Roadmap

Part I: Priority Queues

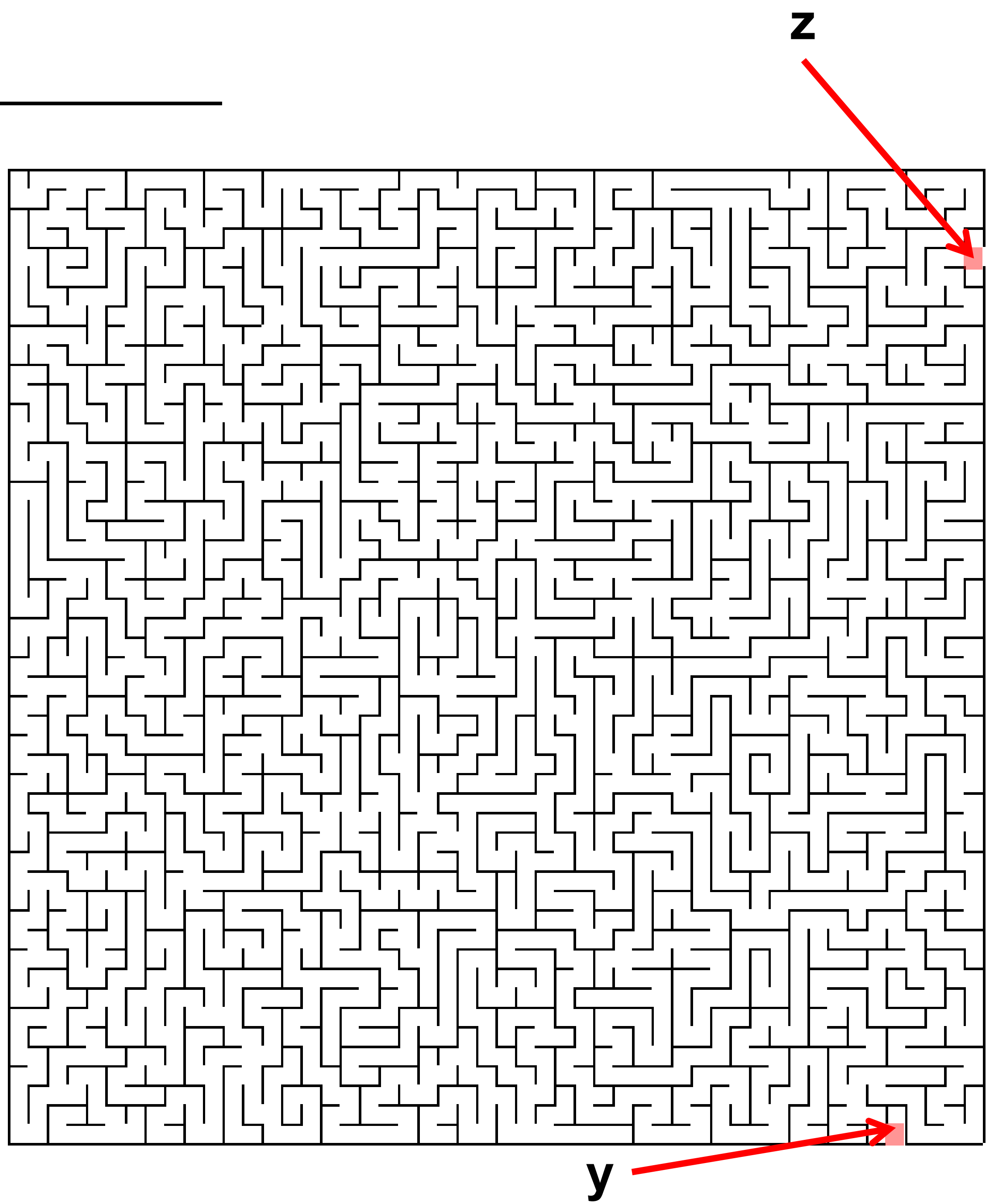
- Binary Heaps
- HeapSort

Part II: Disjoint Set

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

Mazes

Is there any route from y to z?



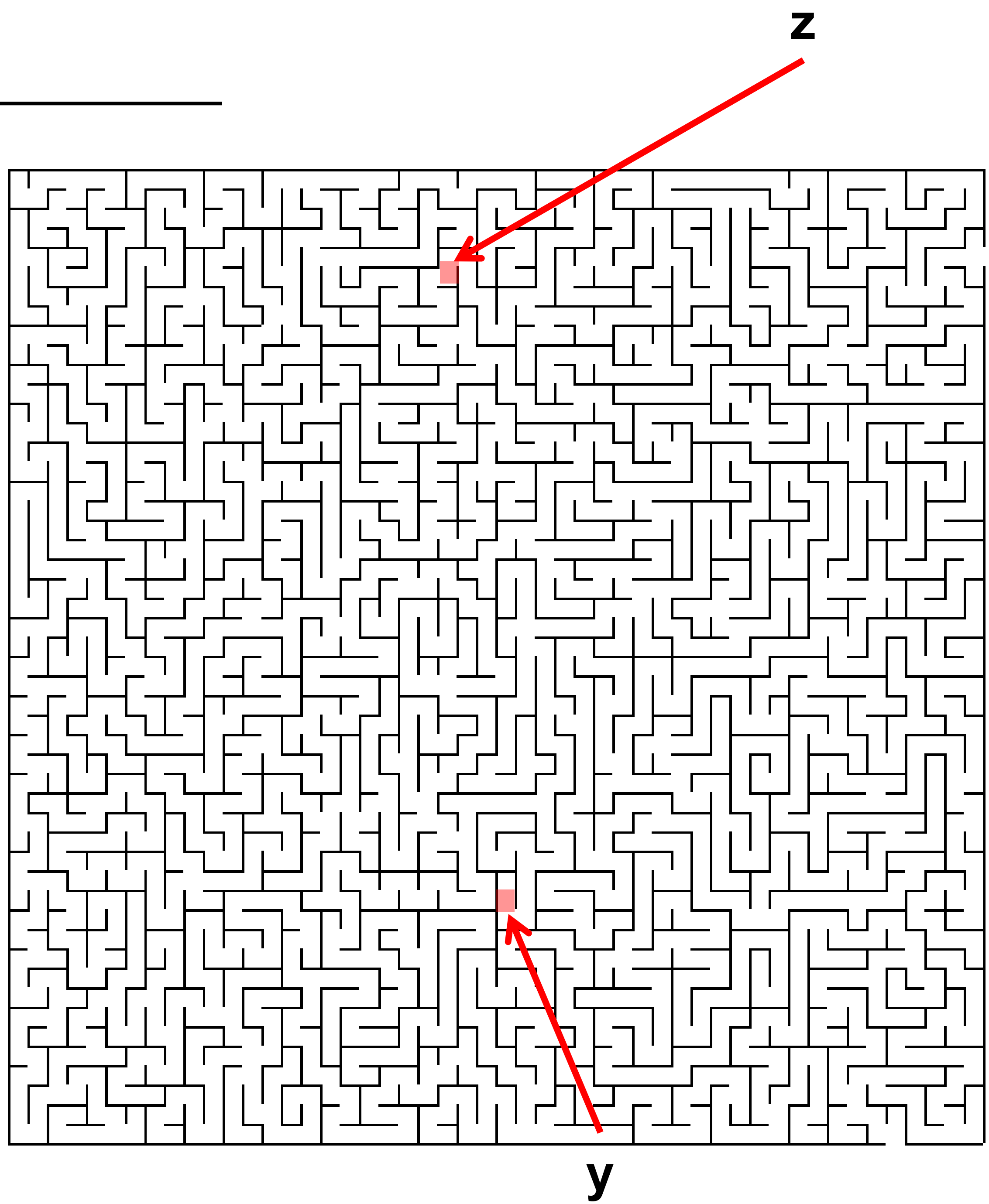
Mazes

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.



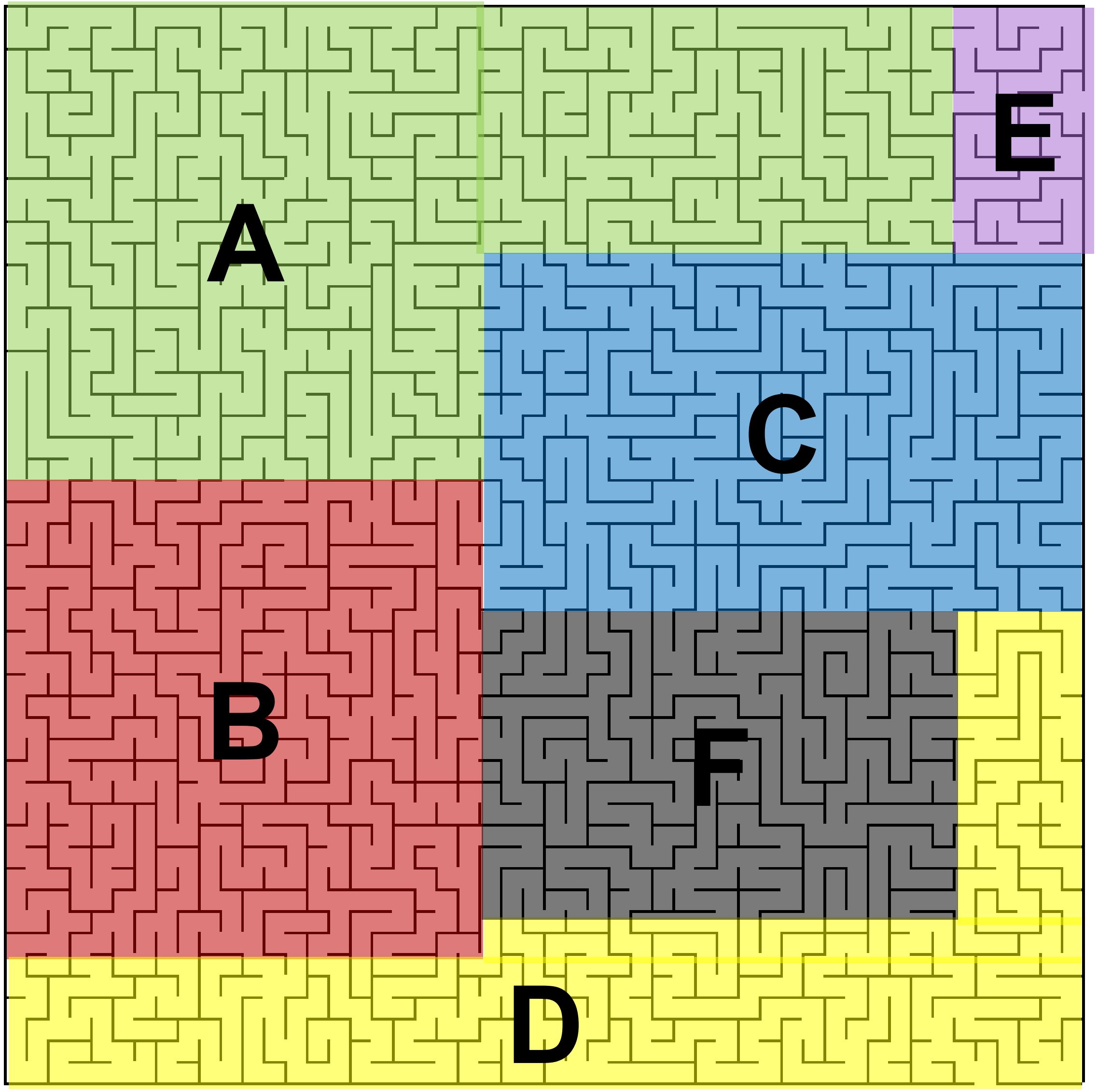
Mazes

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:

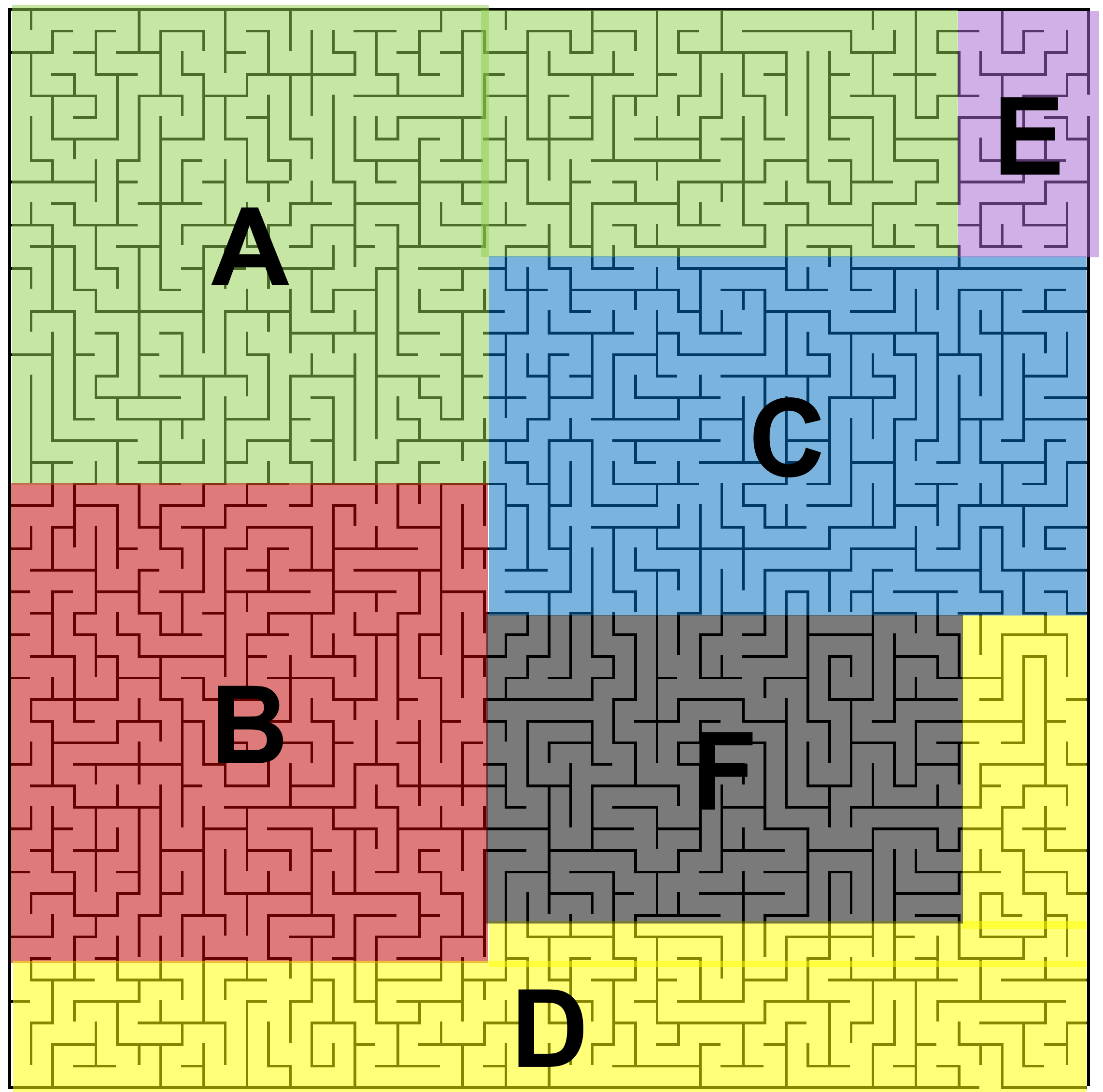
Prepare to answer queries.

destroyWall(x):

Remove walls from the maze using your superpowers.

isConnected(y, z):

Answer connectivity queries.



Mazes

Preprocess:

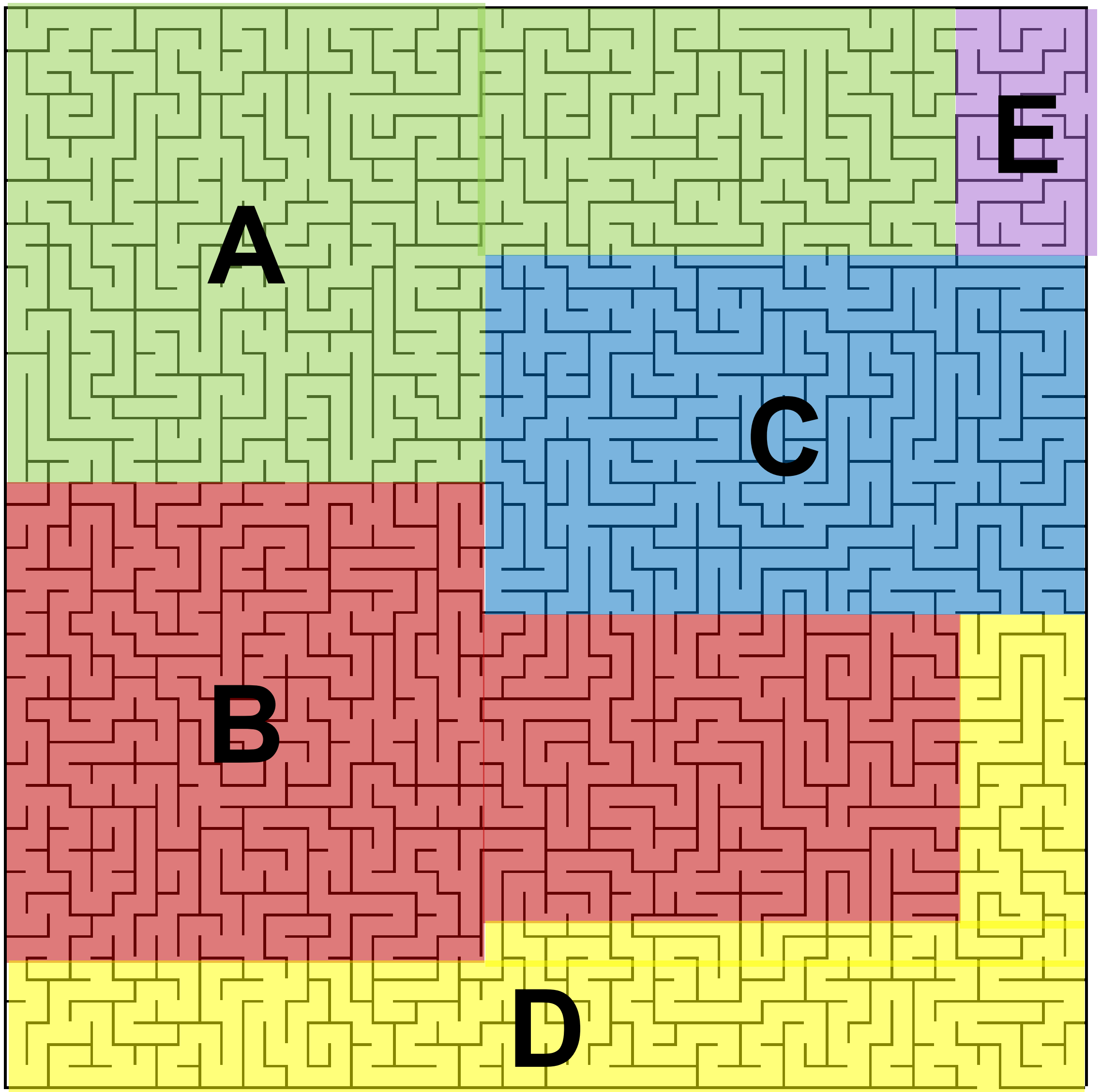
Prepare to answer queries.

destroyWall(x):

Remove walls from the maze using your superpowers.

isConnected(y, z):

Answer connectivity queries.

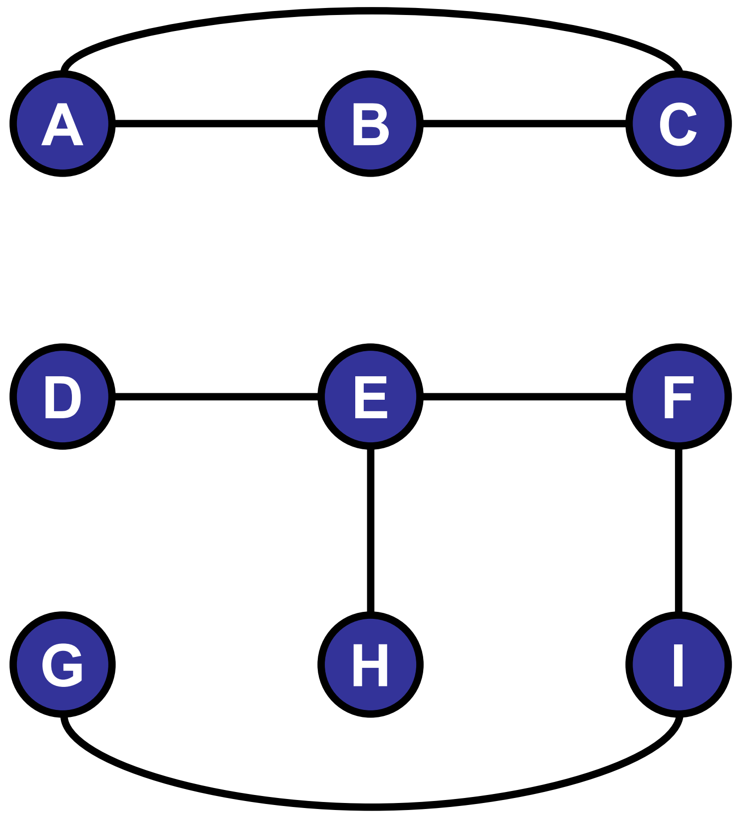


Dynamic Connectivity

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) = **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:

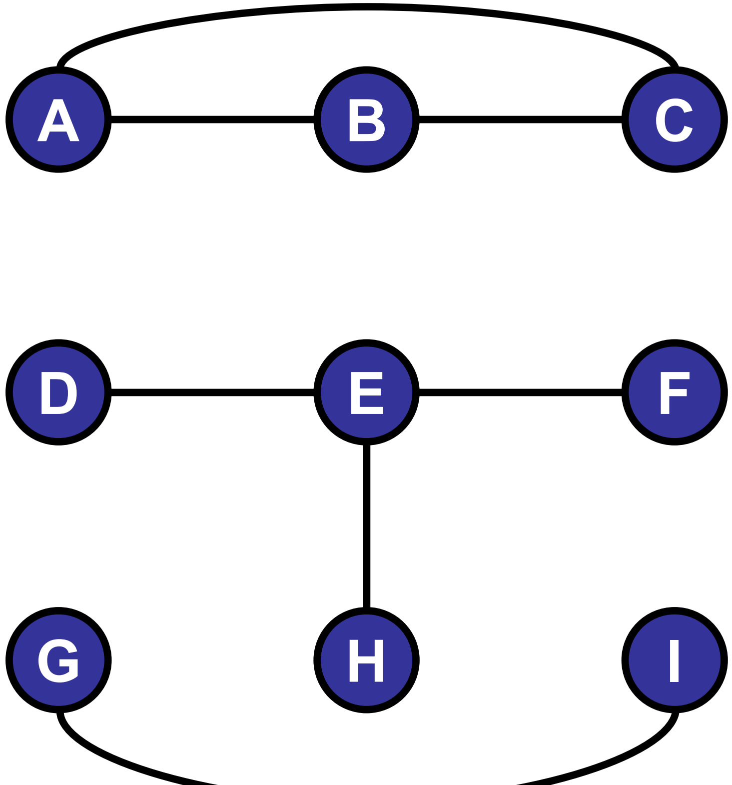
- **Union:** connect two objects
- **Find:** is there a path connecting the two objects?

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.



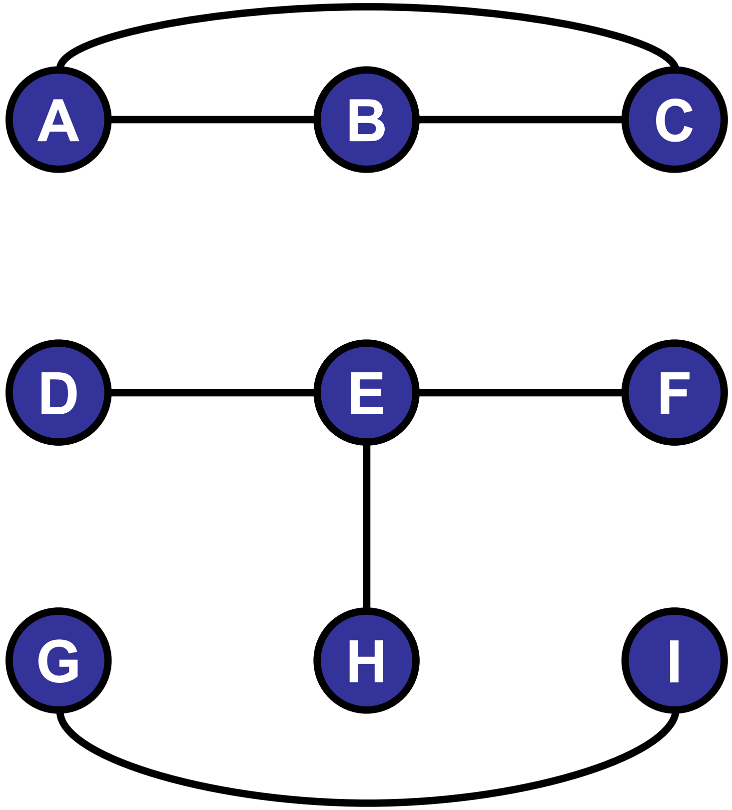
Dynamic Connectivity

Given a set of objects:

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

Maintain sets of connected components:

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



Dynamic Connectivity

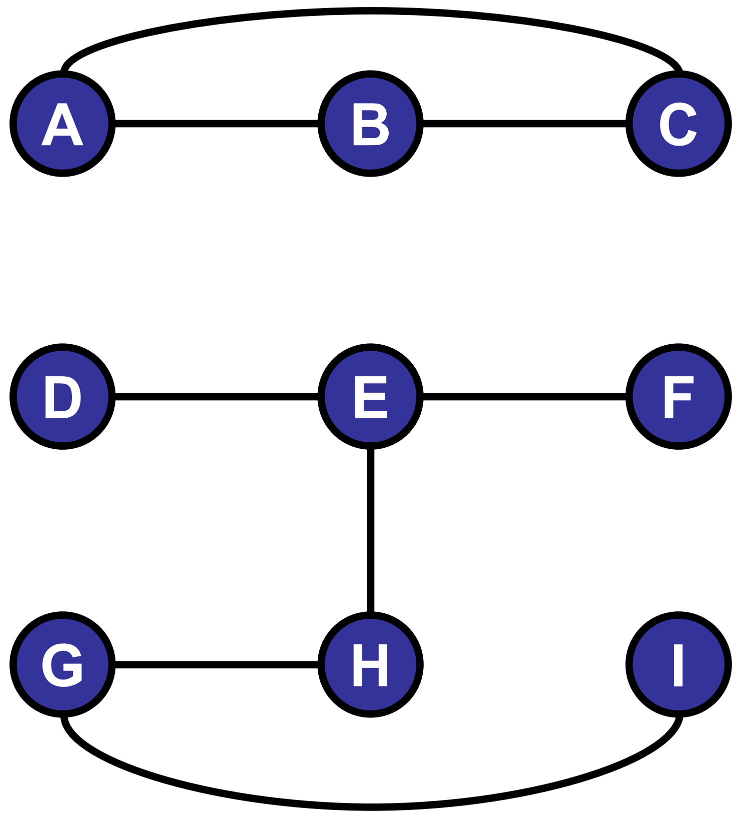
Given a set of objects:

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

Maintain sets of connected components:

{A, B, C}

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



Abstract Data Type

Disjoint Set (Union-Find)

	`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*

Roadmap

Part II: Disjoint Set

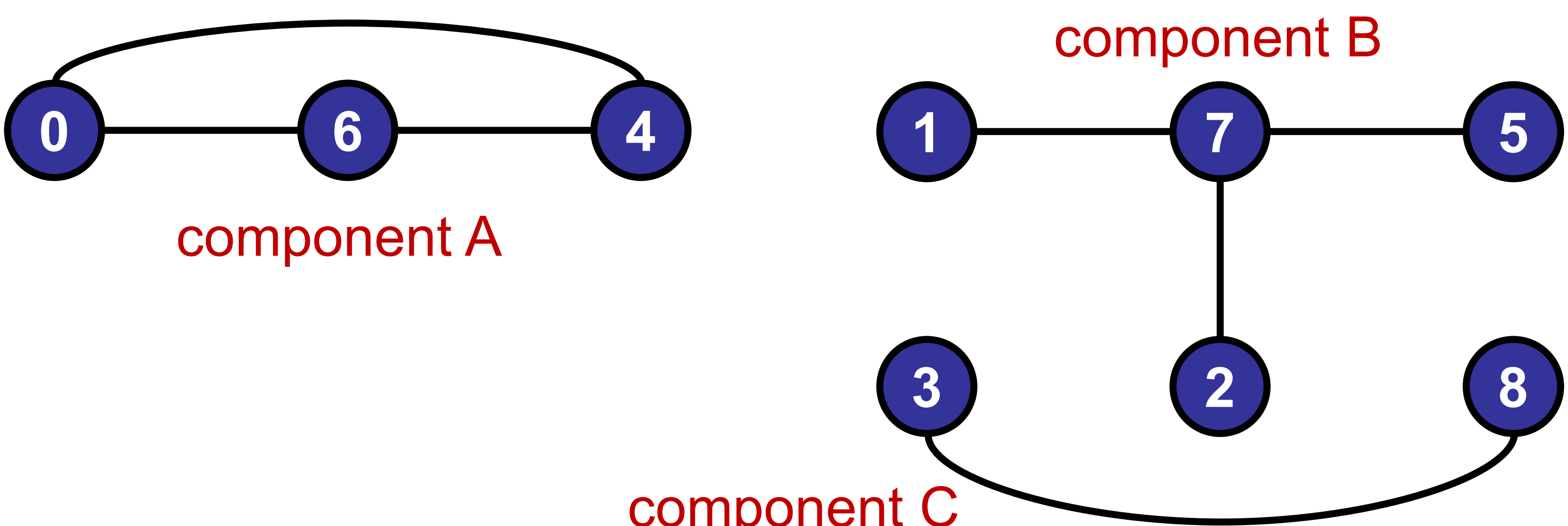
- Problem: Dynamic Connectivity
- Algorithm: Quick-Find
- Algorithm: Quick-Union
- Optimizations
- Applications

Quick Find

Data structure:

- Array: componentId
- Two objects are connected if they have the same component identifier.

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



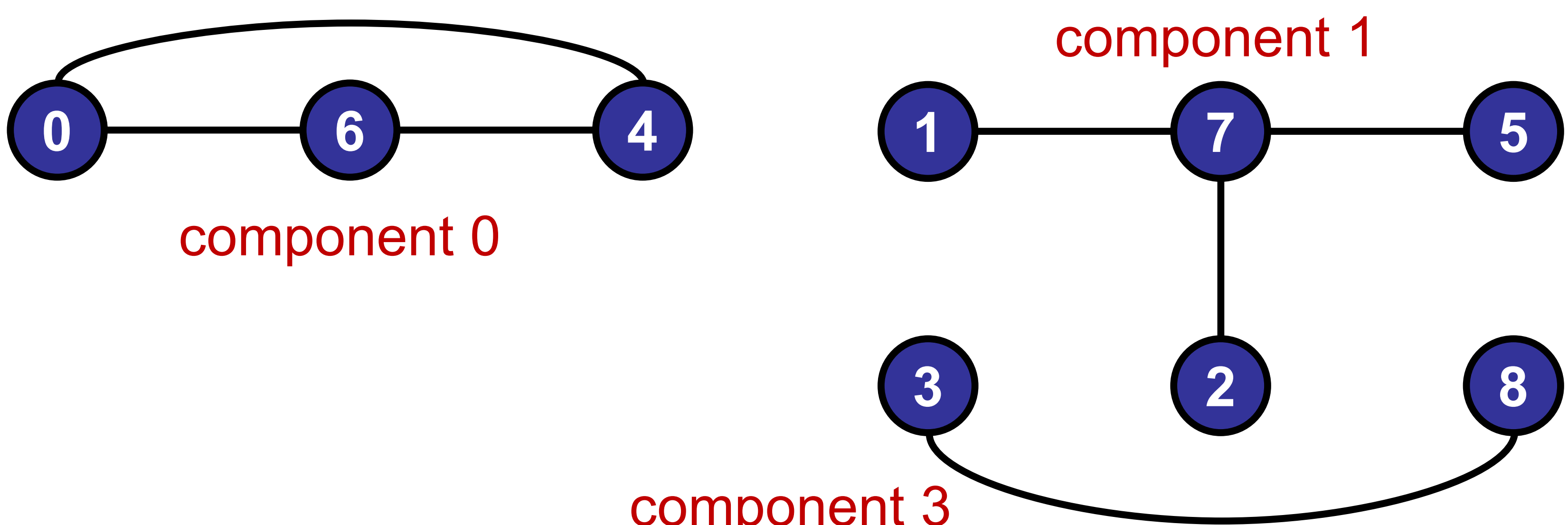
Quick Find

Assume objects are integers

Data structure:

- Integer array: int[] componentId
- Two objects are connected if they have the same component identifier.

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

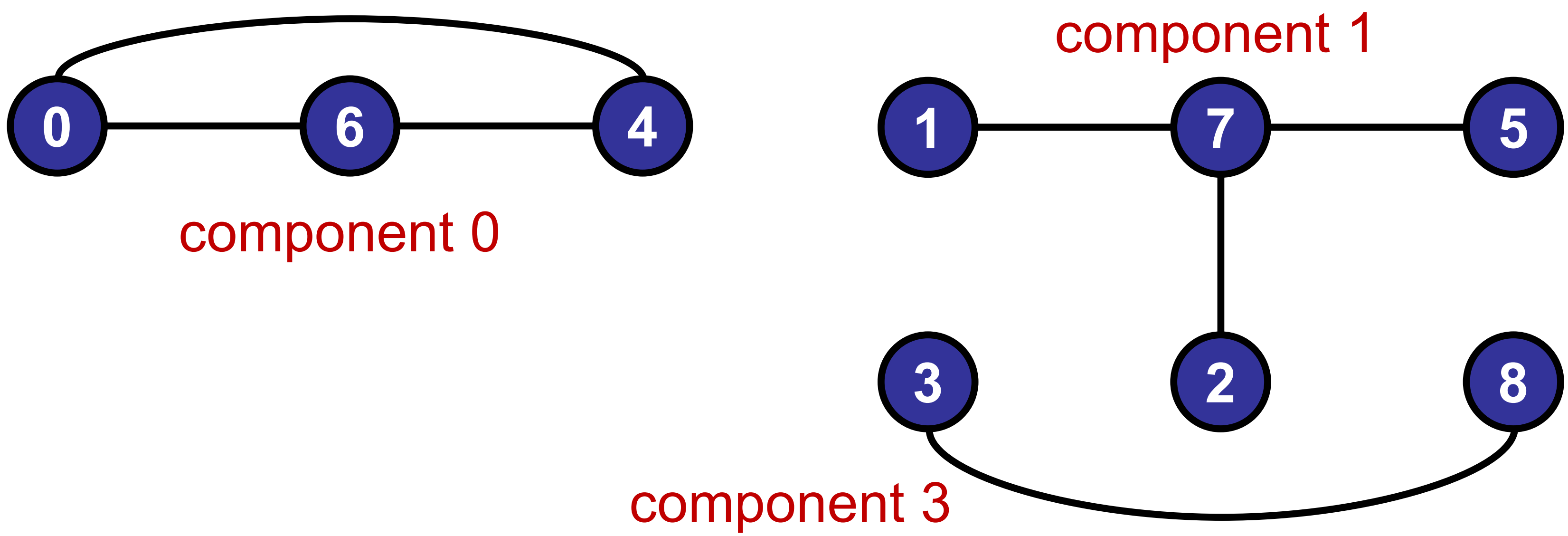


Quick Find

Data structure:

- Integer array: int[] componentId
- Two objects are connected if they have the same component identifier.

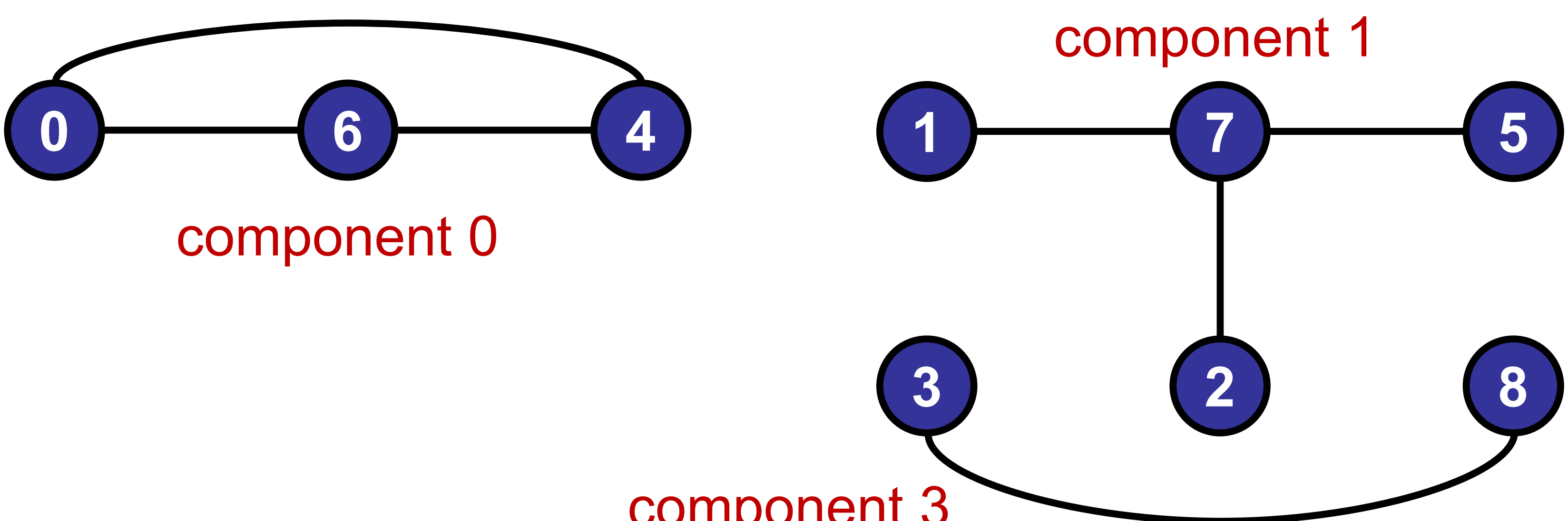
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



Quick Find

```
find(int p, int q)

   return(componentId[p] == componentId[q]);
```

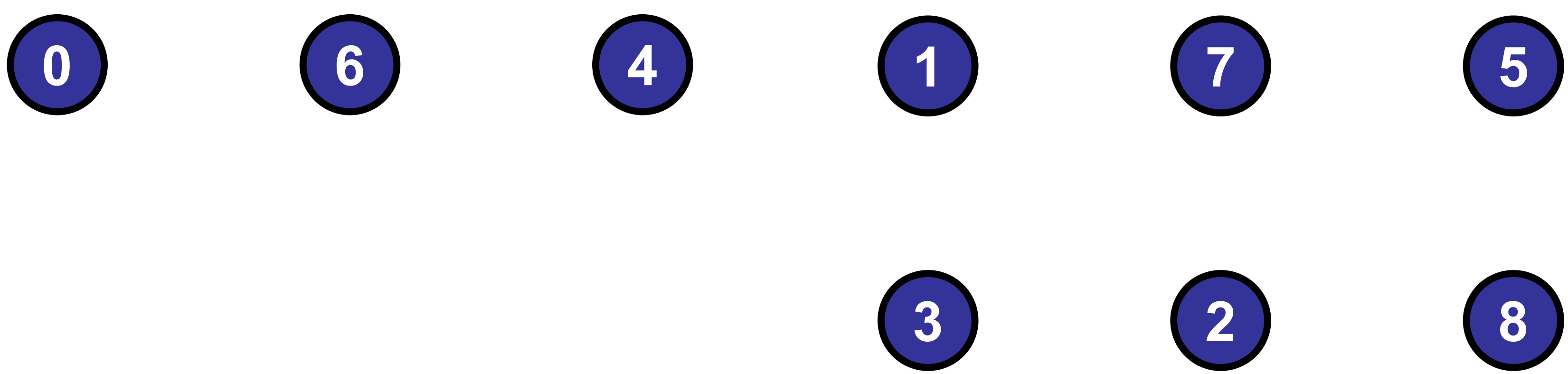
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



Quick Find

Initial state of data structure:

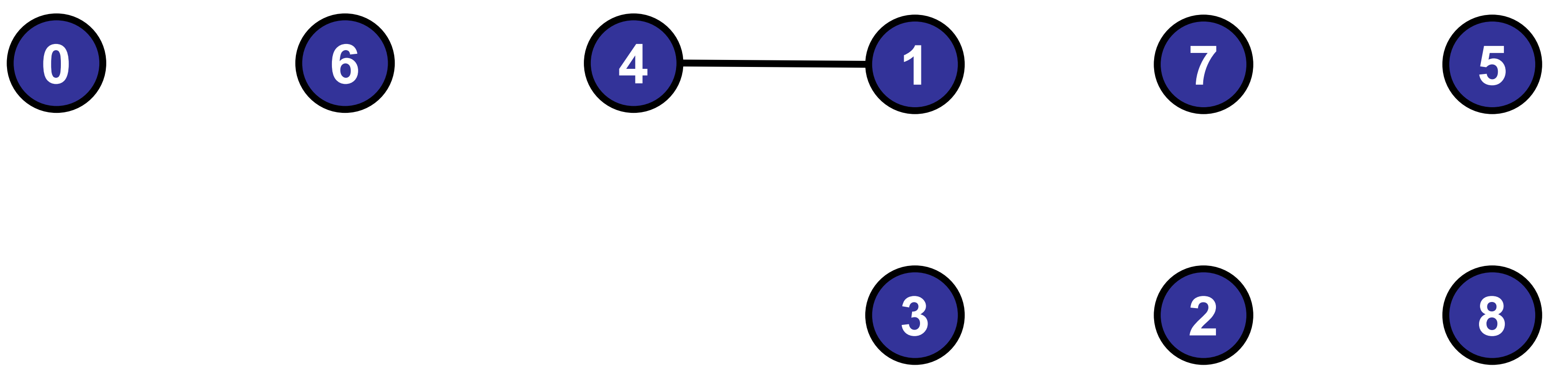
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	4	5	6	7	8



Quick Find

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

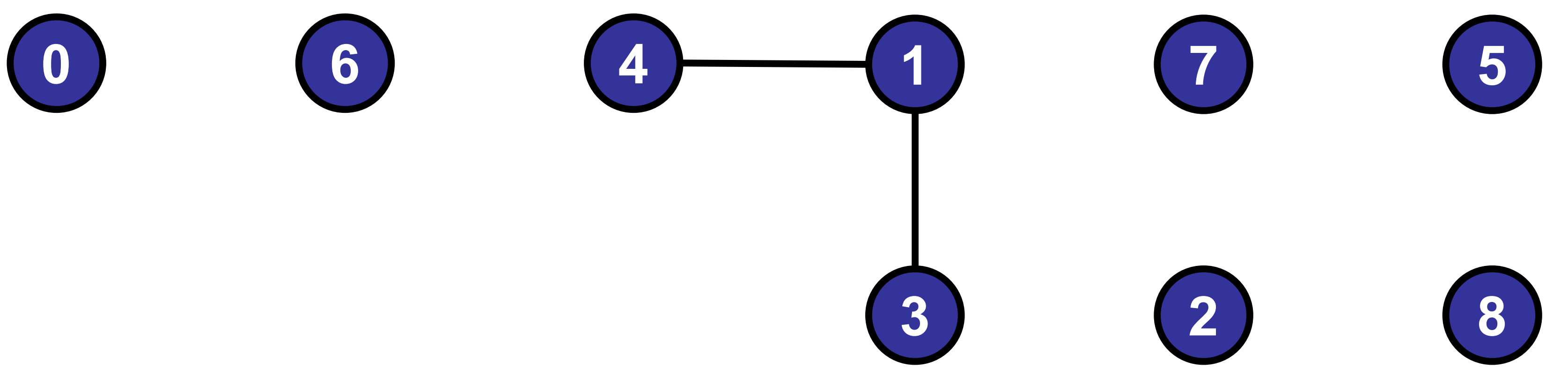
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	3	1	5	6	7	8



Quick Find

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

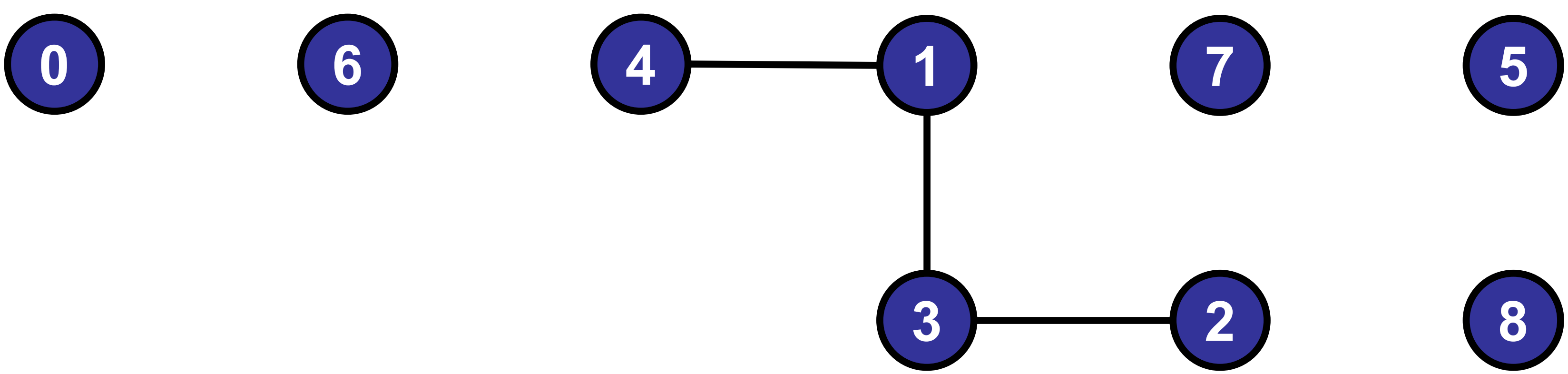
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	2	1	1	5	6	7	8



Quick Find

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

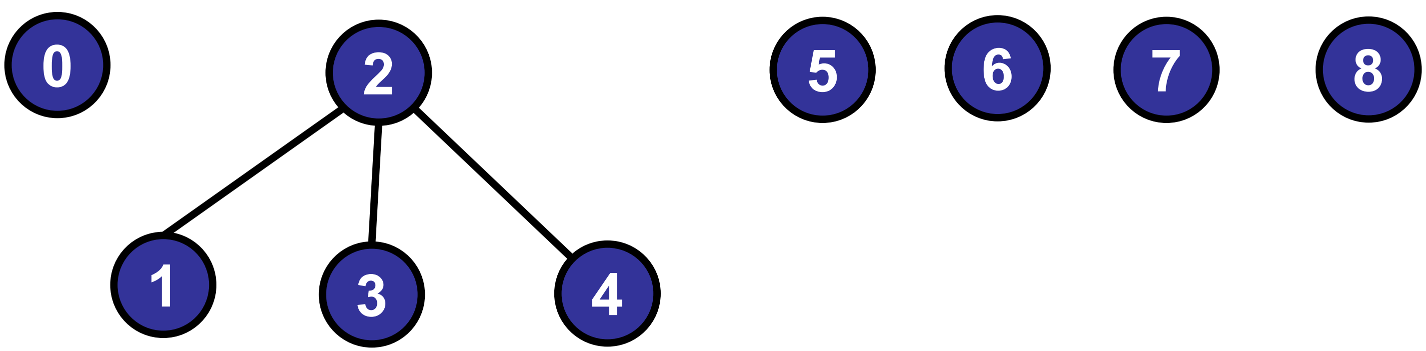
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	6	7	8



Quick Find

Flat trees:

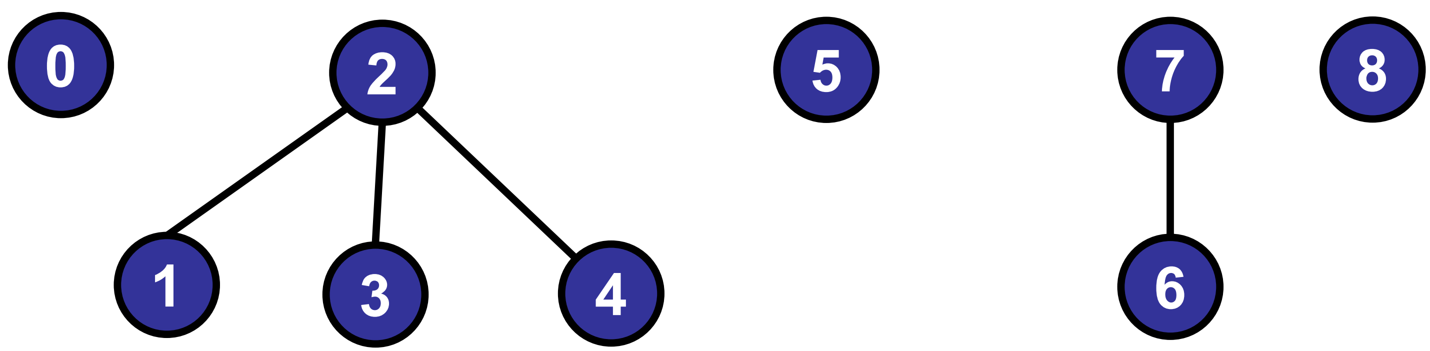
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	6	7	8



Quick Find

```
Flat trees:
```

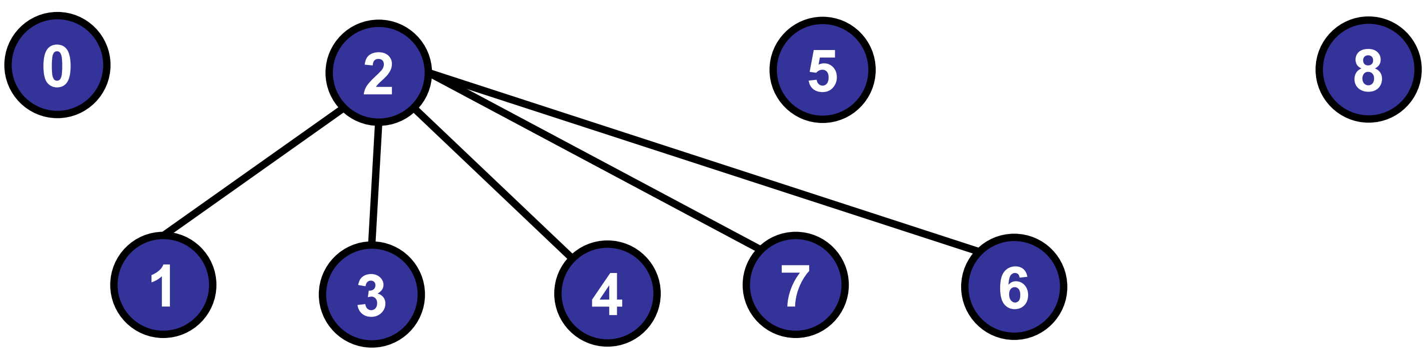
object	0	1	2	3	4	5	6	7	8
component identifier	0	2	2	2	2	5	7	7	8



Quick Find

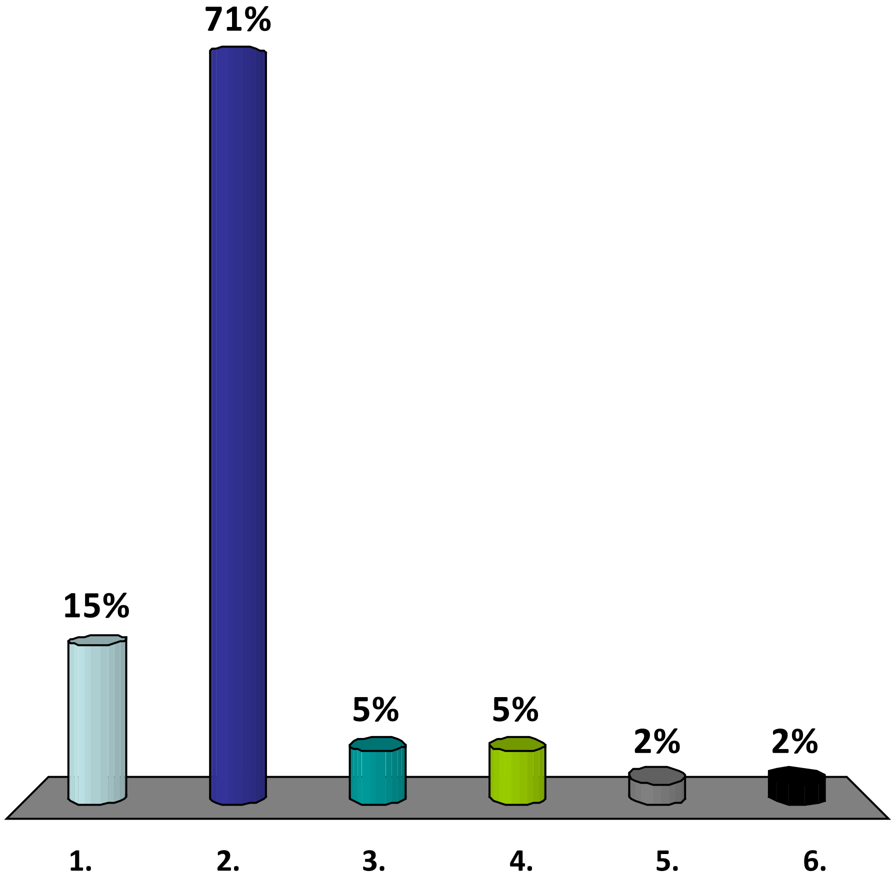
Flat trees:

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



Running time of (Find, Union):

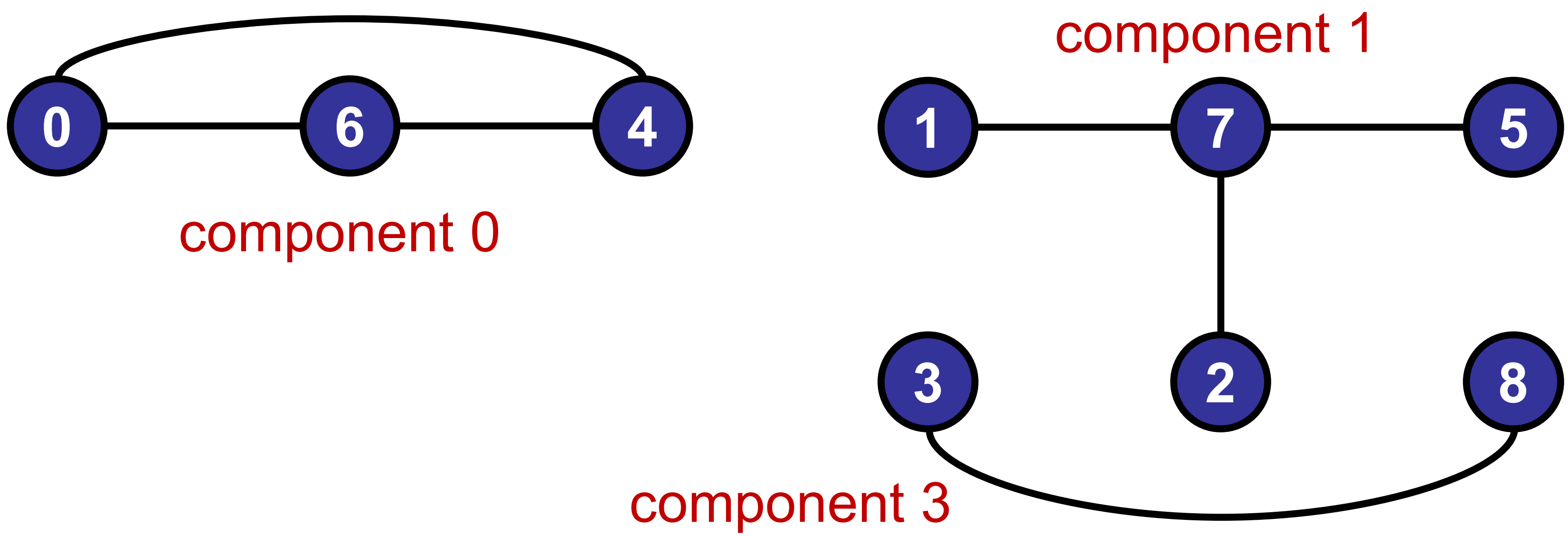
1. O(1), O(1)
2. O(1), O(n)
3. O(n), O(1)
4. O(n), O(n)
5. O(log n), O(log n)
6. None of the above.



Quick Find

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

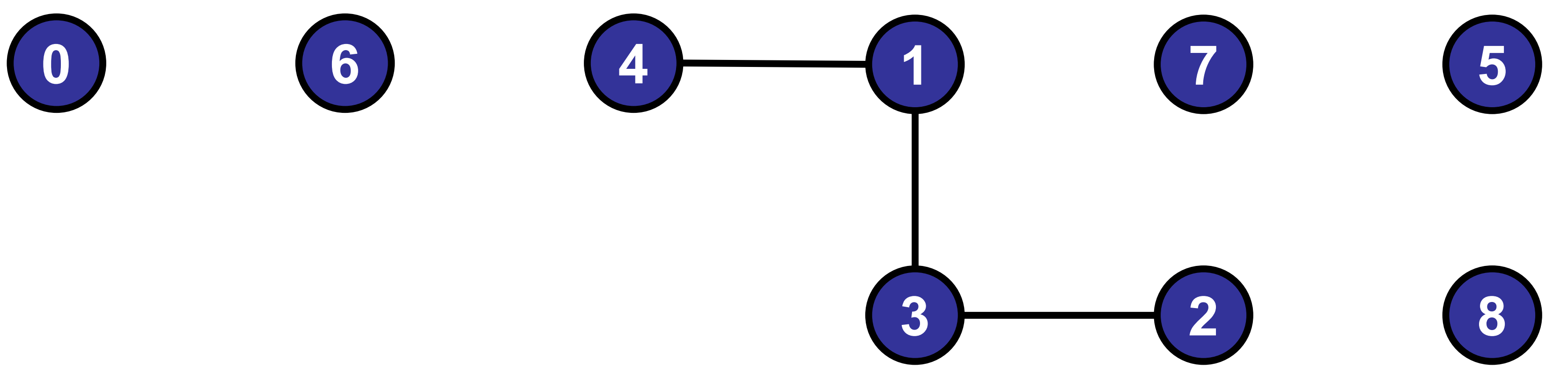
object	0	1	2	3	4	5	6	7	8
component identifier	0	1	1	3	0	1	0	1	3



Quick Find

```
union(int p, int q)
   for (inti=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	2	2	2	2	5	6	7	8



Roadmap

Part II: Disjoint Set

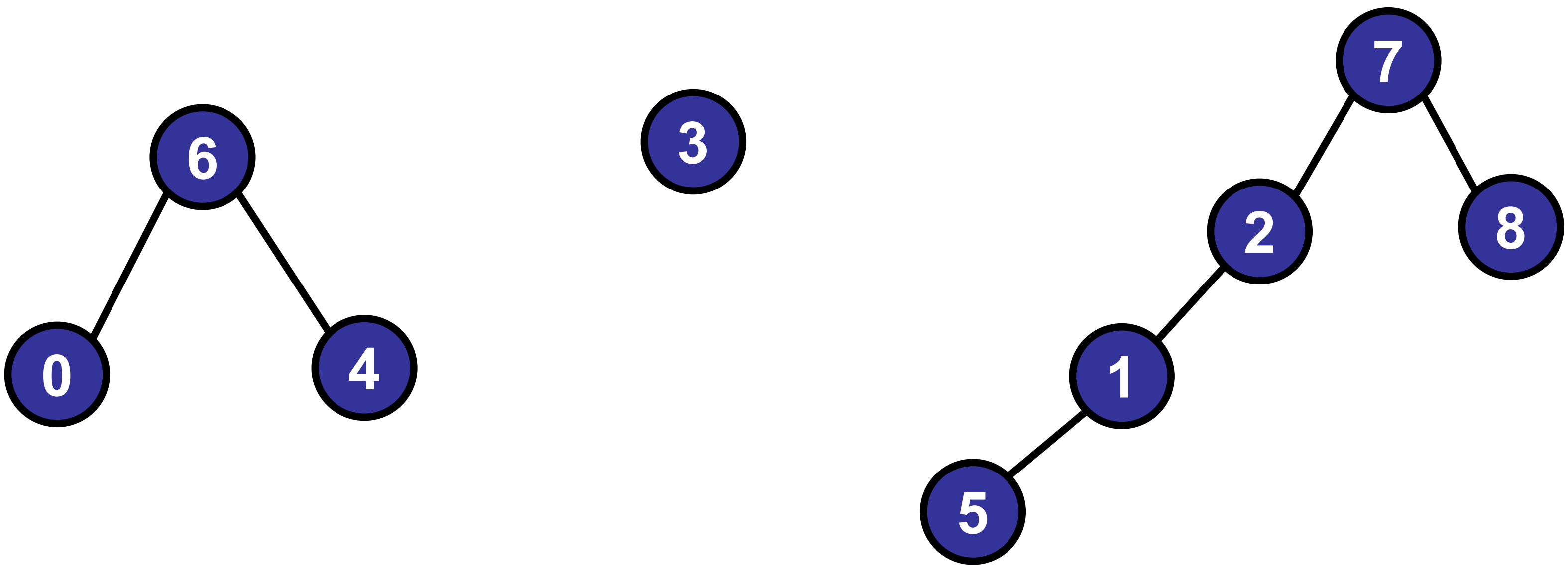
- Problem: Dynamic Connectivity
- Algorithm: Quick-Find
- Algorithm: Quick-Union
- Optimizations
- Applications

Quick Union

Data structure:

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

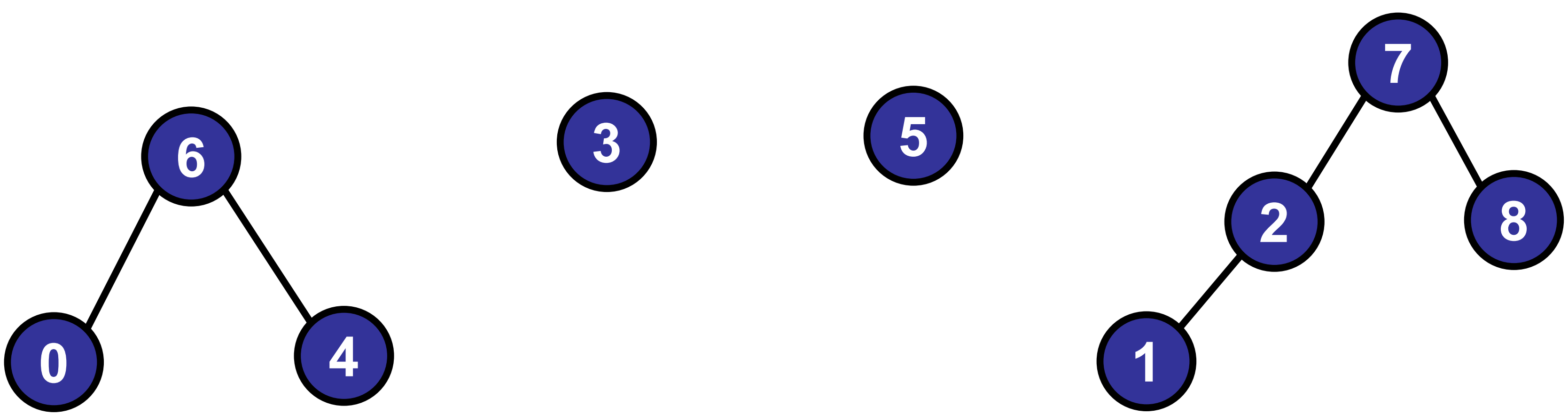
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Quick Union

```
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

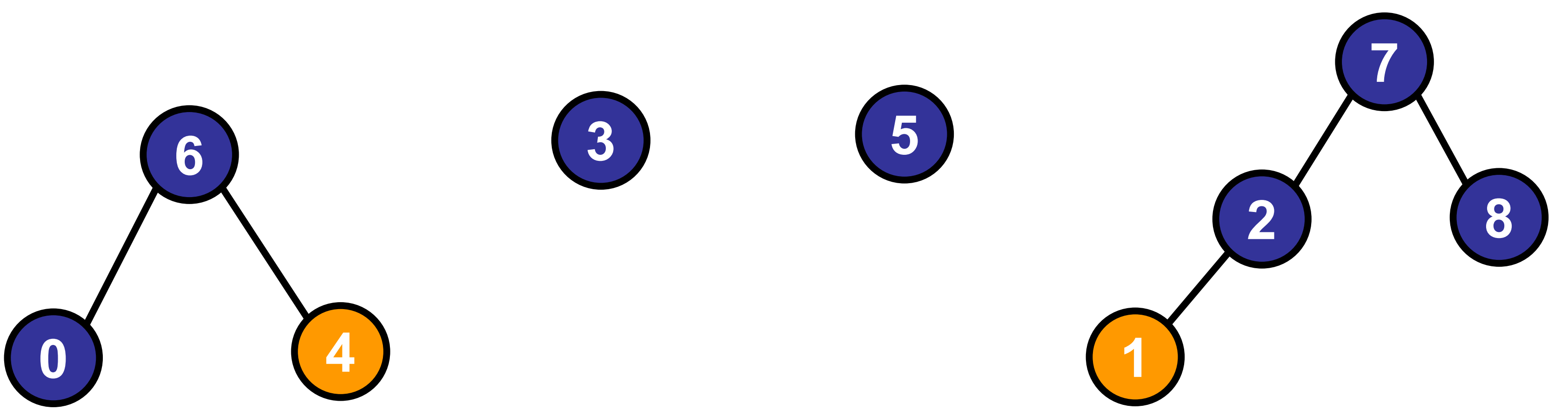


Quick Union

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



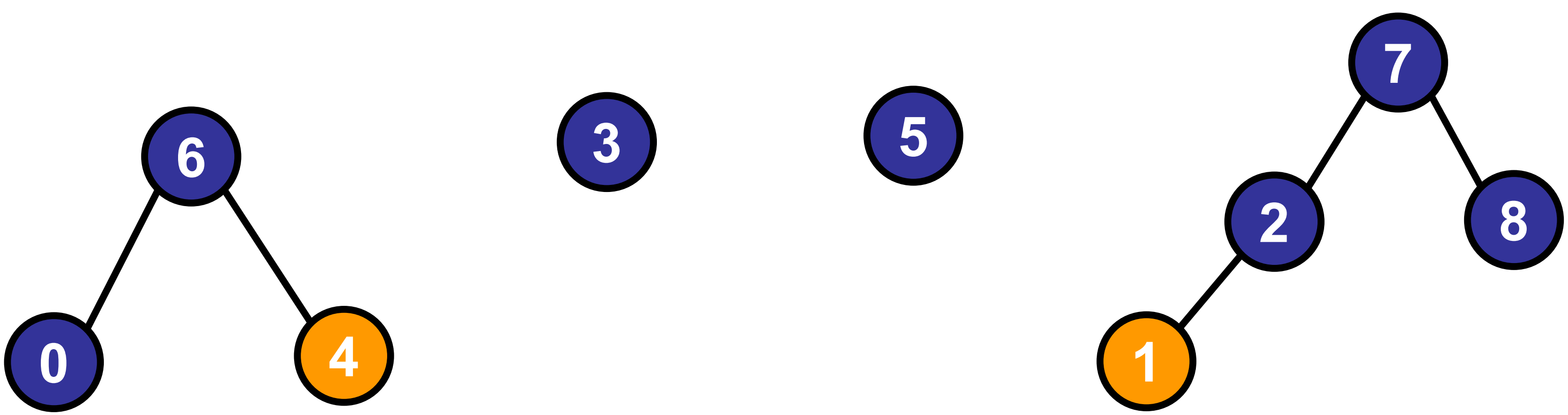
Quick Union

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

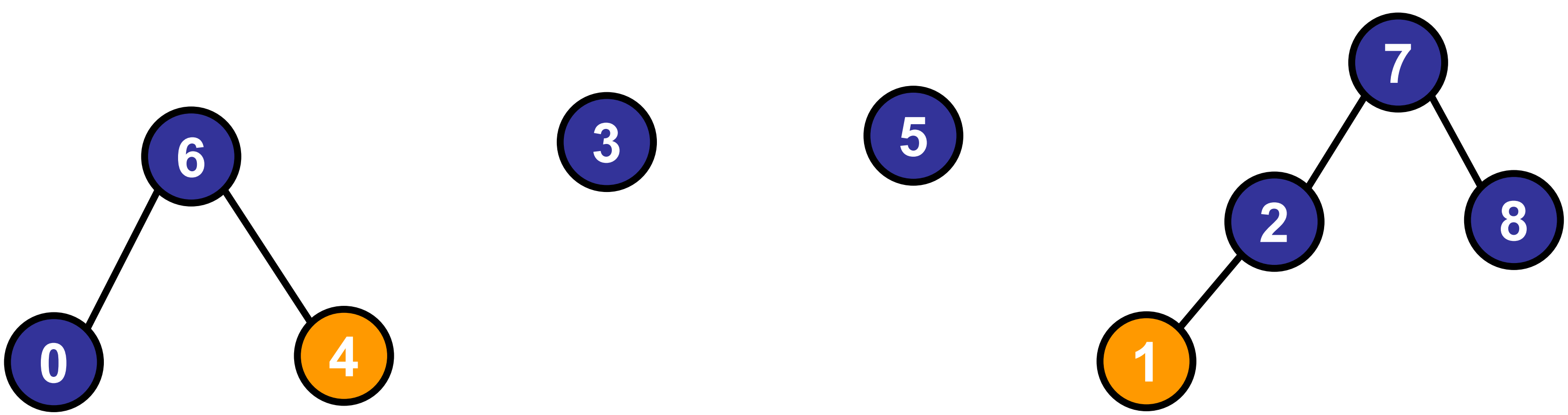


Quick Union

Example: `find(4, 1)`

```
4 → 6 → 6

1 → 2 → 7 → 7

return (6 == 7) → false
```

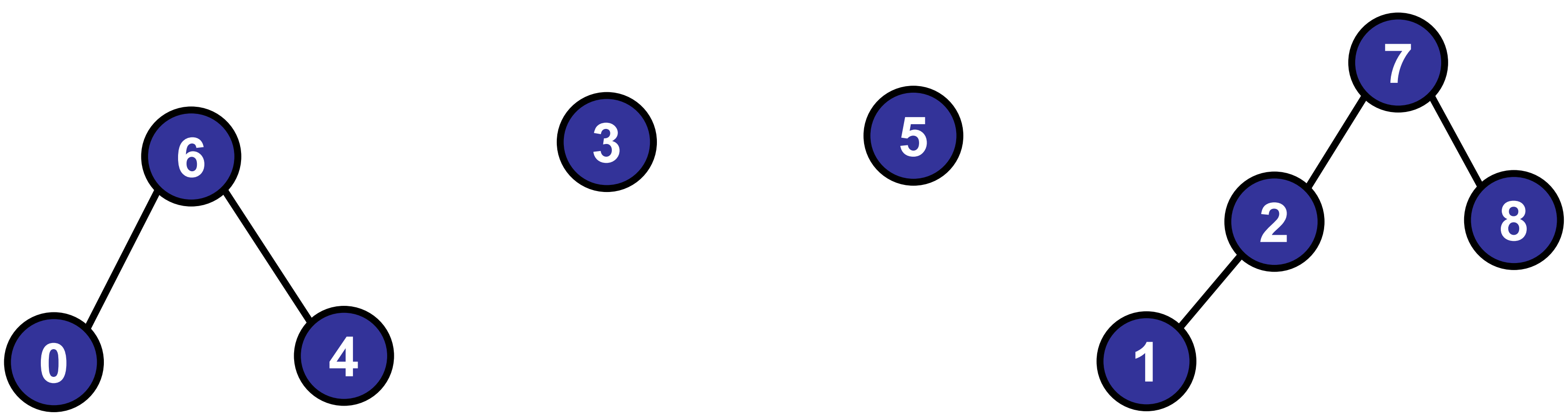
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

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

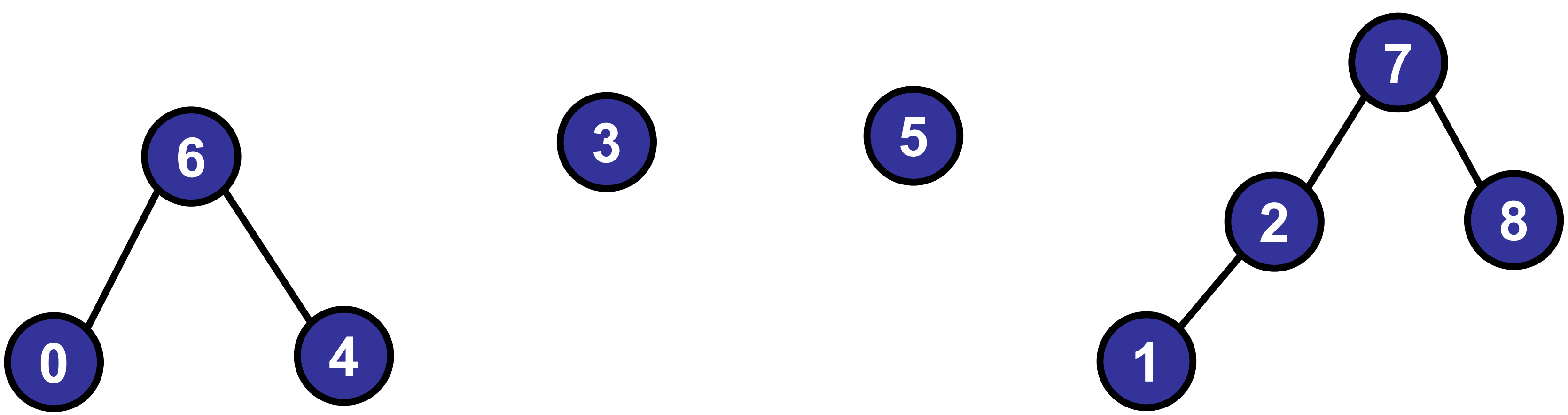
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

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

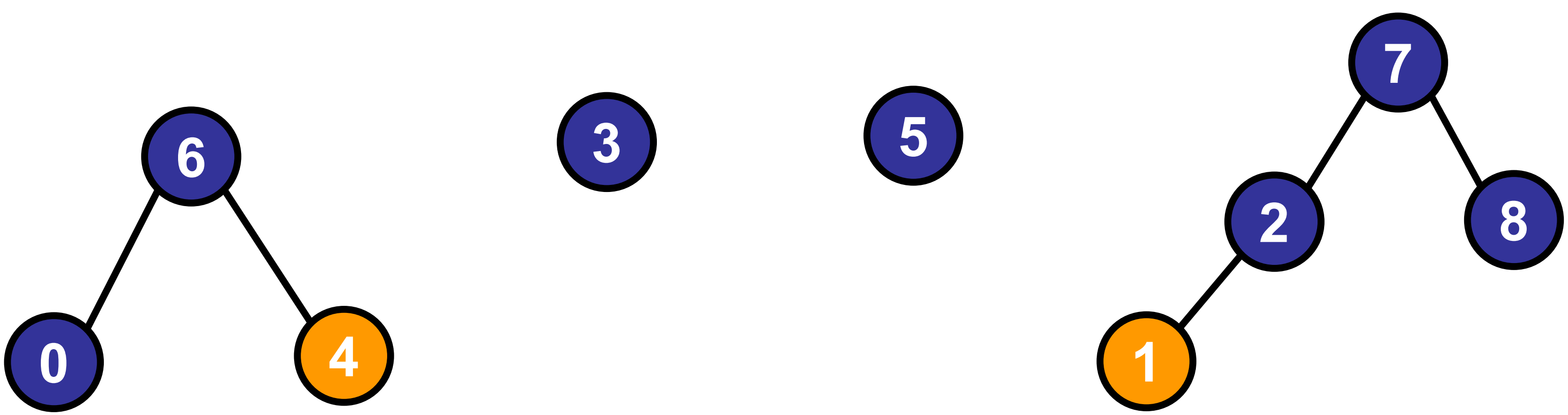
object	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
parent	**6**	**2**	**7**	**3**	**6**	**1**	**6**	**7**	**7**



Quick Union

Example: `union(1, 4)`

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



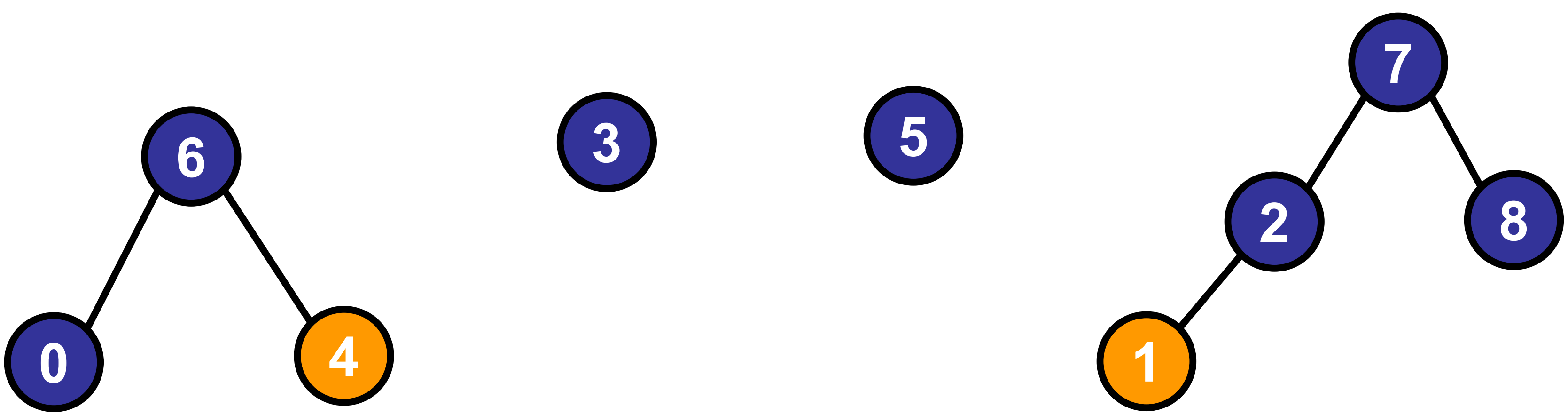
Quick Union

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



Quick Union

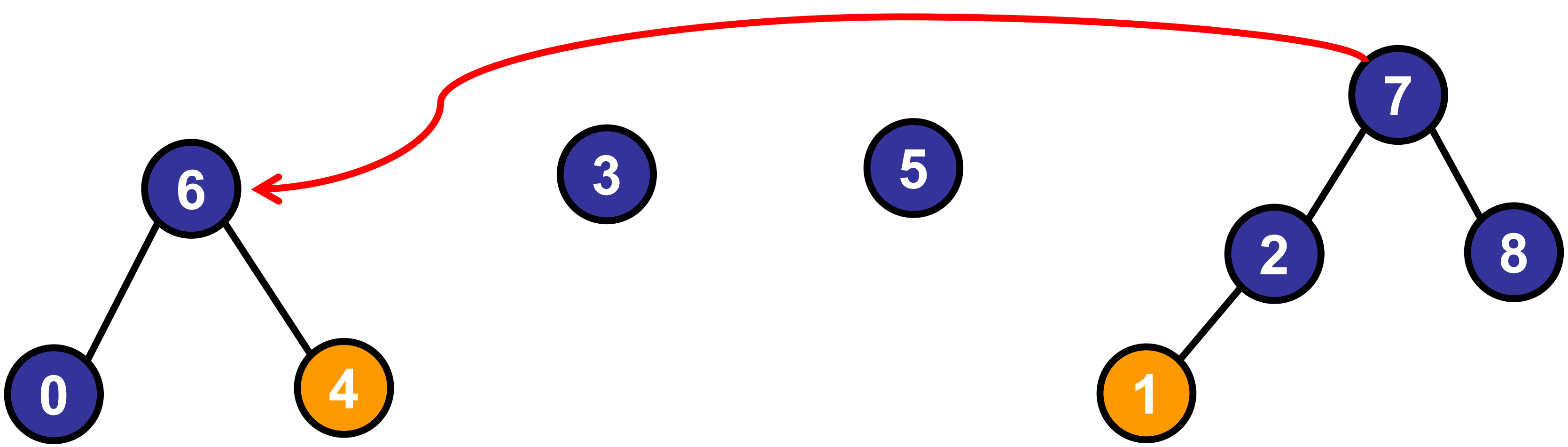
Example: union(1, 4)

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

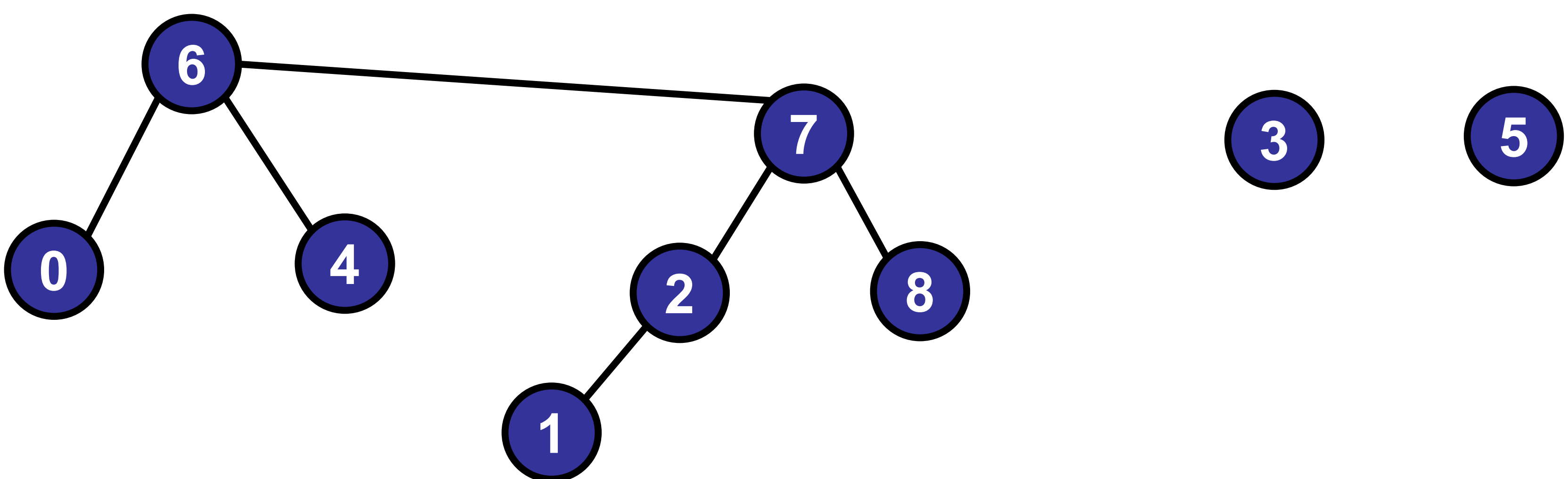


Quick Union

Example: `union(1, 4)`

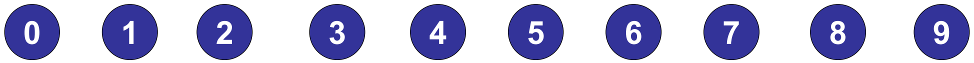
```
4 → 6 → 6

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**



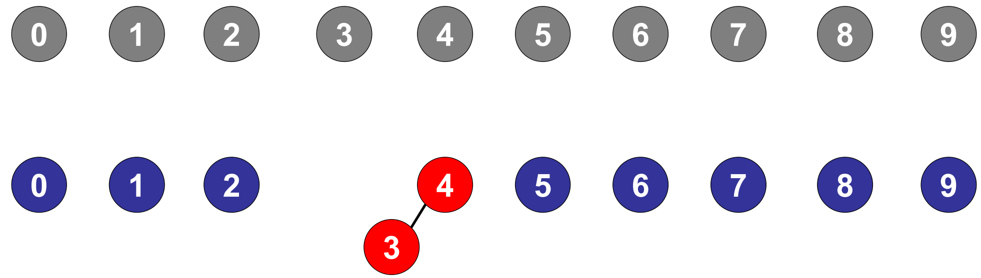
Example:

P	0	1	2	3	4	5	6	7	8	9



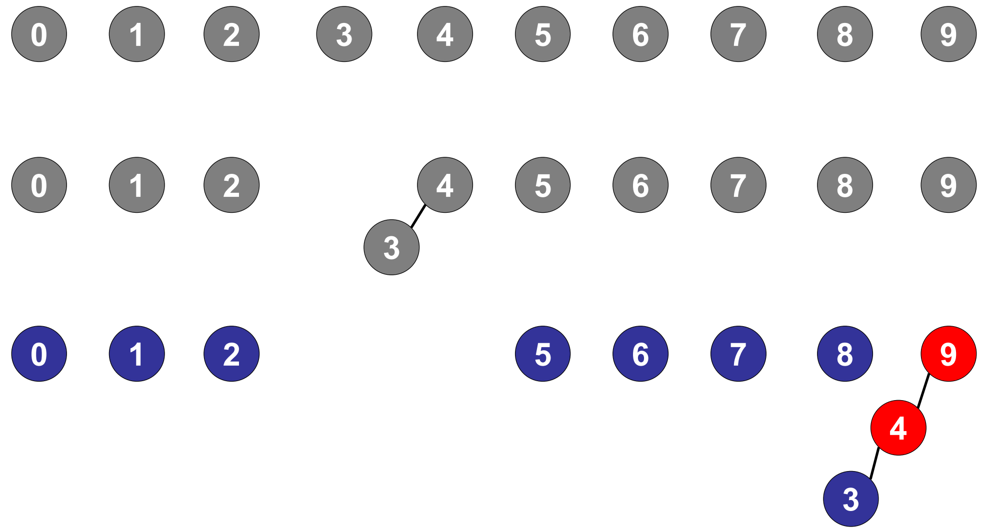
Example:

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



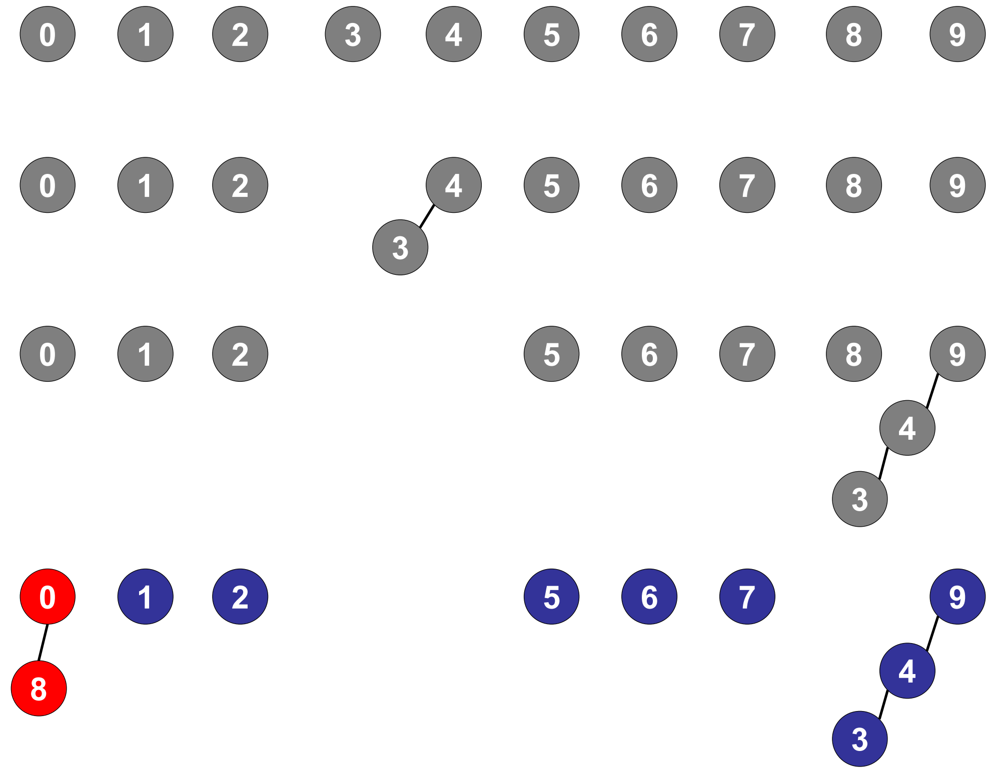
Example:

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



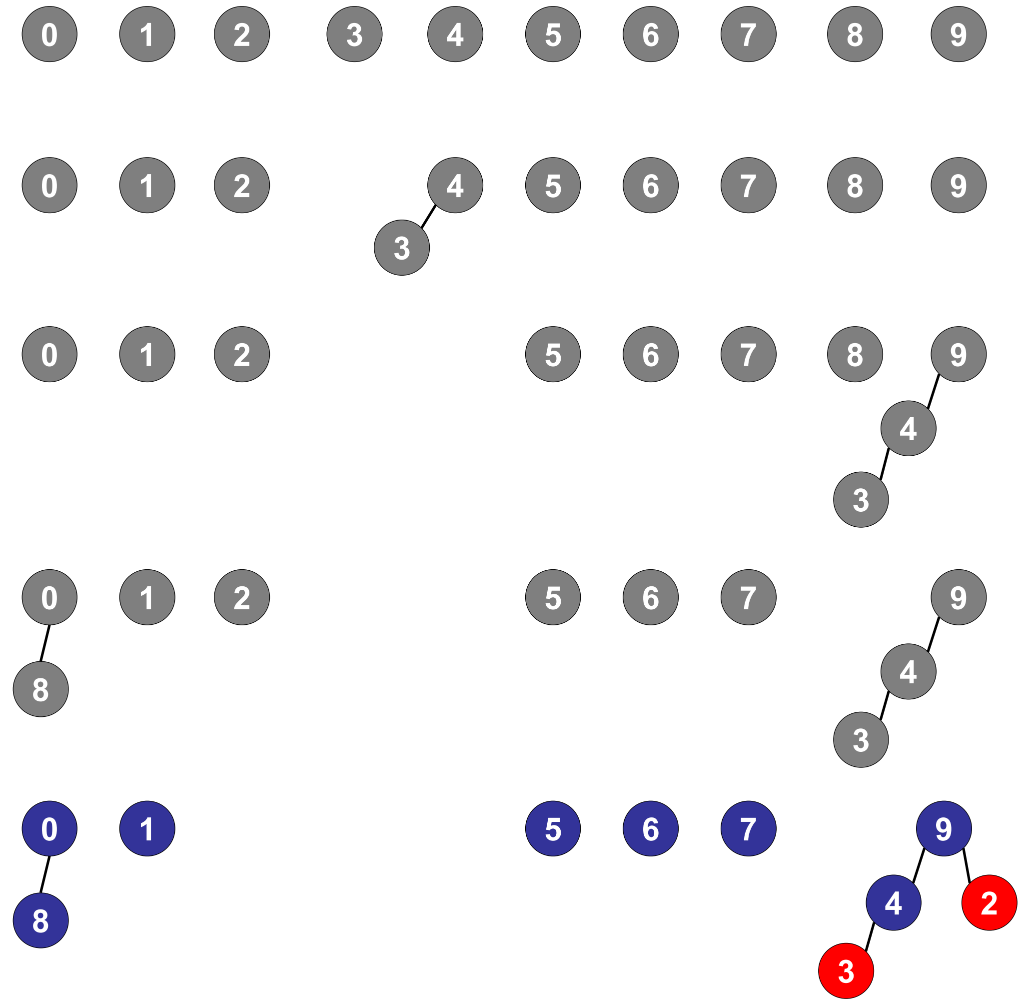
Example:

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



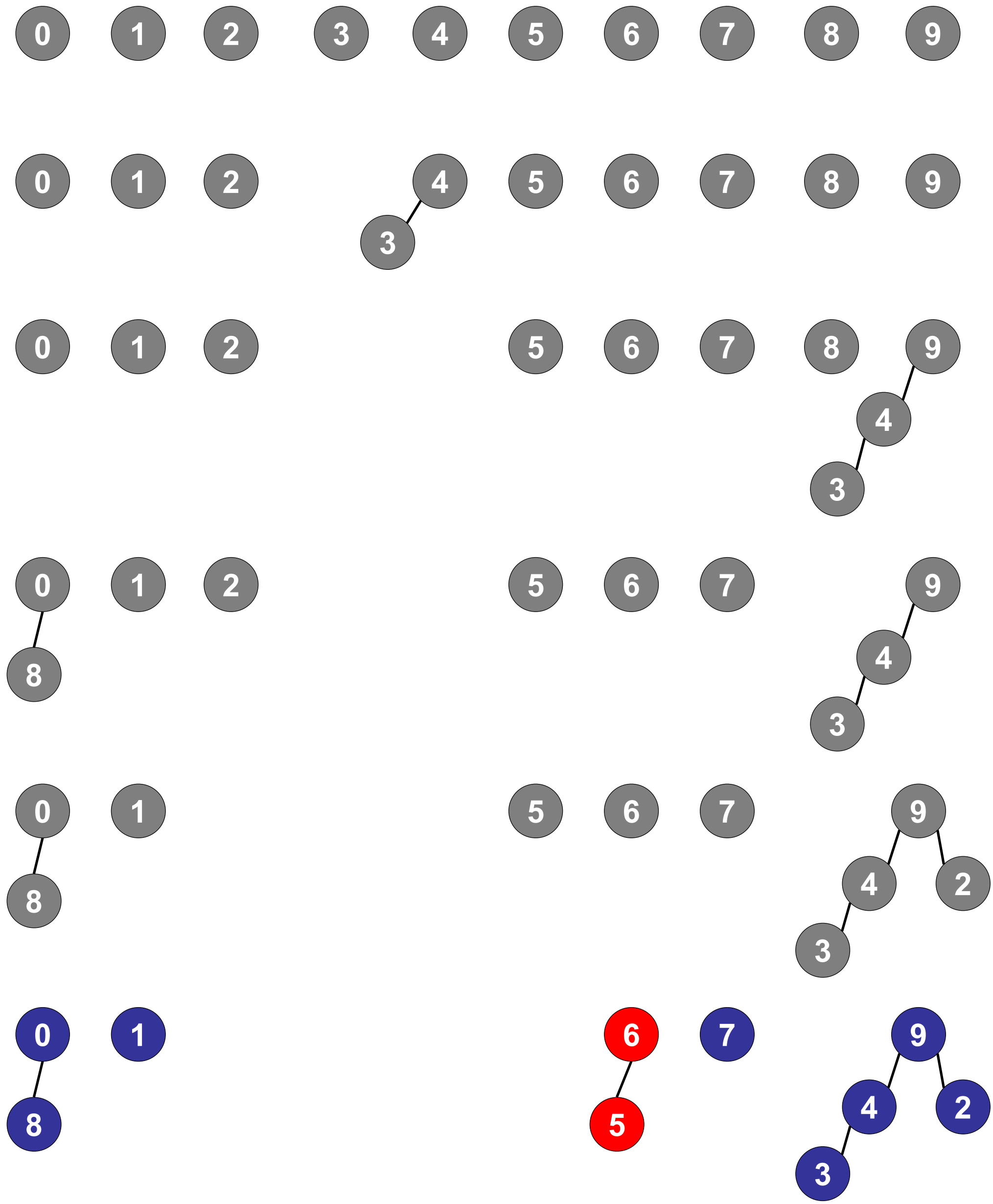
Example:

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

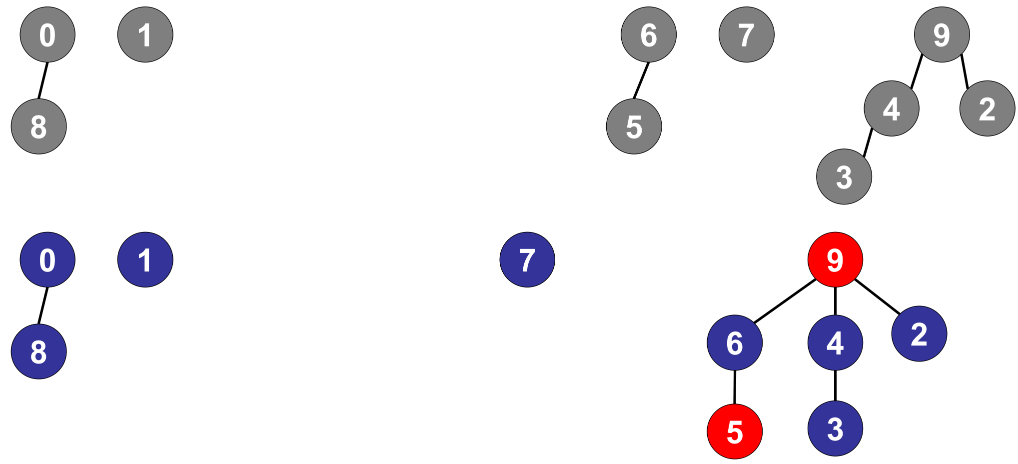


Example:

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

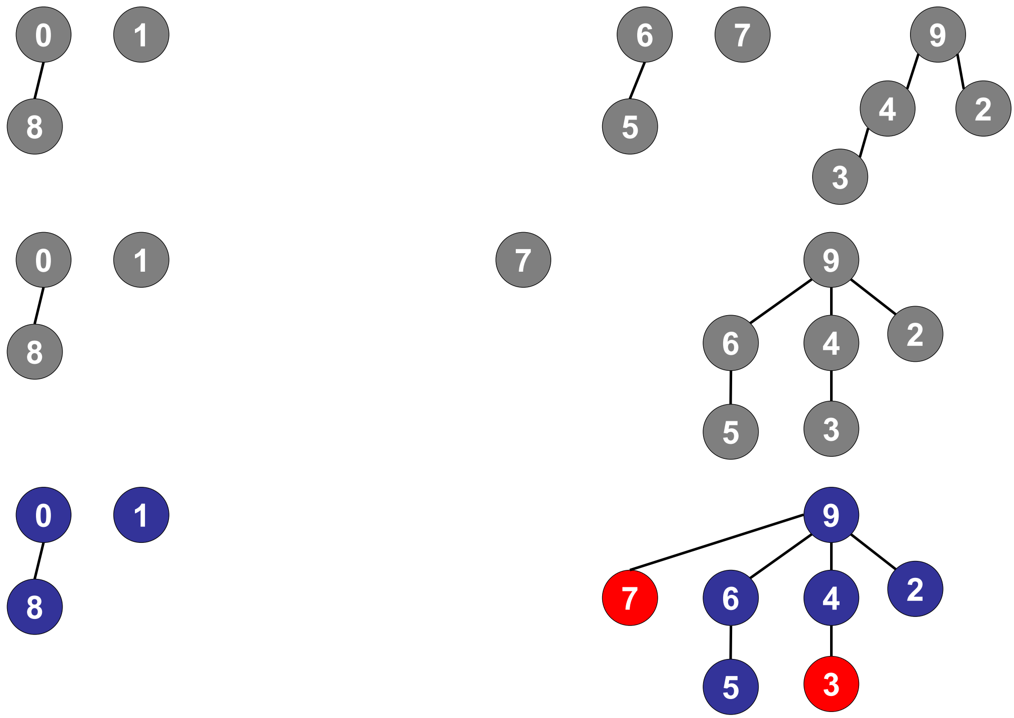
Example:

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



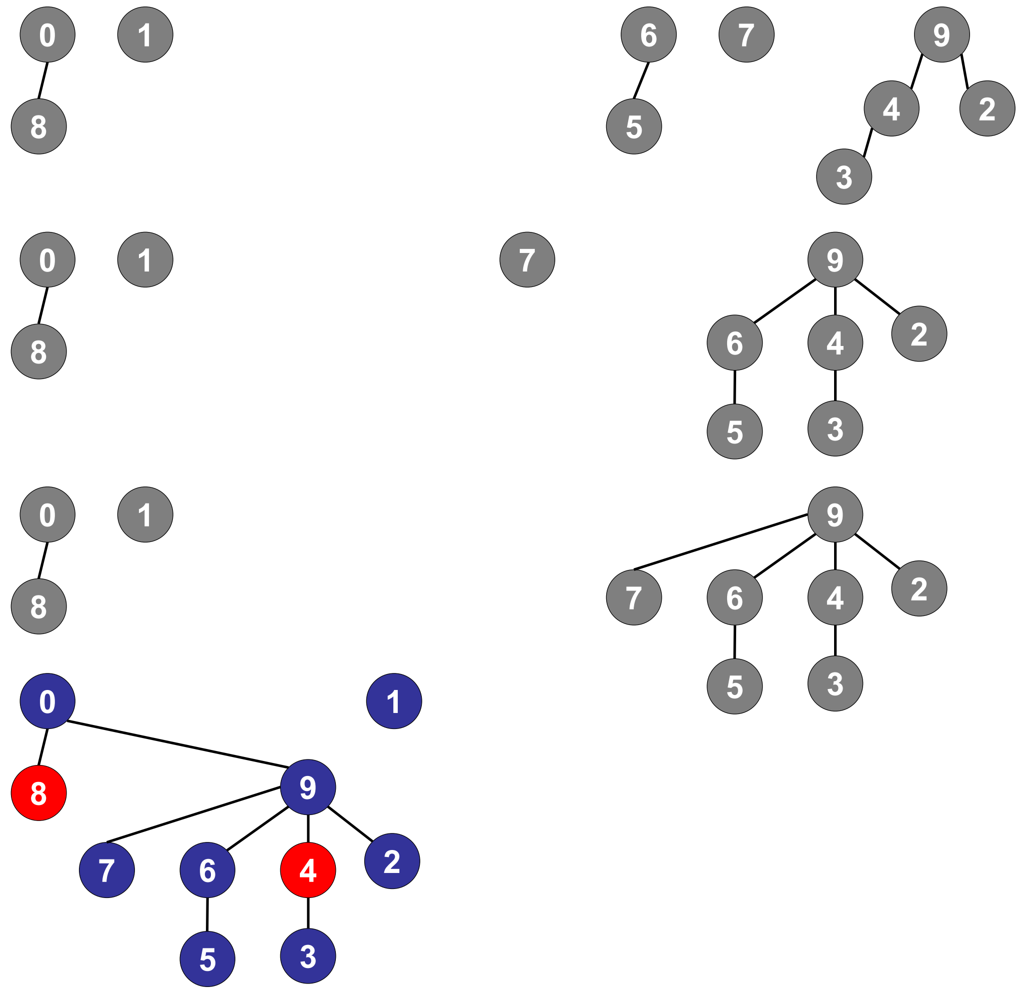
Example:

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



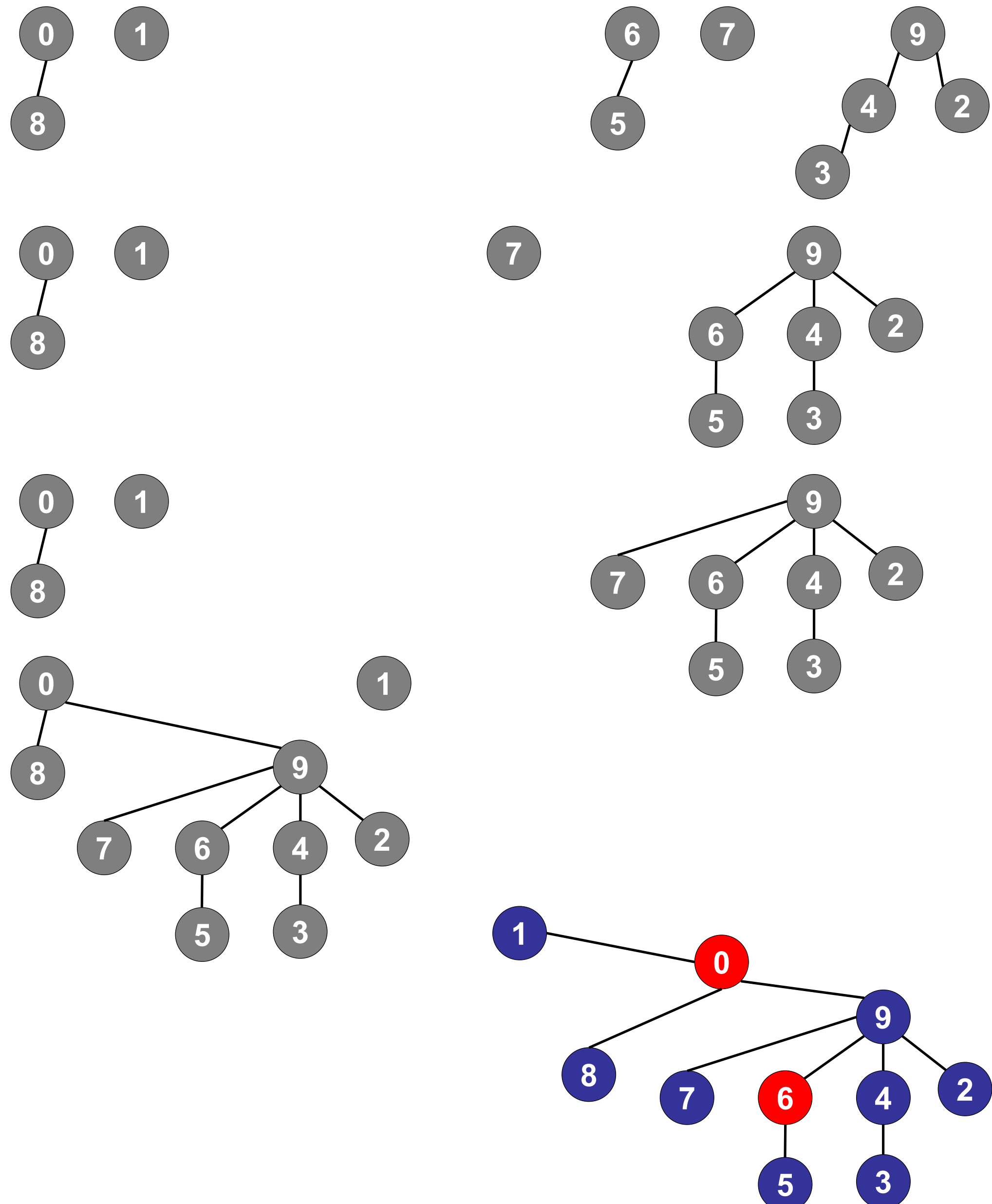
Example:

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:

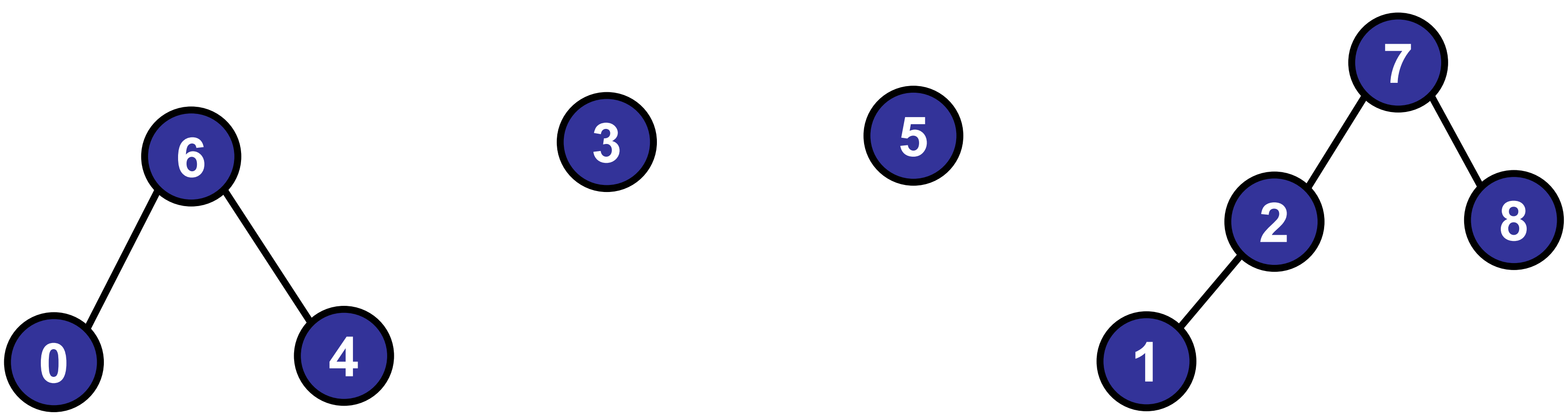
P	0	1	2	3	4	5	6	7	8	9
3-4	0	1	2	4	4	5	6	7	8	9
4-9	0	1	2	4	9	5	6	7	8	9
8-0	0	1	2	4	9	5	6	7	0	9
2-3	0	1	9	4	9	5	6	7	0	9
5-6	0	1	9	4	9	6	6	7	0	9
5-9	0	1	9	4	9	6	9	7	0	9
7-3	0	1	9	4	9	6	9	9	0	9
4-8	0	1	9	4	9	6	9	9	0	0
6-1	1	1	9	4	9	6	9	9	0	0



Quick Union

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

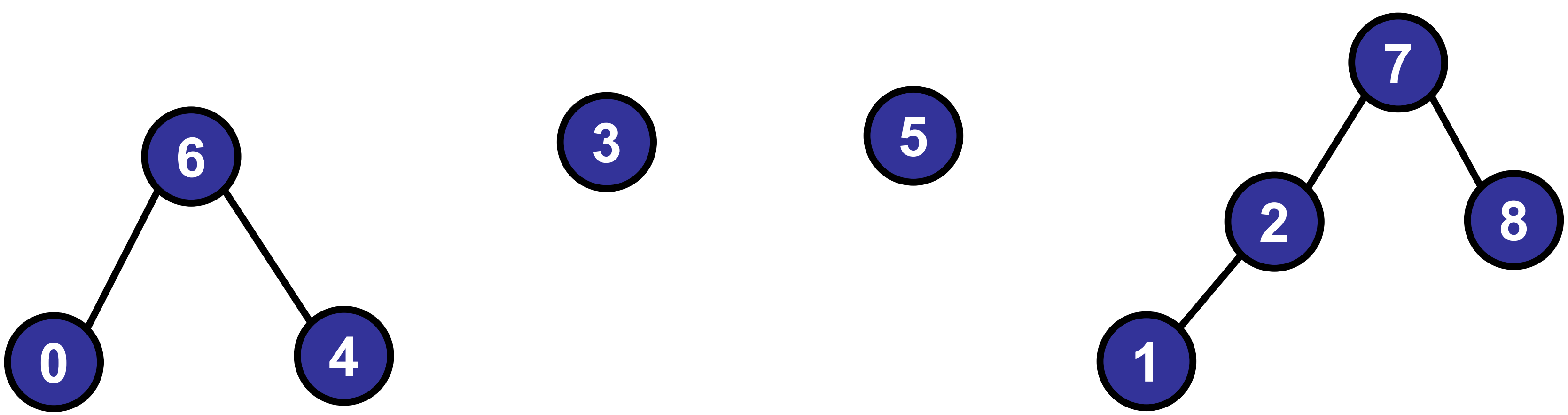
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

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

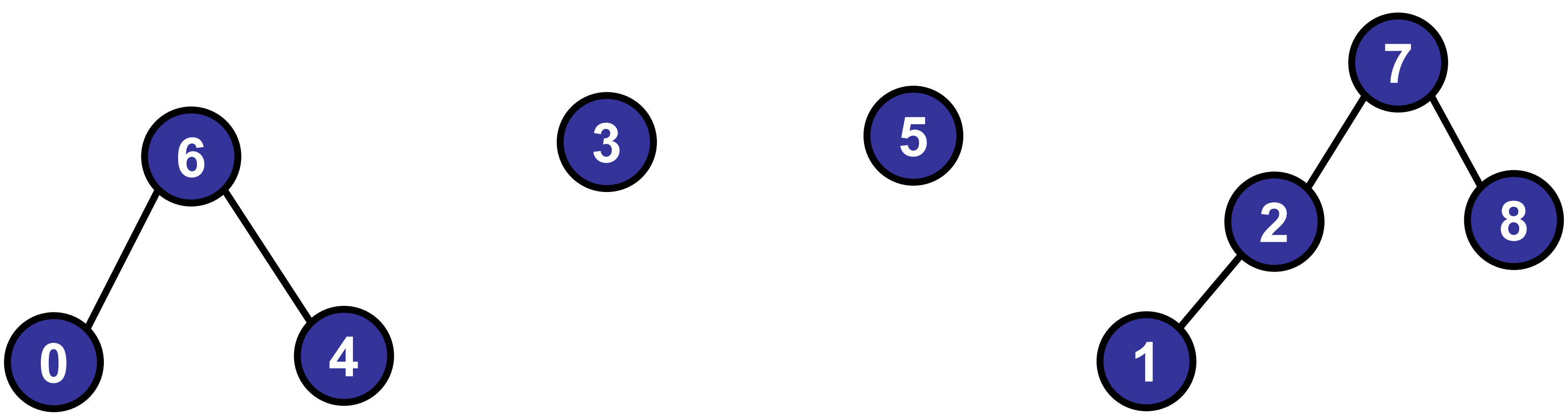
object	0	1	2	3	4	5	6	7	8
parent	6	2	7	3	6	1	6	7	7



Quick Union

```
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

Quick-find is slow:

- Find is fast
- Union is expensive
- Tree is flat

Quick-union is slow:

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

	find	union
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)

Roadmap

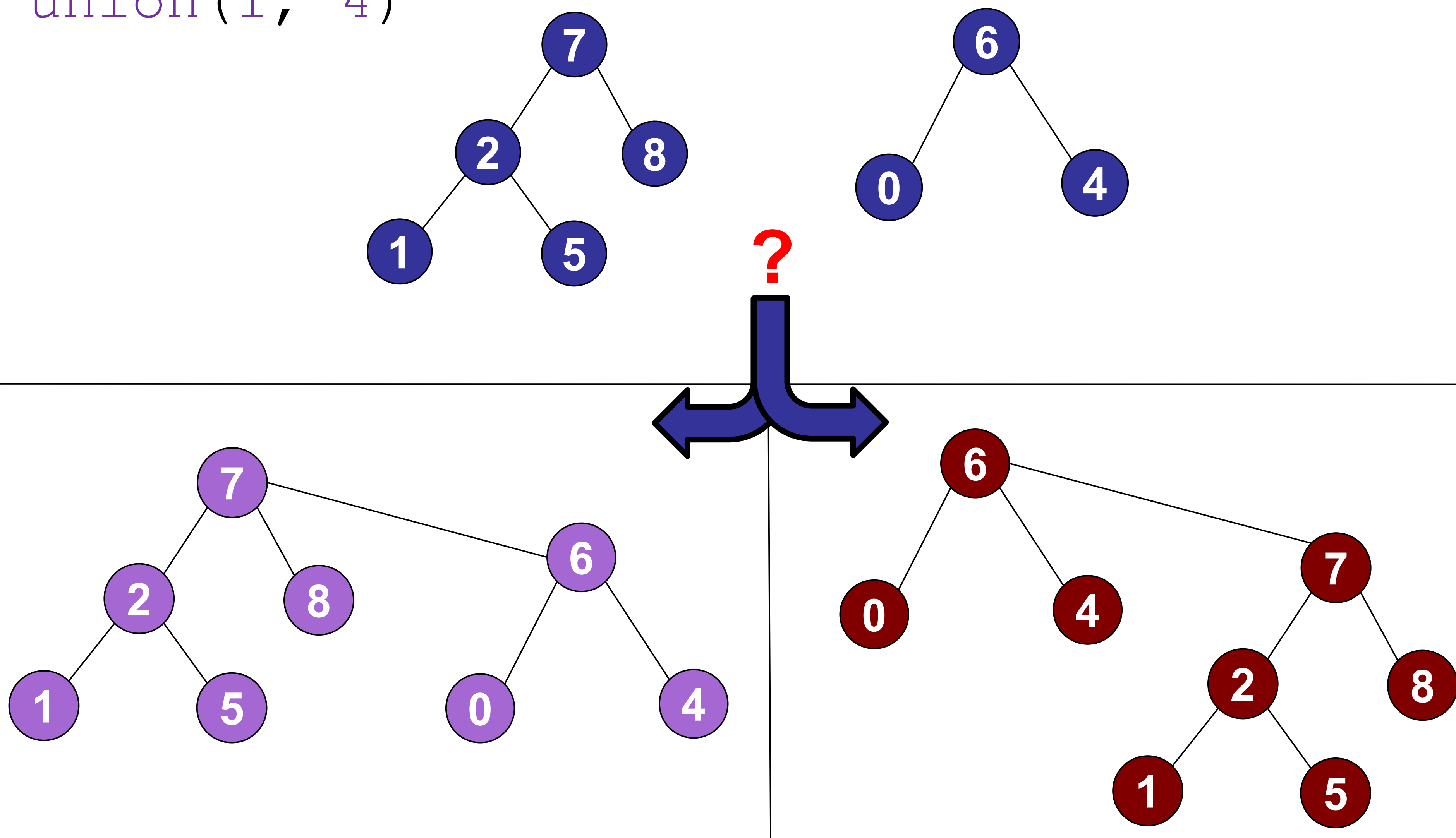
Part II: Disjoint Set

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

Weighted Union

Question: which tree should you make the root?

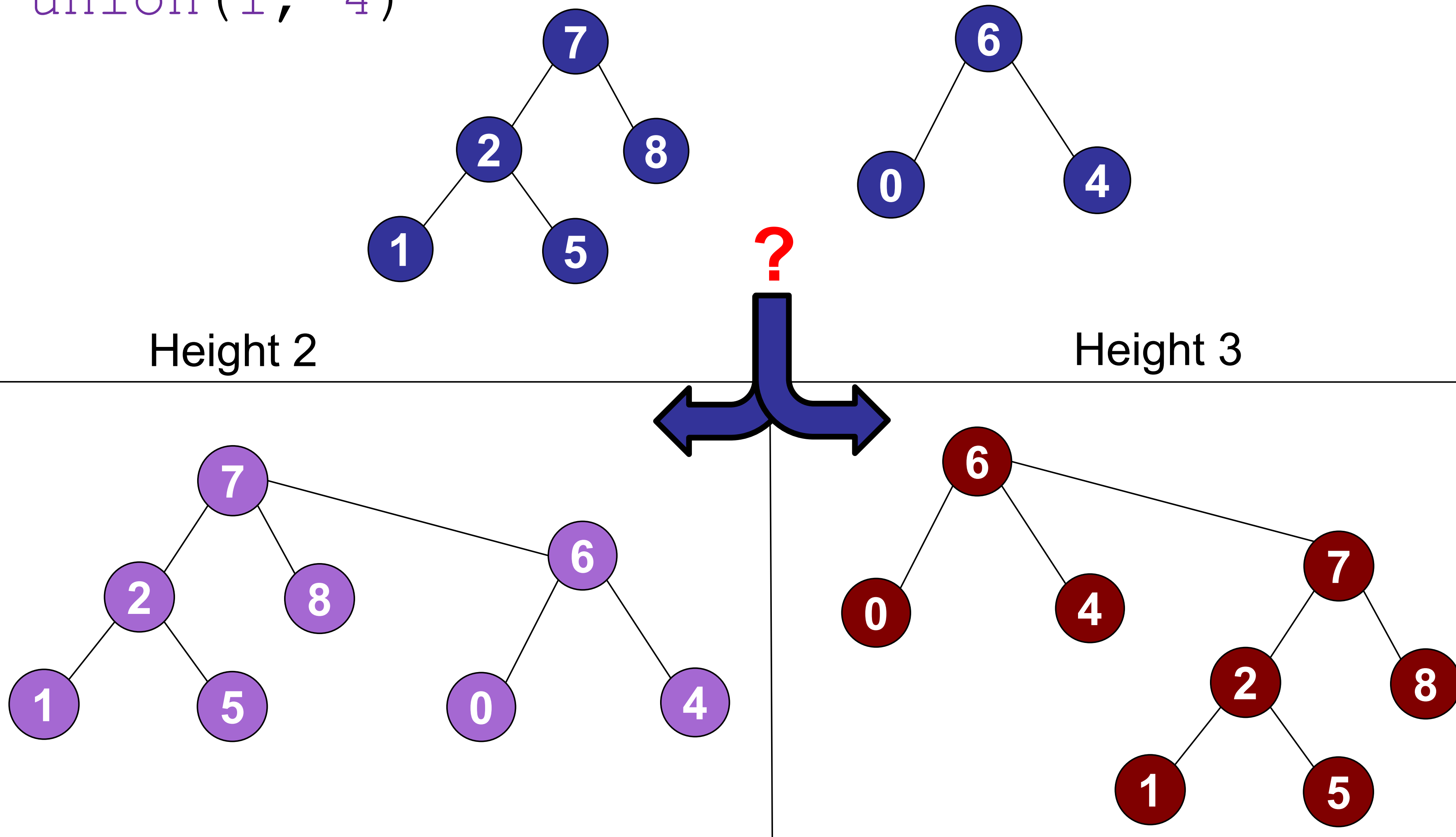
`union(1, 4)`



Weighted Union

Question: which tree should you make the root?

`union(1, 4)`



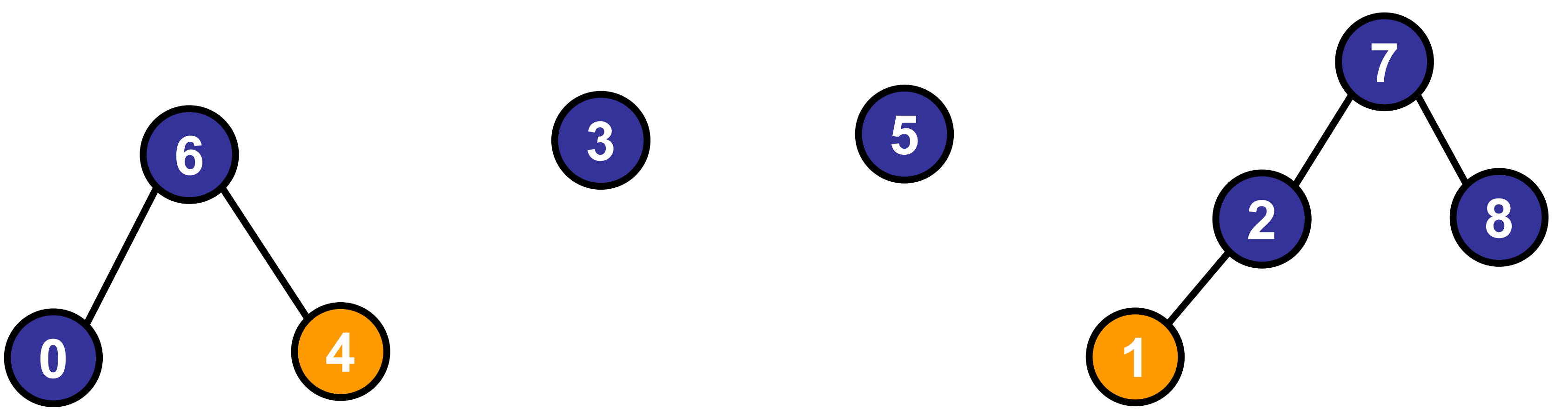
Weighted Union

```
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(1, 4)`

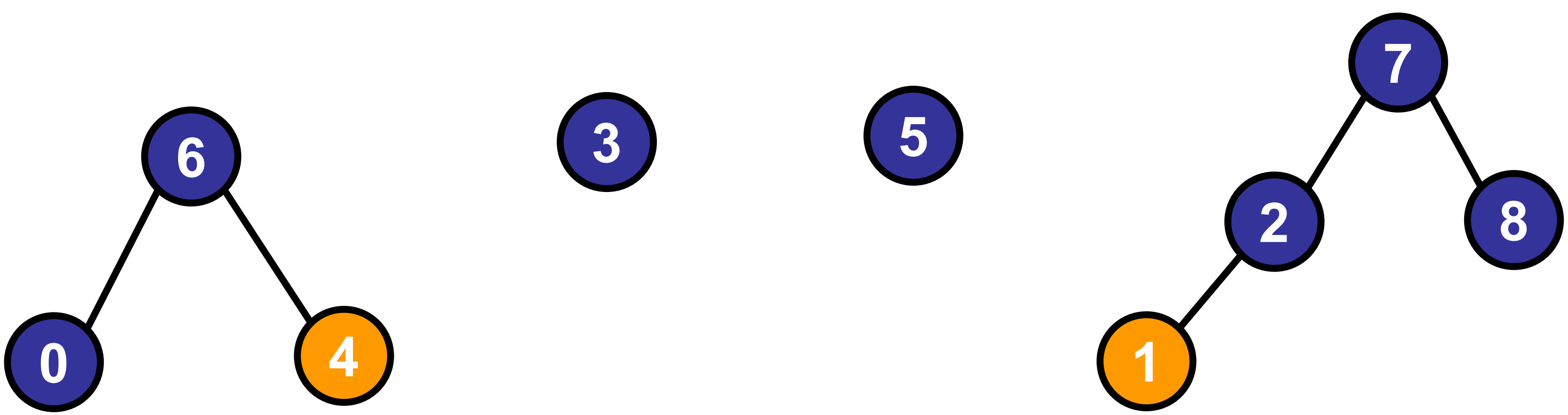
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

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

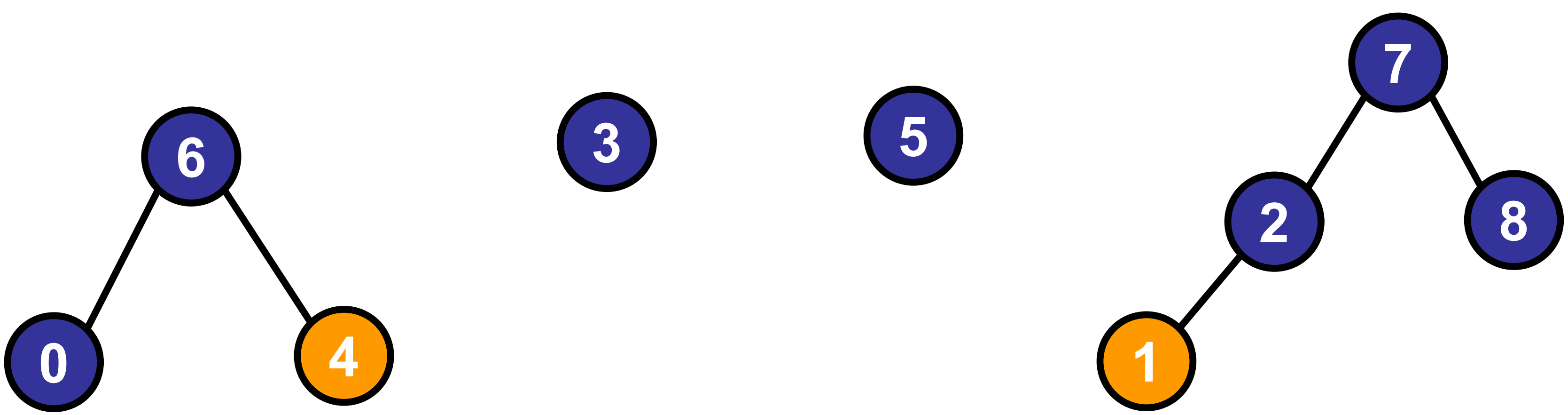
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

`union(1, 4)`

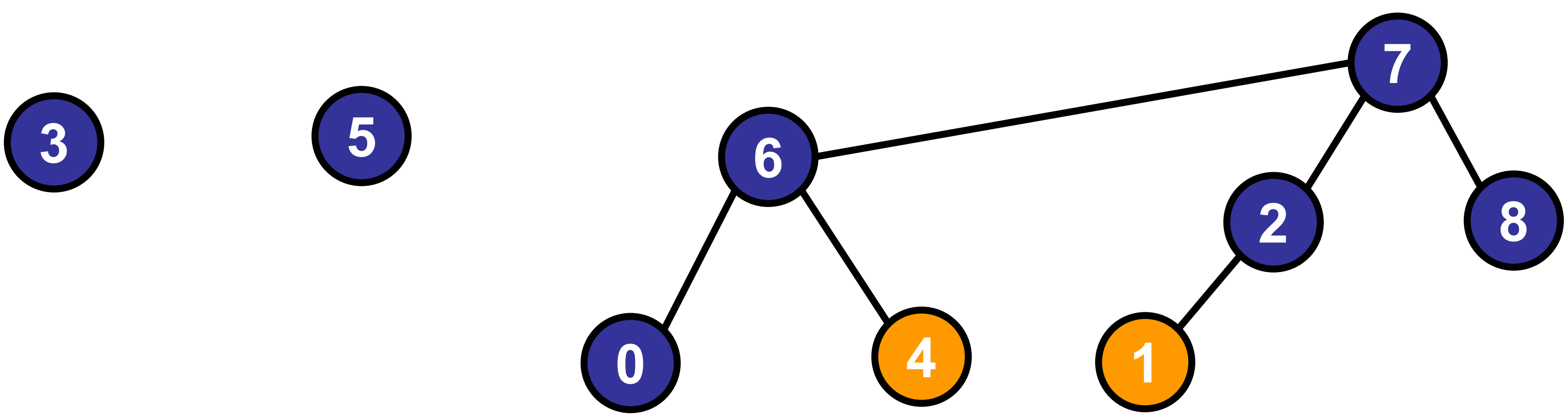
object	0	1	2	3	4	5	6	7	8
size	1	1	2	1	1	1	3	4	1
parent	6	2	7	3	6	1	6	7	7



Weighted Union

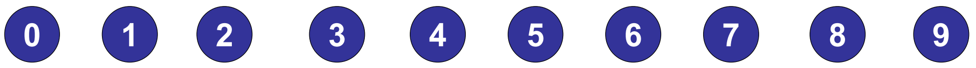
```
union(1, 4)
```

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



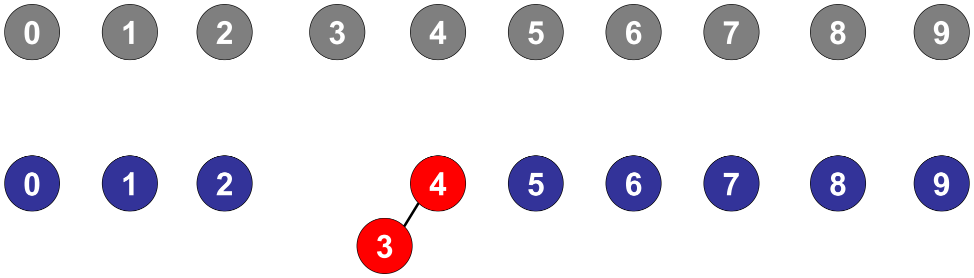
Example: Weighted Union

P	0	1	2	3	4	5	6	7	8	9



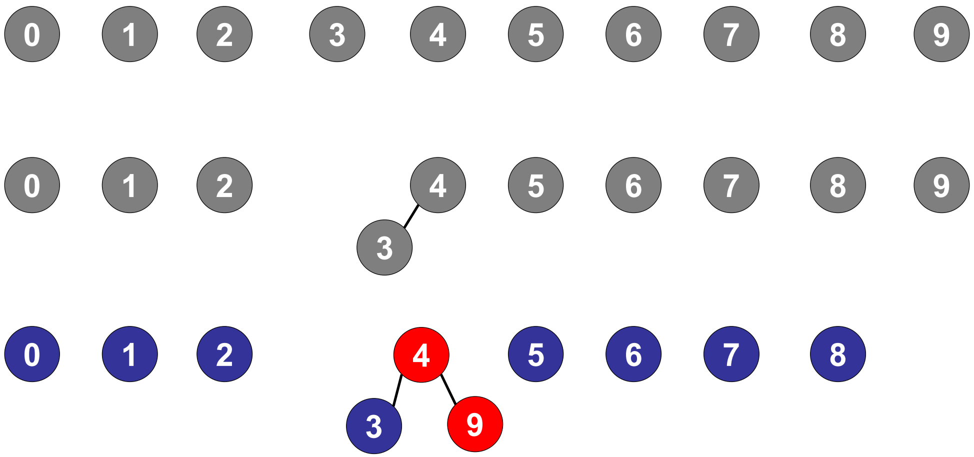
Example: Weighted Union

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



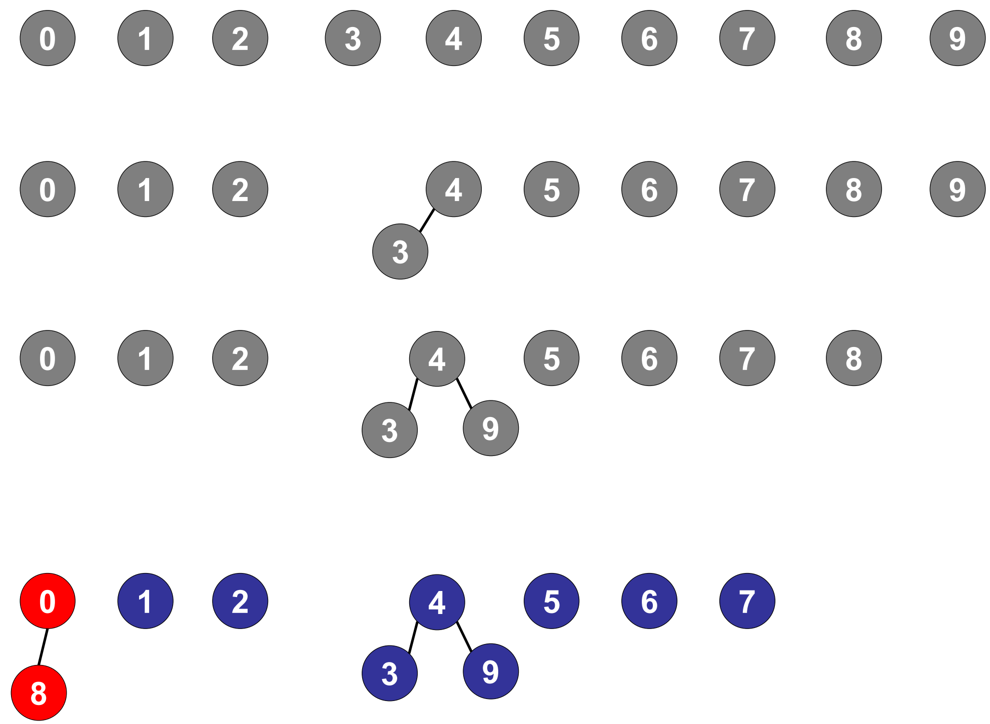
Example: Weighted Union

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



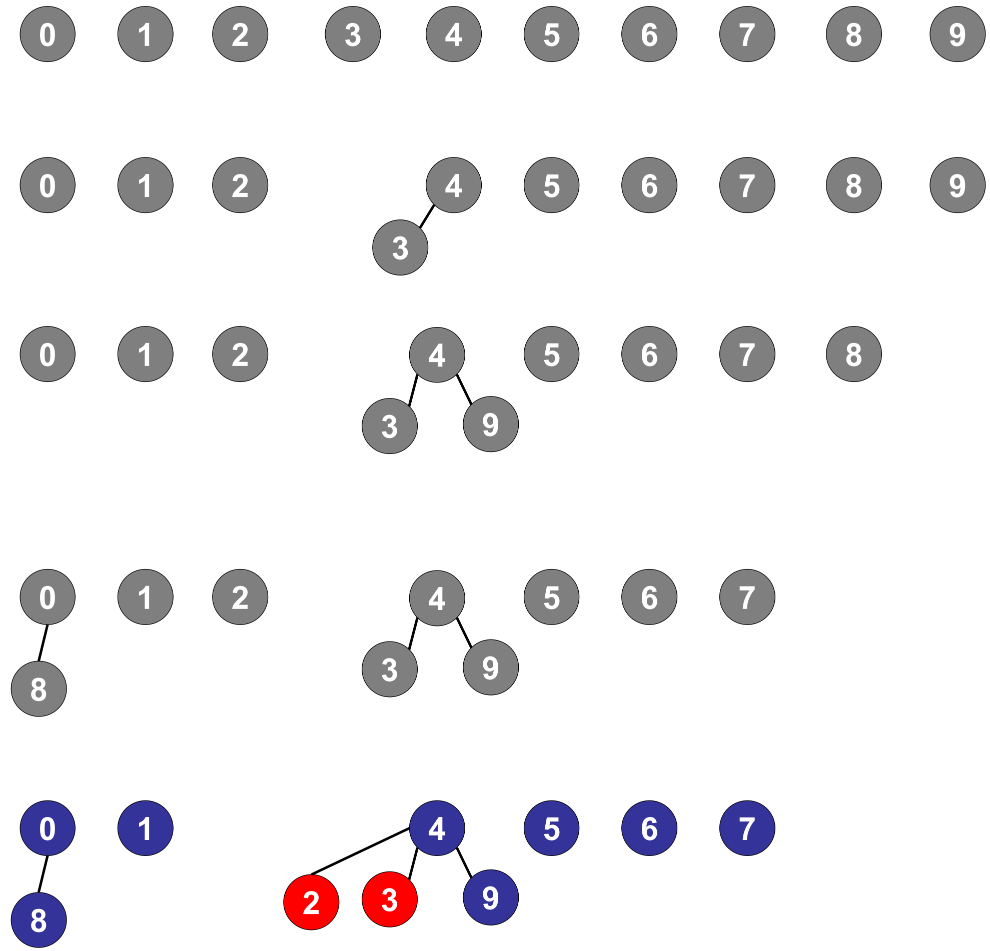
Example: Weighted Union

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



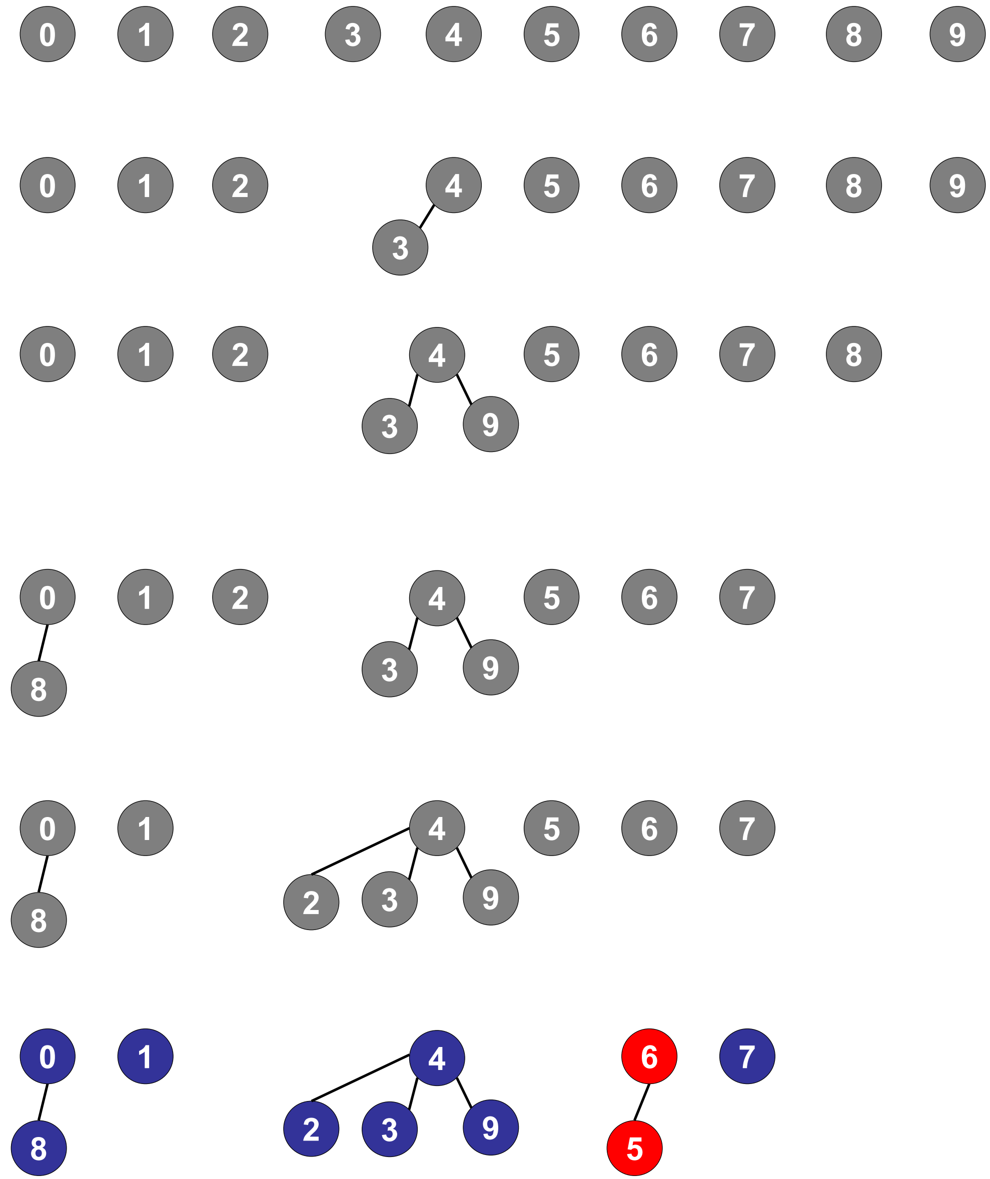
Example: Weighted Union

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



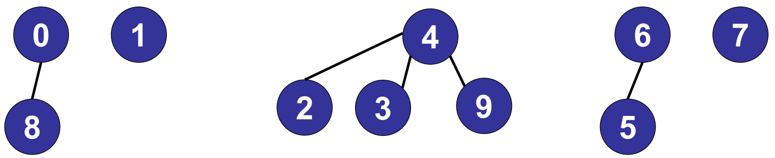
Example: Weighted Union

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



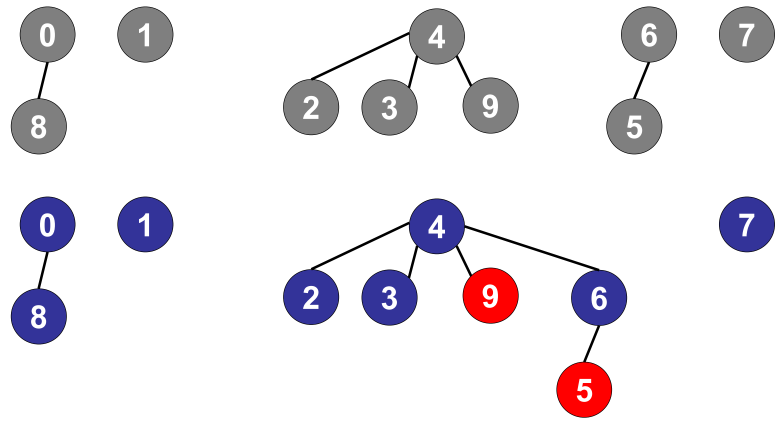
Example: Weighted Union

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



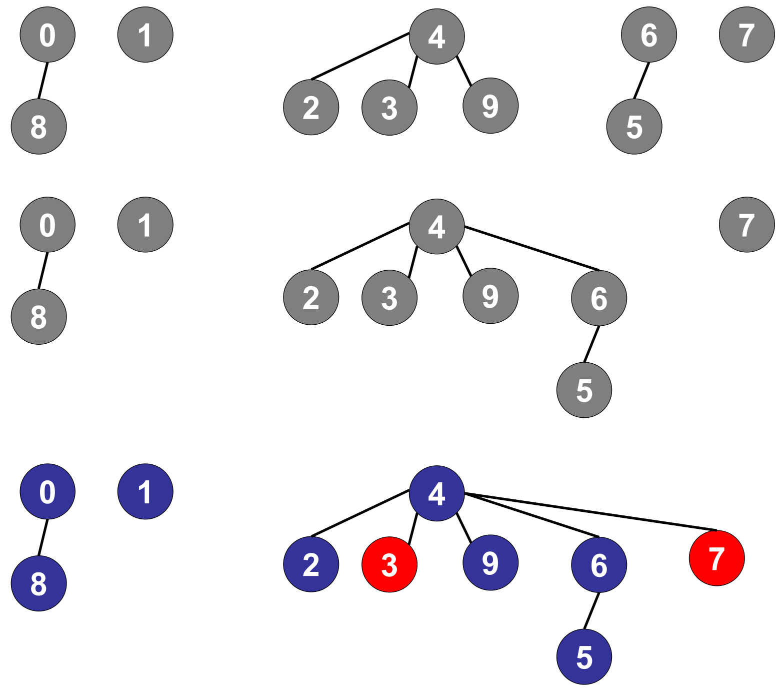
Example: Weighted Union

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



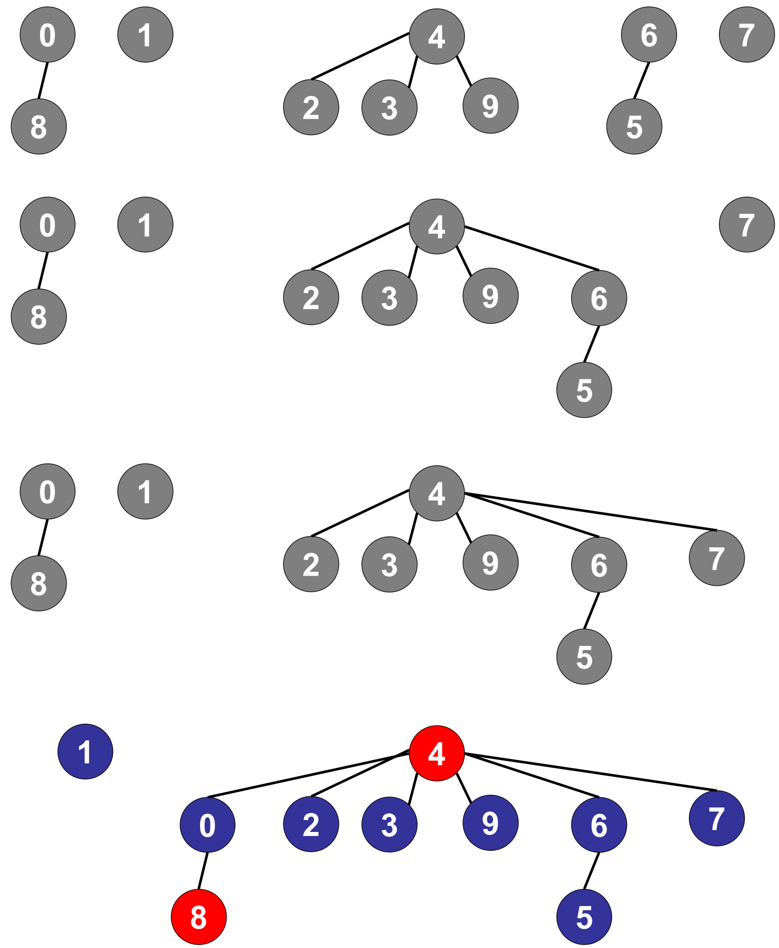
Example: Weighted Union

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



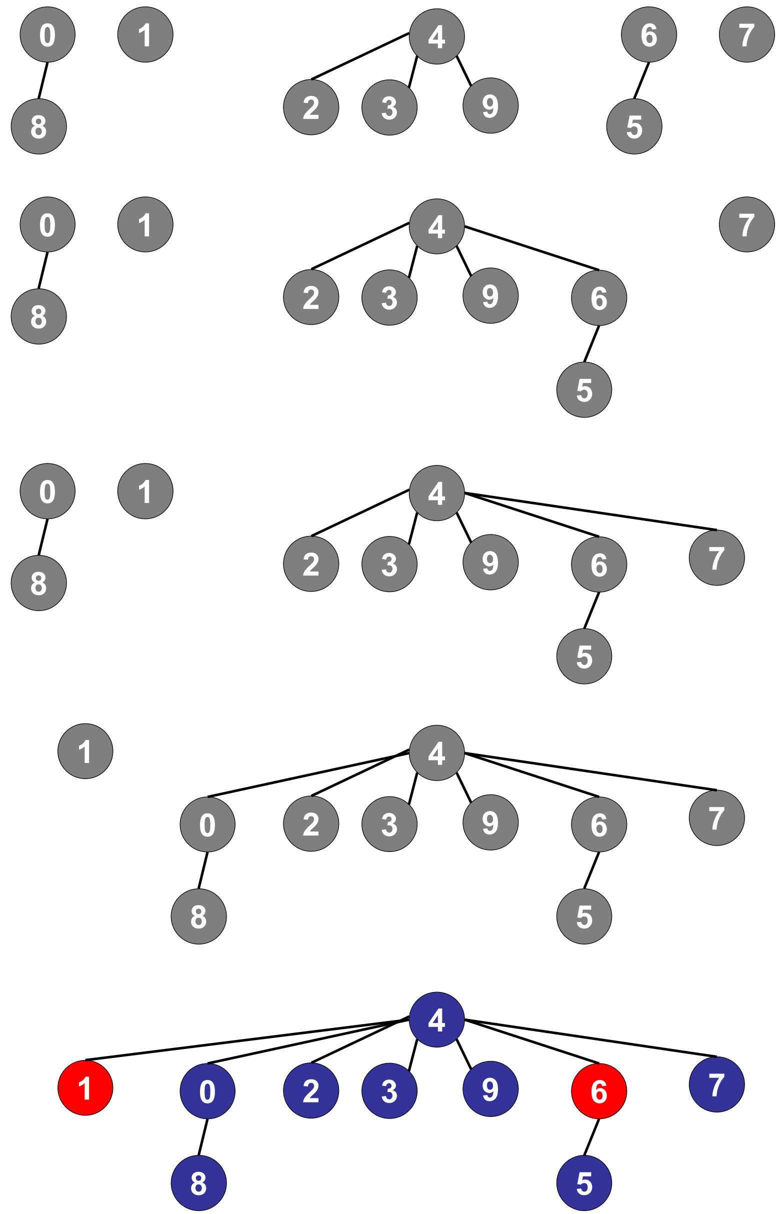
Example: Weighted Union

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



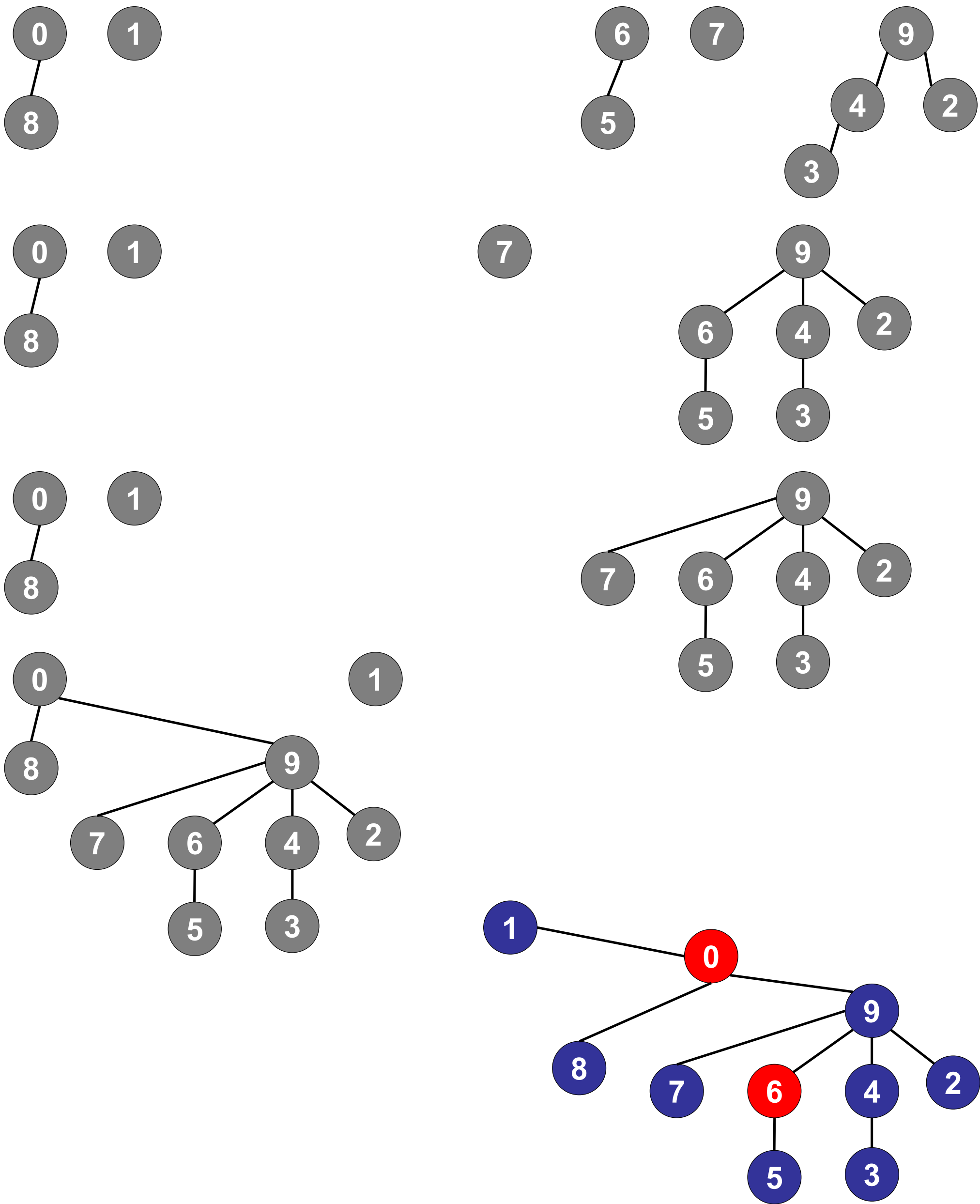
Example: Weighted Union

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



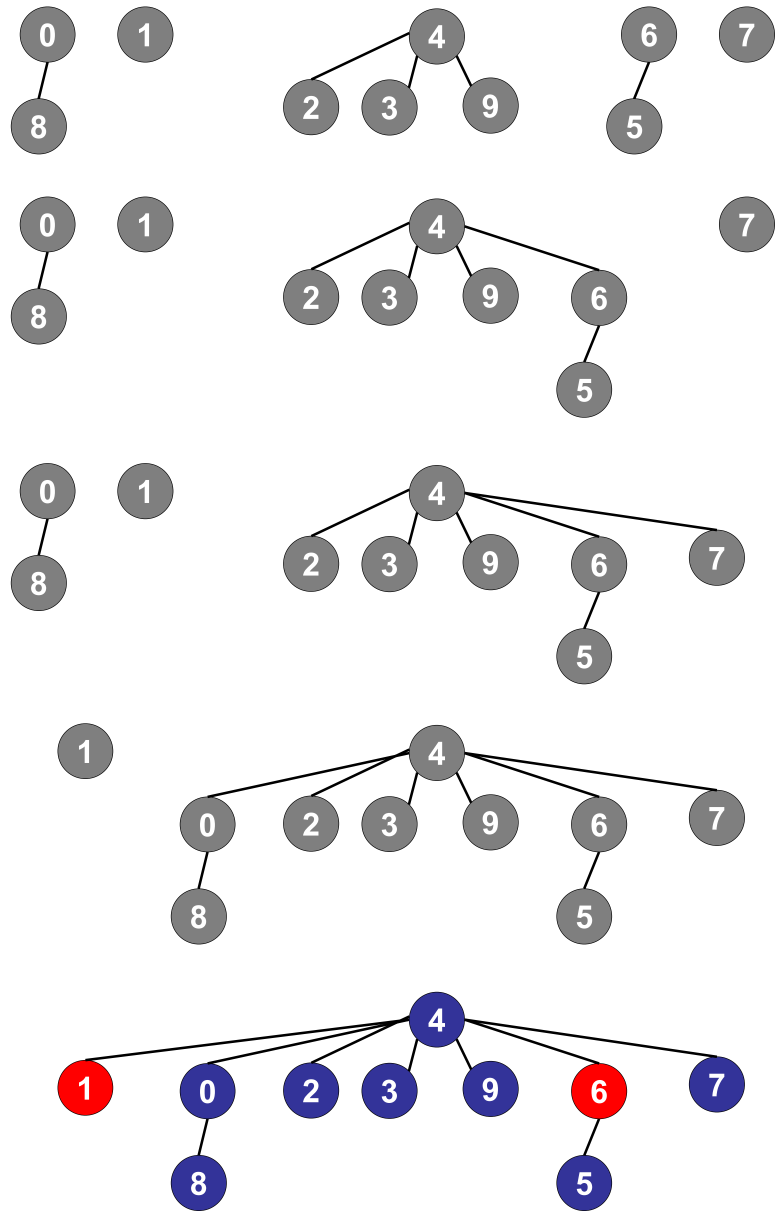
Example: (Unweighted) Quick Union

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



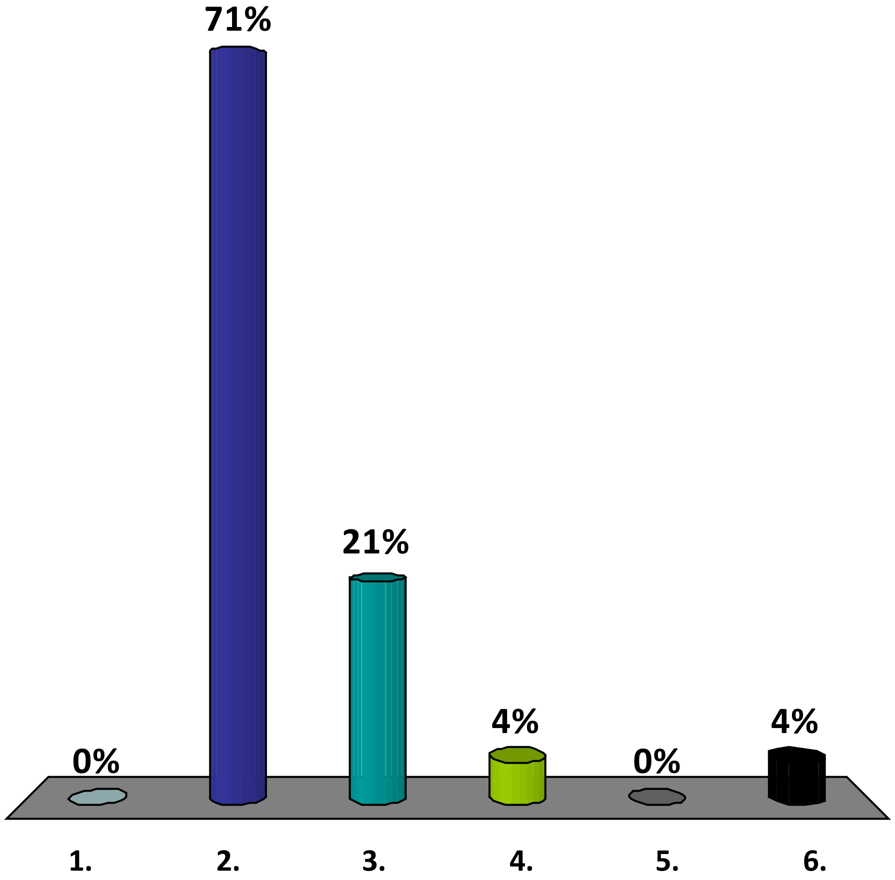
Example: Weighted Union

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?

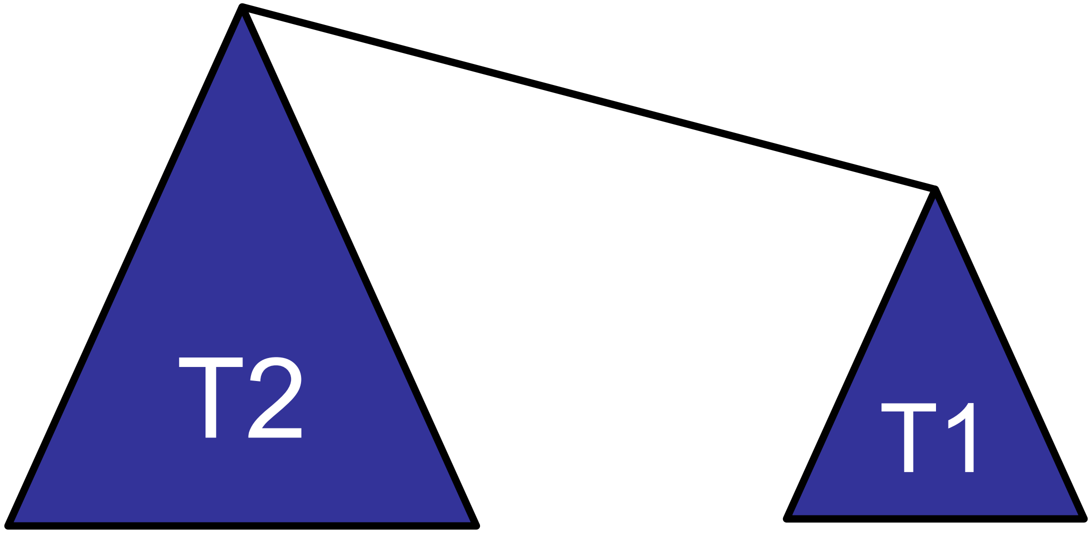
1. O(1)
2. O(log n)
3. O(n)
4. O(n log n)
5. $O(n^2)$
6. None of the above.



Weighted Union

Analysis:

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



Weighted Union

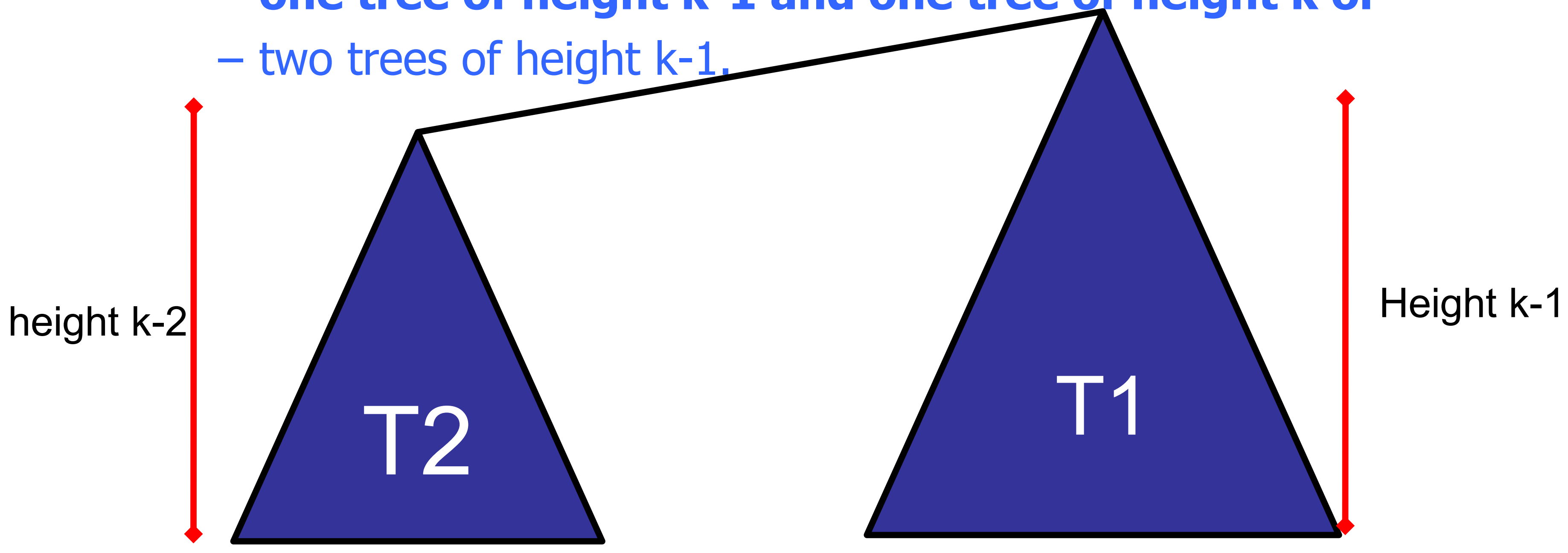
Analysis:

- Base case: tree of height 0 contains 1 object.
- Induction:
  - Induction: a tree of height k-1 contains at least $2^{(k-1)}$ objects.
  - A tree of height k is built from
    - one tree of height k-1 and one tree of height k or
    - two trees of height k-1.

Weighted Union

Analysis:

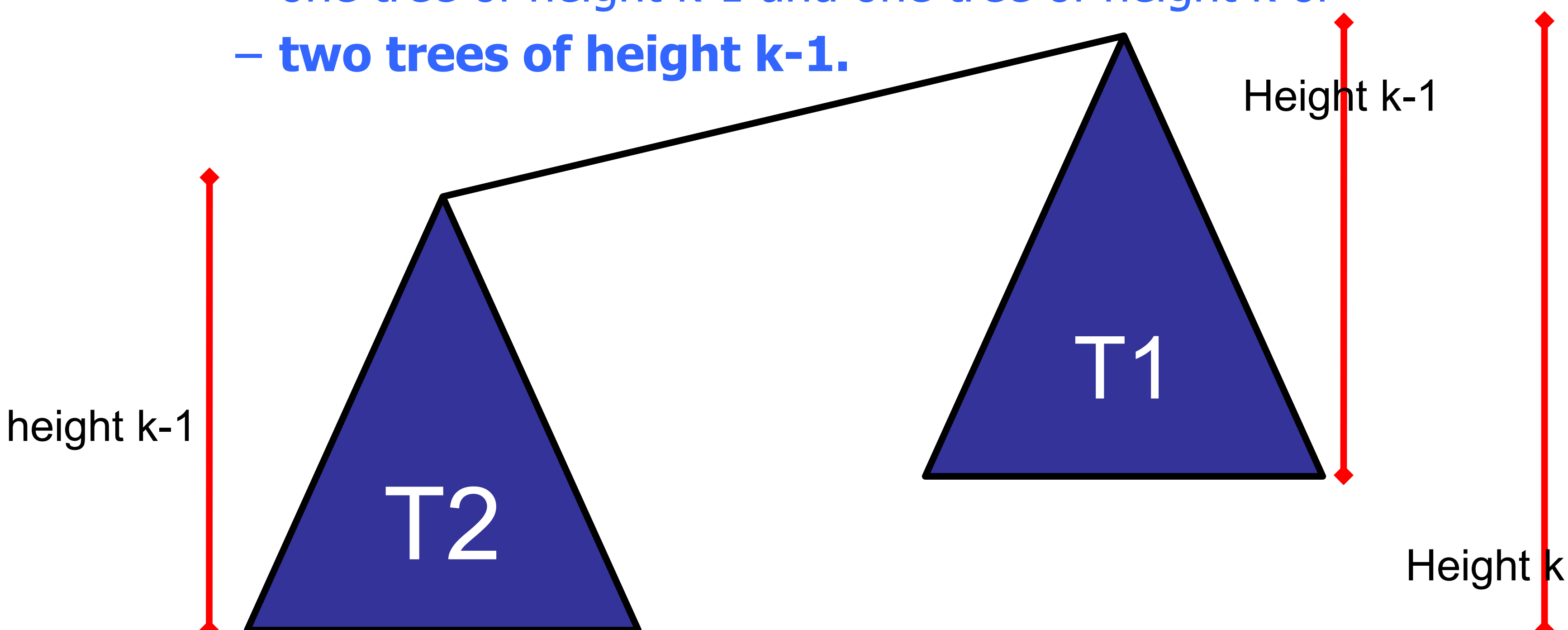
- Base case: tree of height 0 contains 1 object.
- Induction:
  - A tree of height k is built from
    - **one tree of height k-1 and one tree of height k or**
    - two trees of height k-1.



Weighted Union

Analysis:

- Base case: tree of height 0 contains 1 object.
- Induction:
  - A tree of height k is built from
    - one tree of height k-1 and one tree of height k or
    - **two trees of height k-1.**



Weighted Union

Analysis:

- Base case: tree of height 0 contains 1 object.
- Induction:
  - Tree of height k is built from two trees of height k-1.
  - Induction: a tree of height k-1 contains at least $2^{(k-1)}$ objects.
  - Conclusion: a tree of height k contains 2^k objects.

Weighted Union

Analysis:

- Base case: tree of height 0 contains 1 object.
- Induction:
  - Tree of height k is built from two trees of height k-1.
  - Induction: a tree of height k-1 contains at least $2^{(k-1)}$ objects.
  - Conclusion: a tree of height k contains 2^k objects.
- Conclusion:
  - Each tree is of height O(log n)

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

Quick-find and Quick-union are slow:

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

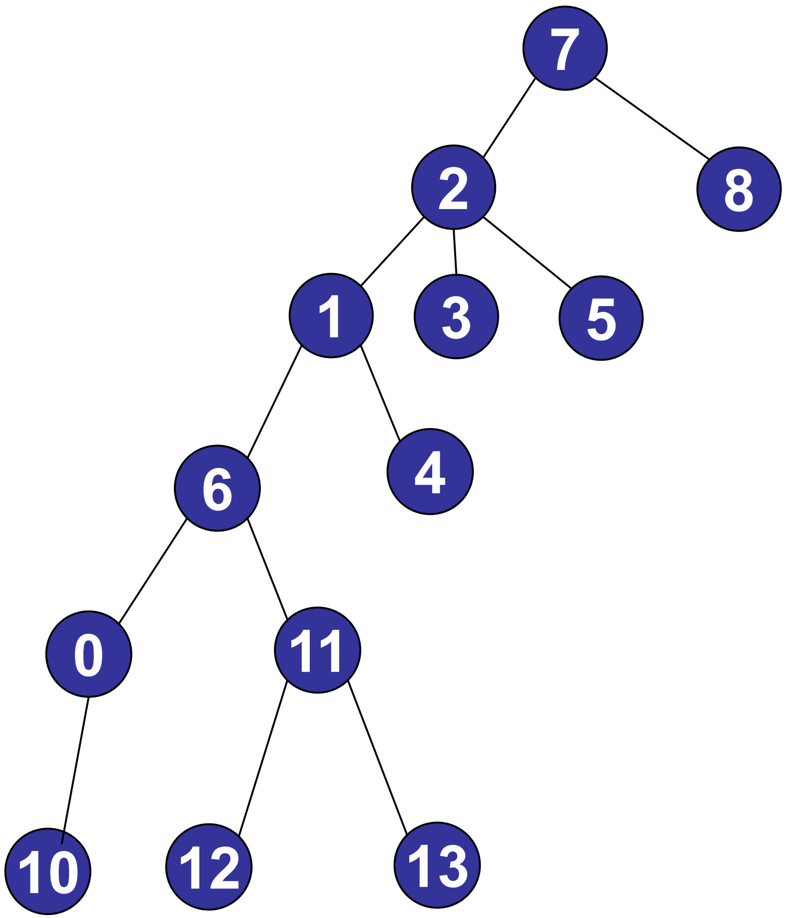
Weighted-union is faster:

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

	find	**union**
quick-find	O(1)	O(n)
quick-union	O(n)	O(n)
weighted-union	O(log n)	O(log n)

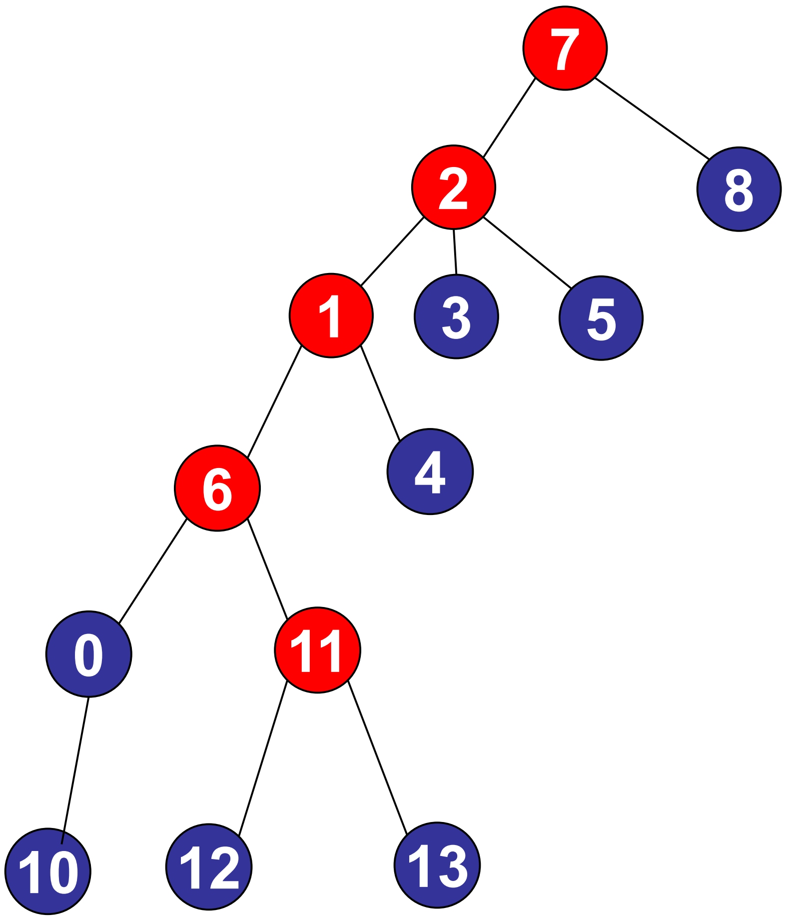
Path Compression

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



Path Compression

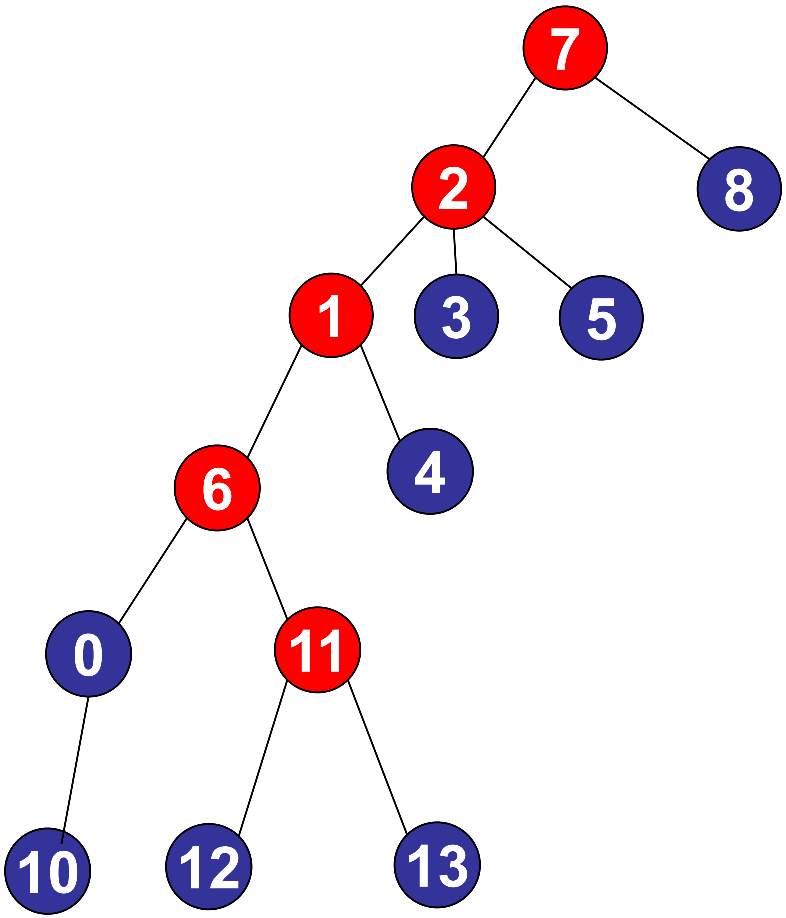
After finding the root: set the parent of each traversed node to the root.



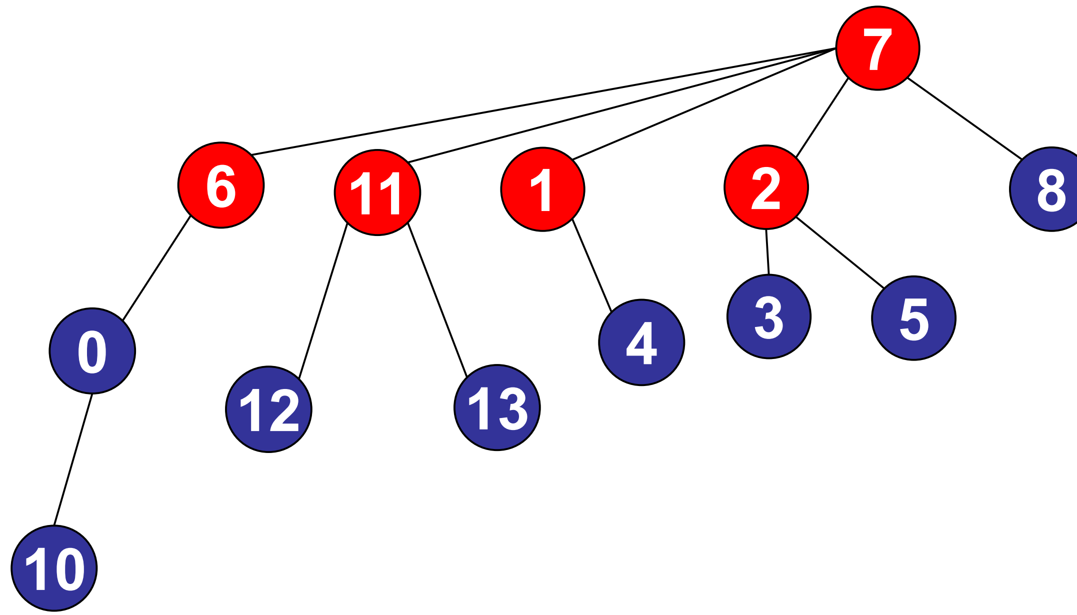
`find(11, 32)`

Path Compression

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



find(11, 32)



Path Compression

```
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!

How ~~fart~~ fast can it be?!

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.

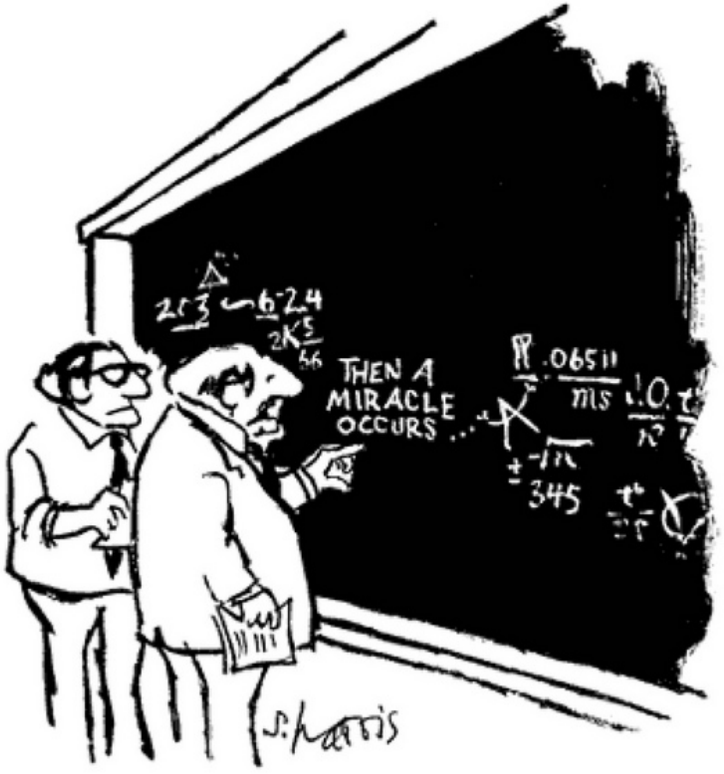
n	α(n, n)
4	0
8	1
32	2
8,192	3
2^{65533}	4

Weight Union with Path Compression

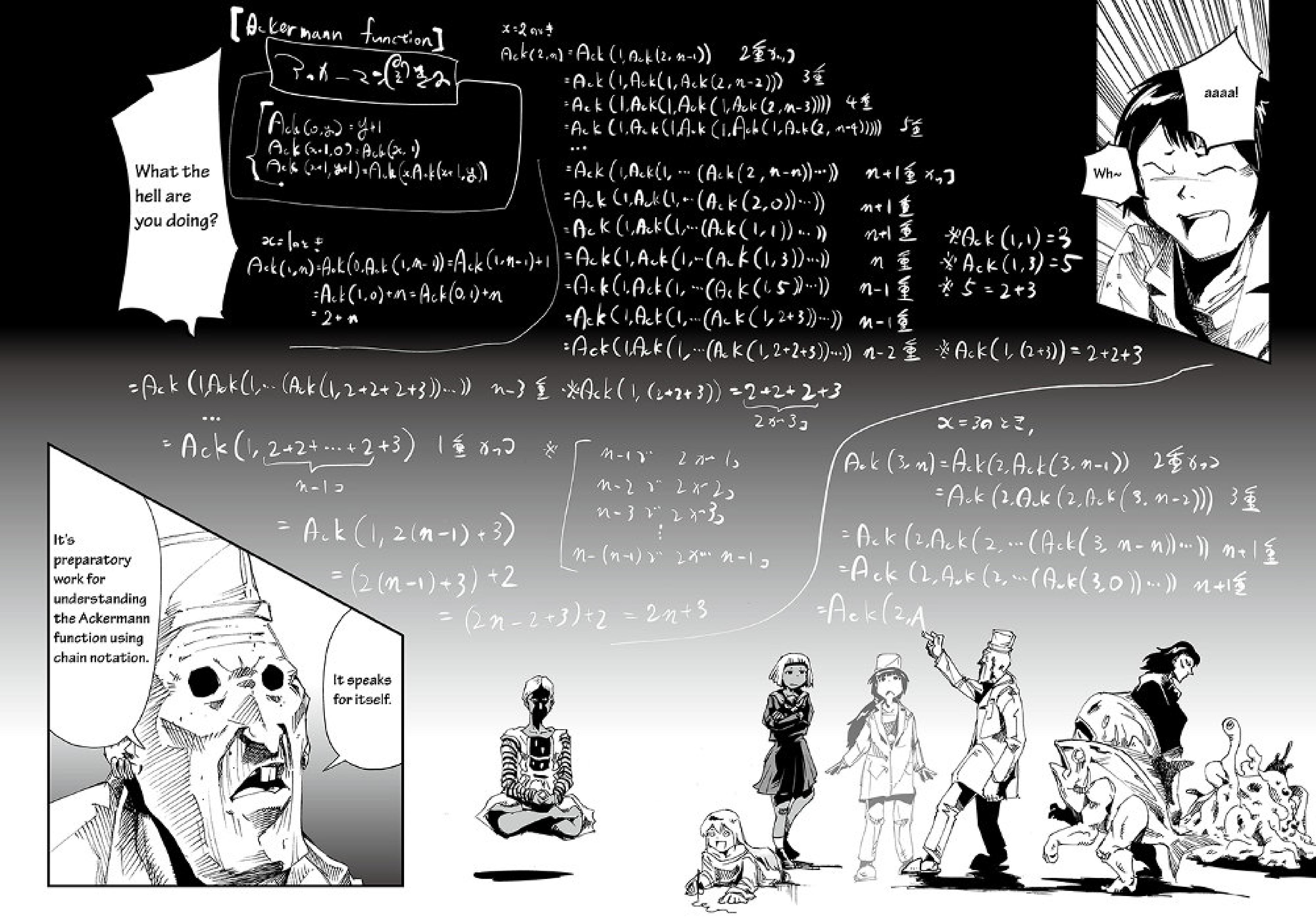
Theorem: [Tarjan 1975]

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

Proof:



"I think you should be more explicit here in step two."



What the hell are you doing?
[Ackermann function]
Ack(0,y)=y+1
Ack(x+1,0)=Ack(x,1)
Ack(x+1,y+1)=Ack(x,Ack(x+1,y))
Ack(1,n)=Ack(0,Ack(1,n-1))=Ack(1,n-1)+1
=Ack(1,0)+n=Ack(0,1)+n
=2+n
Ack(2,n)=Ack(1,Ack(2,n-1))
=Ack(1,Ack(1,Ack(2,n-2)))
=Ack(1,Ack(1,Ack(1,Ack(2,n-3))))
=Ack(1,Ack(1,Ack(1,Ack(1,Ack(2,n-4)))))
=Ack(1,Ack(1,…(Ack(2,n-n))…))
=Ack(1,Ack(1,…(Ack(2,0))…))
=Ack(1,Ack(1,…(Ack(1,1))…))
=Ack(1,Ack(1,…(Ack(1,3))…))
=Ack(1,Ack(1,…(Ack(1,5))…))
=Ack(1,Ack(1,…(Ack(1,2+3))…))
=Ack(1,Ack(1,…(Ack(1,2+2+3))…))
※Ack(1,1)=3
※Ack(1,3)=5
※5=2+3
※Ack(1,(2+3))=2+2+3
=Ack(1,Ack(1,…(Ack(1,2+2+2+3))…))
※Ack(1,(2+2+3))=2+2+2+3
=Ack(1,2+2+…+2+3)
=Ack(1,2(n-1)+3)
=(2(n-1)+3)+2
=(2n-2+3)+2=2n+3
Ack(3,n)=Ack(2,Ack(3,n-1))
=Ack(2,Ack(2,Ack(3,n-2)))
=Ack(2,Ack(2,…(Ack(3,n-n))…))
=Ack(2,Ack(2,…(Ack(3,0))…))
=Ack(2,A
Wh~
aaaa!
It's preparatory work for understanding the Ackermann function using chain notation.
It speaks for itself.

Weight Union with Path Compression

Theorem: [Tarjan 1975]

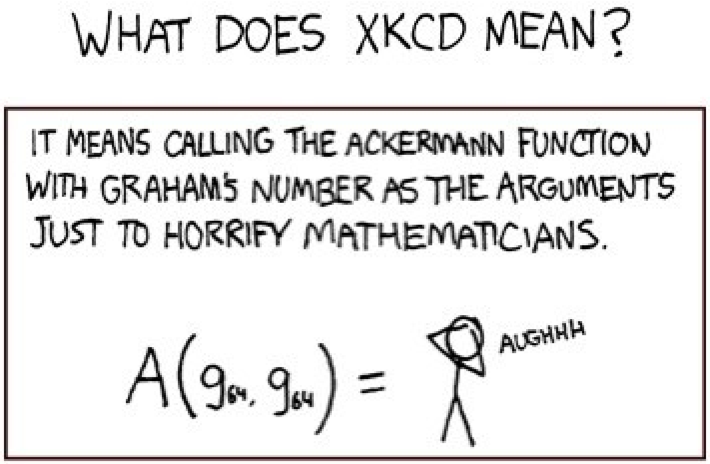
Starting from empty, any sequence of m union/find operations on n objects takes: $O(n + m\alpha(m, n))$ time.

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? No!

- Proof: impossible to achieve linear time.

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

What about Union-Split-Find?

- Insert and delete edges.
- New result: 2013!!

Dynamic graph connectivity in polylogarithmic worst case time

Bruce M. Kapron * Valerie King * Ben Mountjoy *

Abstract
The dynamic graph connectivity problem is the following: given a graph on a fixed set of n nodes which is undergoing a sequence of edge insertions and deletions, answer queries of the form $q(a, b)$: "Is there a path between nodes a and b?" While data structures for this problem with polylogarithmic *amortized* time per operation have been known since the mid-1990's, these data structures have $\Theta(n)$ worst case time. In fact, no previously known solution has worst case time per operation which is $o(\sqrt{n})$.

We present a solution with worst case times $O(\log^4 n)$ per edge insertion, $O(\log^5 n)$ per edge deletion, and $O(\log n / \log \log n)$ per query. The answer to each query is correct if the answer is "yes" and is correct with high probability if the answer is "no". The data structure is based on a simple novel idea which can be used to quickly identify an edge in a cutset.

Our technique can be used to simplify and significantly

Though the problem of improving the worst case update time from $O(\sqrt{n})$ has been posed in the literature many times, there has been no improvement since 1985. In the words of Pătraşcu and Thorup, it is "perhaps the most fundamental challenge in dynamic graph algorithms today" [11].

Nearly every dynamic connectivity data structure maintains a spanning forest F. Dealing with edge insertions is relatively easy. The challenge is to find a replacement edge when a tree edge is deleted, splitting a tree into two subtrees. A replacement edge is an edge reconnecting the two subtrees, or, in other words, in the cutset of the cut $(T, V \setminus T)$ where T is one of the subtrees. An edge with both endpoints in the same subtree we call *internal* to the tree.

Roadmap

Part I: Priority Queues

- Binary Heaps
- HeapSort

Part II: Disjoint Set

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

Applications

Many applications:

- Mazes
  - Are two locations connected?
- Games:
  - Can you get from one state to another?

Applications

Many applications:

- Networks
  - Are two locations connected?
- Least-common-ancestor:
  - Which node in a tree network is the closest ancestor?

Applications

Many applications:

- Programming languages
  - Hinley-Milner polymorphic type inference
  - Equivalence of finite state automata
  - Image processing in Matlab

- Physics:
  - Hoshen-Kopelman algorithm
  - Percolation
  - Conductance / insulation

Applications

Many applications:

– Topology in Molecular Design

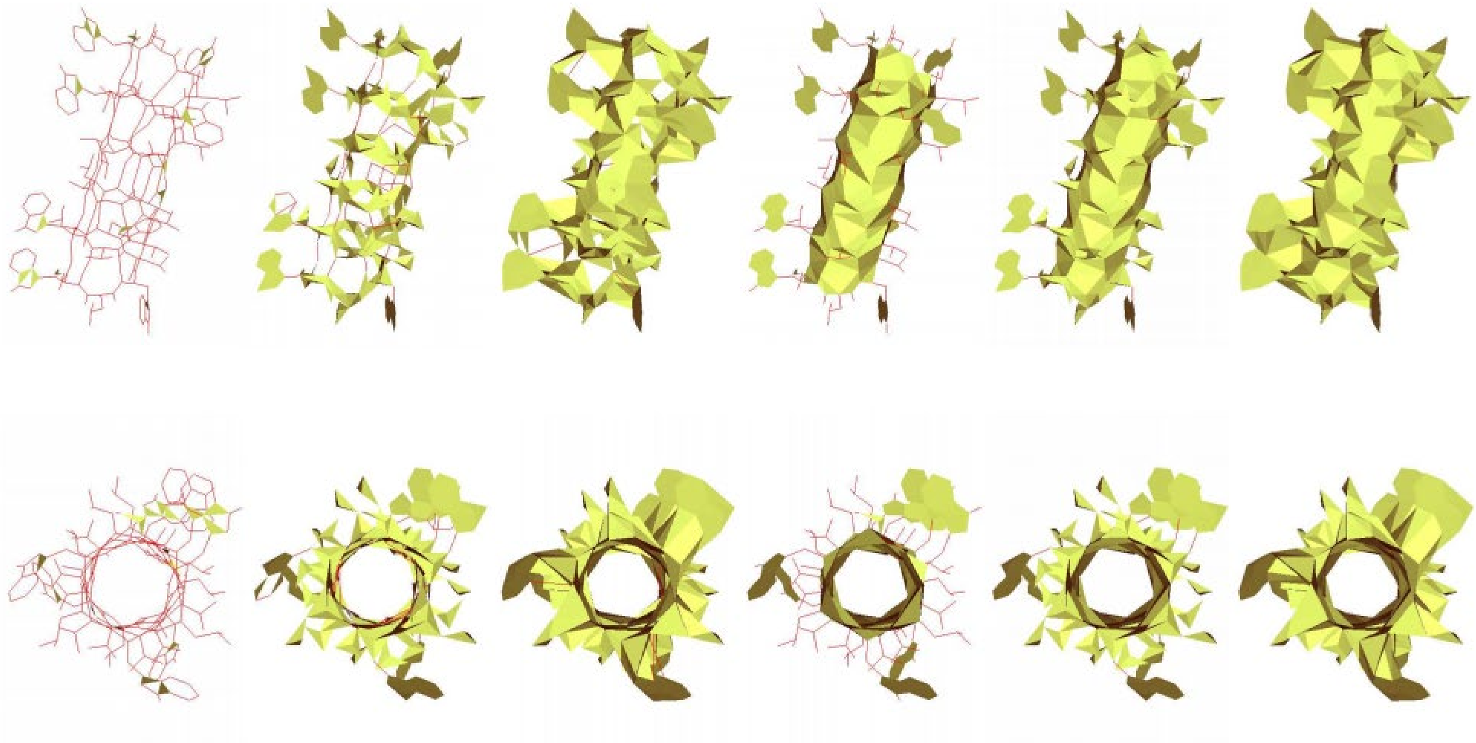


Figure 19. Side and top views of complexes $K_{715}, K_{1431}, K_{2682}$ of Gramicidin A are shown in the left three columns. The corresponding 2688-persistent complexes are shown on the right.