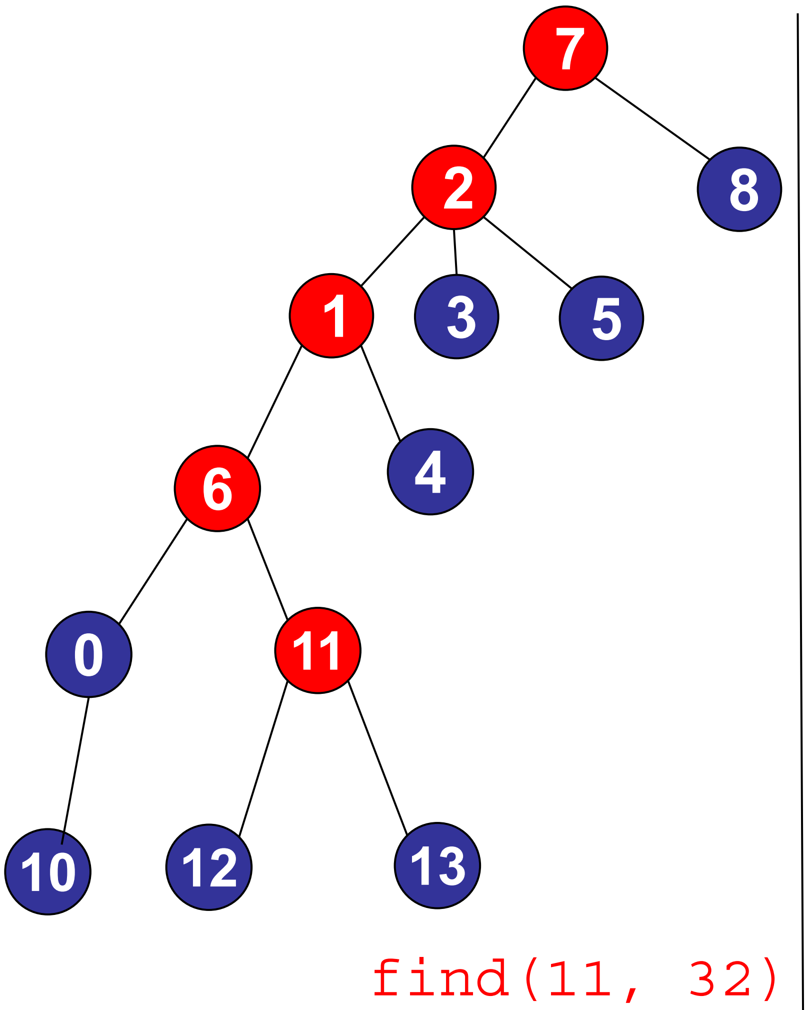
Path Compression

After finding the root: set the parent of each traversed node to the root.



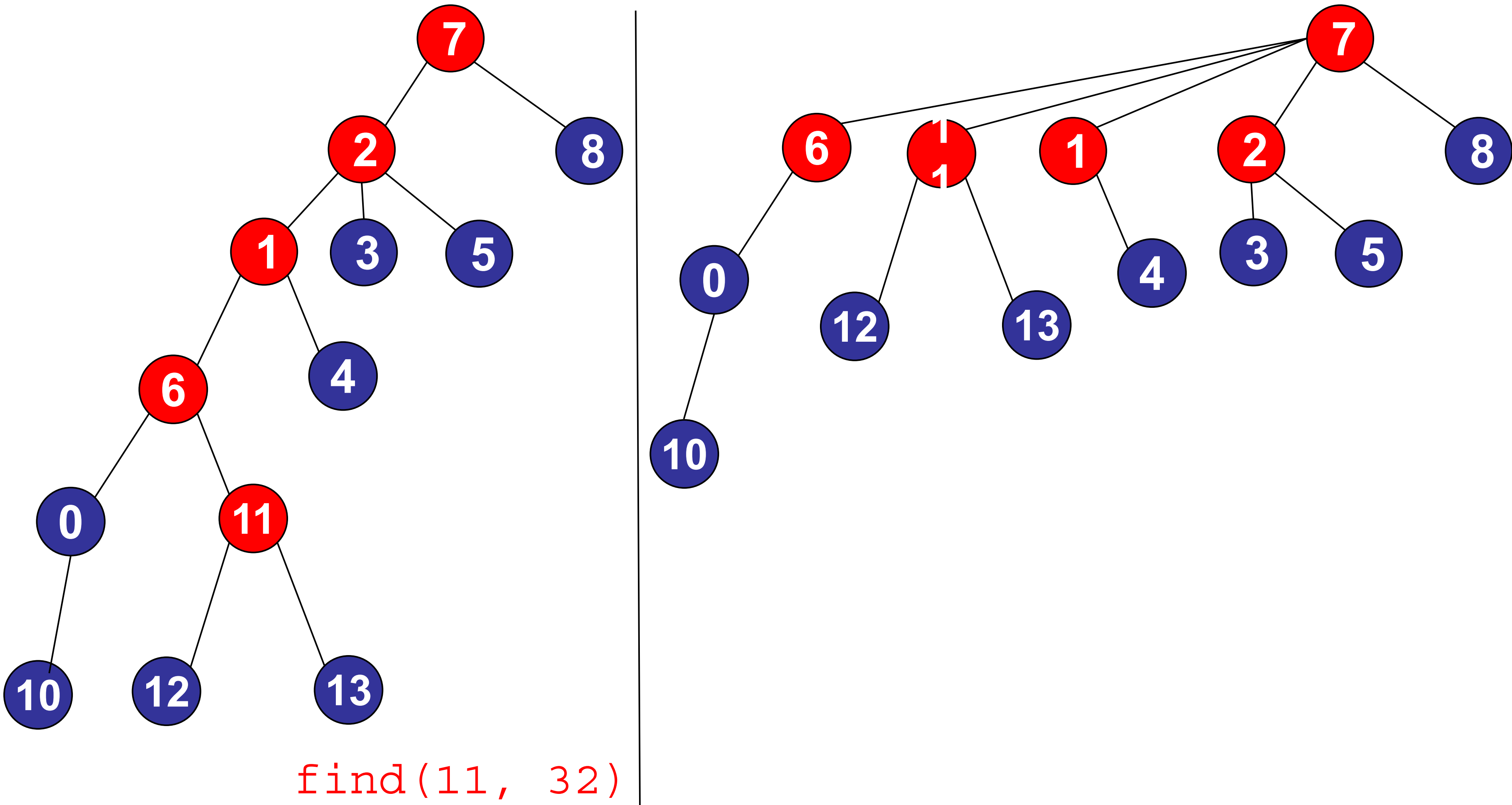
Path Compression

After finding the root: set the parent of each traversed node to the root.



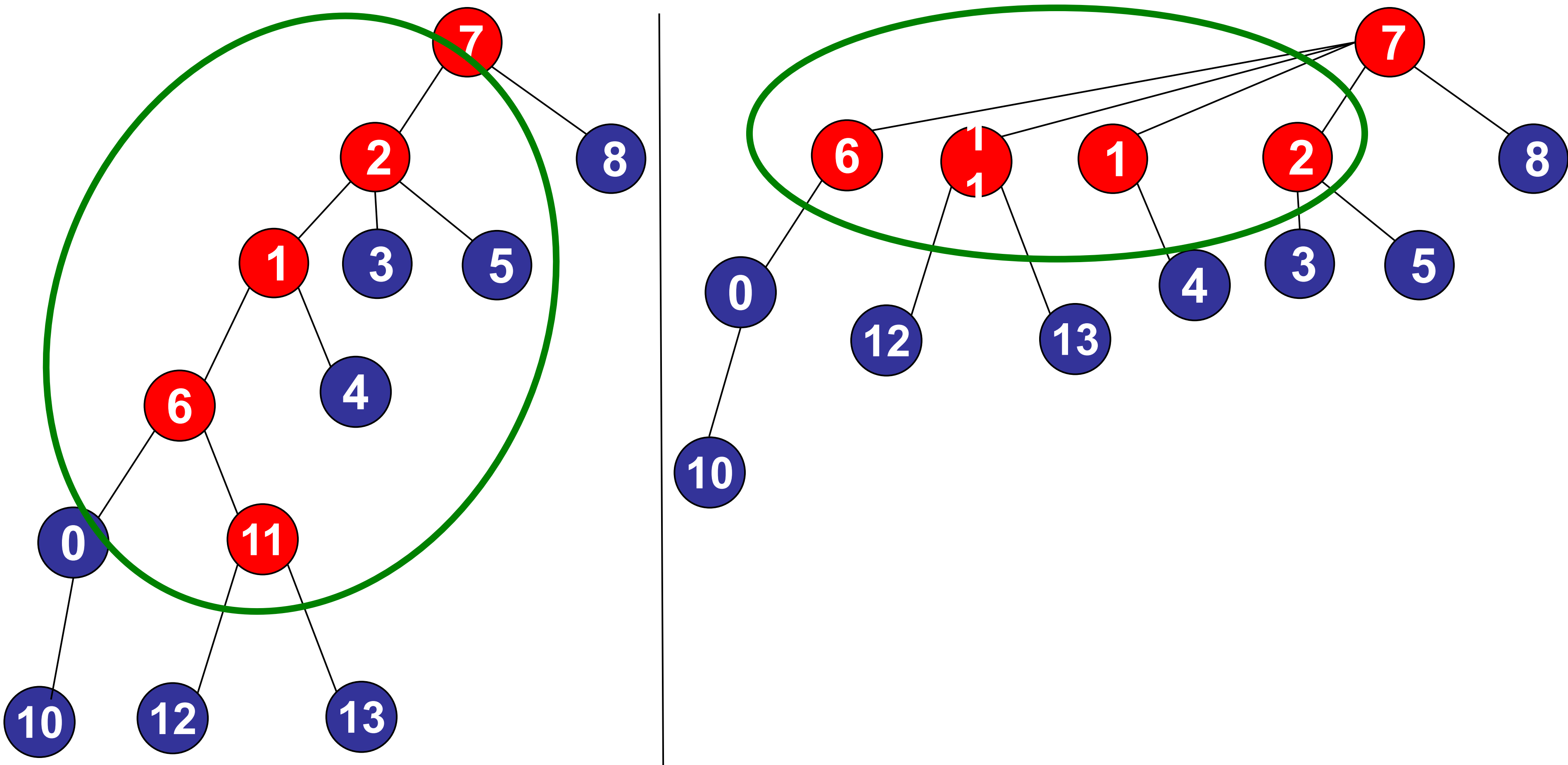
Path Compression

After finding the root: set the parent of each traversed node to the root.



Path Compression

After finding the root: set the parent of each traversed node to the root.



`find(11, 32)`

Path Compression: Old Algorithm

```
findRoot(int p) {
   root = p;
   while (parent[root] != root) root = parent[root];
   return root;
}
```

Path Compression

```
findRoot(int p) {
   root = p;
   while (parent[root] != root) root = parent[root];
   while (parent[p] != p) {
         temp = parent[p];
         parent[p] = root;
         p = temp;
   }
   return root;
}
```

Alternative Path Compression

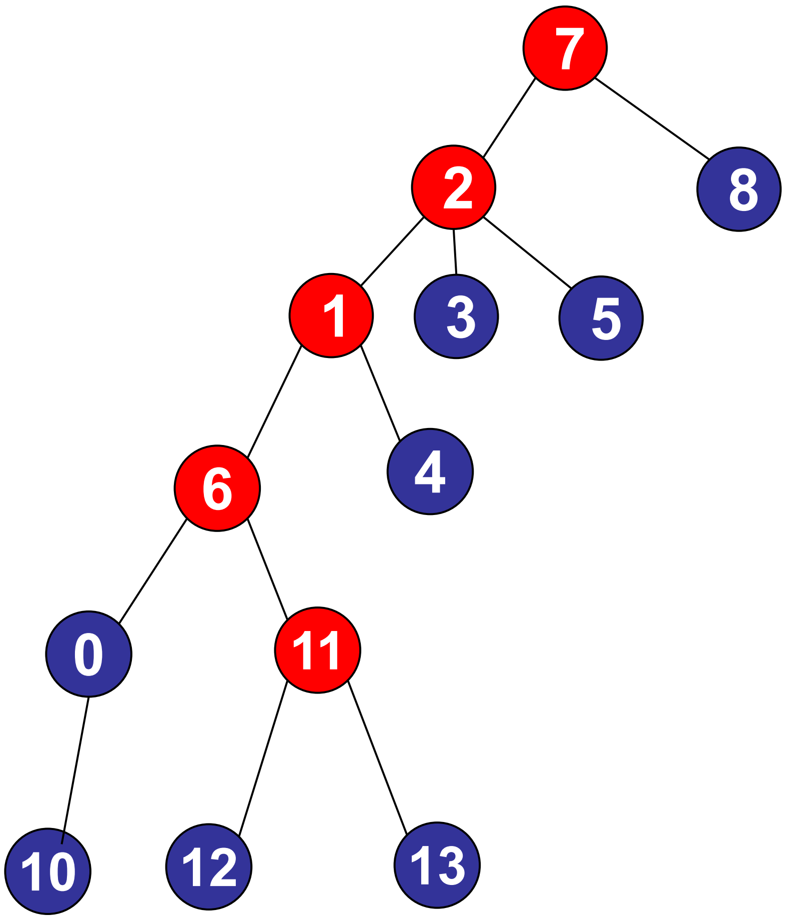
```
findRoot(int p) {
   root = p;
   while (parent[root] != root) {
        parent[root] = parent[parent[root]];
        root = parent[root];
   }
   return root;
}
```

Make every other node in the path point to its grandparent!

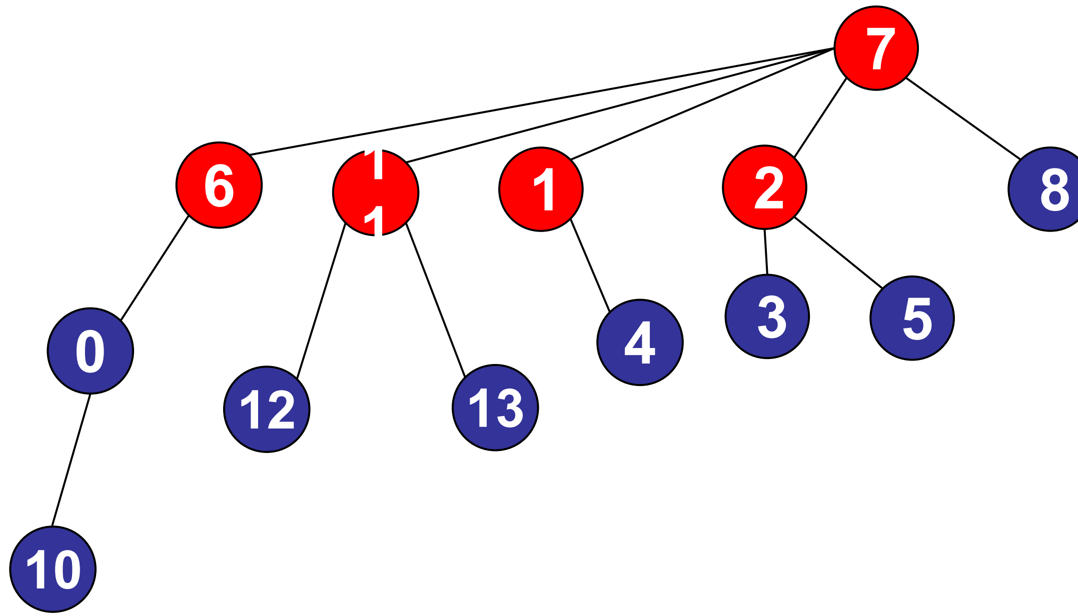
- Simple
- Works as well!

Path Compression

After finding the root: set the parent of each traversed node to the root.



`find(11, 32)`



How fast does it run now?

Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

Inverse Ackermann function: always ≤ 5 in this universe.

n	α(n, n)
4	0
8	1
32	2
8,192	3
2^{65533}	4

Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

Proof:

Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

Proof:

- Very difficult.
- Algorithm: very simple to implement.

Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

Proof:

- Very difficult.
- Algorithm: very simple to implement.

Can we do better?

Weight Union with Path Compression

Theorem: [Tarjan 1975]

Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

Proof:

- Very difficult.
- Algorithm: very simple to implement.

Can we do better? No!

- Proof: Fredman and Saks 1989 The cell probe complexity of dynamic data structures

Union-Find Summary

Weighted-union is faster:

- Trees are flat: O(log n)
- Union *and* find are O(log n)

Weighted Union + Path Compression is very fast:

- Trees very flat.
- On average, almost linear performance per operation.

	find	**union**
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
weighted-union with path-compression	$\alpha(m, n)$	$\alpha(m, n)$

Union-Find Summary

Path Compression **without** weighted union?

	find	union
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
path compression	O(log n)	O(log n)
weighted-union with path-compression	$\alpha(m, n)$	$\alpha(m, n)$

Union-Find Summary

Path Compression **without** weighted union?

	find	**union**
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)
path compression	O(log n)	O(log n)
weighted-union with path-compression	α(m, n)	α(m, n)

Why Union Find?

What's the point of union find? What if I don't care about breaking walls in mazes?

Why Union Find?

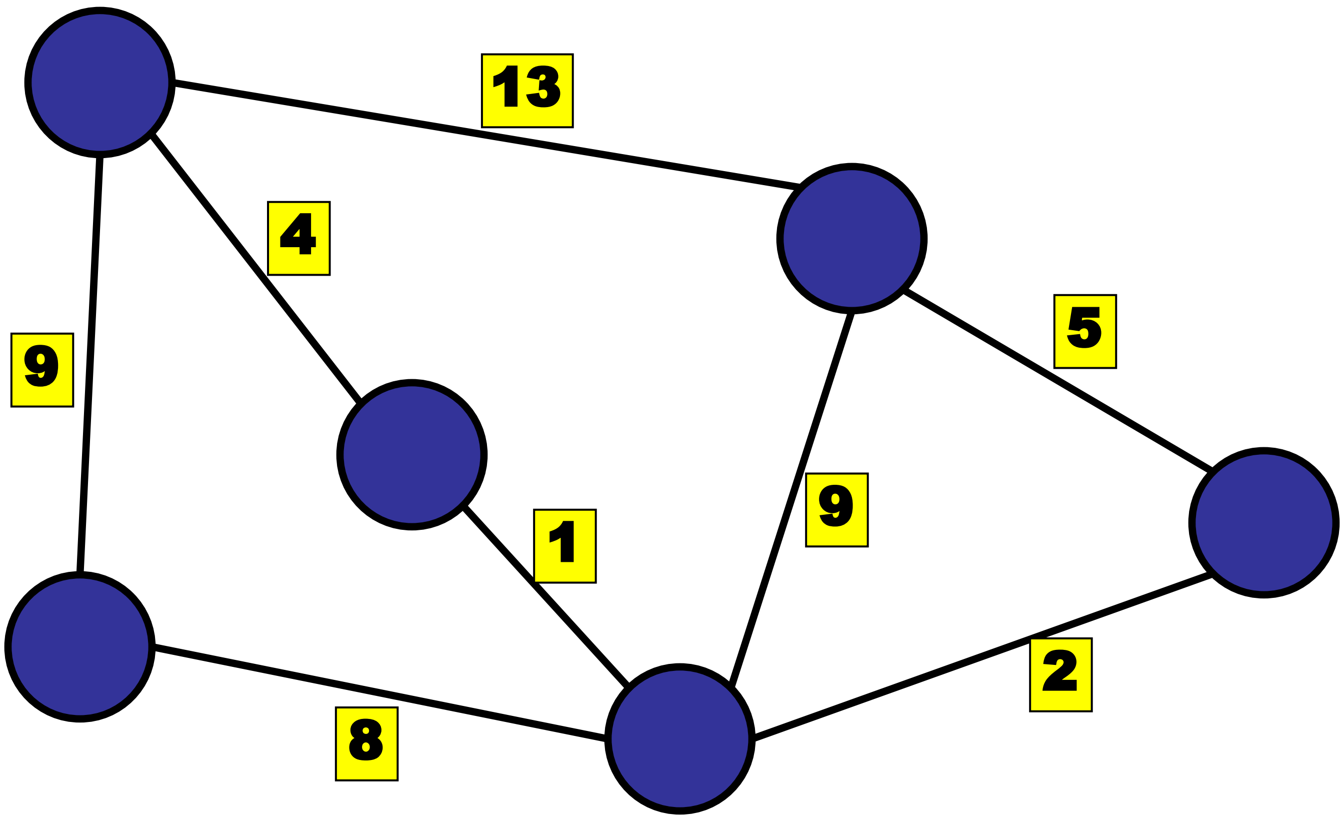
What's the point of union find? What if I don't care about breaking walls in mazes?

Remember MST and Kruskal's from CS1231?

Spanning Tree

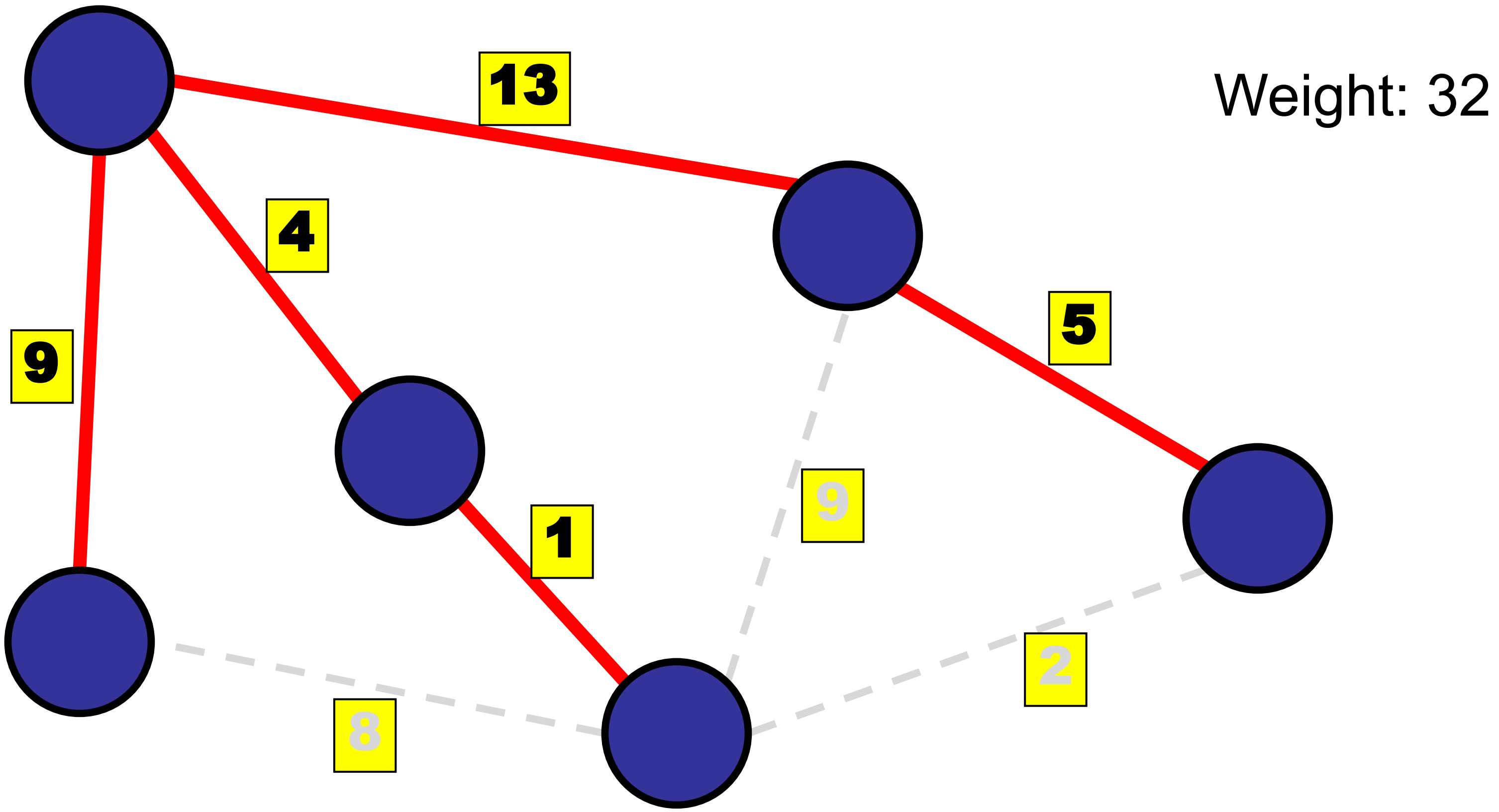
Weighted, <u>undirected</u> graph:

<u>To think about:</u>
Why is this more complicated with directed graphs?



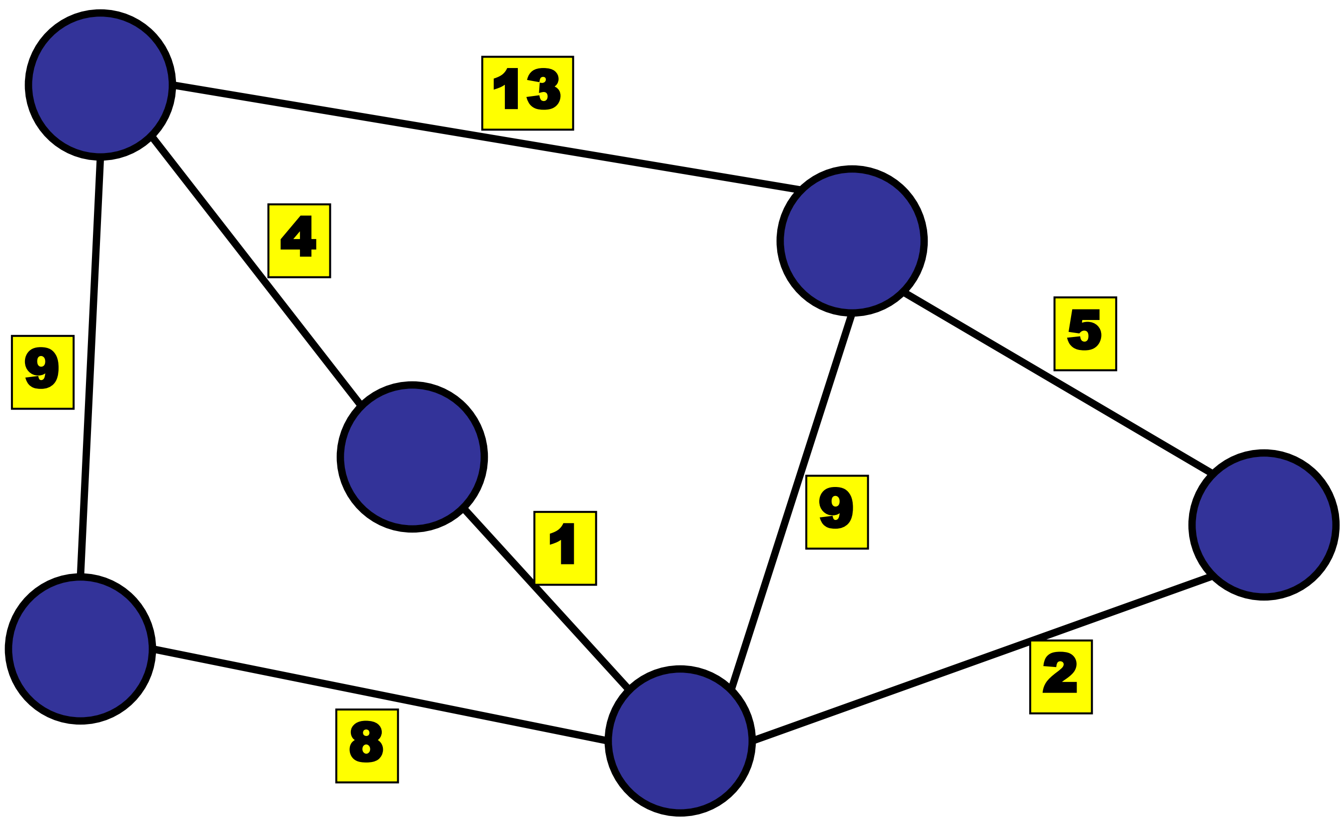
Spanning Tree

Definition: a spanning tree is an acyclic subset of the edges that connects all nodes



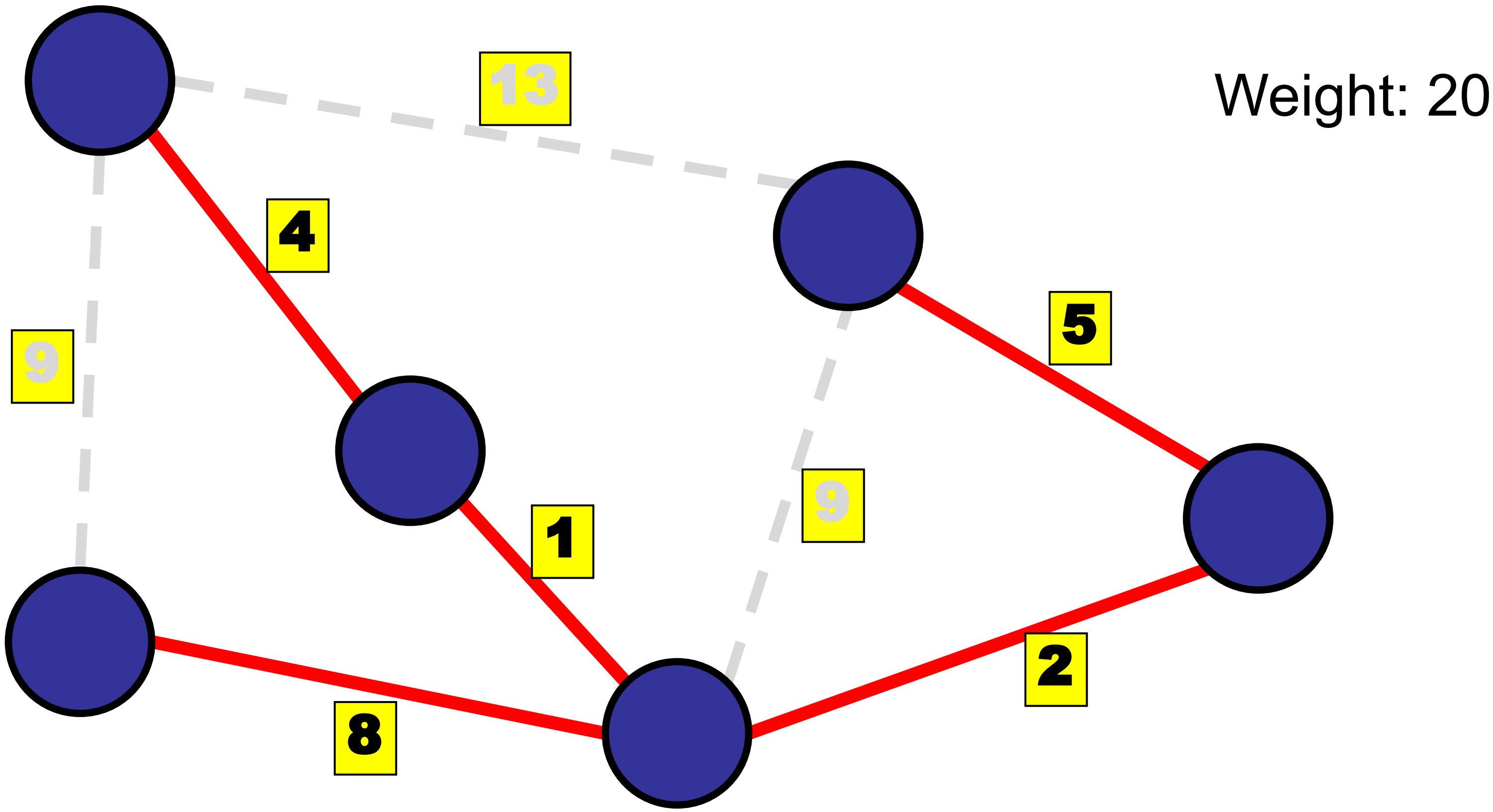
Minimum Spanning Tree

Definition: a spanning tree with minimum weight



Minimum Spanning Tree

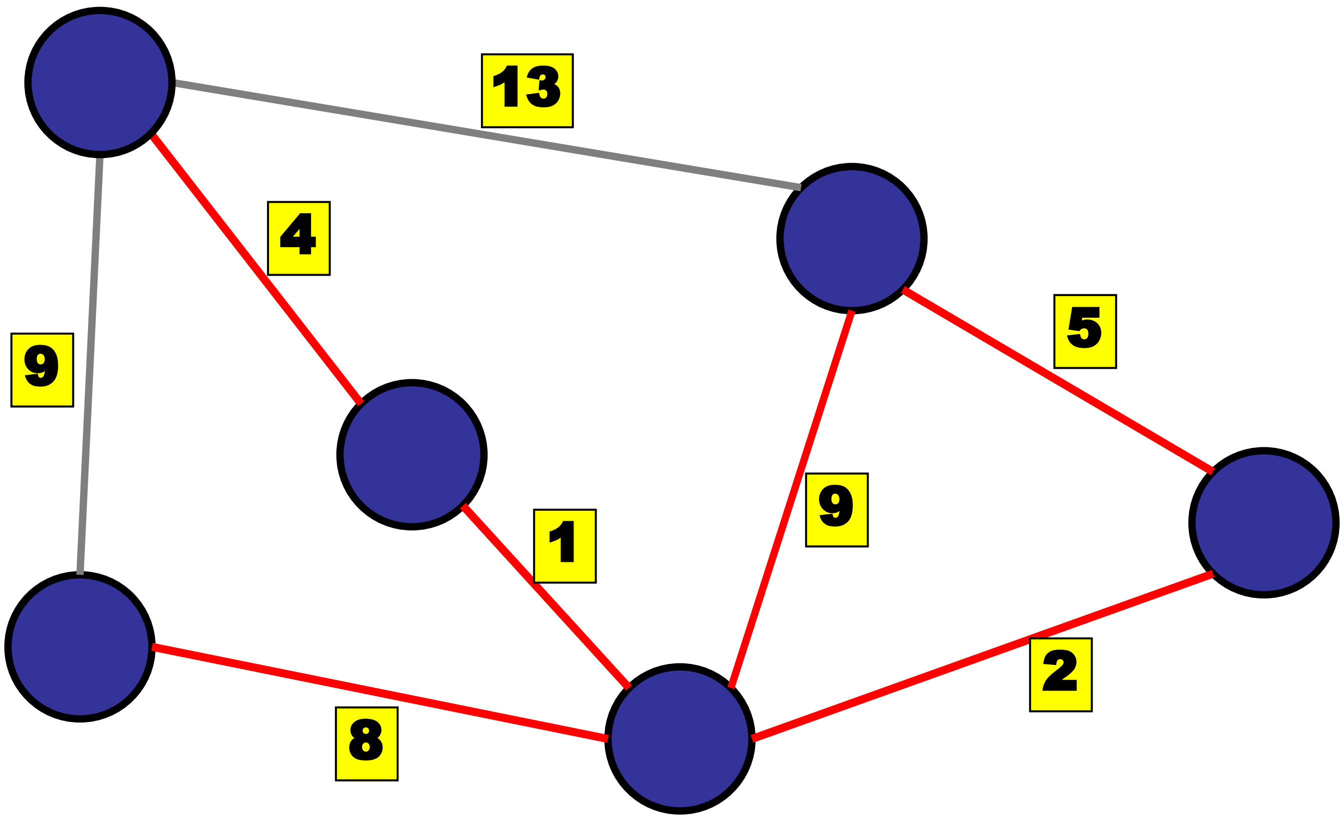
Definition: a spanning tree with minimum weight



Minimum Spanning Tree

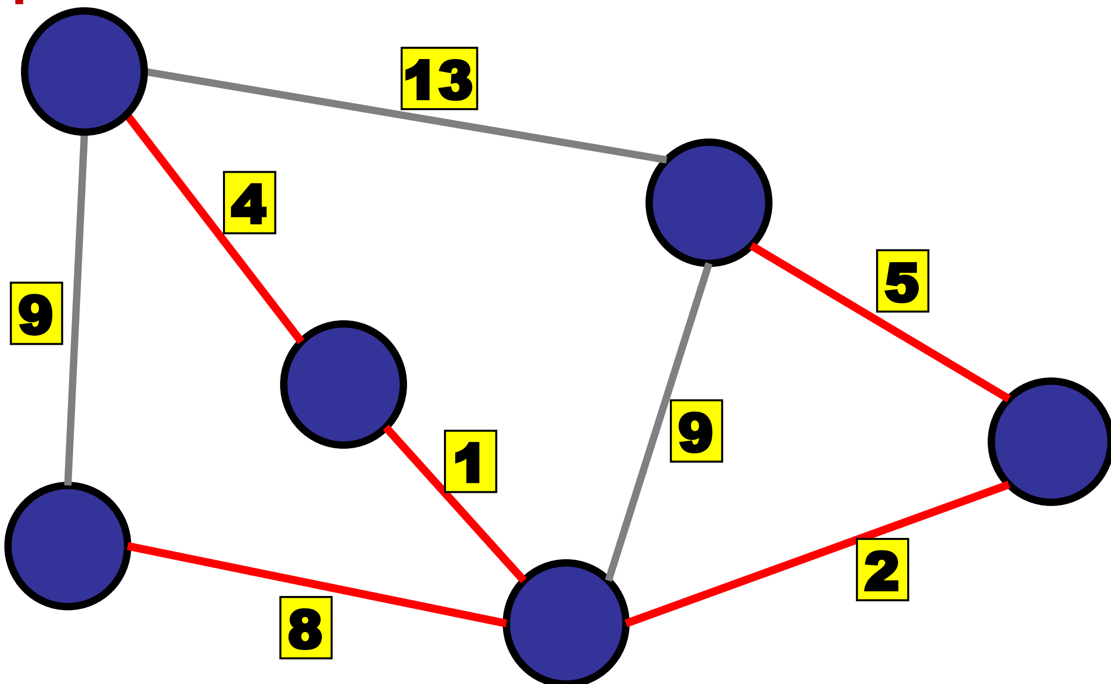
Note: no cycles

Why? If there were cycles, we could remove one edge and reduce the weight!



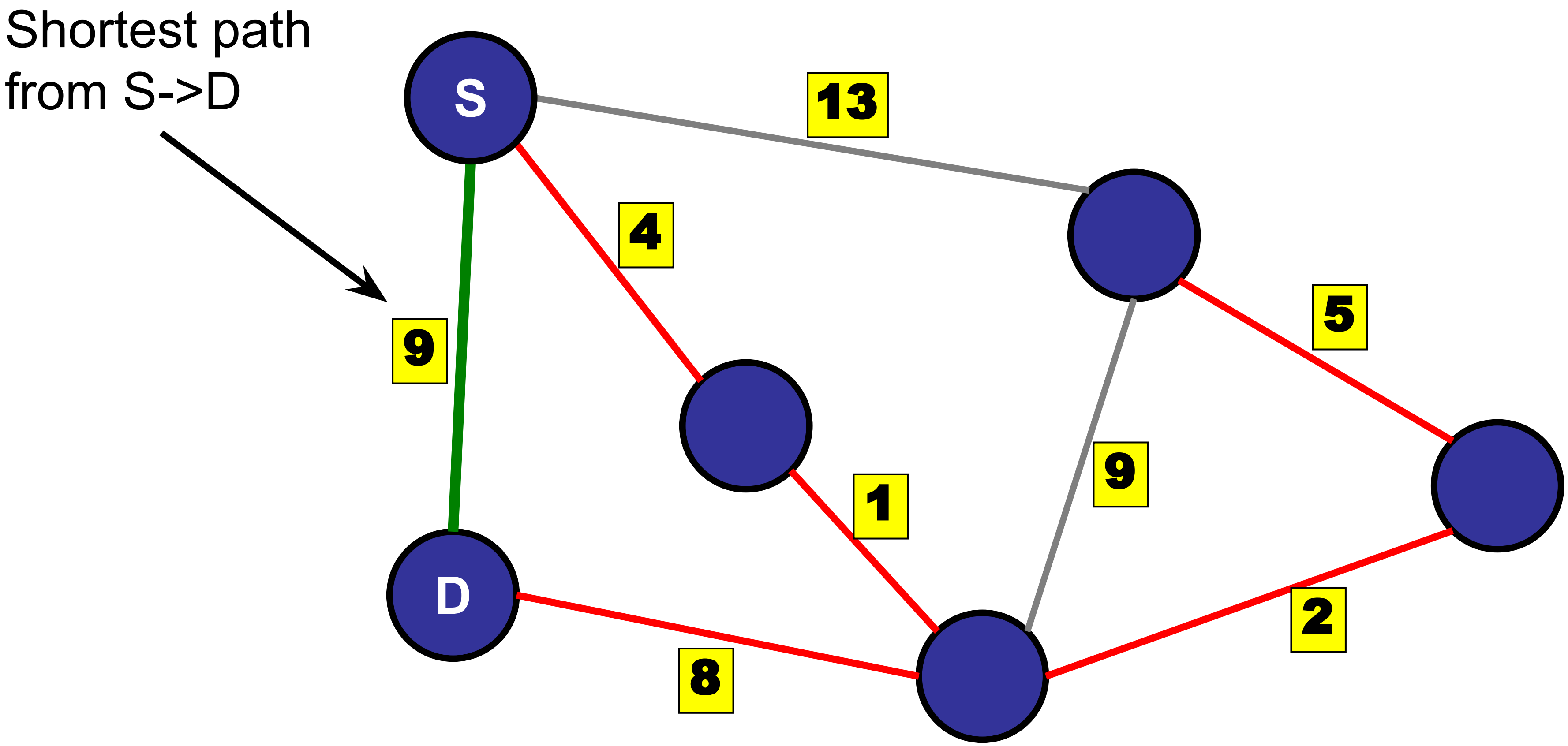
Can we use MST to find shortest paths?

1. Yes
2. Only on connected graphs.
3. Only on dense graphs.
4. No.
5. I need to see a picture.



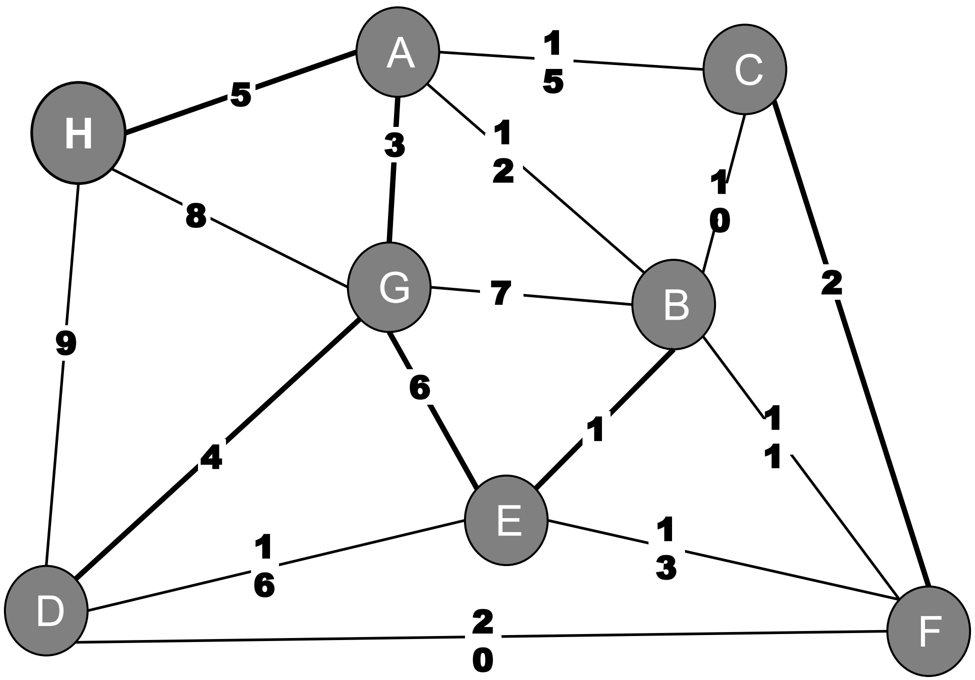
Minimum Spanning Tree

Not the same a shortest paths:



Kruskal's Algorithm

Kruskal's Algorithm. (Kruskal 1956)

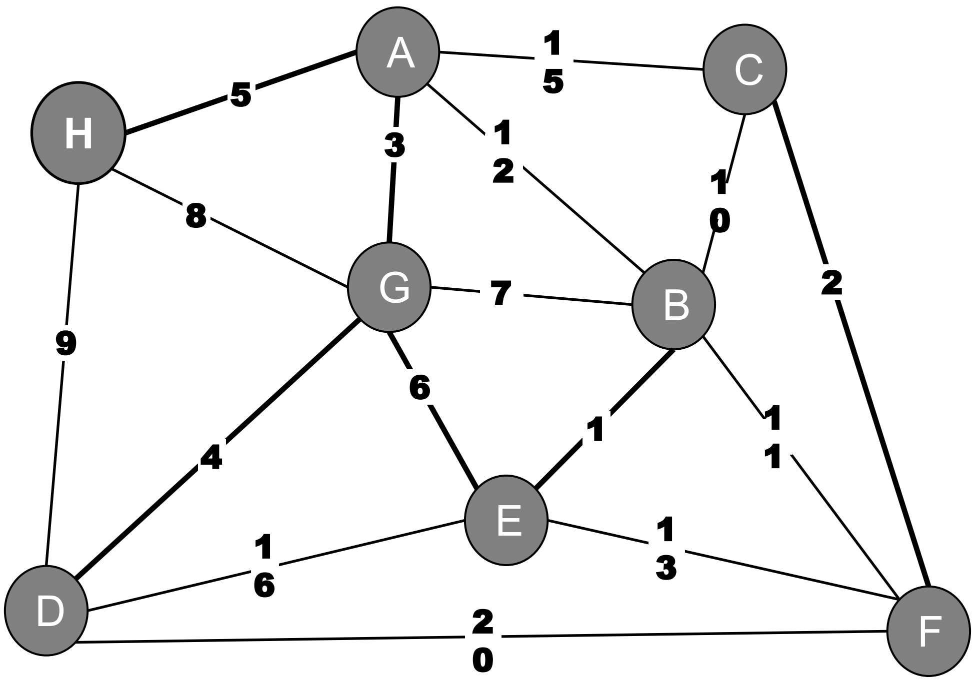


Kruskal's Algorithm

Kruskal's Algorithm. (Kruskal 1956)

Basic idea:

- Initially each node is disconnected and their own component.

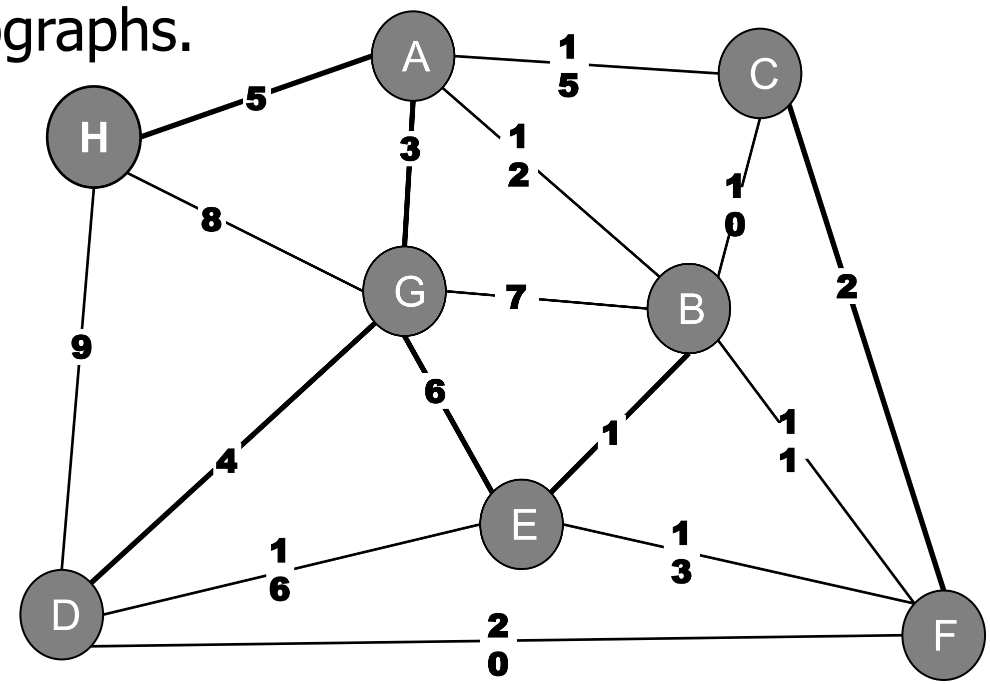


Kruskal’s Algorithm

Kruskal’s Algorithm. (Kruskal 1956)

Basic idea:

- Sort edges by weight from smallest to biggest.
- Consider edges in ascending order:
  - If edge is joining two disconnected subgraphs, include the edge. Union the two subgraphs.
  - Else, ignore the edge.



Kruskal’s Algorithm

```
// Sort edges and initialize
```

1. Initialise a UFDS for n nodes, all initially disjoint.

Kruskal's Algorithm

```
// Sort edges and initialize
```

1. Initialise a UFDS for n nodes, all initially disjoint.

2. Sort the edges by their weights, in ascending order.

Kruskal's Algorithm

```
// Sort edges and initialize
```

1. Initialise a UFDS for n nodes, all initially disjoint.

2. Sort the edges by their weights, in ascending order.

3. For each edge e = (u, v)

Kruskal's Algorithm

```
// Sort edges and initialize
```

1. Initialise a UFDS for n nodes, all initially disjoint.

2. Sort the edges by their weights, in ascending order.

3. For each edge e = (u, v)
   a. If u and v belong to the same component:
      i. Skip!

Kruskal's Algorithm

```
// Sort edges and initialize
```

1. Initialise a UFDS for n nodes, all initially disjoint.

2. Sort the edges by their weights, in ascending order.

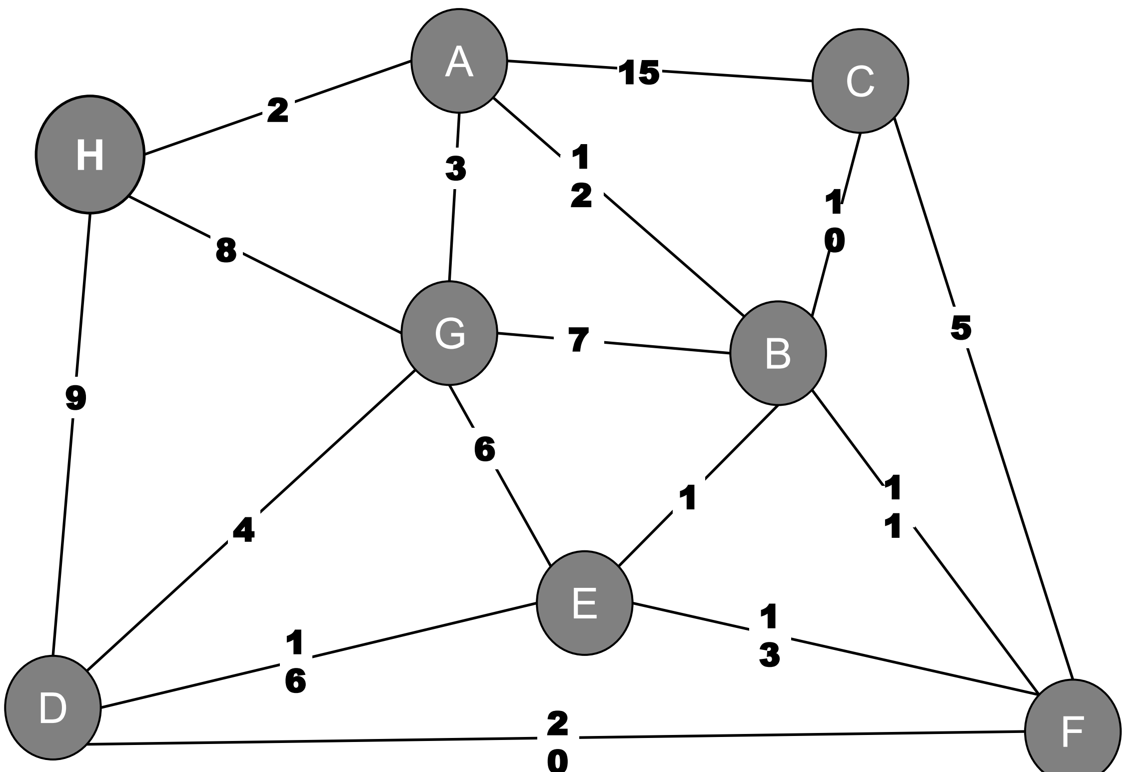
3. For each edge e = (u, v)
   a. If u and v belong to the same component:
      i. Skip!

   b. Otherwise, add the edge in, union u and v's component.

Kruskal's Algorithm

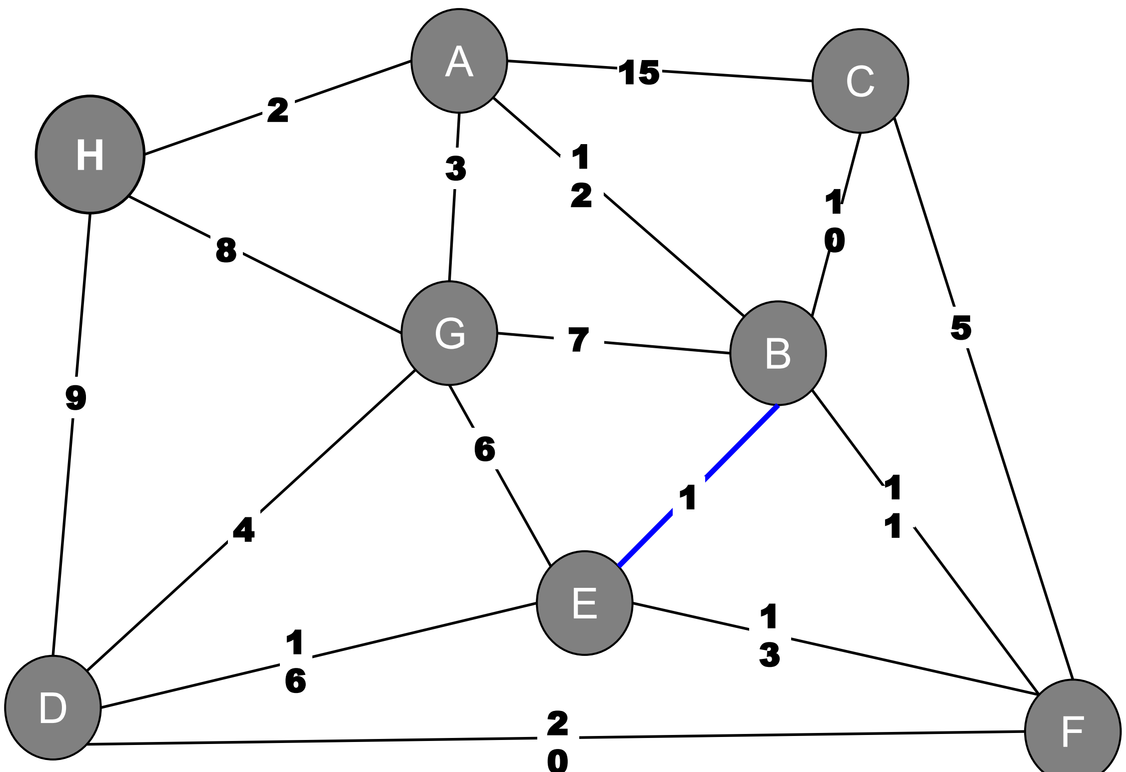
```
// Sort edges and initialize
Edge[] sortedEdges = sort(G.E());
ArrayList<Edge> mstEdges = new ArrayList<Edge>();
UnionFind uf = new UnionFind(G.V());


// Iterate through all the edges, in order
for (int i=0; i<sortedEdges.length; i++) {
      Edge e = sortedEdges[i]; // get edge
      Node v = e.one(); // get node endpoints
      Node w = e.two();

      if (!uf.find(v,w)) { // in the same tree?
         mstEdges.add(e); // save edge
         uf.union(v,w); // combine trees
      }
}
```

Kruskal's Example



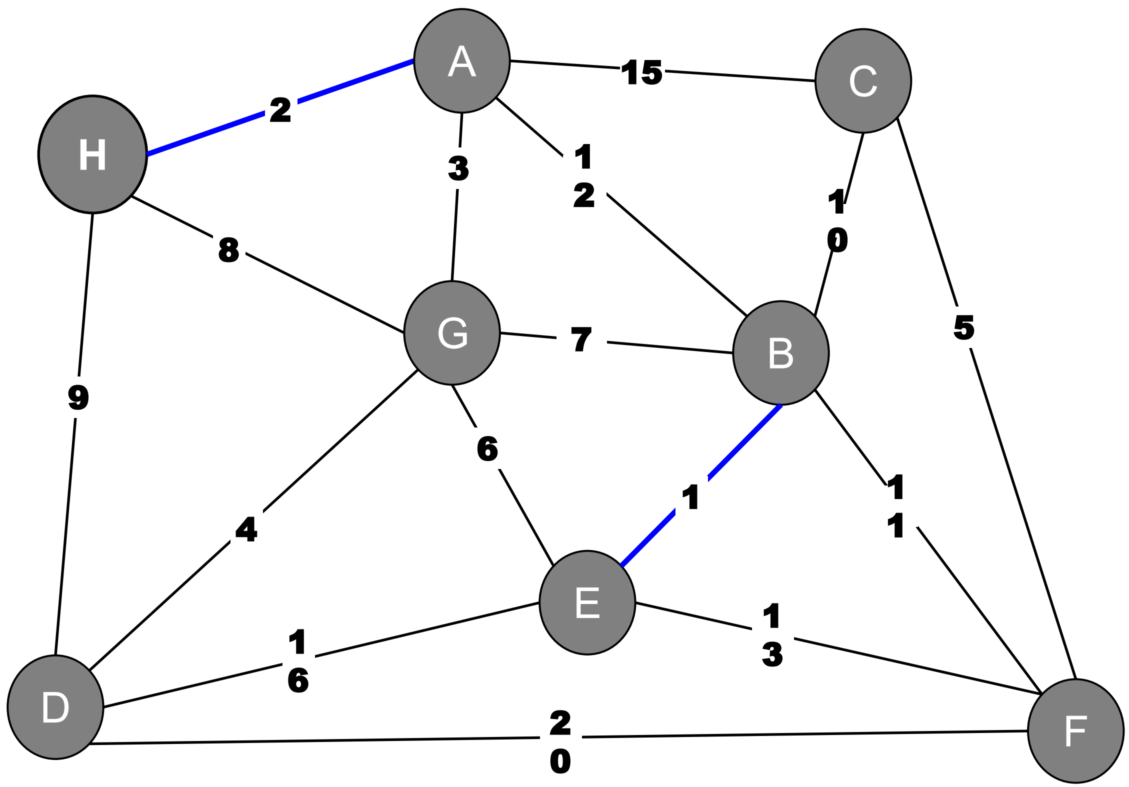
Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal's Example



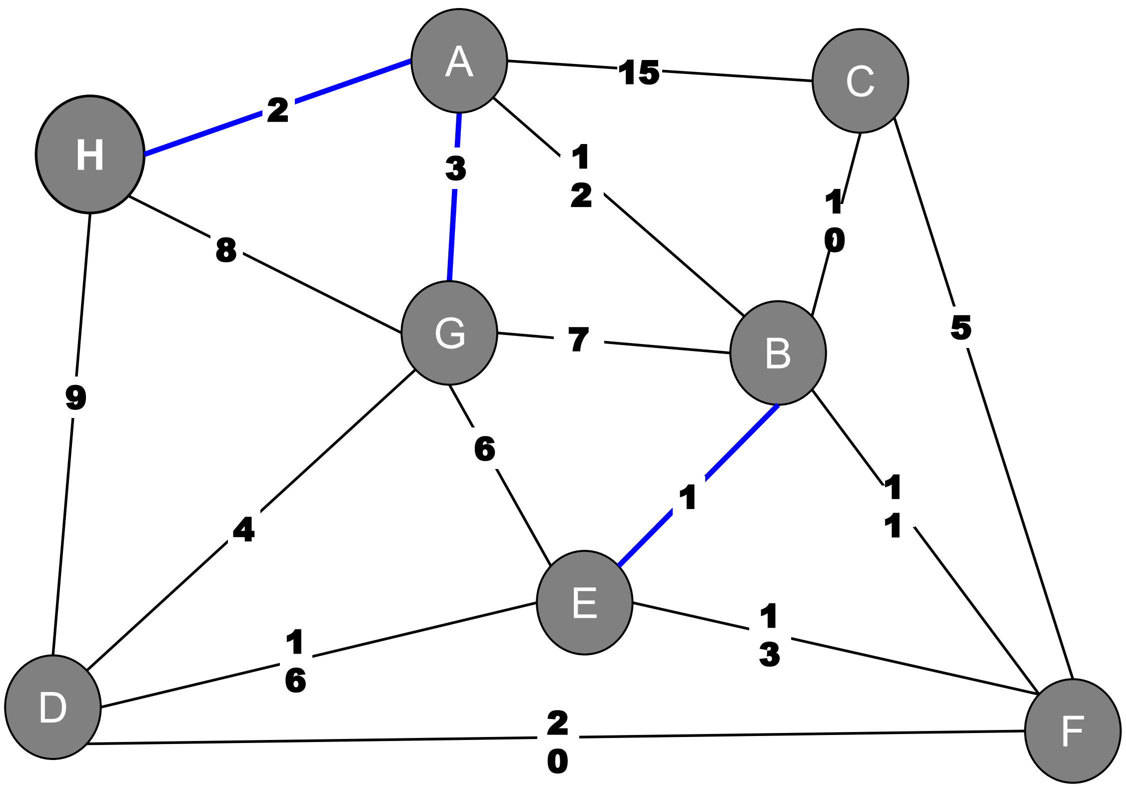
Weight	Edge
1	**(E,B)**
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal's Example



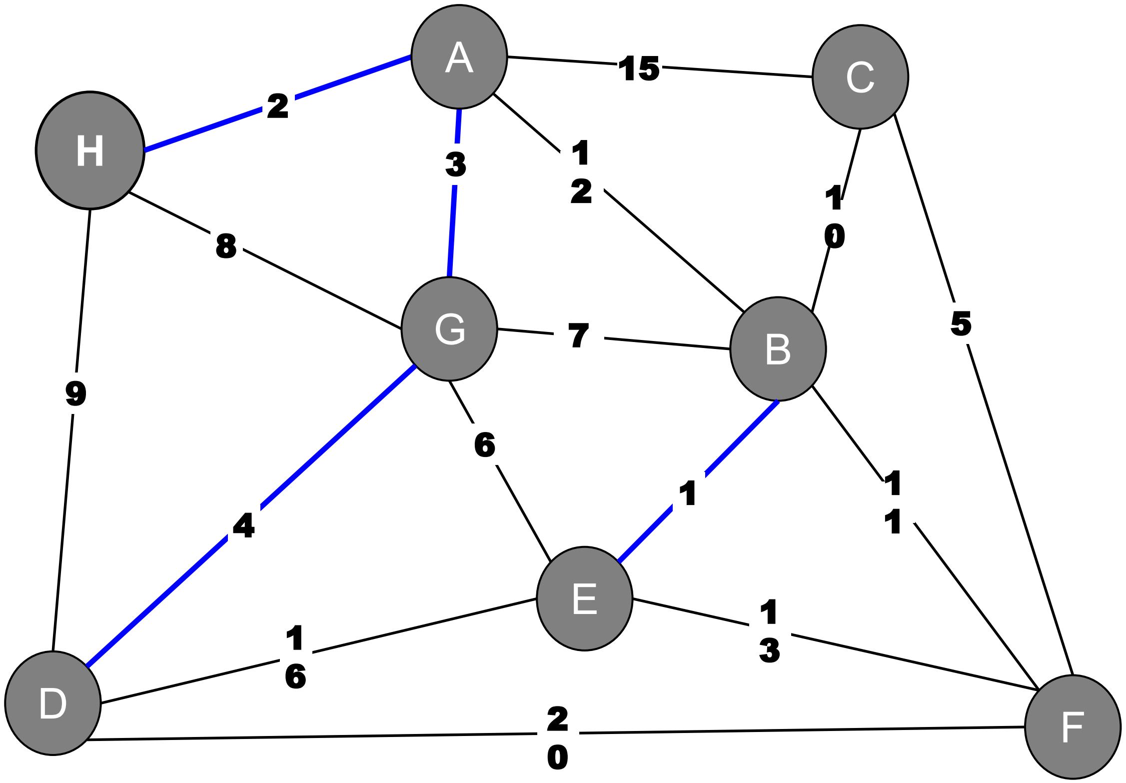
Weight	Edge
1	(E,B)
2	**(C,F)**
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal's Example



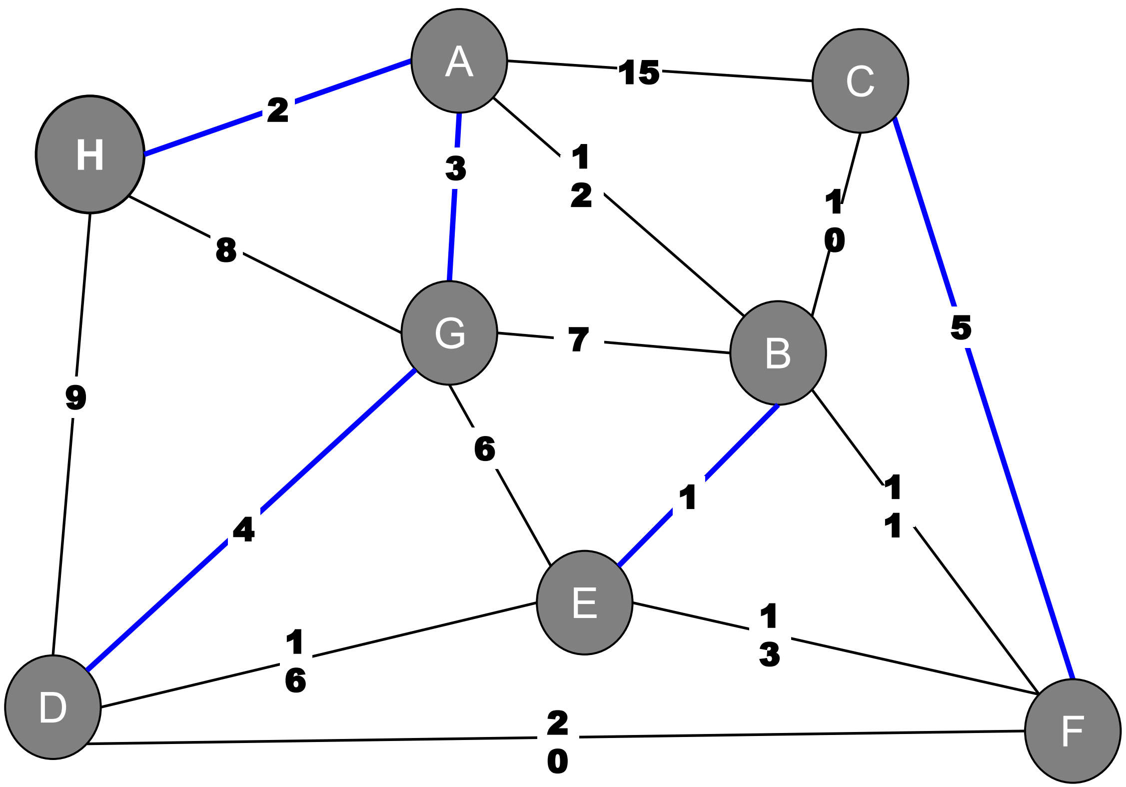
Weight	Edge
1	(E,B)
2	(C,F)
3	**(A,G)**
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal's Example



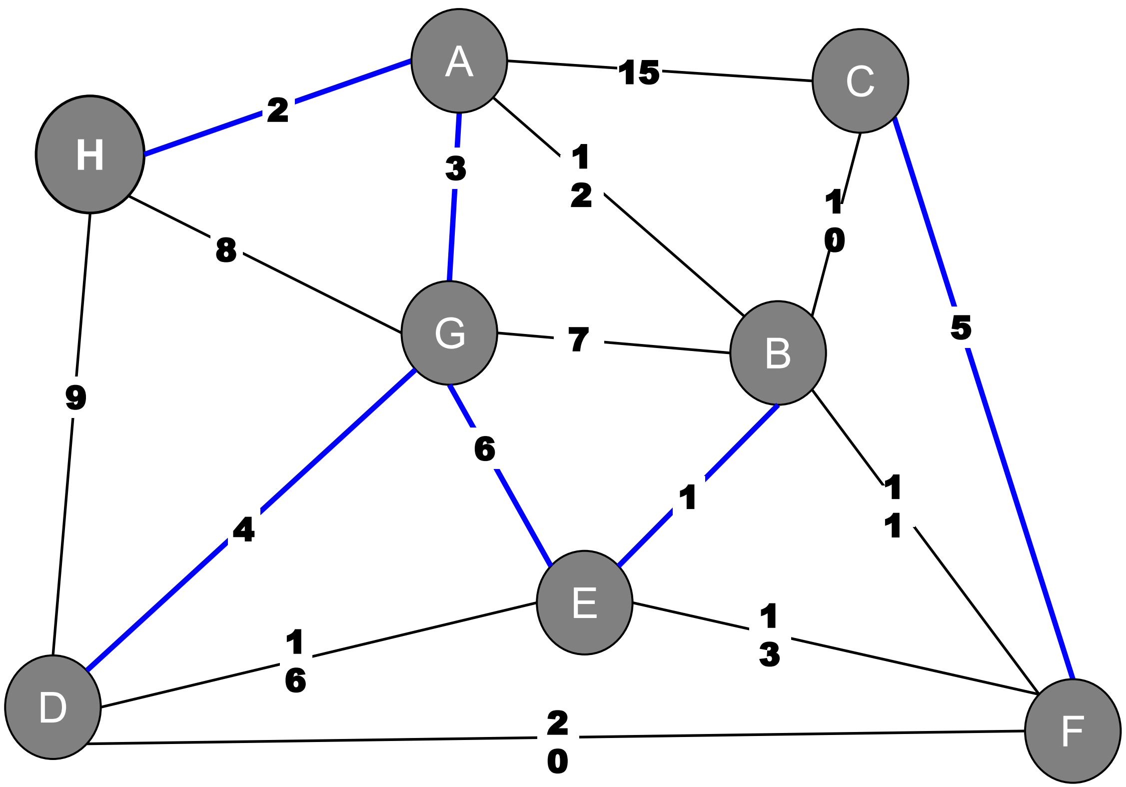
Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	**(D,G)**
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal’s Example



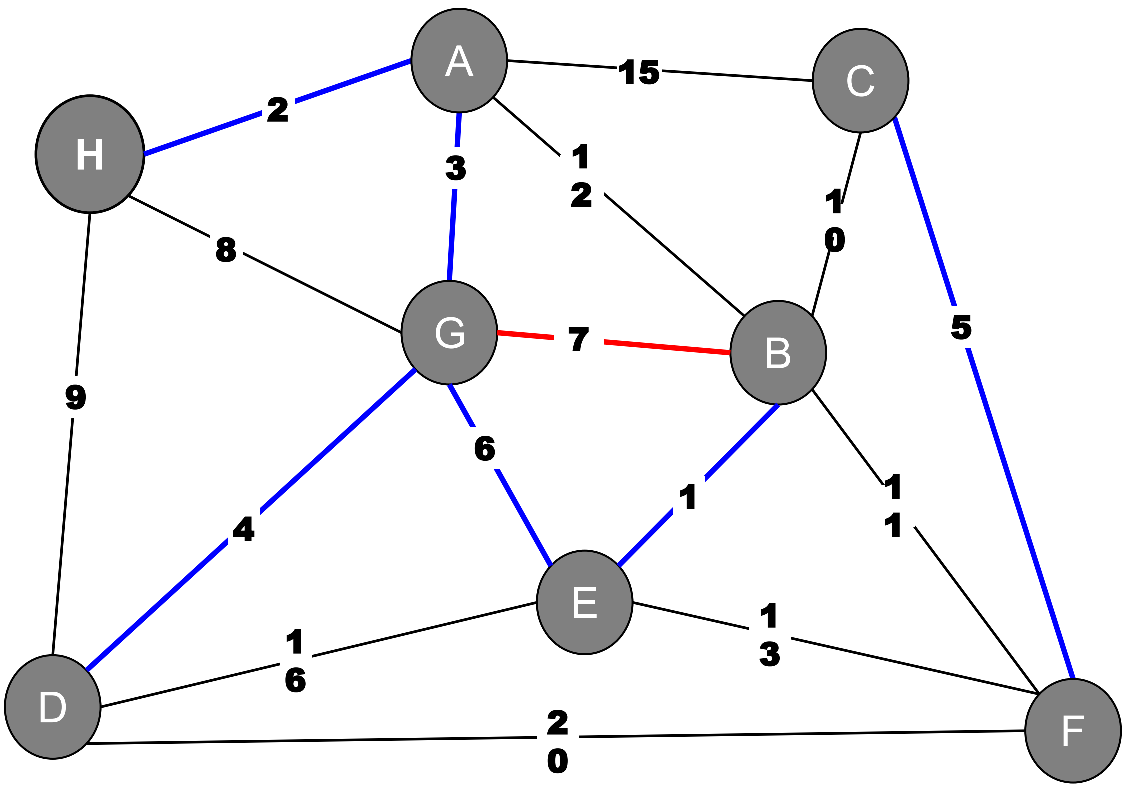
Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	**(C,F)**
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal's Example



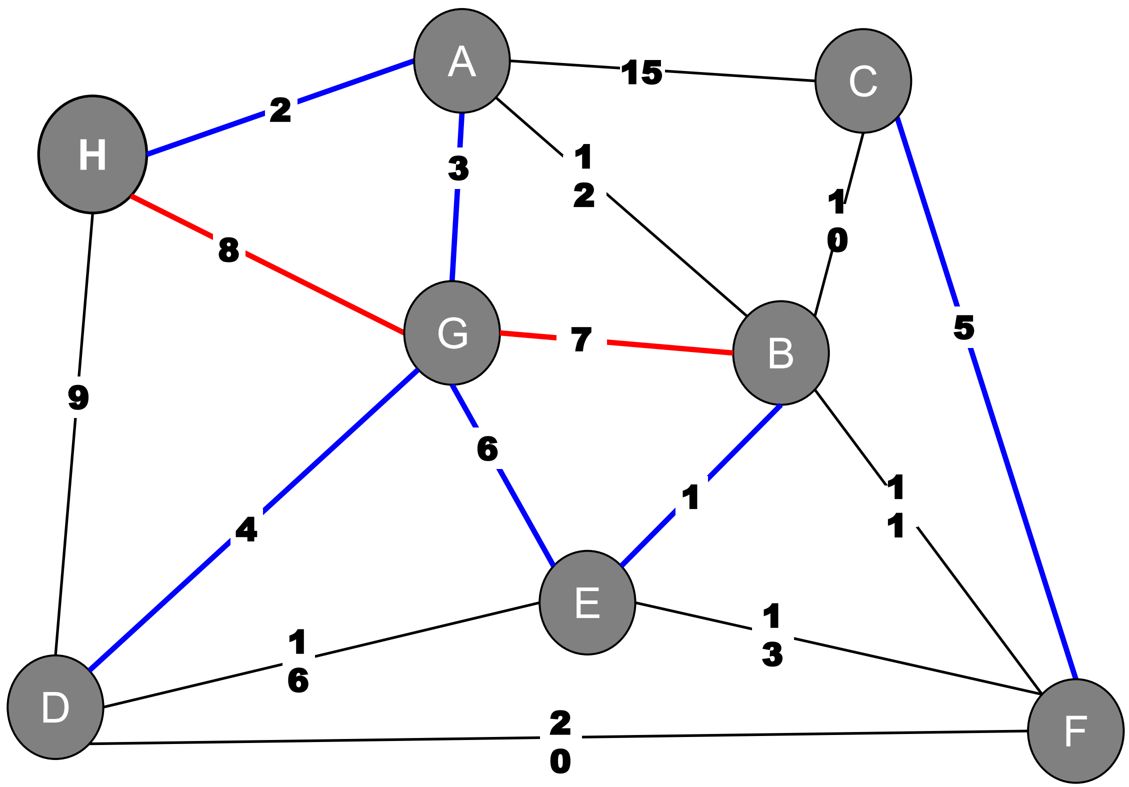
Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	**(E,G)**
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal's Example



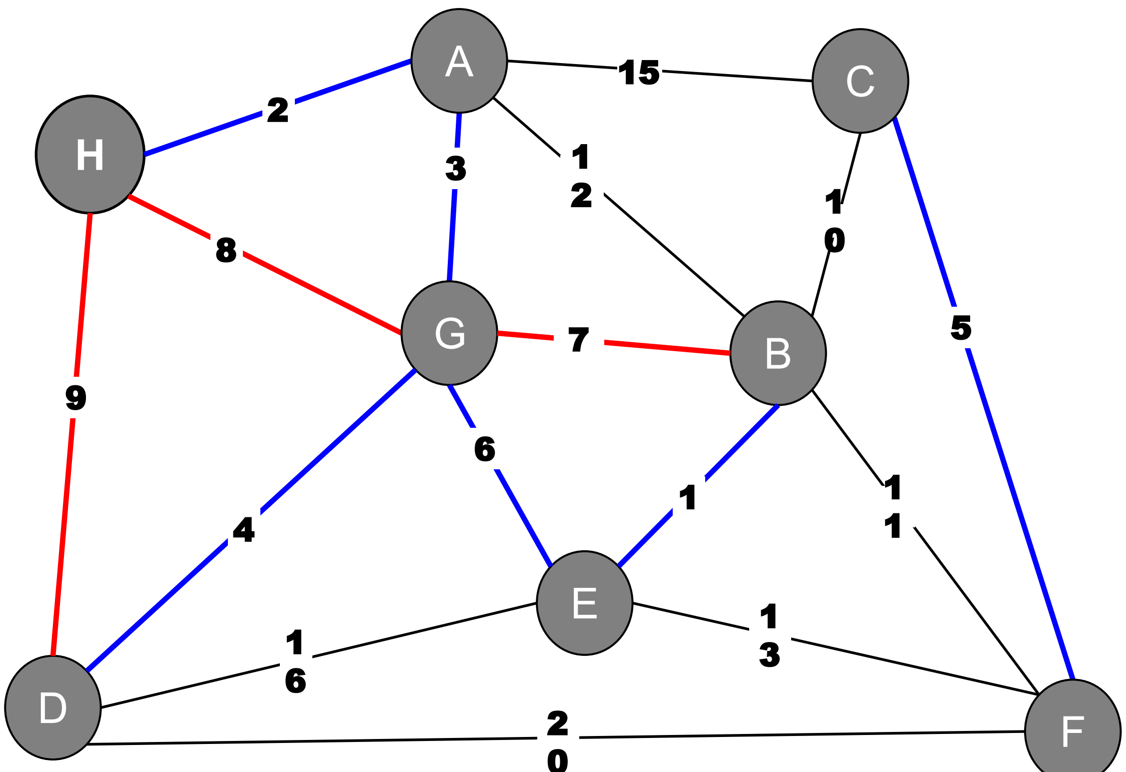
Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	**(B,G)**
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal’s Example



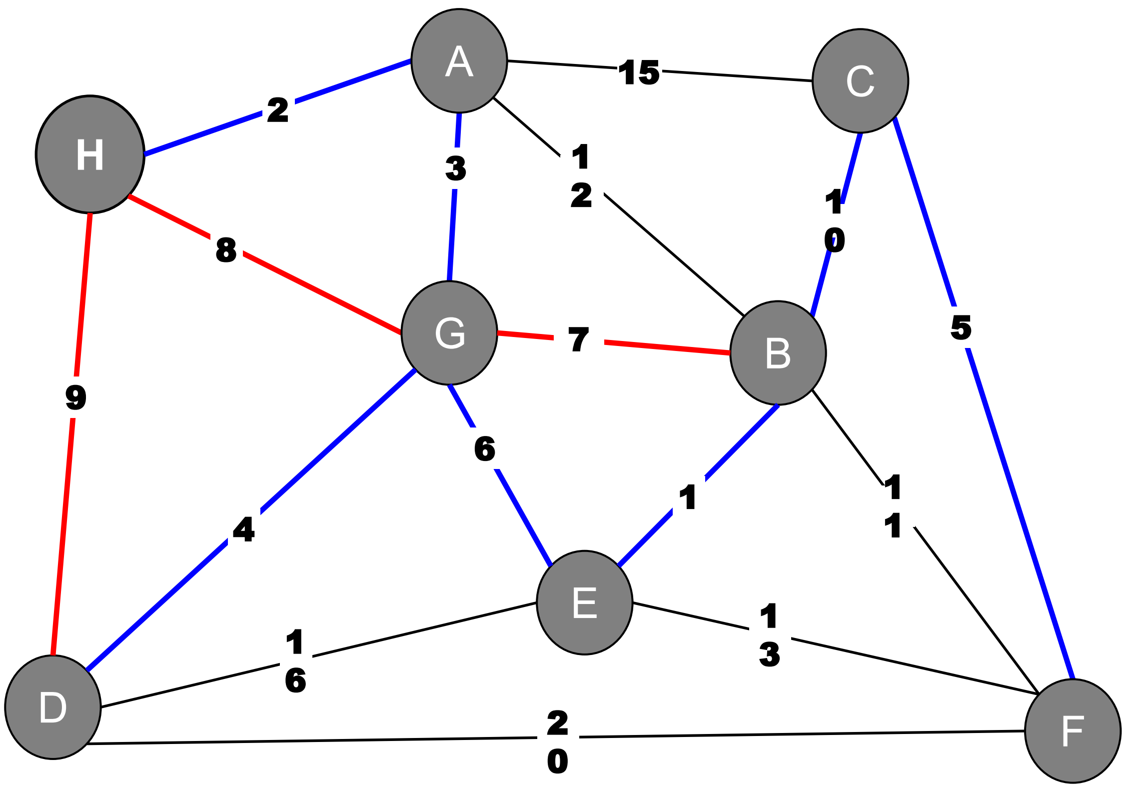
Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	**(G,H)**
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal's Example



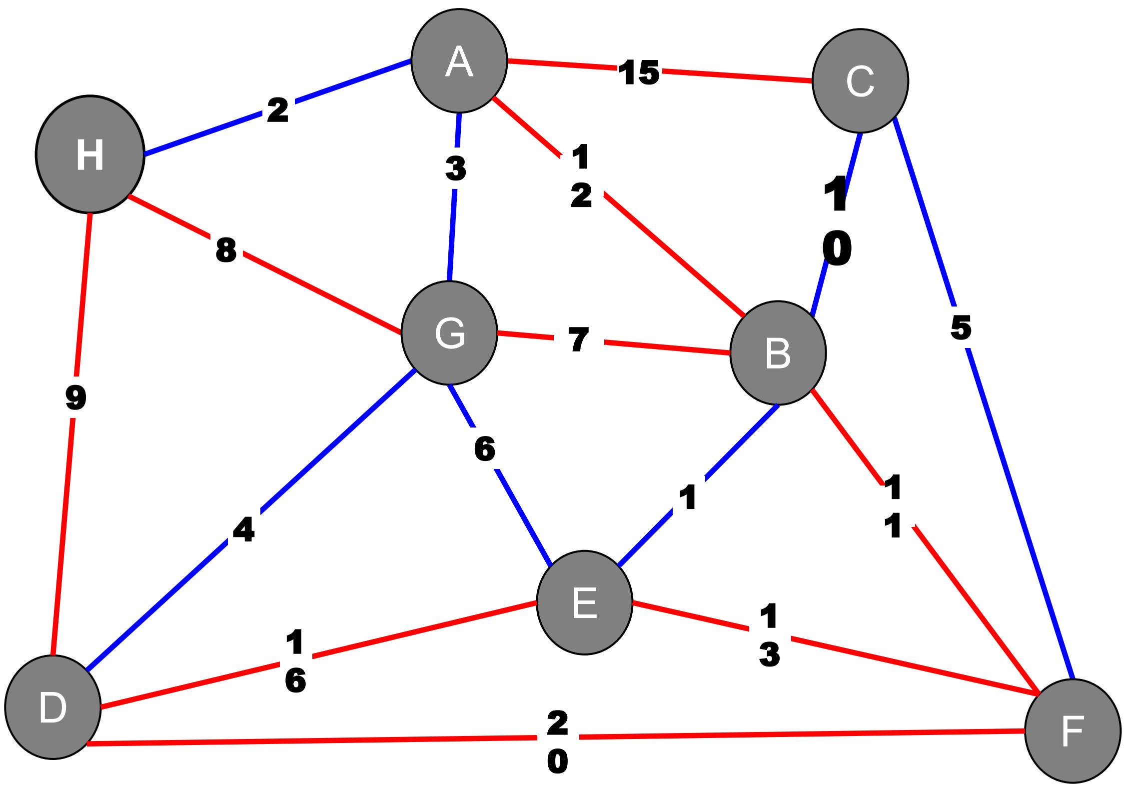
Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	**(D,G)**
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal's Example



Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

Kruskal’s Example



Weight	Edge
1	(E,B)
2	(C,F)
3	(A,G)
4	(D,G)
5	(C,F)
6	(E,G)
7	(B,G)
8	(G,H)
9	(D,G)
10	(B,C)
11	(B,F)
12	(A,B)
13	(E,F)
15	(A,C)
16	(D,E)
20	(D,F)

What is the running time of Kruskal's Algorithm on a connected graph?

1. $O(V)$
2. $O(E)$
3. $O(E\,\alpha)$
4. $O(V\,\alpha)$
5. $O(E \log V)$
6. $O(V \log E)$

Kruskal's Algorithm

```
// Sort edges and initialize
```

1. Initialise a UFDS for n nodes, all initially disjoint.

2. Sort the edges by their weights, in ascending order.

3. For each edge e = (u, v)
   a. If u and v belong to the same component:
      i. Skip!

   b. Otherwise, add the edge in, union u and v's component.

O(E α(E))

O(E log E)

Kruskal's Algorithm

```
// Sort edges and initialize
```

1. Initialise a UFDS for n nodes, all initially disjoint.

2. Sort the edges by their weights, in ascending order.

3. For each edge e = (u, v)
   a. If u and v belong to the same component:
      i. Skip!

   b. Otherwise, add the edge in, union u and v's component.

$O(E\,\alpha(E))$

$O(E \log E)$

$O(E\,\alpha(E) + E \log (E)) = O(E \log E)$

Correctness?

Deferred until next Monday!

(Also Prim’s algorithm)