# SPRING 2023

# Information Technology Research

Yi Han

Department of Information Management
National Sun Yat-sen University

Lecture slides are based on the supplemental materials of the textbook: https://algs4.cs.princeton.edu

# Subtext of today's lecture (and this course)

Steps to developing a usable algorithm.

- Model the problem.
- Find an algorithm to solve it.
- Fast enough? Fits in memory?
- If not, figure out why not.
- Find a way to address the problem.
- Iterate until satisfied.

A little mathematical analysis.

**http://algs4.cs.princeton.edu**

# 1.5 Union-Find

- ▸ *dynamic connectivity*
- ▸ *quick find*
- ▸ *quick union*
- ▸ *improvements*
- ▸ *applications*

# Dynamic connectivity problem

Given a set of N objects, support two operation:

- Connect two objects.
- Is there a path connecting the two objects?

*connect 4 and 3*

*connect 3 and 8*

*connect 6 and 5*

*connect 9 and 4*

*connect 2 and 1*

*are 0 and 7 connected?* ✗

*are 8 and 9 connected?* ✔

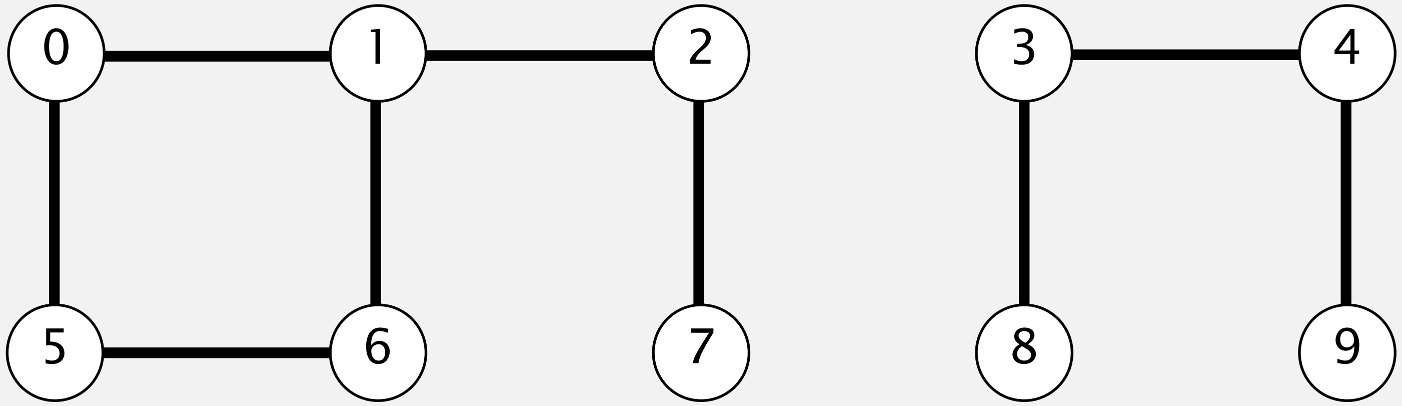
*connect 5 and 0*

*connect 7 and 2*

*connect 6 and 1*

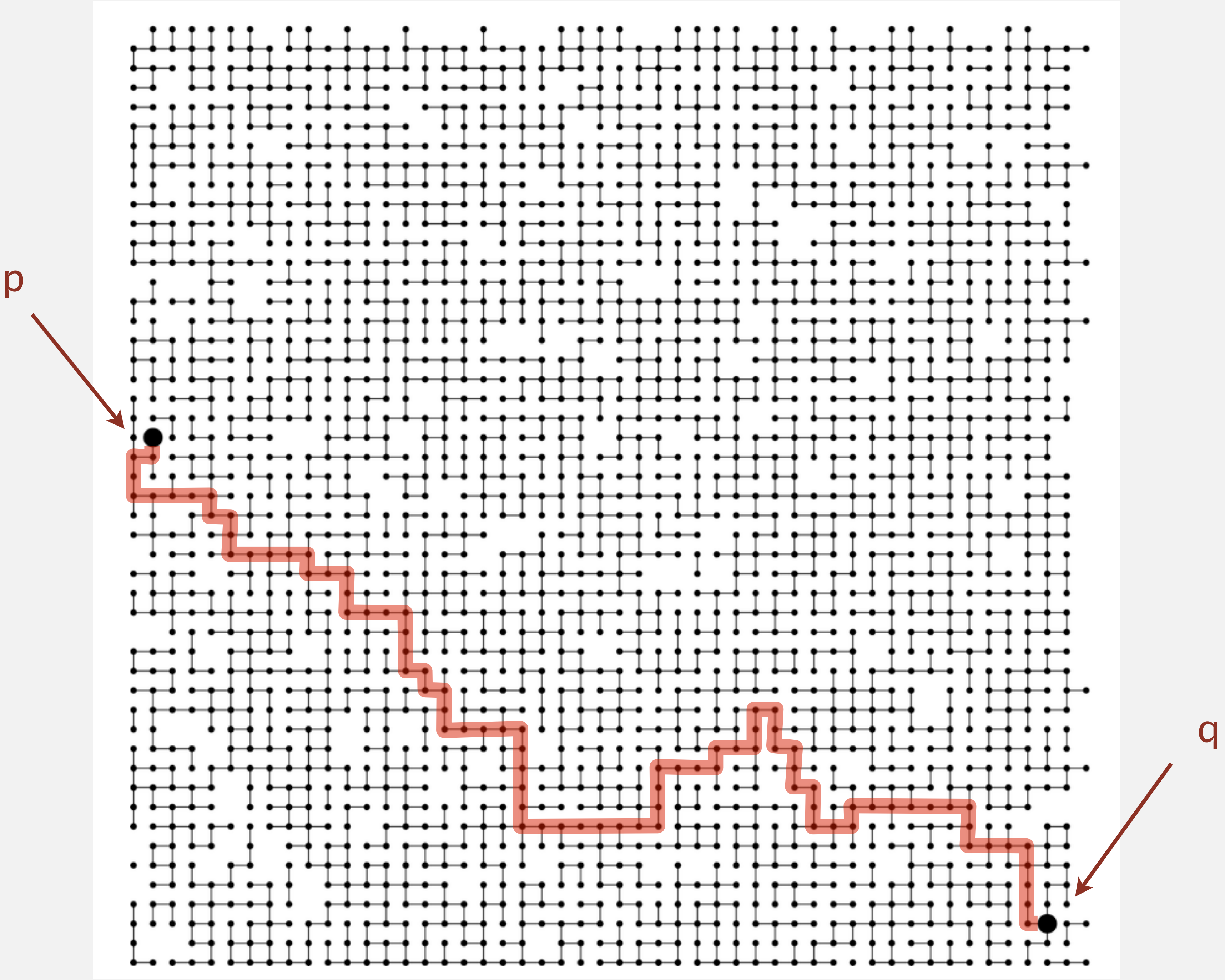
*connect 1 and 0*

*are 0 and 7 connected?* ✔



# A larger connectivity example

Q. Is there a path connecting $p$ and $q$ ?



A. Yes.

# Modeling the objects

Applications involve manipulating objects of all types.

- Pixels in a digital photo.
- Computers in a network.
- Friends in a social network.
- Transistors in a computer chip.
- Elements in a mathematical set.
- Variable names in a Fortran program.
- Metallic sites in a composite system.

When programming, convenient to name objects 0 to N – 1.

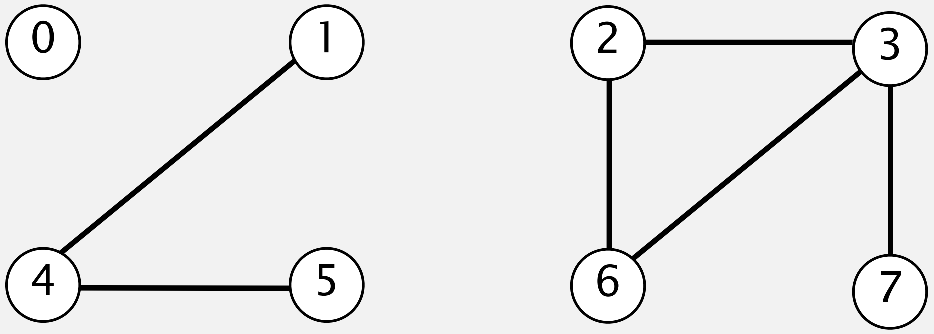
- Use integers as array index.
- Suppress details not relevant to union-find.

# Modeling the connections

We assume "is connected to" is an equivalence relation:

- Reflexive: $p$ is connected to $p$.
- Symmetric: if $p$ is connected to $q$, then $q$ is connected to $p$.
- Transitive: if $p$ is connected to $q$ and $q$ is connected to $r$, then $p$ is connected to $r$.

Connected component. Maximal set of objects that are mutually connected.



{ 0 } { 1 4 5 } { 2 3 6 7 }

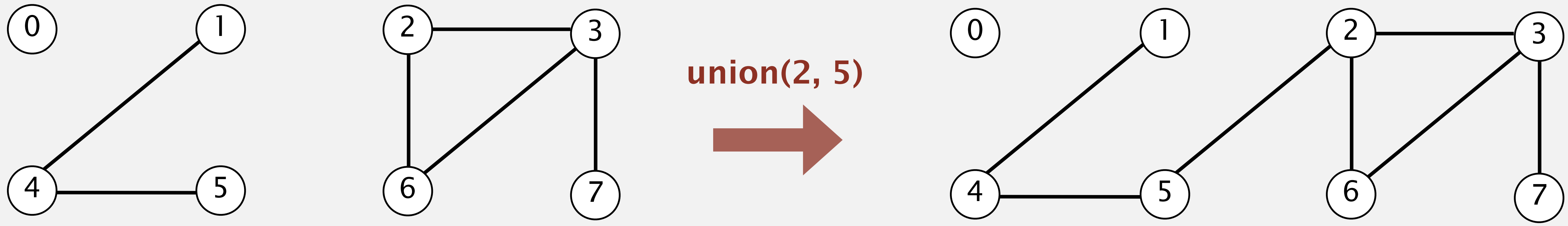
3 connected components

# Implementing the operations

Find. In which component is object $p$ ?

Connected. Are objects $p$ and $q$ in the same component?

Union. Replace components containing objects $p$ and $q$ with their union.



{ 0 } { 1 4 5 } { 2 3 6 7 }

3 connected components

{ 0 } { 1 2 3 4 5 6 7 }

2 connected components

# Union-find data type (API)

Goal. Design efficient data structure for union-find.

- Number of objects $N$ can be huge.
- Number of operations $M$ can be huge.
- Union and find operations may be intermixed.

| public class UF | | |
|---|---|---|
| | `UF(int N)` | *initialize union-find data structure with N singleton objects (0 to N – 1)* |
| `void` | `union(int p, int q)` | *add connection between p and q* |
| `int` | `find(int p)` | *component identifier for p (0 to N – 1)* |
| `boolean` | `connected(int p, int q)` | *are p and q in the same component?* |

```
public boolean connected(int p, int q)
{  return find(p) == find(q);  }
```

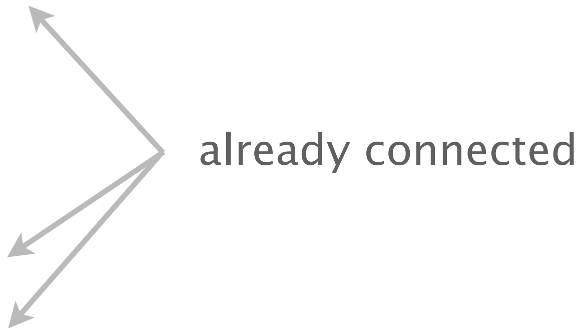
**1-line implementation of connected()**

# Dynamic-connectivity client

- Read in number of objects $N$ from standard input.
- Repeat:
  - read in pair of integers from standard input
  - if they are not yet connected, connect them and print out pair

```
public static void main(String[] args)
{
   int N = StdIn.readInt();
   UF uf = new UF(N);
   while (!StdIn.isEmpty())
   {
      int p = StdIn.readInt();
      int q = StdIn.readInt();
      if (!uf.connected(p, q))
      {
         uf.union(p, q);
         StdOut.println(p + " " + q);
      }
   }
}
```

```
% more tinyUF.txt
10
4 3
3 8
6 5
9 4
2 1
8 9
5 0
7 2
6 1
1 0
6 7
```



* StdIn() & StDOut() are from the book authors: https://algs4.cs.princeton.edu/code/

http://algs4.cs.princeton.edu

# 1.5 Union-Find

- ‣ *dynamic connectivity*
- ‣ *quick find*
- ‣ *quick union*
- ‣ *improvements*
- ‣ *applications*

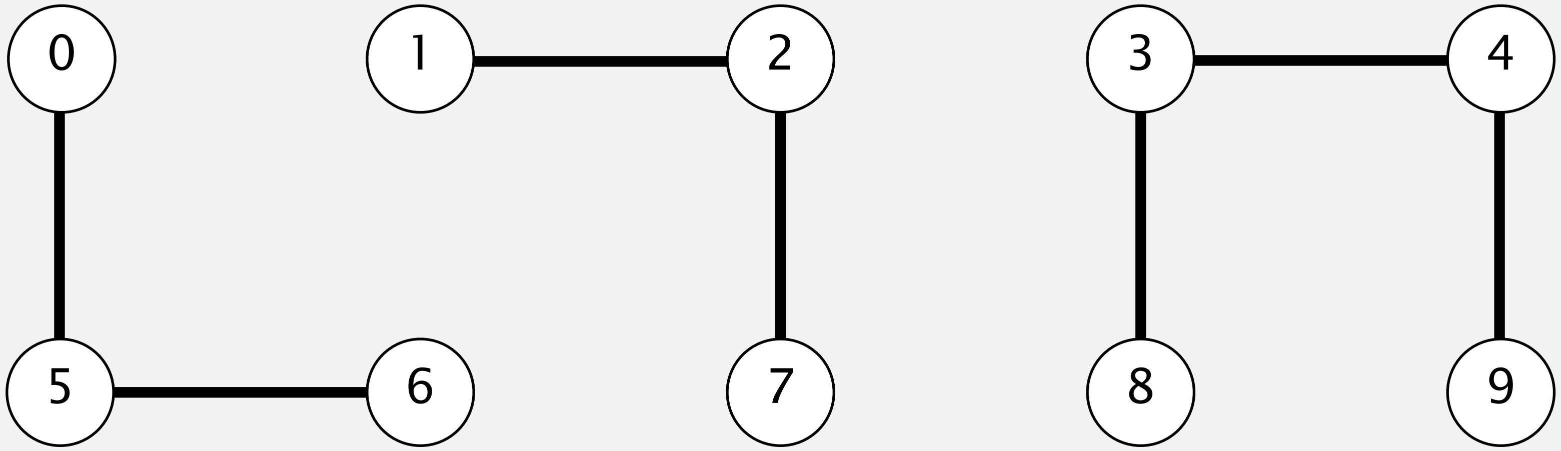
# Quick-find [eager approach]

Data structure.

- Integer array `id[]` of length `N`.
- Interpretation: `id[p]` is the id of the component containing `p`.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| `id[]` | 0 | 1 | 1 | 8 | 8 | 0 | 0 | 1 | 8 | 8 |

0, 5 and 6 are connected
1, 2, and 7 are connected
3, 4, 8, and 9 are connected



# Quick-find [eager approach]

## Data structure.

- Integer array `id[]` of length `N`.
- Interpretation: `id[p]` is the id of the component containing `p`.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| `id[]` | 0 | 1 | 1 | 8 | 8 | 0 | 0 | 1 | 8 | 8 |

**Find.** What is the id of `p`?

**Connected.** Do `p` and `q` have the same id?

`id[6] = 0`; `id[1] = 1`
6 and 1 are not connected

**Union.** To merge components containing `p` and `q`, change all entries whose id equals `id[p]` to `id[q]`.

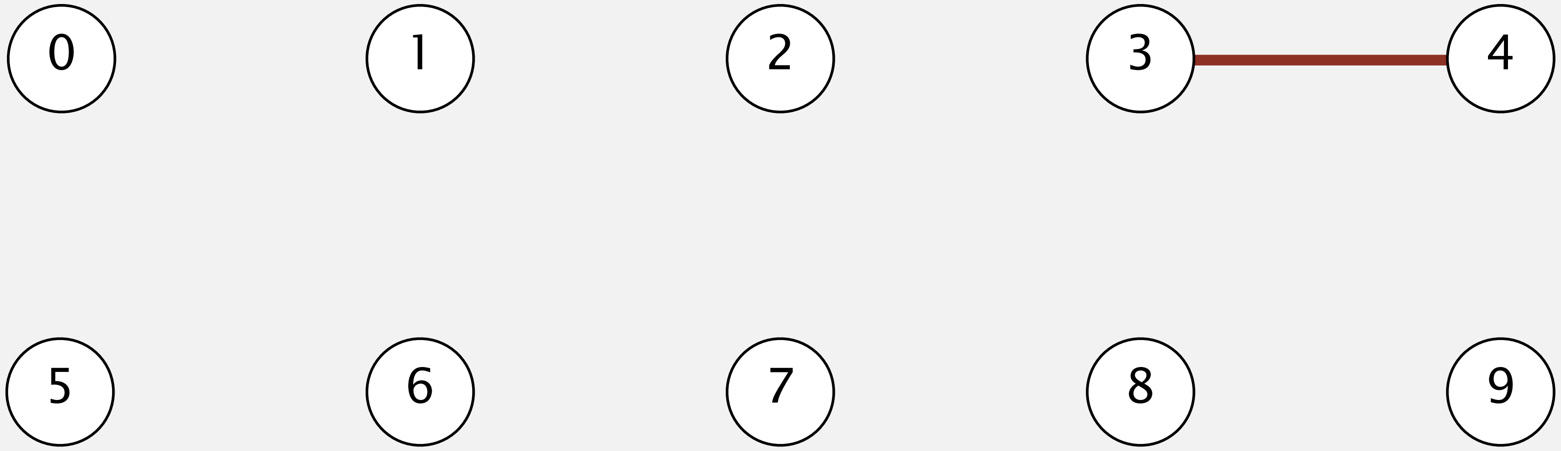
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| `id[]` | 1 | 1 | 1 | 8 | 8 | 1 | 1 | 1 | 8 | 8 |

**after union of 6 and 1**

problem: many values can change

# Quick-find demo

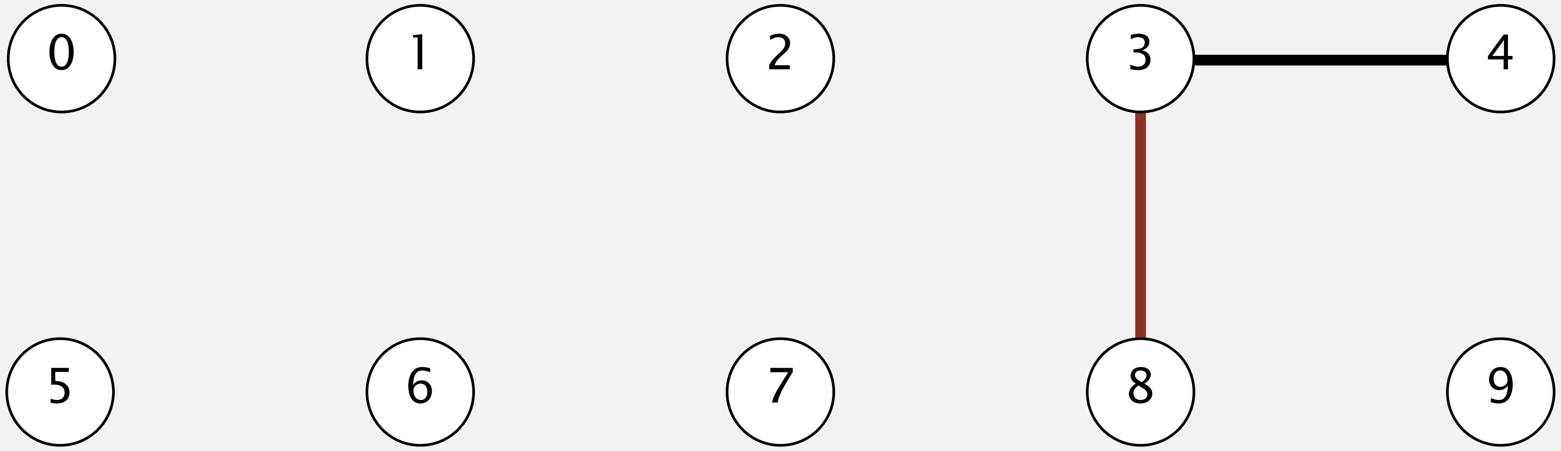
**union(4, 3)**



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 |

# Quick-find demo

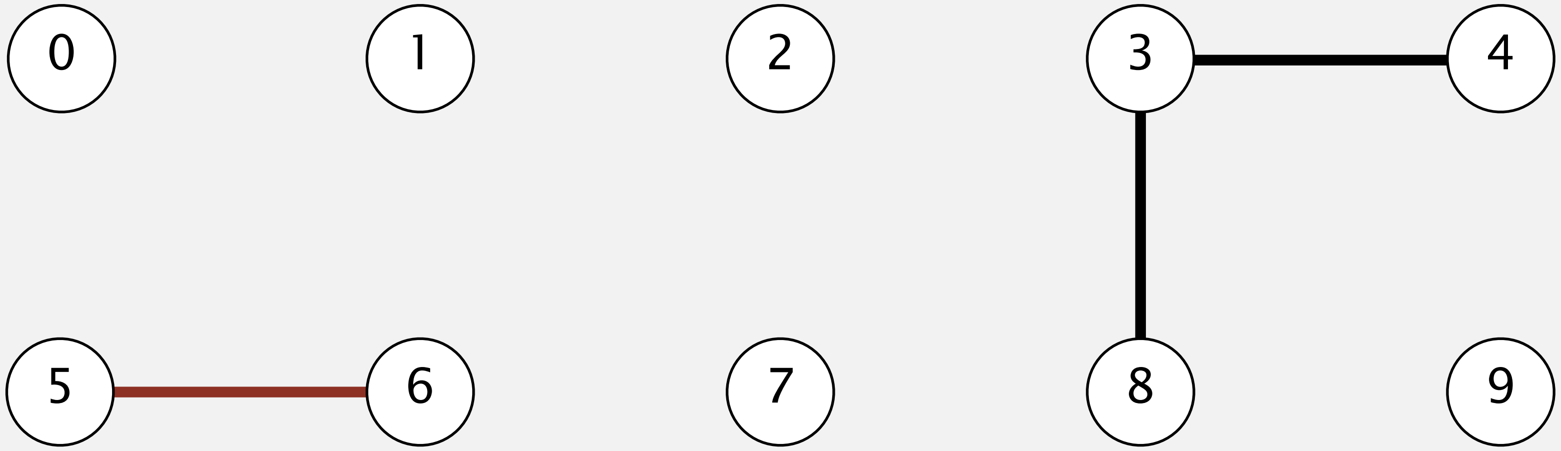
**union(3, 8)**



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 8 | 8 | 5 | 6 | 7 | 8 | 9 |
| | | | | ↑ | | | | | ↑ | |

# Quick-find demo

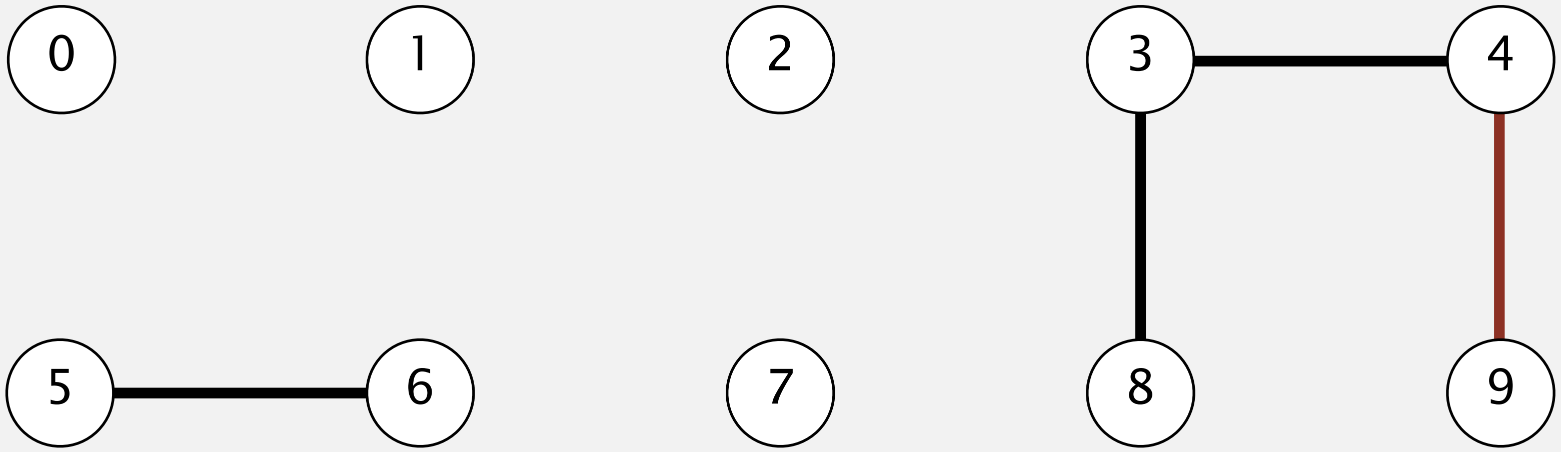
**union(6, 5)**



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 8 | 8 | 5 | 5 | 7 | 8 | 9 |

# Quick-find demo

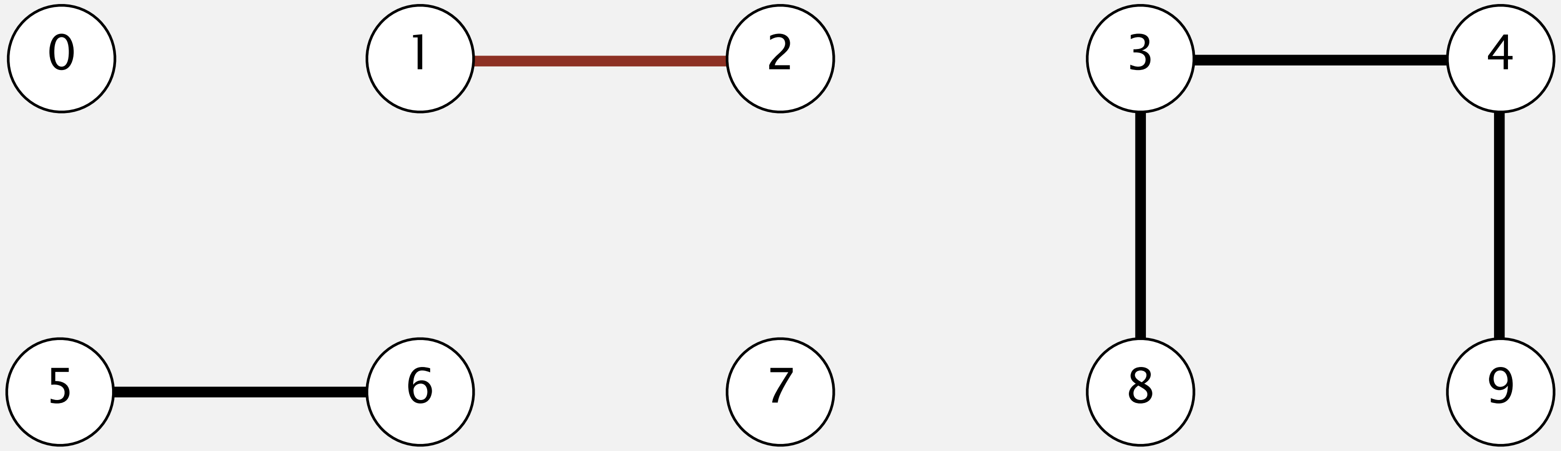
union(9, 4)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 8 | 8 | 5 | 5 | 7 | 8 | 8 |

# Quick-find demo

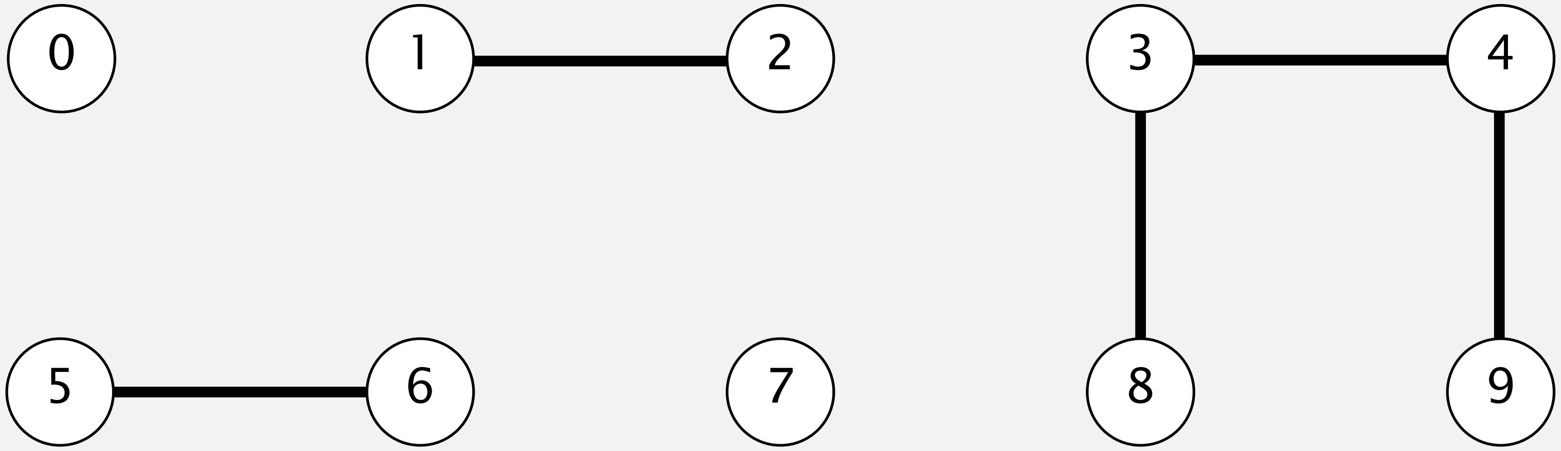
**union(2, 1)**



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 8 | 5 | 5 | 7 | 8 | 8 |

# Quick-find demo

**connected(8, 9)**



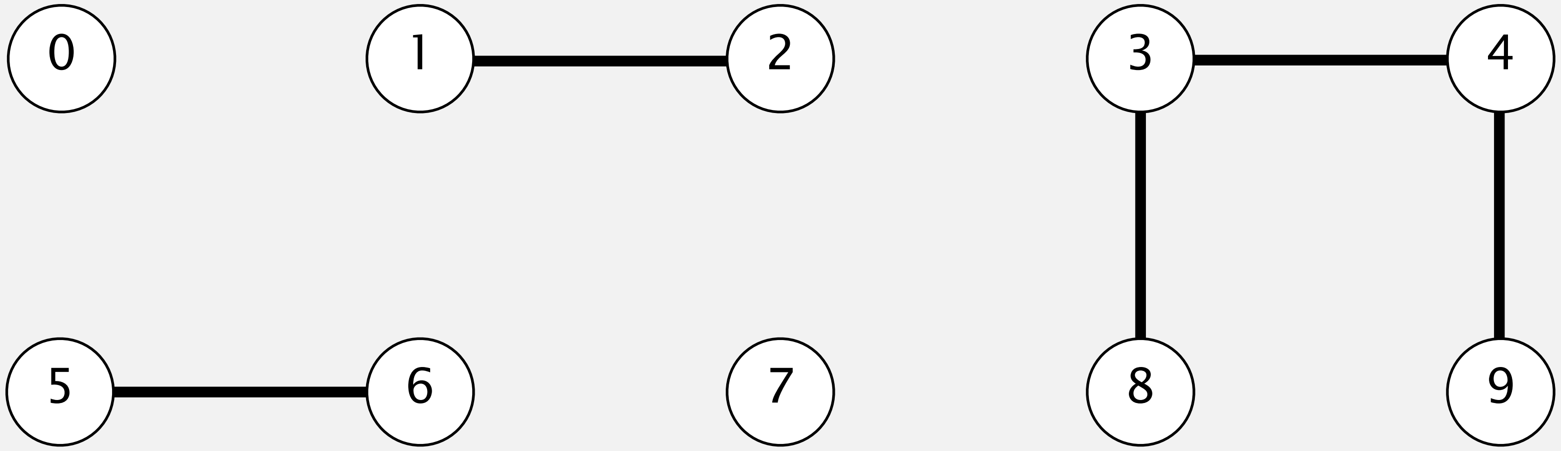
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 8 | 5 | 5 | 7 | 8 | 8 |

↑ ↑

**already connected**

# Quick-find demo

**connected(5, 0)**

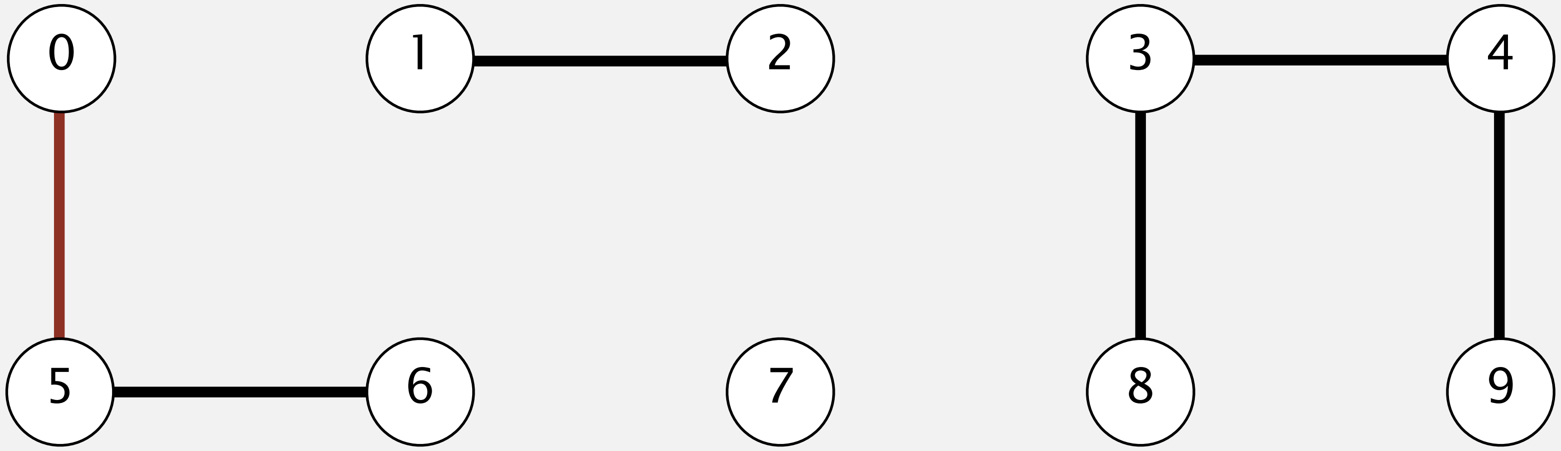


| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 8 | 5 | 5 | 7 | 8 | 8 |

**not connected**

# Quick-find demo

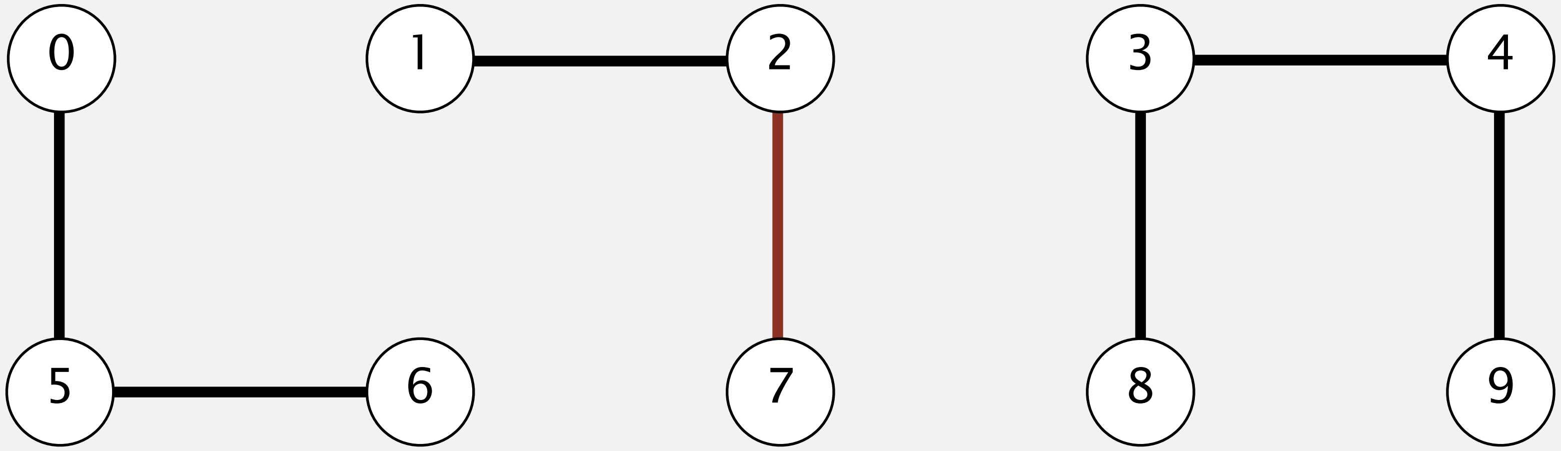
**union(5, 0)**



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 8 | 0 | 0 | 7 | 8 | 8 |

# Quick-find demo

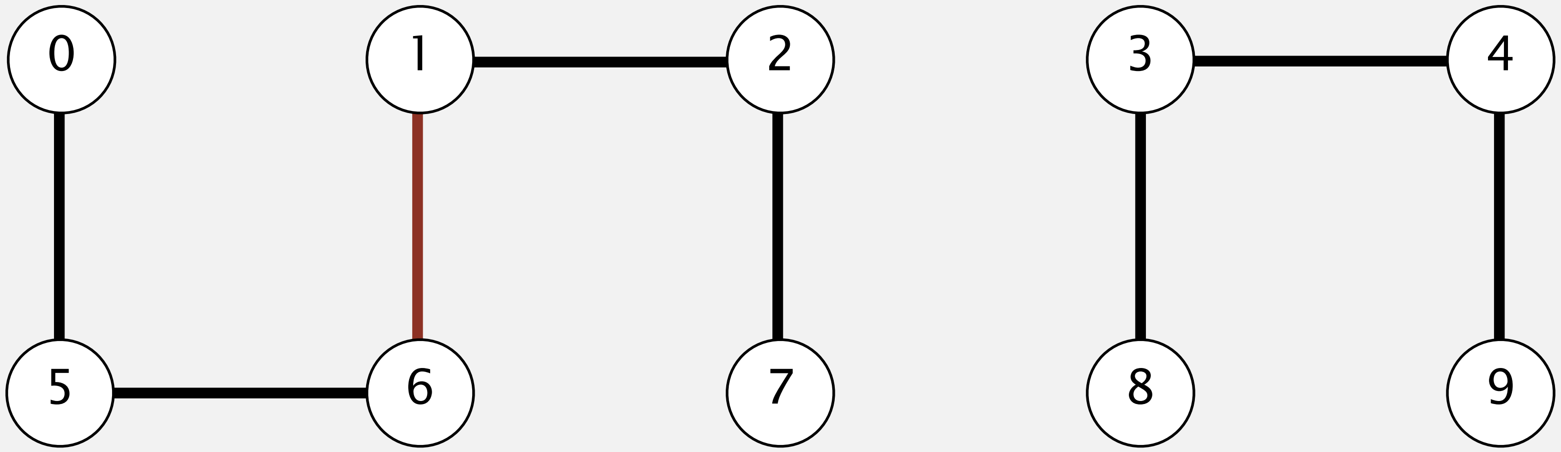
**union(7, 2)**



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 8 | 0 | 0 | 1 | 8 | 8 |

# Quick-find demo

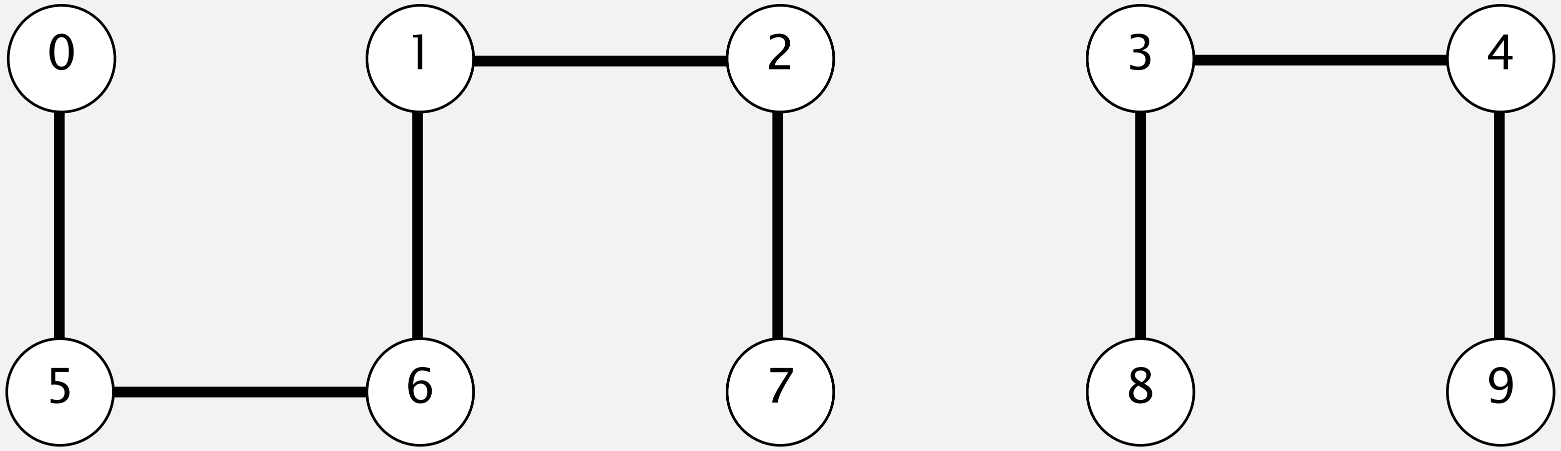
**union(6, 1)**



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 8 | 1 | 1 | 1 | 8 | 8 |

# Quick-find demo

**connected(1, 0)**

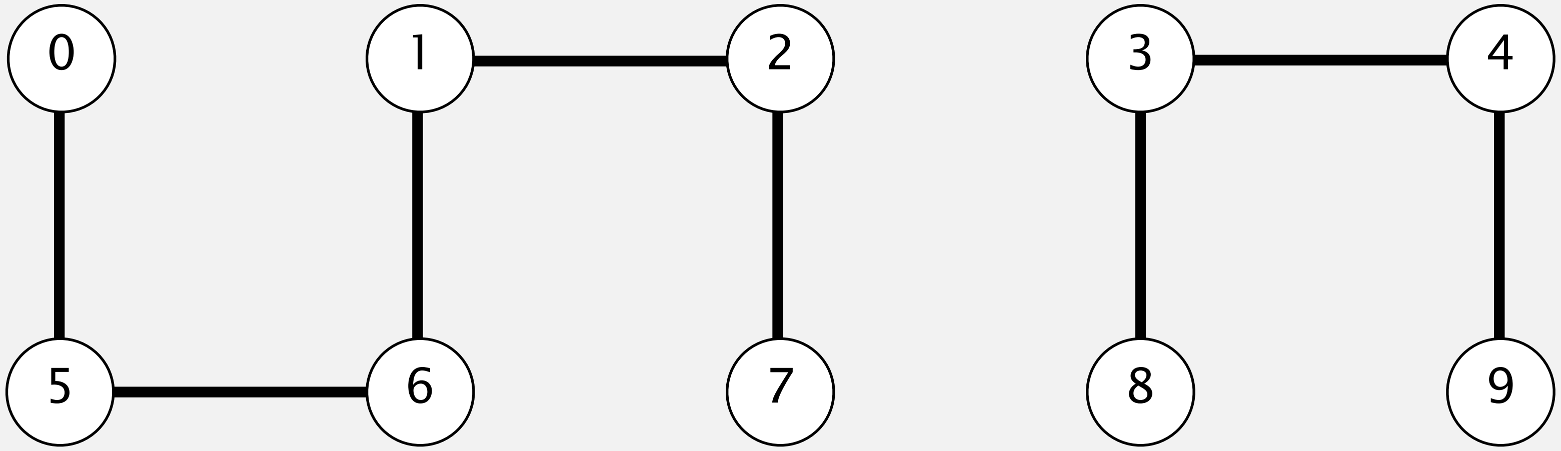


| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 8 | 1 | 1 | 1 | 8 | 8 |

**already connected**

# Quick-find demo

**connected(6, 7)**

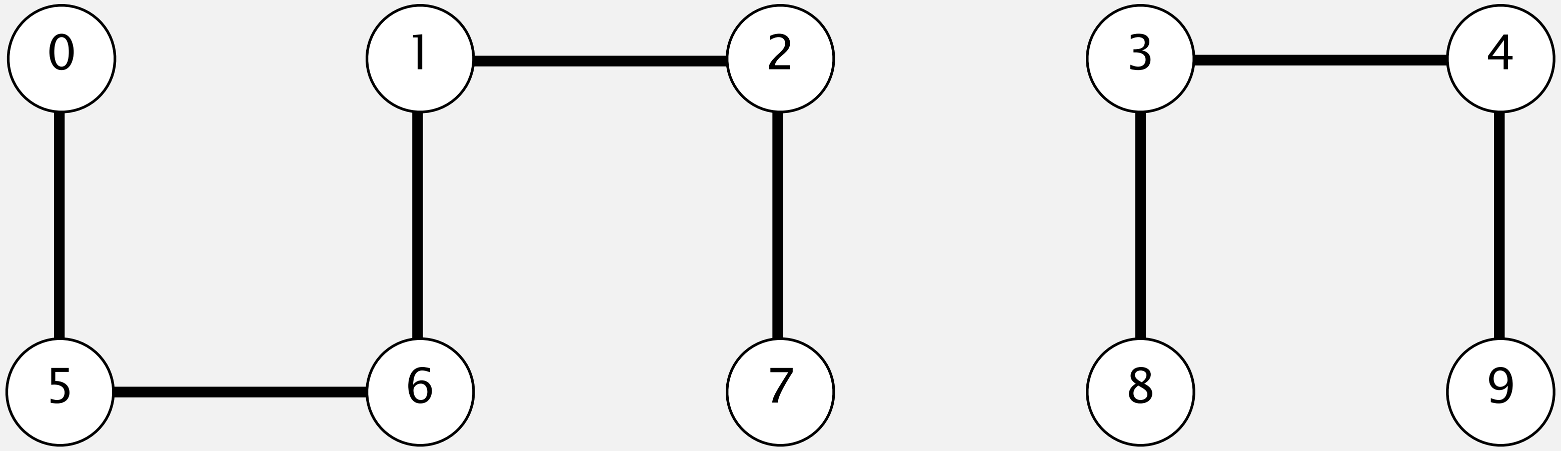


|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 8 | 1 | 1 | 1 | 8 | 8 |

↑ ↑

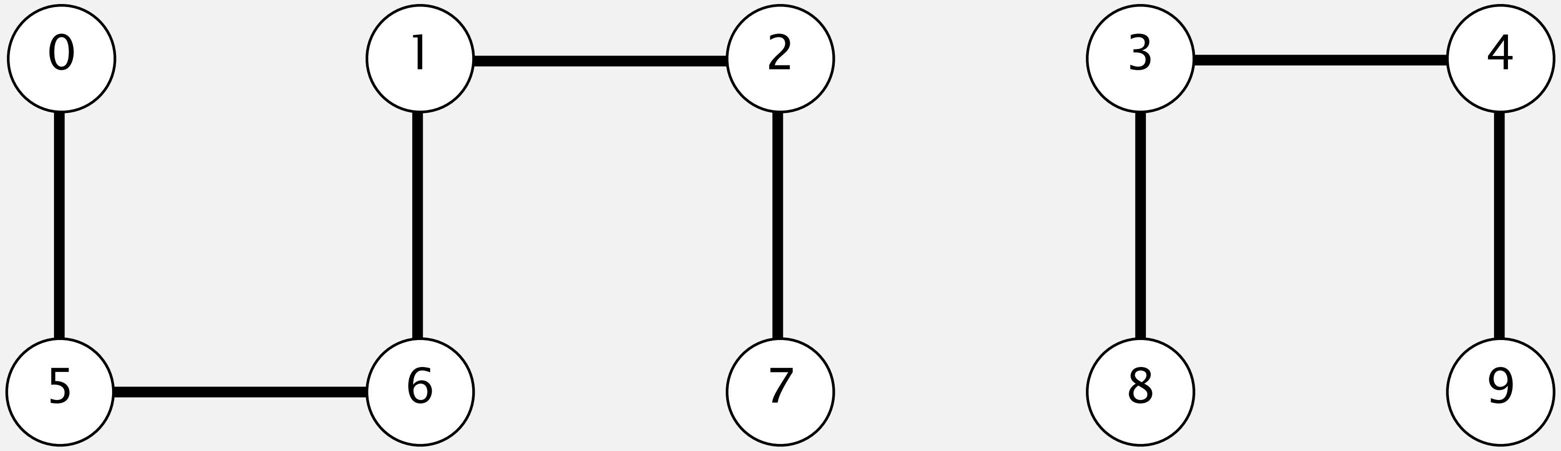
**already connected**

# Quick-find demo



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 8 | 1 | 1 | 1 | 8 | 8 |

# Quick-find demo



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 8 | 1 | 1 | 1 | 8 | 8 |

# Quick-find: Java implementation

```
public class QuickFindUF
{
   private int[] id;

   public QuickFindUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++)
      id[i] = i;
   }

   public int find(int p)
   {  return id[p];  }

   public void union(int p, int q)
   {

     What to write here? 5 mins.

   }
}
```

set id of each object to itself
(N array accesses)

return the id of p
(1 array access)

change all entries with id[p] to id[q]
(at most 2N + 2 array accesses)

# Quick-find: Java implementation

```
public class QuickFindUF
{
   private int[] id;

   public QuickFindUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++)
      id[i] = i;
   }

   public int find(int p)
   {  return id[p];  }

   public void union(int p, int q)
   {
      int pid = id[p];
      int qid = id[q];
      for (int i = 0; i < id.length; i++)
         if (id[i] == pid) id[i] = qid;
   }
}
```

set id of each object to itself
(N array accesses)

return the id of p
(1 array access)

change all entries with id[p] to id[q]
(at most 2N + 2 array accesses)

# Quick-find is too slow

Cost model. Number of array accesses (for read or write).

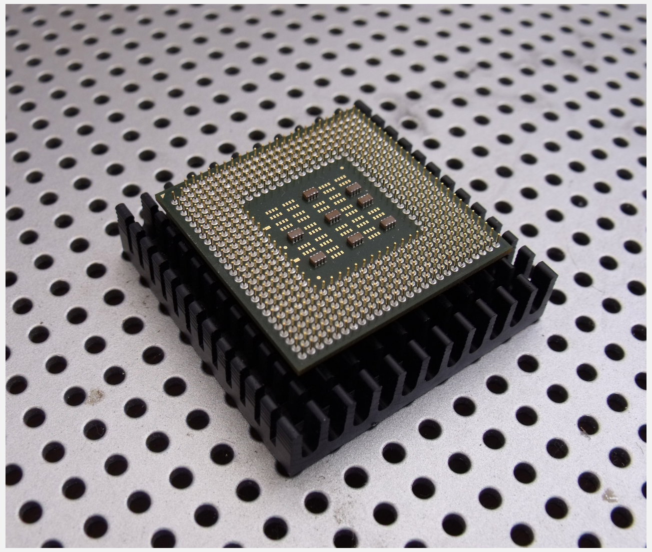
| algorithm | initialize | union | find | connected |
|---|---|---|---|---|
| **quick-find** | N | N | 1 | 1 |

**order of growth of number of array accesses**

quadratic

Union is too expensive. It takes $N^2$ array accesses to process a sequence of $N$ union operations on $N$ objects.

# Quadratic algorithms do not scale



## Rough standard (for now).

- $10^9$ operations per second.
- $10^9$ words of main memory.
- Touch all words in approximately 1 second.

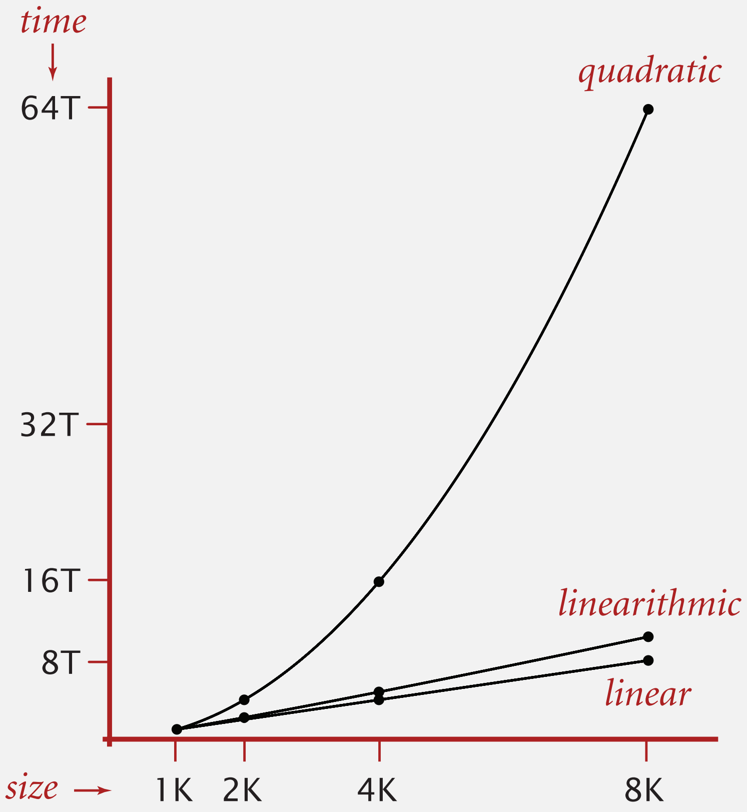
a truism (roughly) since 1950!

## Ex. Huge problem for quick-find.

- $10^9$ union commands on $10^9$ objects.
- Quick-find takes more than $10^{18}$ operations.
- 30+ years of computer time!

## Quadratic algorithms don't scale with technology.

- New computer may be 10x as fast.
- But, has 10x as much memory ⇒ want to solve a problem that is 10x as big.
- With quadratic algorithm, takes 10x as long!



http://algs4.cs.princeton.edu

# 1.5 Union-Find

- ▸ *dynamic connectivity*
- ▸ *quick find*
- ▸ *quick union*
- ▸ *improvements*
- ▸ *applications*

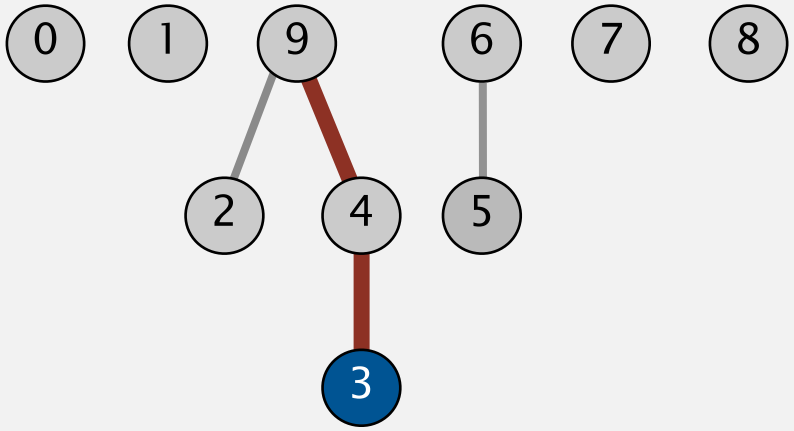
# Quick-union [lazy approach]

## Data structure.

- Integer array `id[]` of length `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

keep going until it doesn't change
(algorithm ensures no cycles)

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 8 | 9 |



parent of 3 is 4

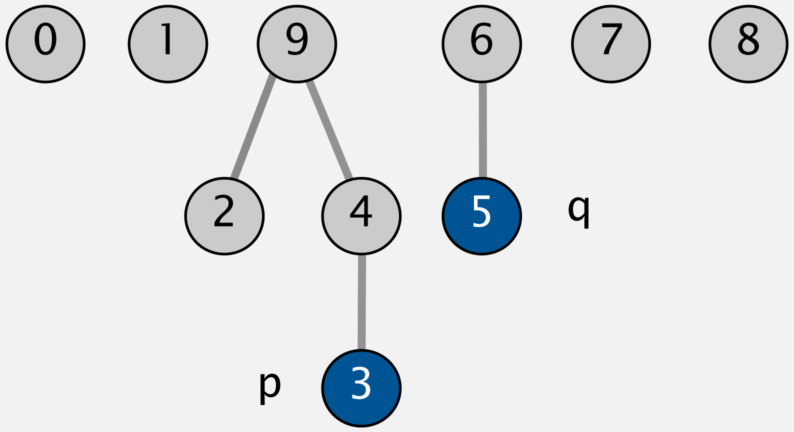
root of 3 is 9

# Quick-union [lazy approach]

Data structure.

- Integer array `id[]` of length `N`.
- Interpretation: `id[i]` is parent of `i`.
- Root of `i` is `id[id[id[...id[i]...]]]`.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 8 | 9 |



root of 3 is 9
root of 5 is 6
3 and 5 are not connected

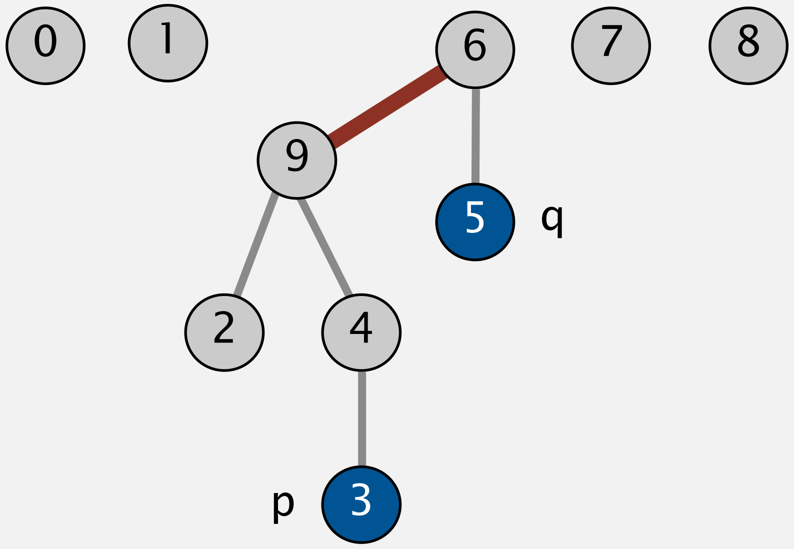
Find. What is the root of `p`?

Connected. Do `p` and `q` have the same root?

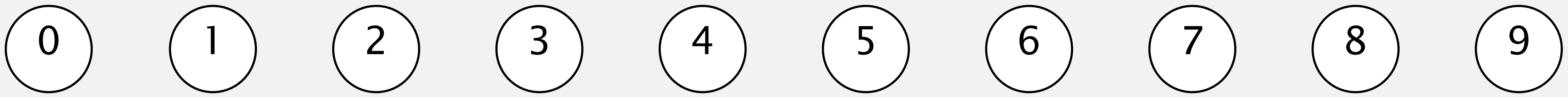
Union. To merge components containing `p` and `q`, set the id of `p`'s root to the id of `q`'s root.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 9 | 4 | 9 | 6 | 6 | 7 | 8 | 6 |

only one value changes



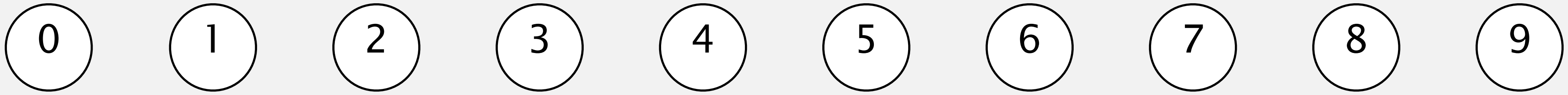
# Quick-union demo



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

# Quick-union demo

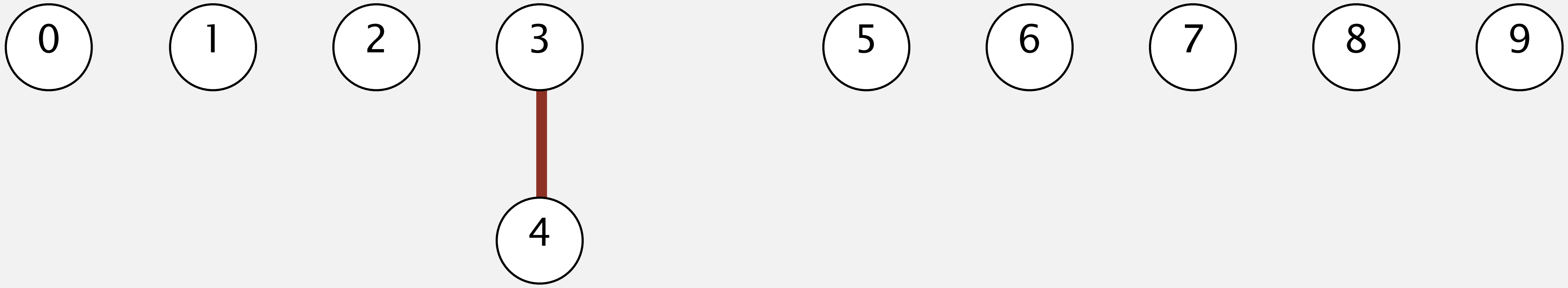
union(4, 3)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

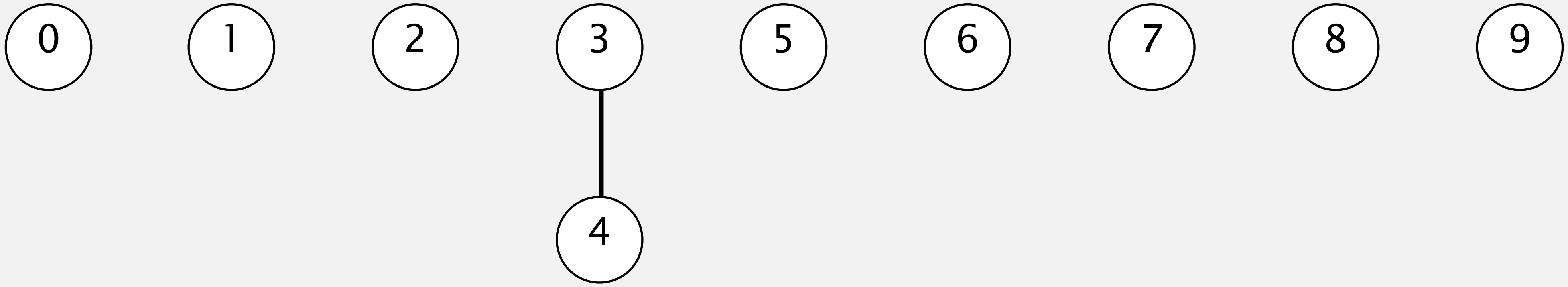
# Quick-union demo

union(4, 3)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[] | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 |

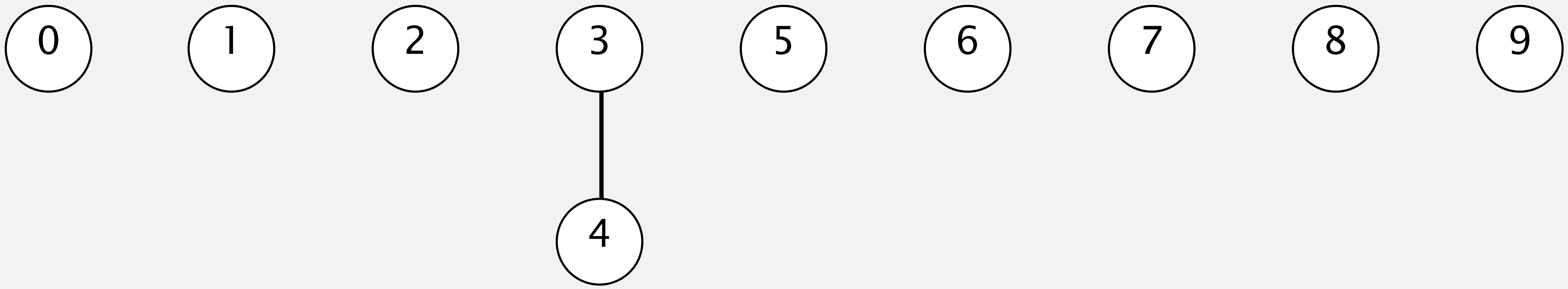
# Quick-union demo



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 |

# Quick-union demo

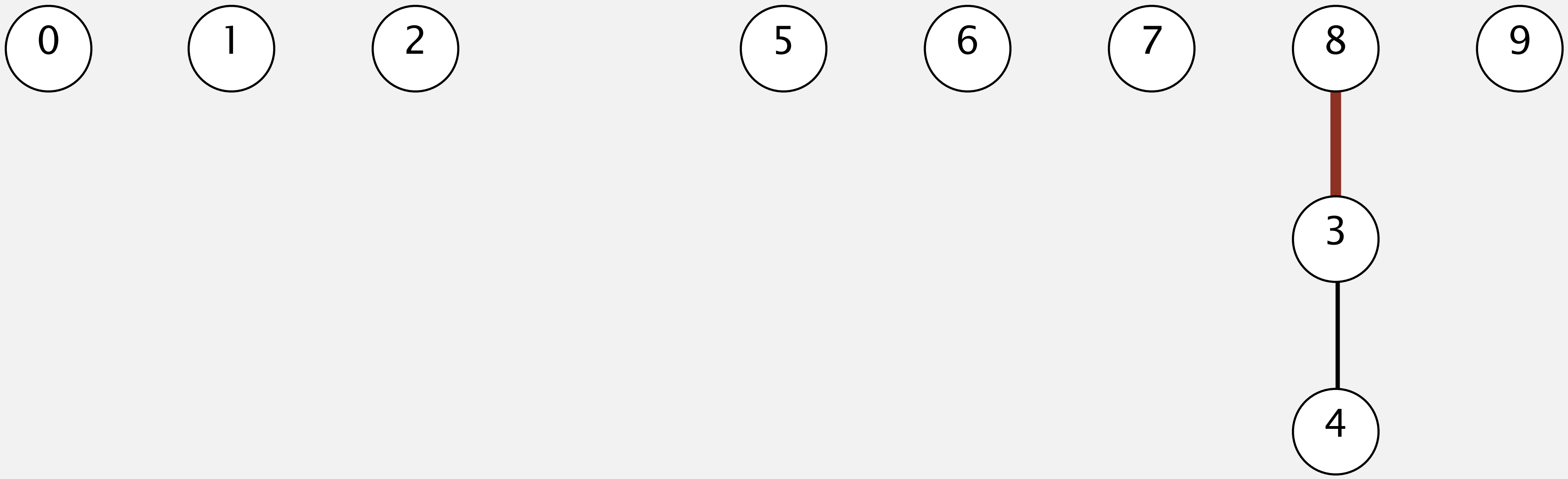
union(3, 8)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[] | 0 | 1 | 2 | 3 | 3 | 5 | 6 | 7 | 8 | 9 |

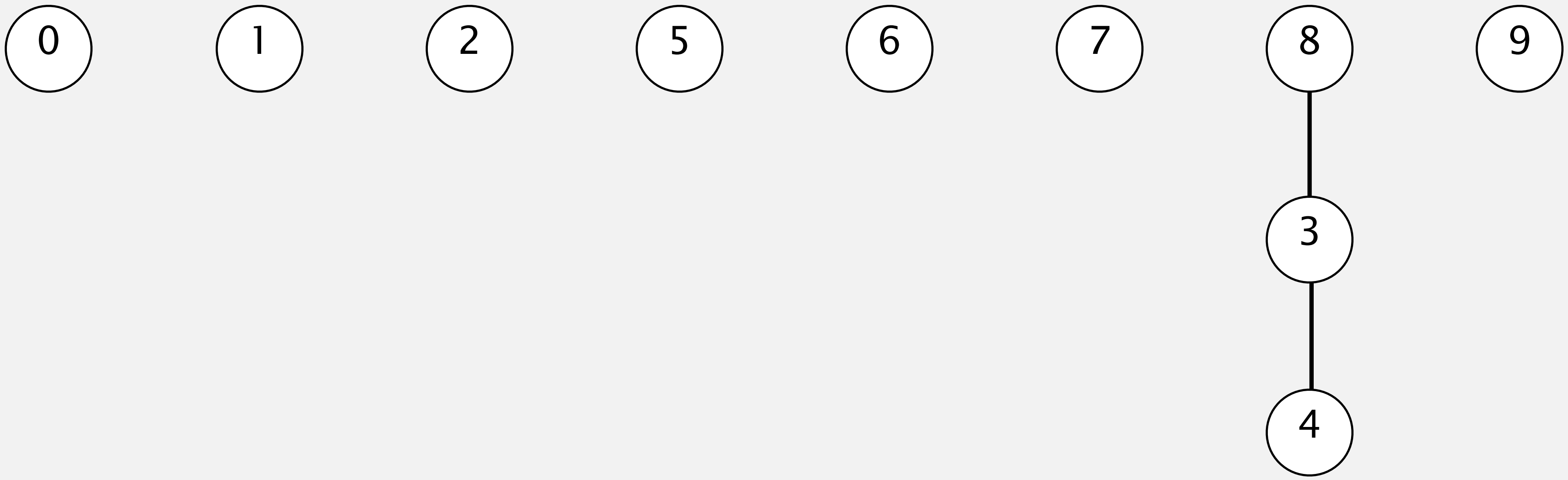
# Quick-union demo

union(3, 8)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 8 | 3 | 5 | 6 | 7 | 8 | 9 |

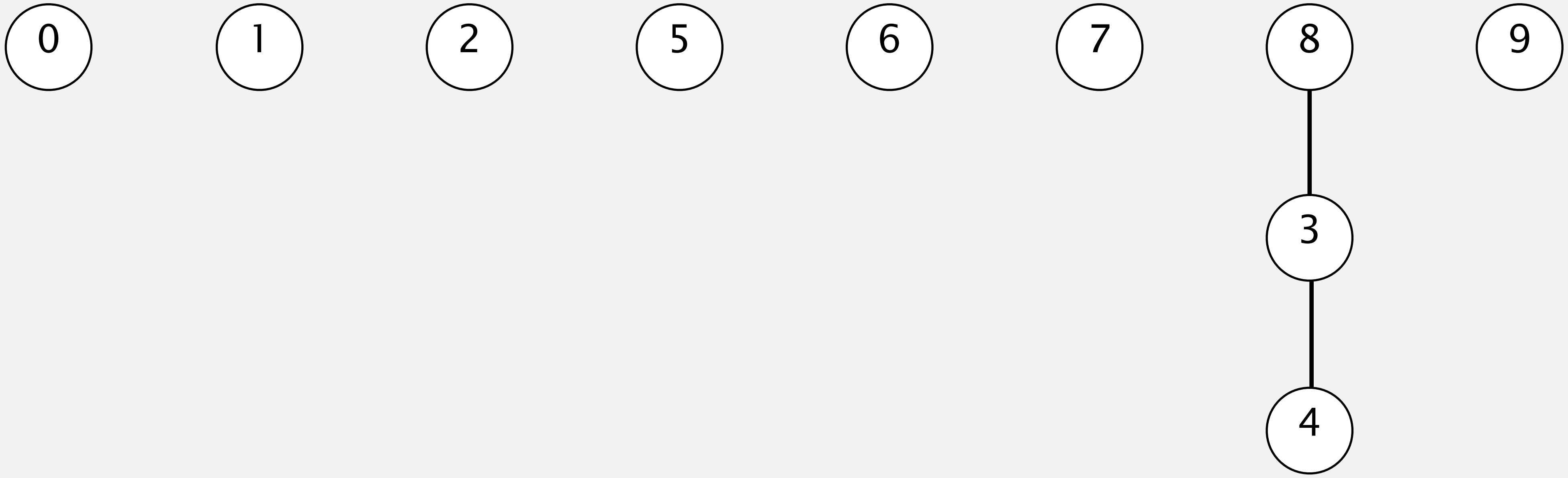
# Quick-union demo



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 8 | 3 | 5 | 6 | 7 | 8 | 9 |

# Quick-union demo

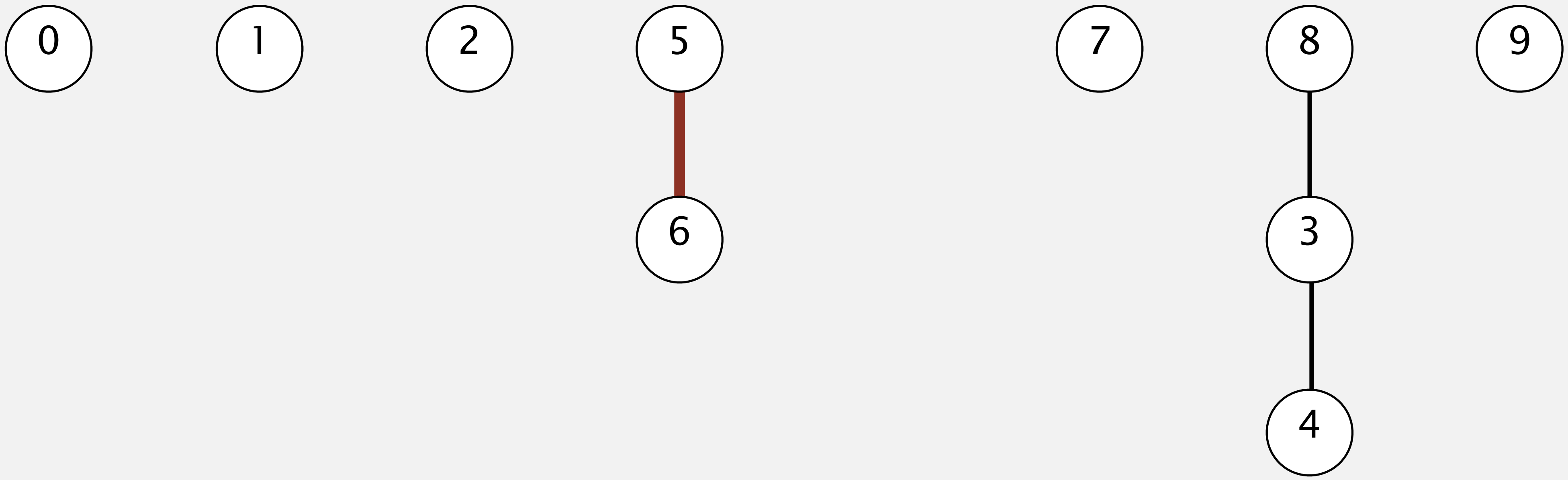
union(6, 5)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 8 | 3 | 5 | 6 | 7 | 8 | 9 |

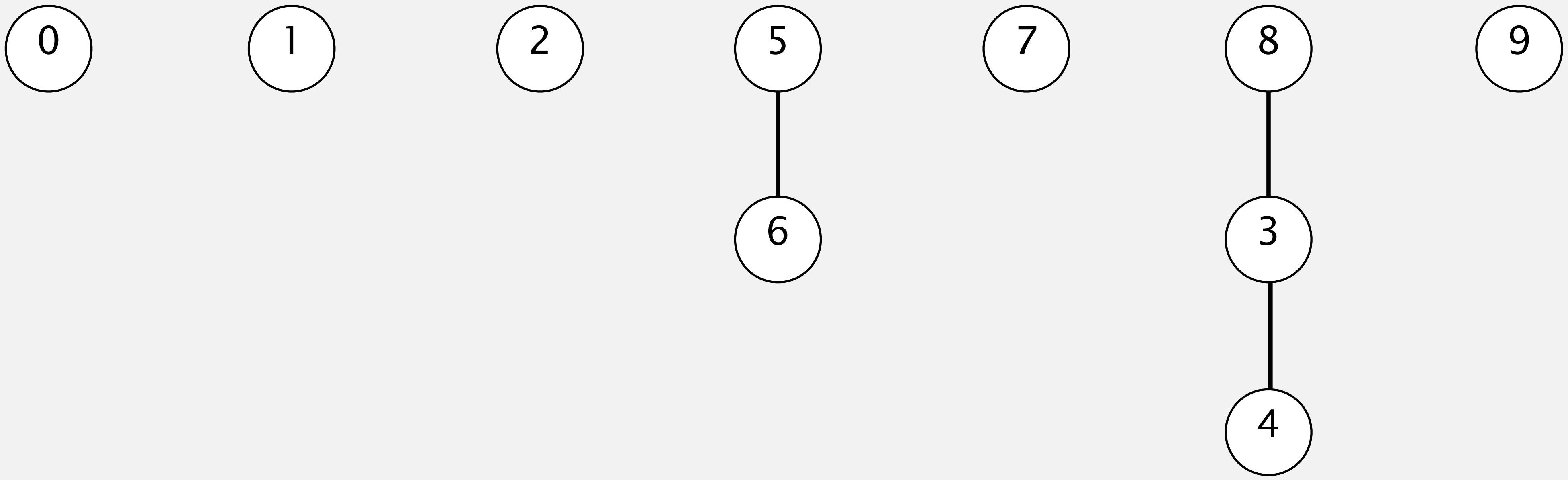
# Quick-union demo

union(6, 5)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[] | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 9 |

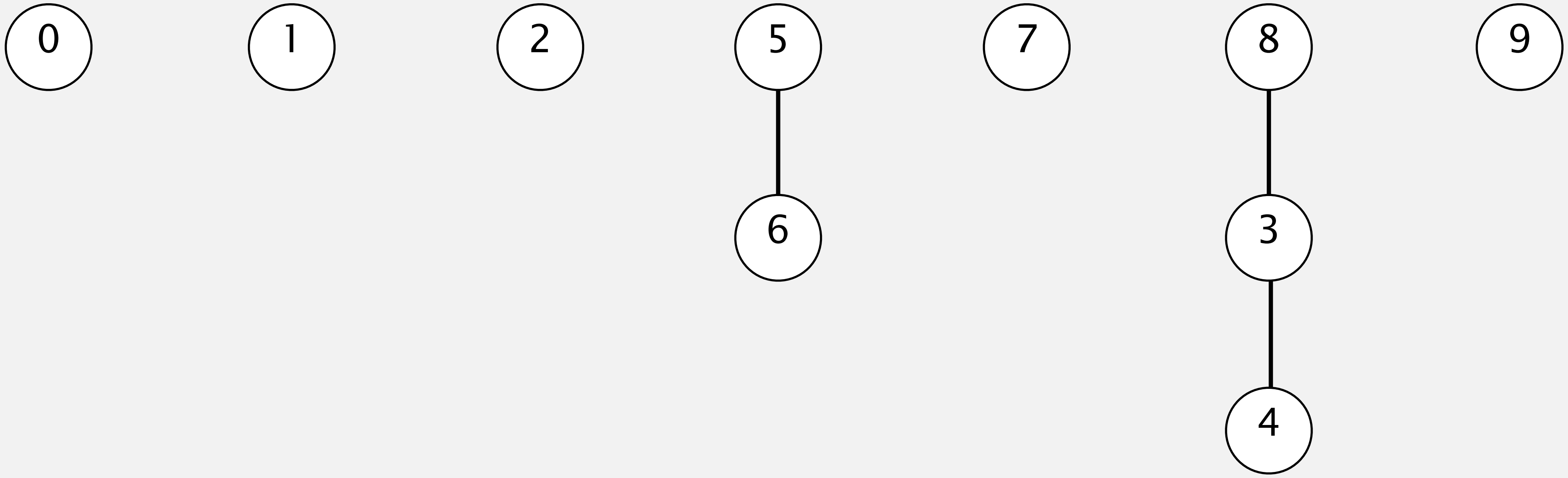
# Quick-union demo



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 9 |

# Quick-union demo

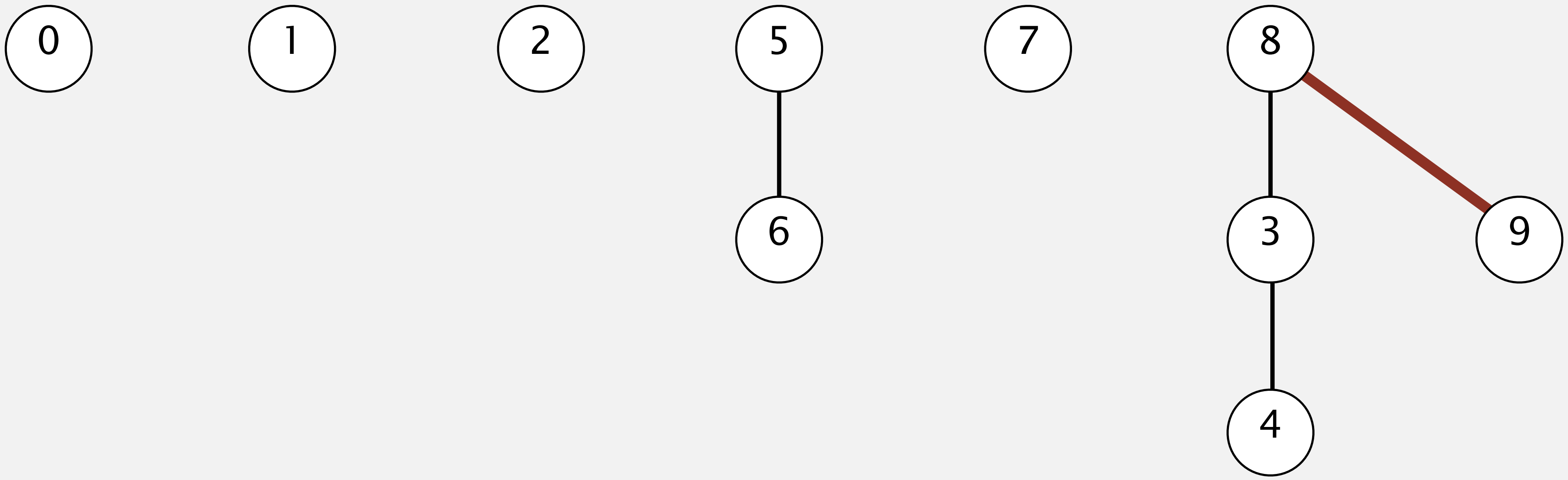
union(9, 4)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 9 |

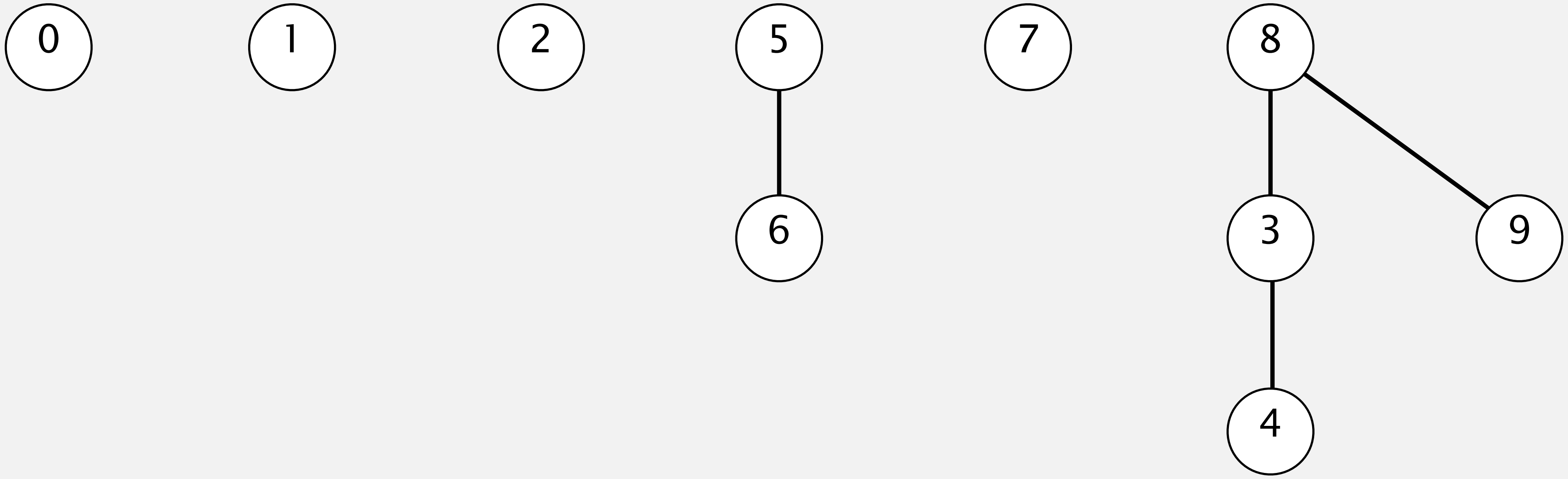
# Quick-union demo

union(9, 4)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[] | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

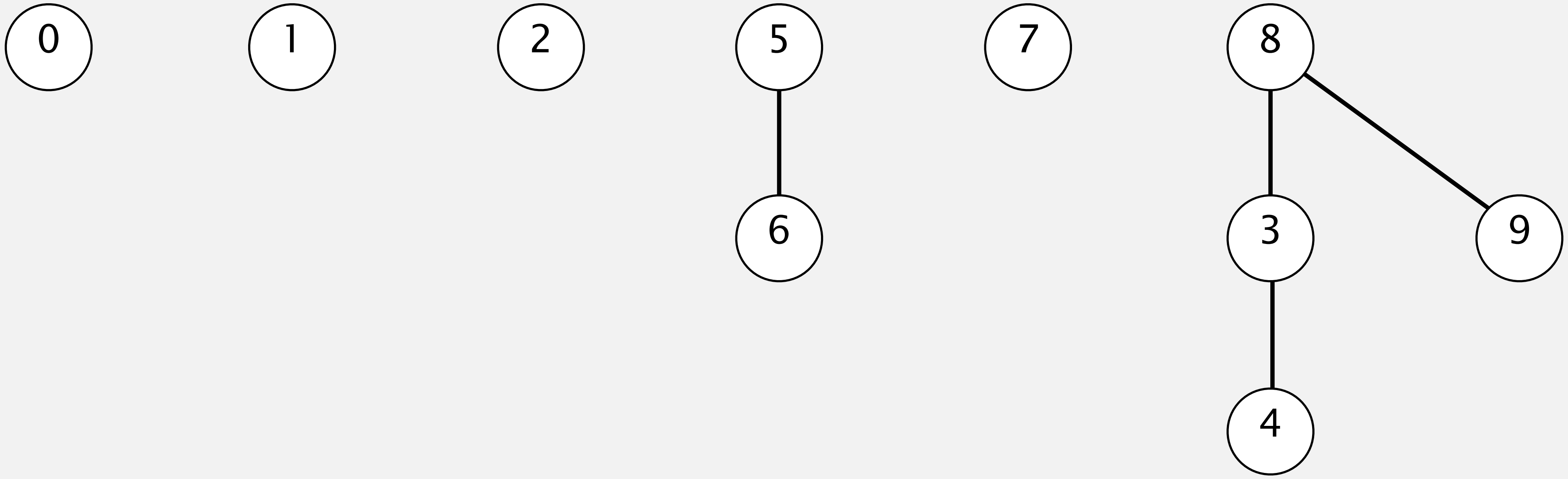
# Quick-union demo



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

# Quick-union demo

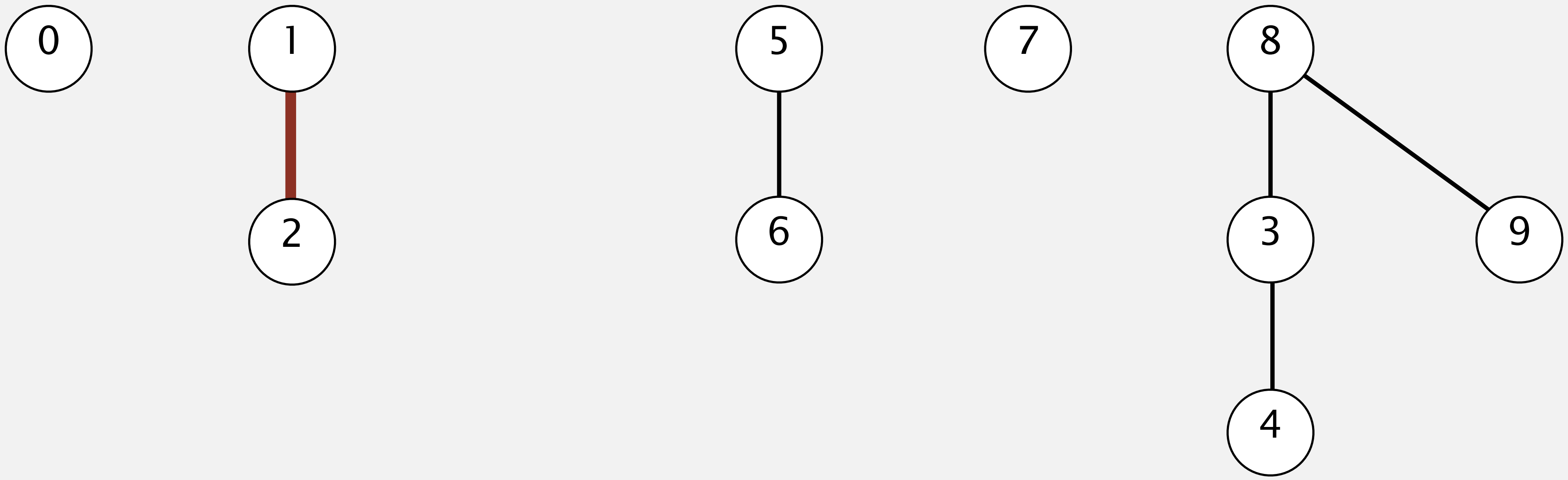
union(2, 1)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[] | 0 | 1 | 2 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

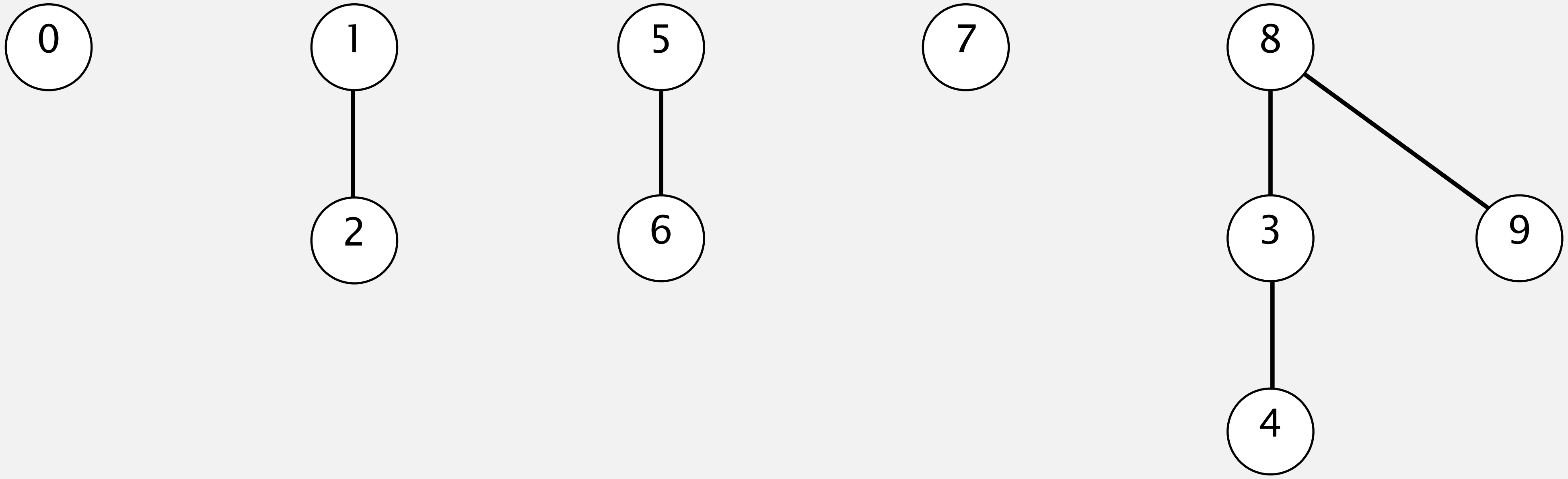
# Quick-union demo

union(2, 1)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[] | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

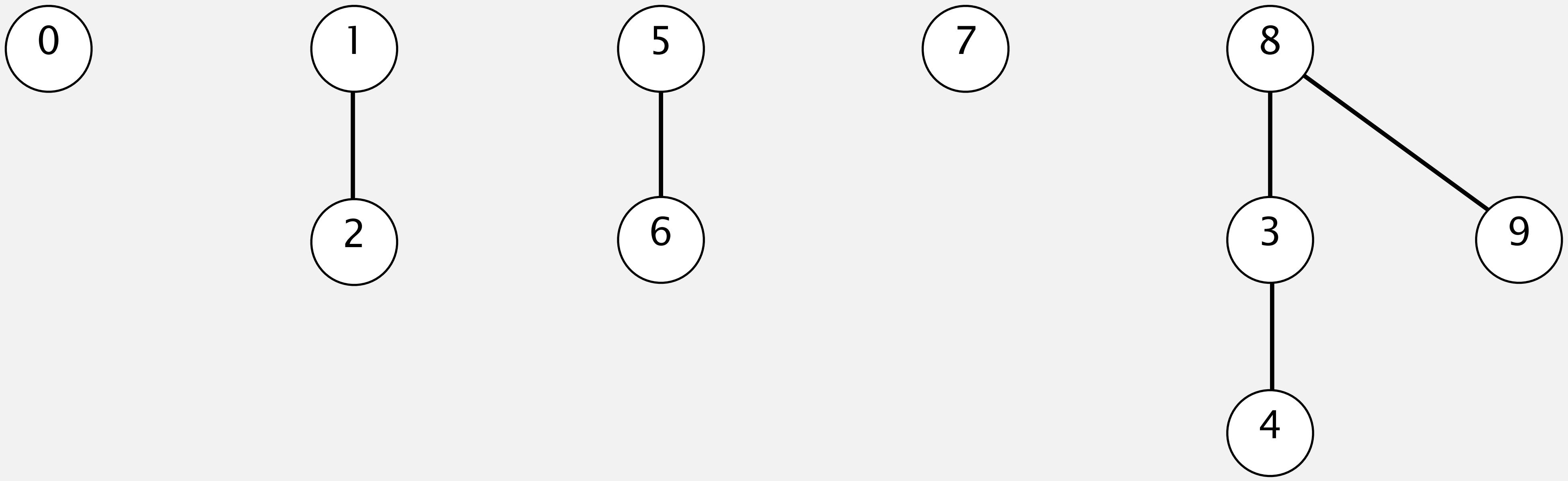
# Quick-union demo



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

# Quick-union demo

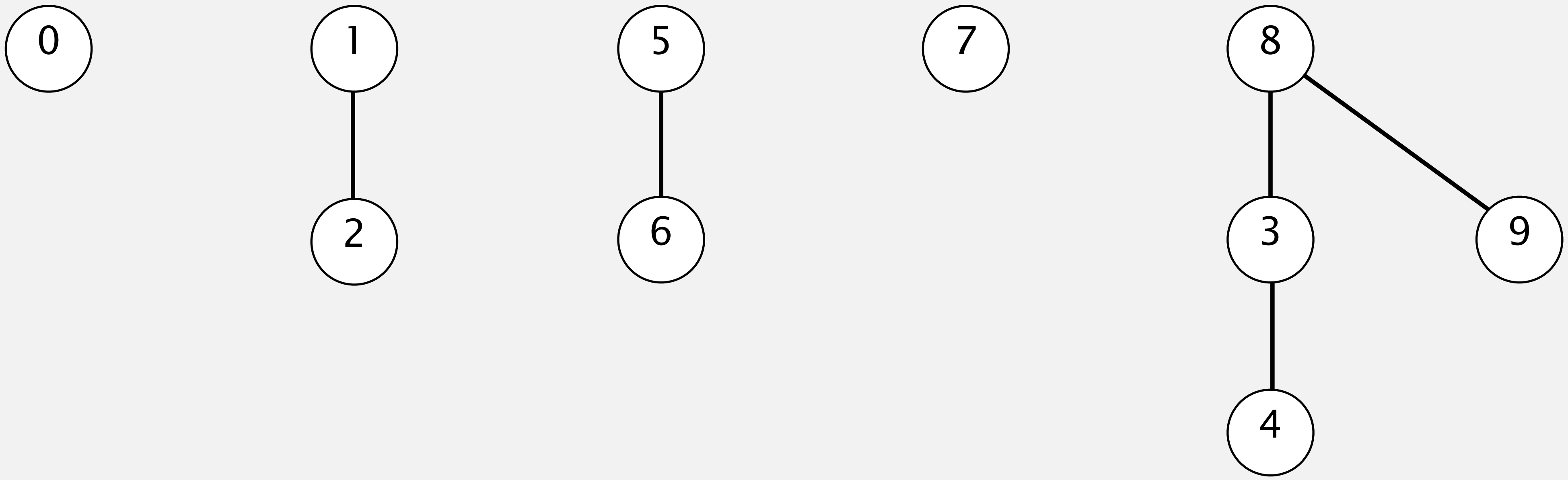
connected(8, 9) ✔



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[] | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

# Quick-union demo

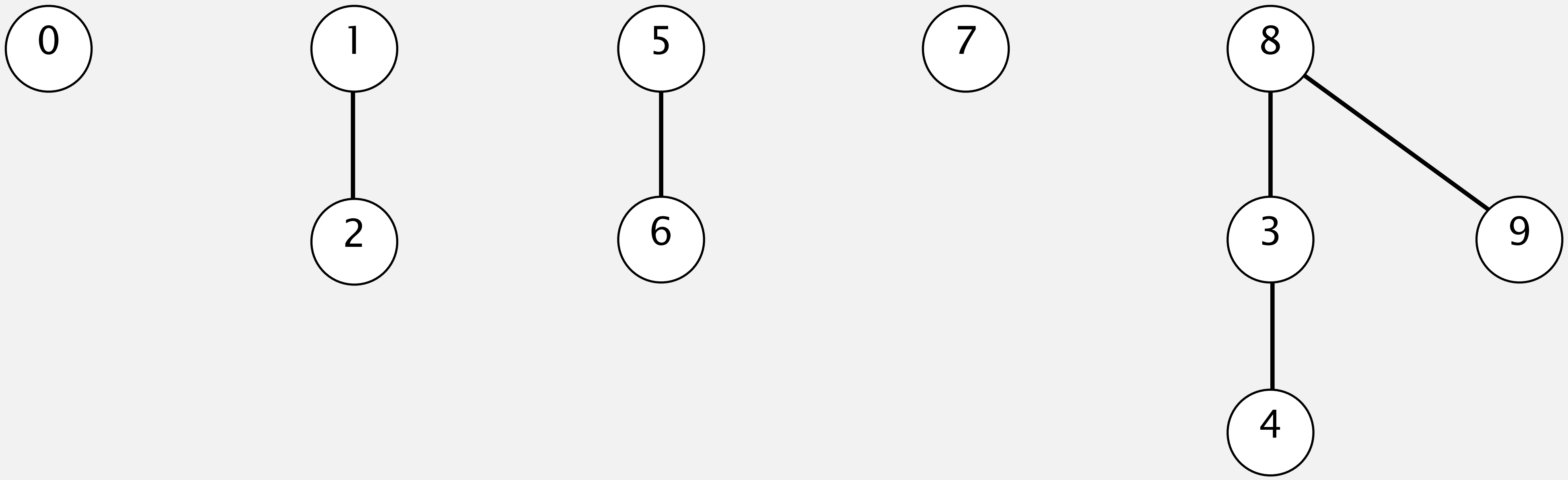
connected(5, 4) ✗



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

# Quick-union demo

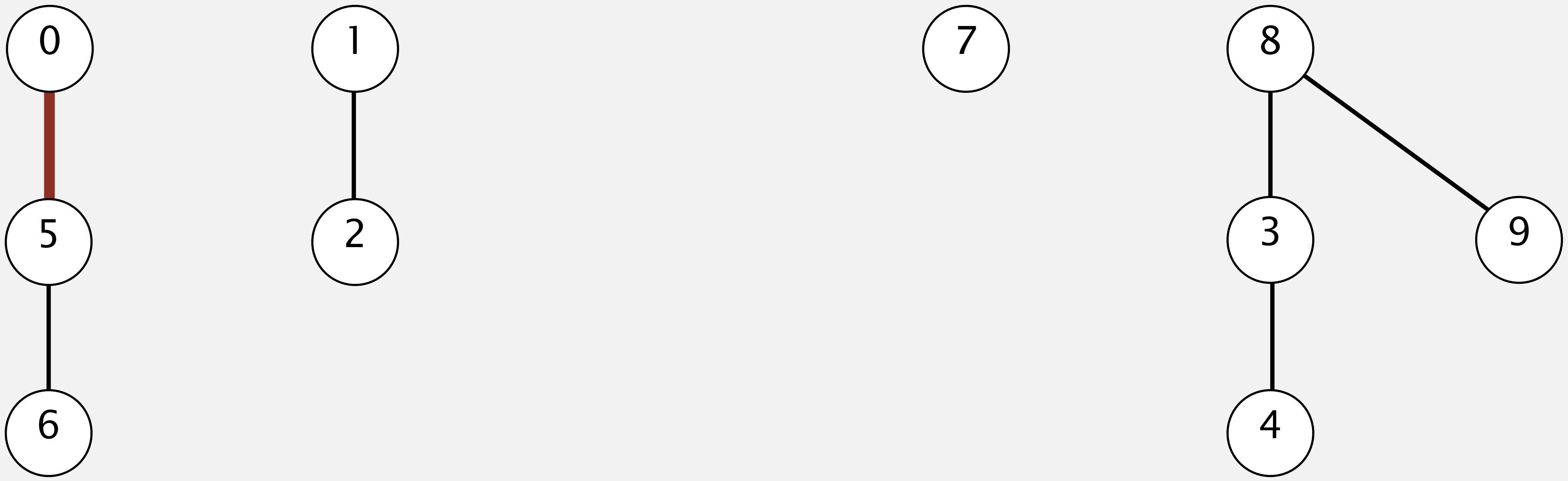
union(5, 0)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 3 | 5 | 5 | 7 | 8 | 8 |

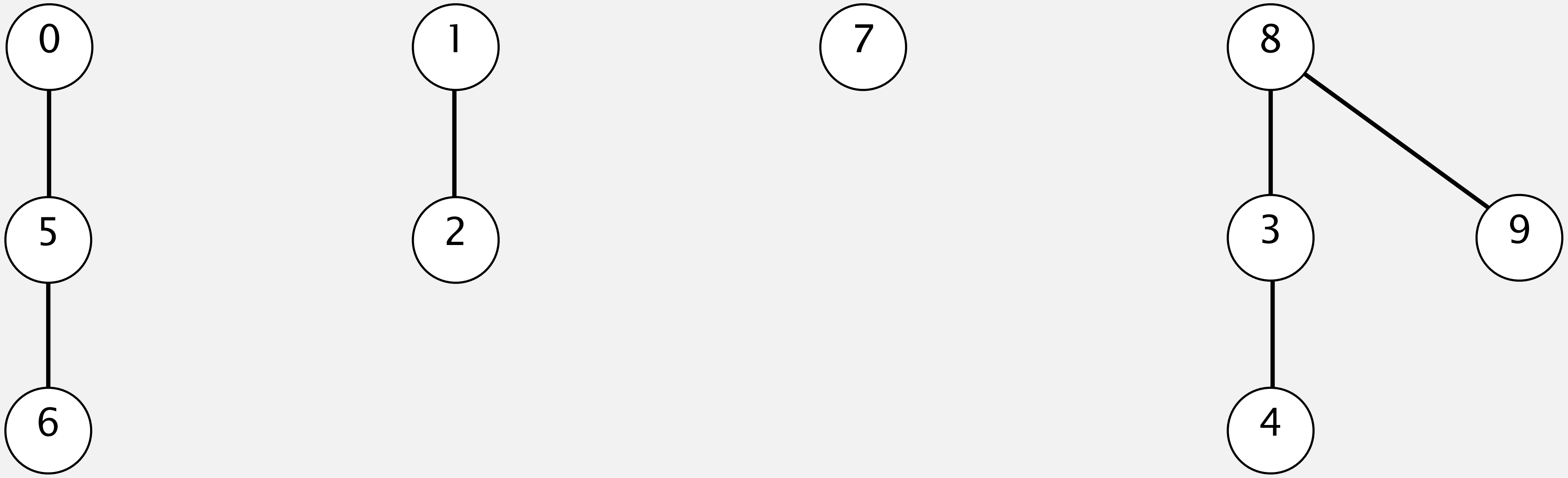
# Quick-union demo

union(5, 0)



|      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|------|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 3 | 0 | 5 | 7 | 8 | 8 |

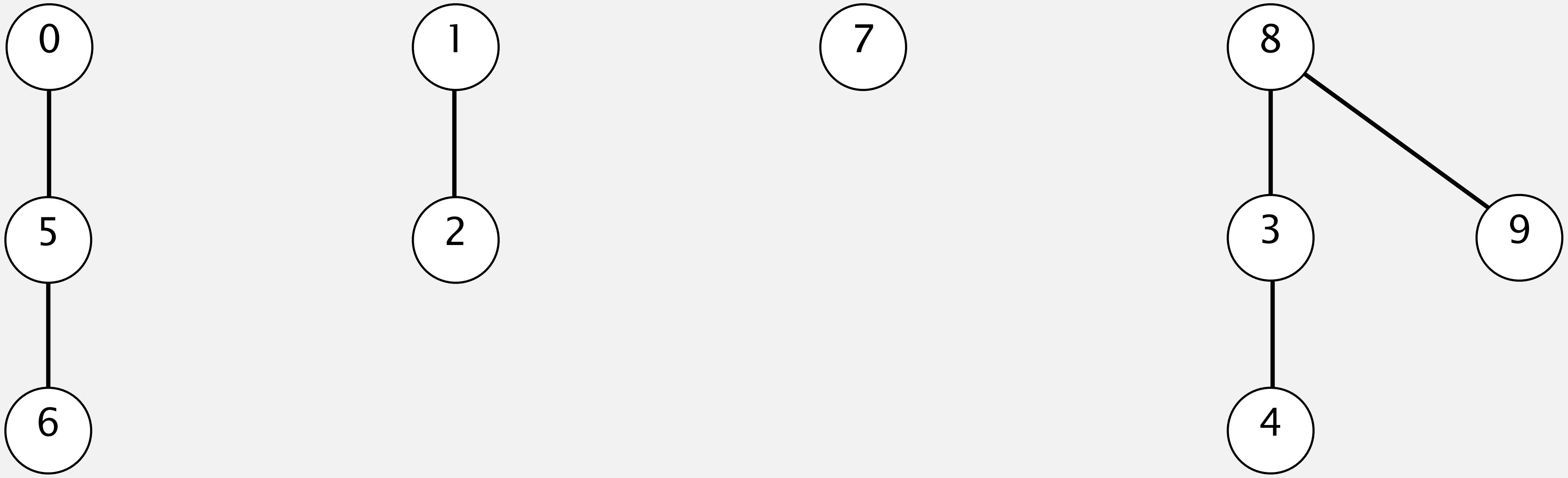
# Quick-union demo



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 3 | 0 | 5 | 7 | 8 | 8 |

# Quick-union demo

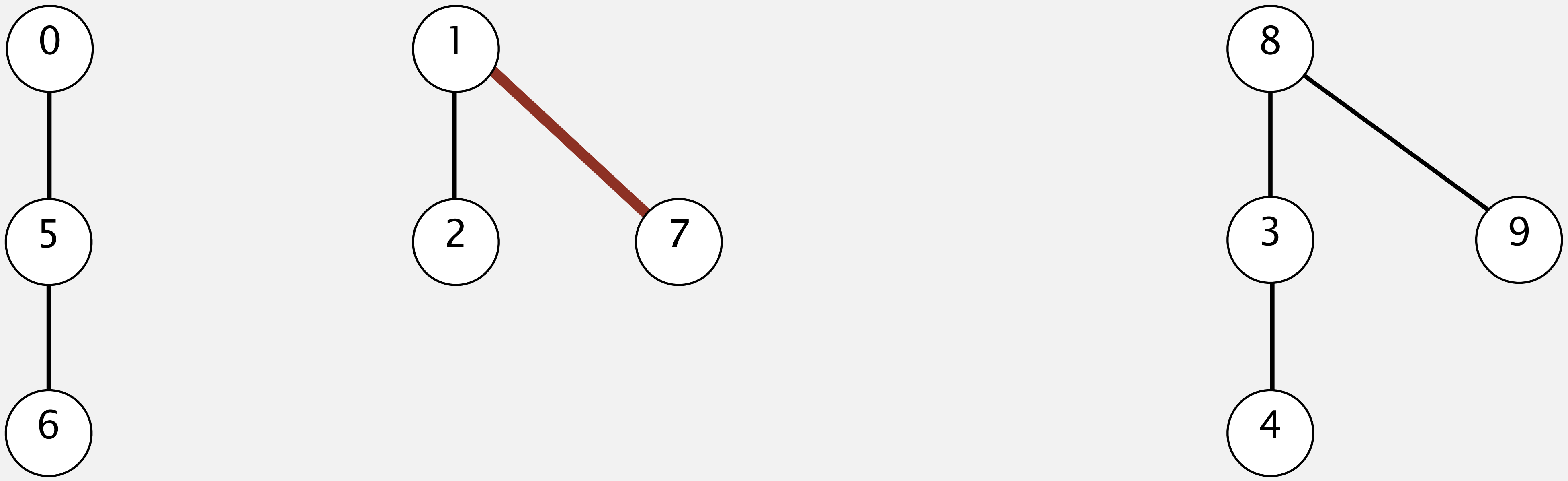
union(7, 2)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[] | 0 | 1 | 1 | 8 | 3 | 0 | 5 | 7 | 8 | 8 |

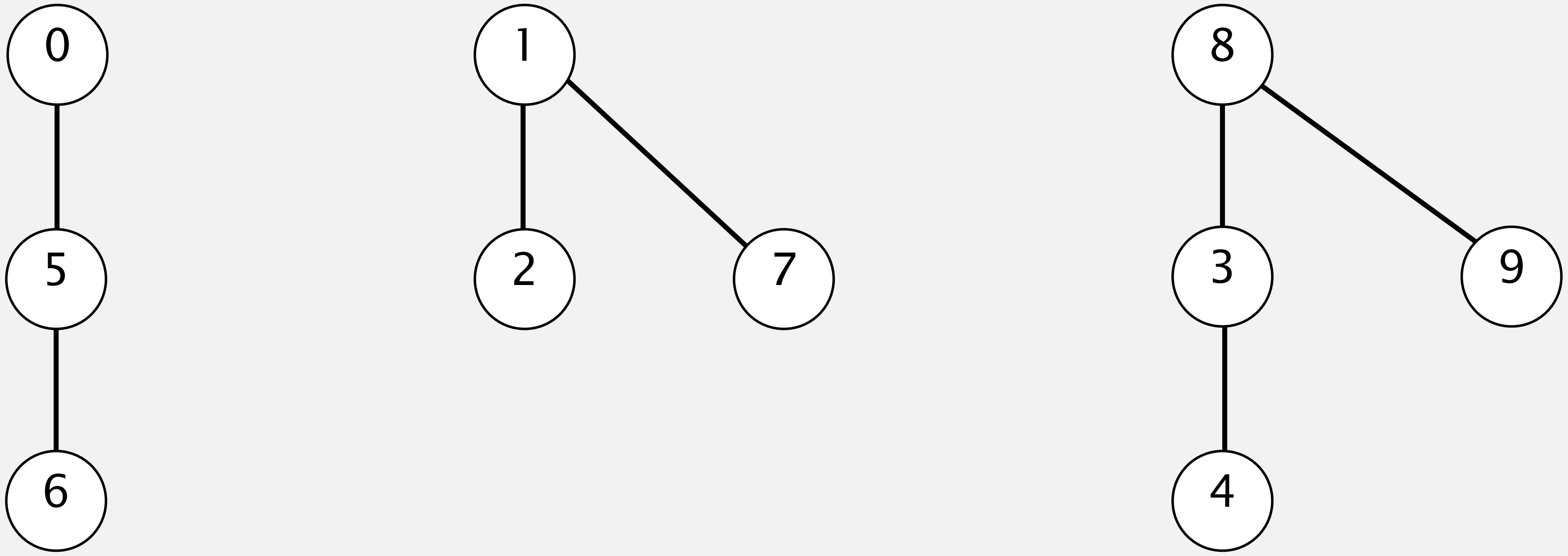
# Quick-union demo

union(7, 2)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

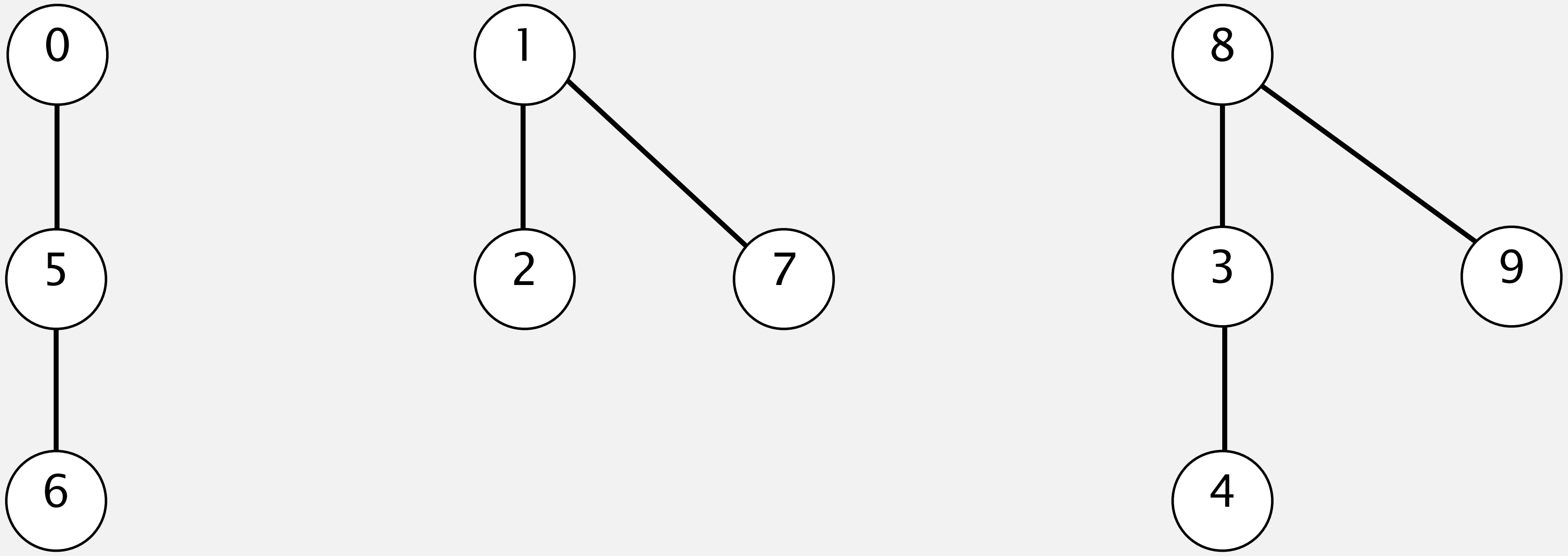
# Quick-union demo



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

# Quick-union demo

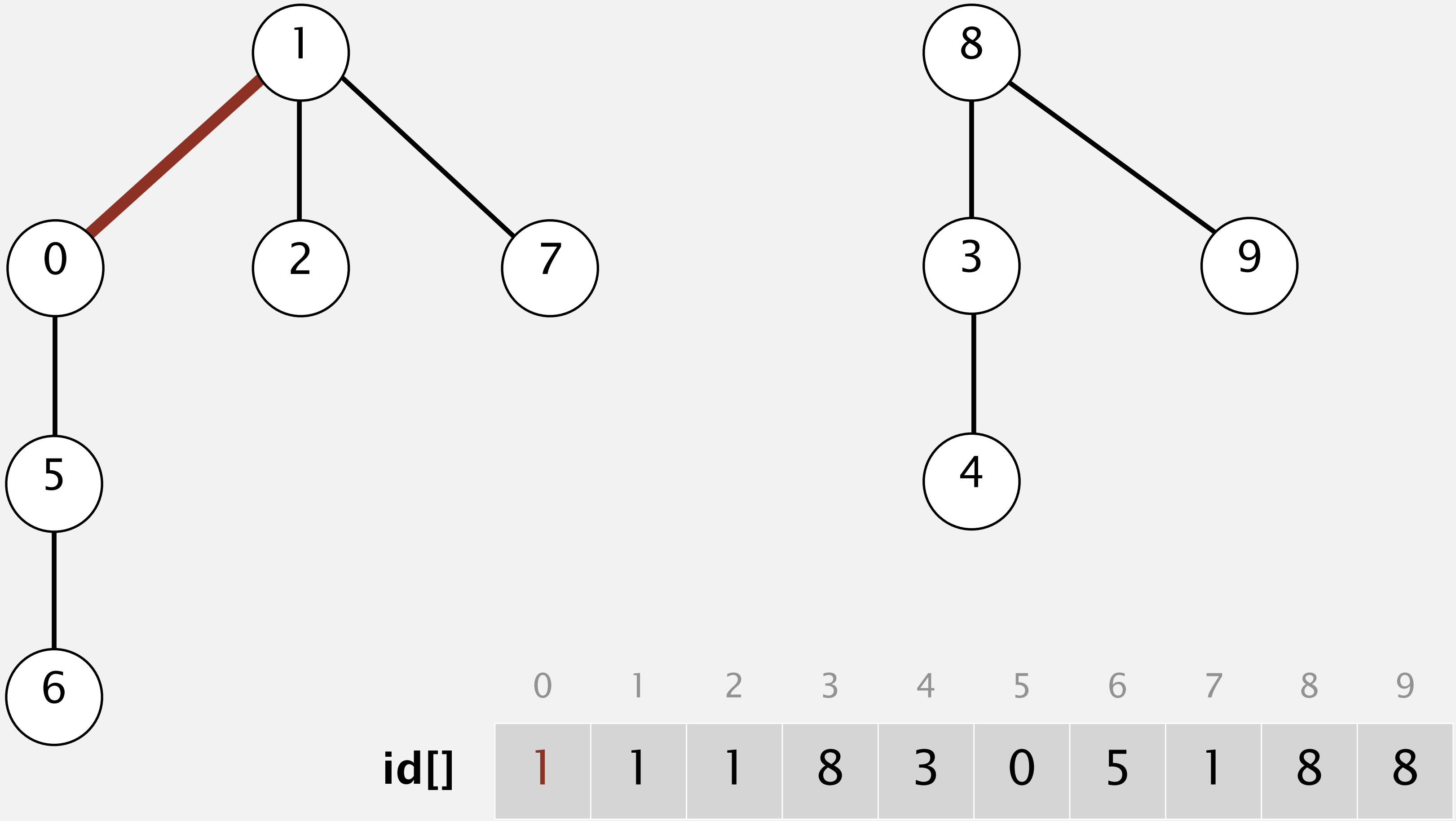
union(6, 1)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 0 | 1 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

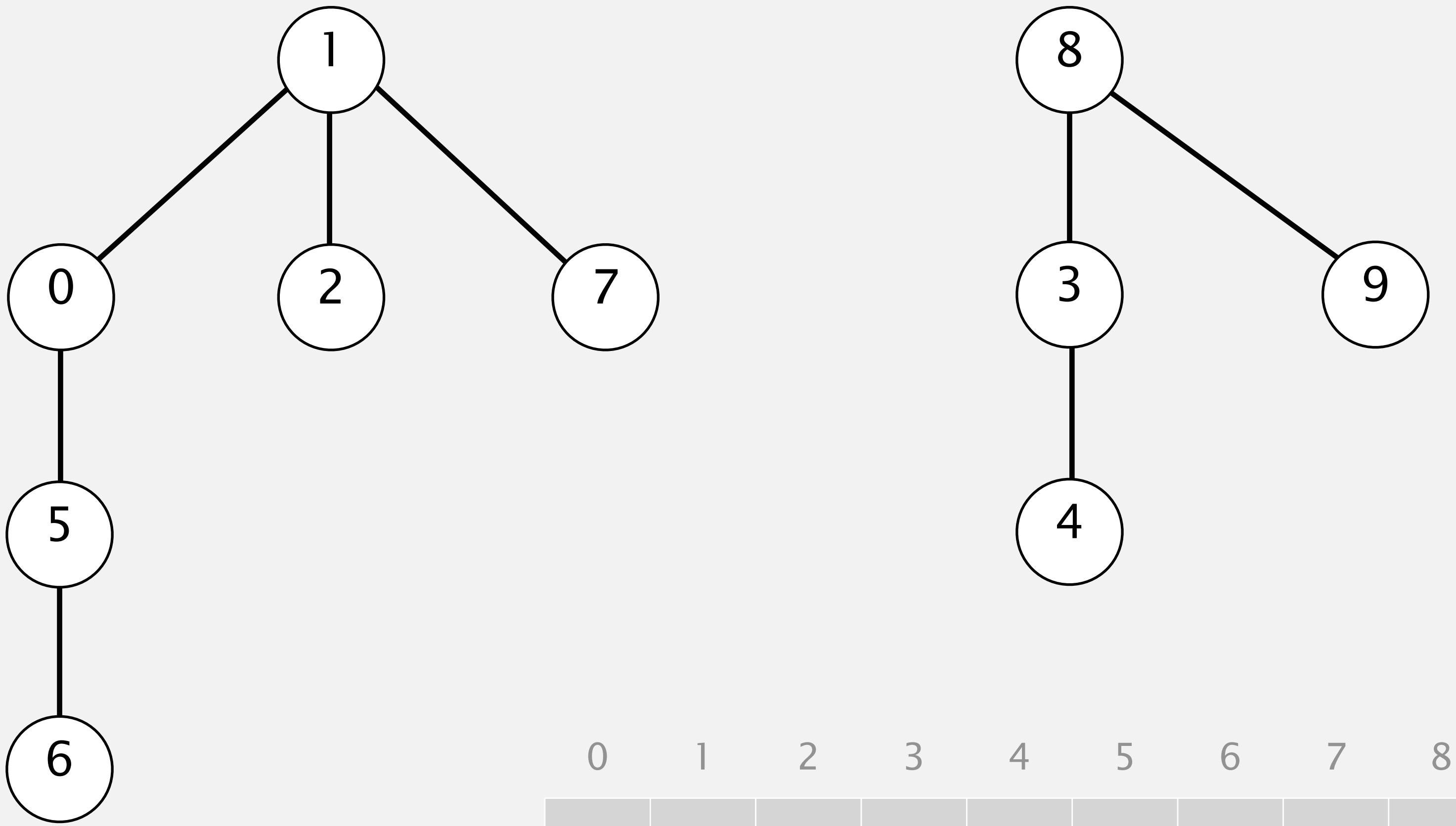
# Quick-union demo

union(6, 1)



| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

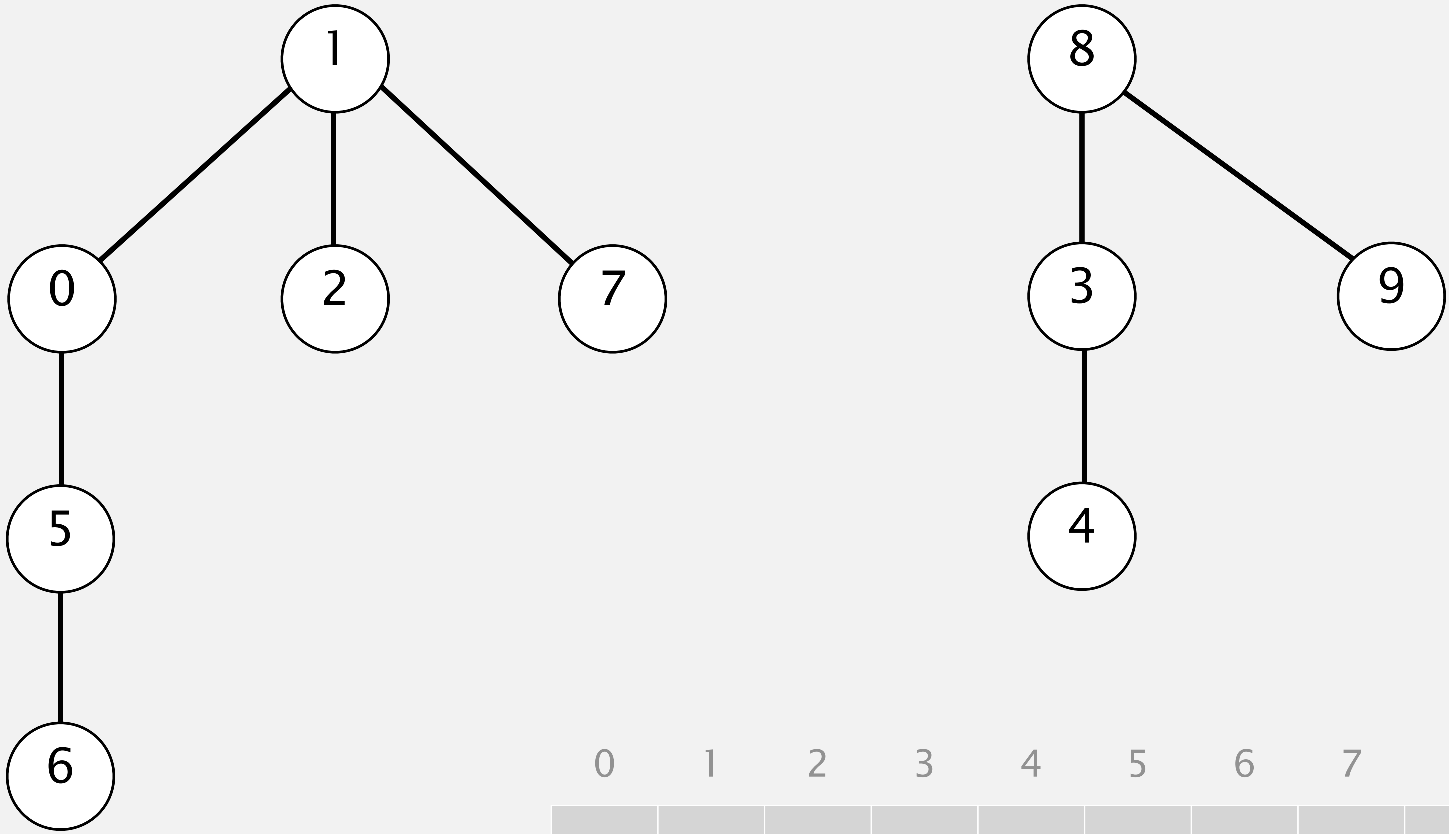
# Quick-union demo



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

# Quick-union demo

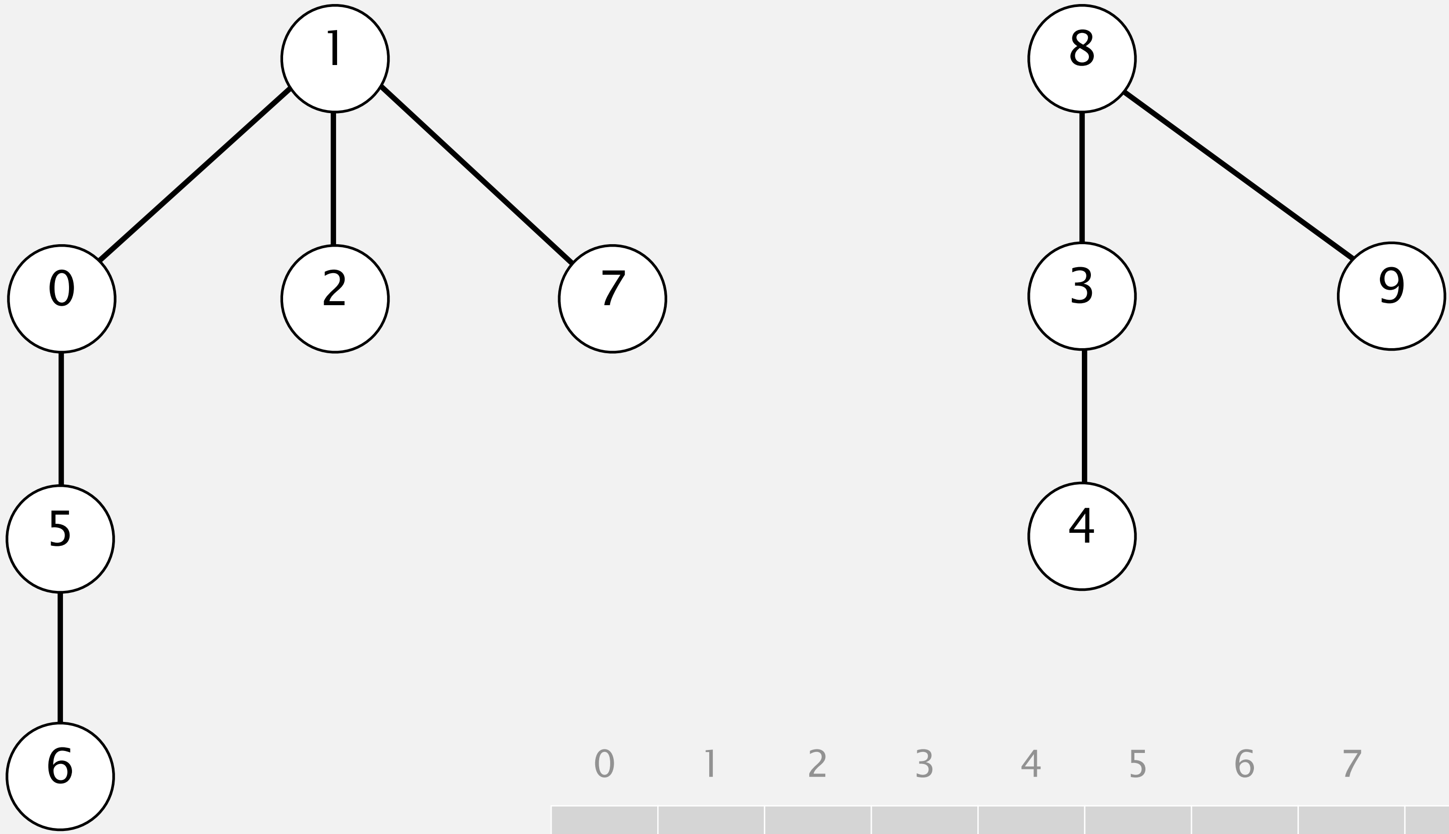
connected(1, 0)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

# Quick-union demo

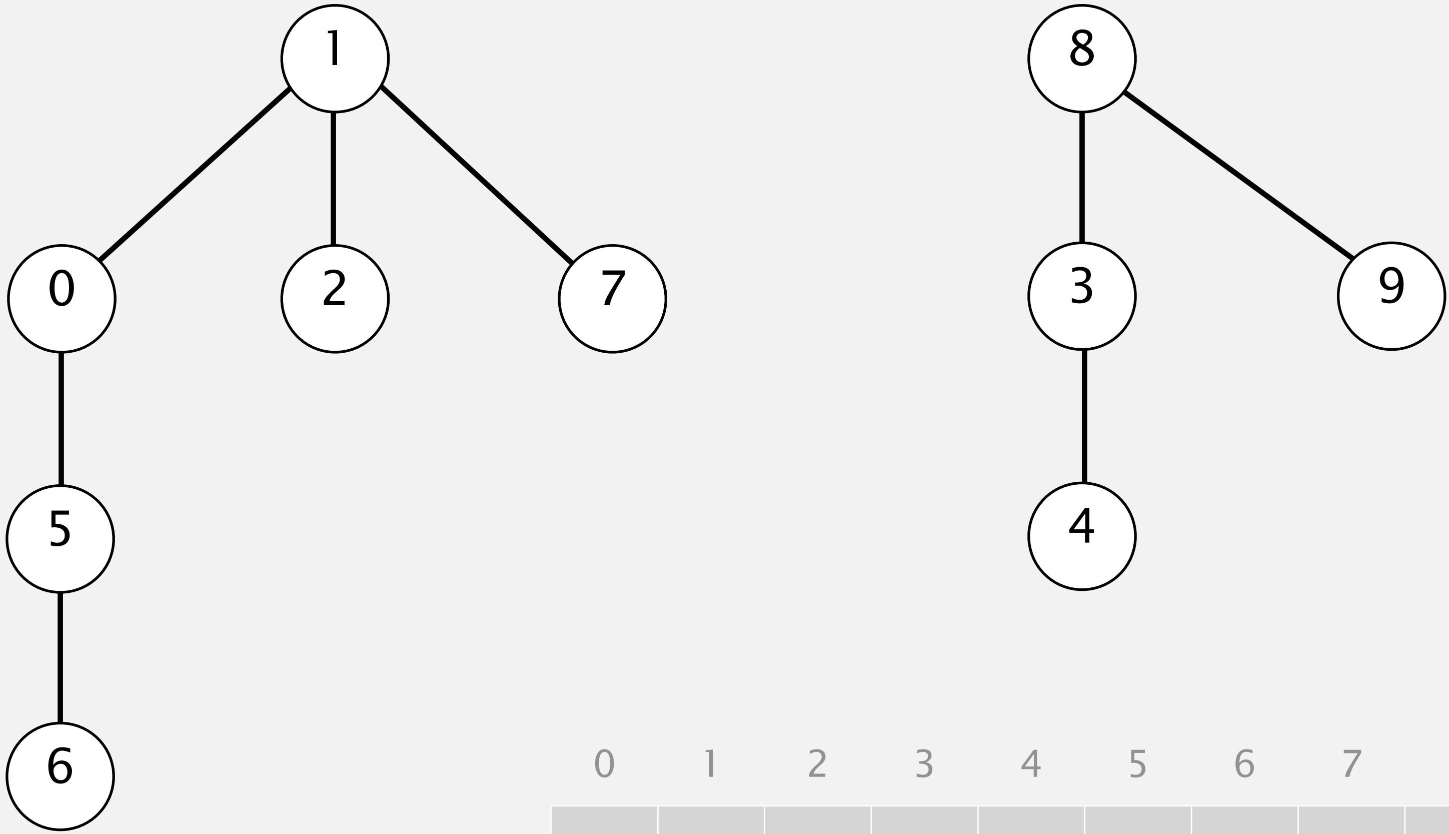
connected(6, 7)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

# Quick-union demo

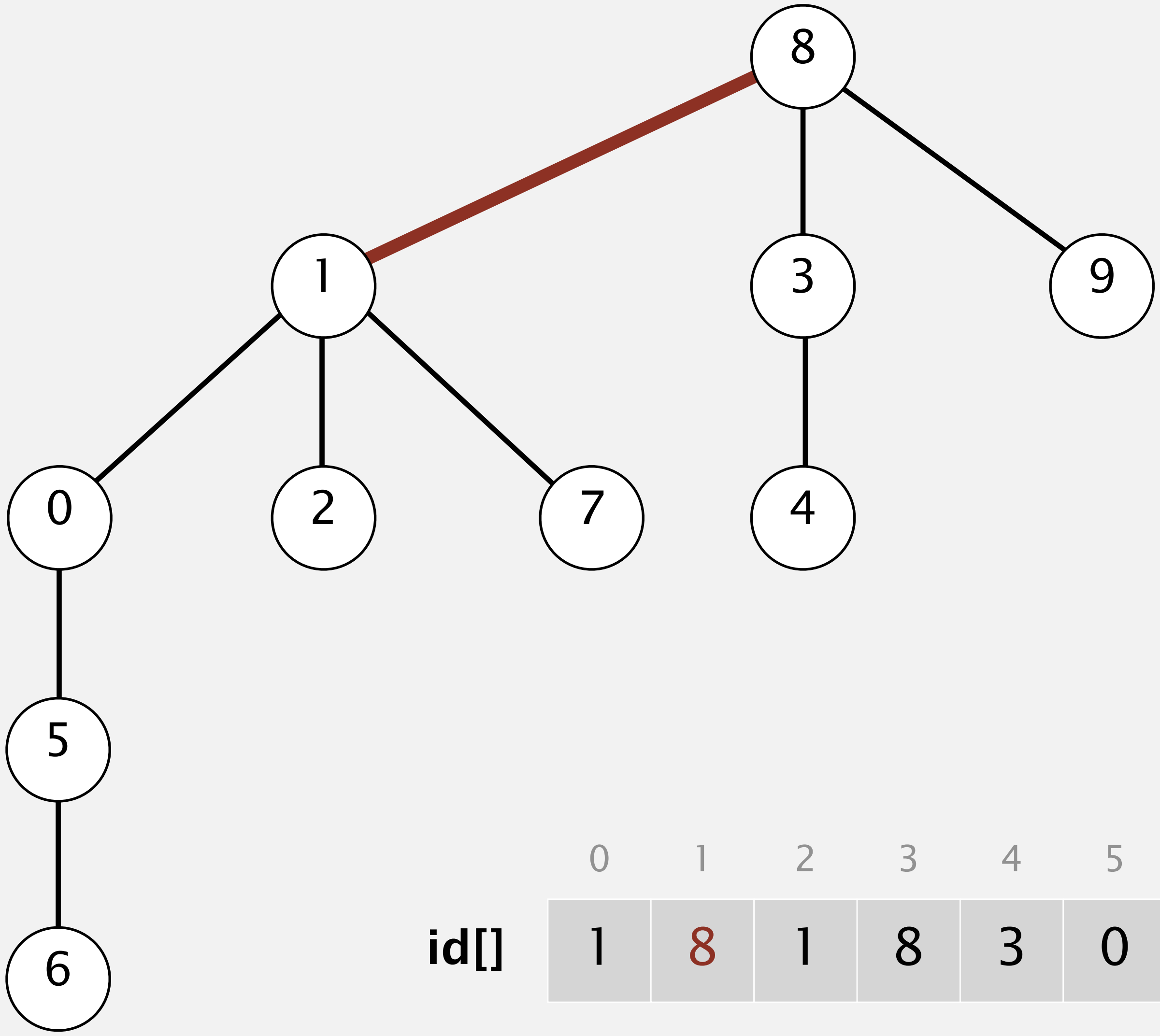
union(7, 3)



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 1 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

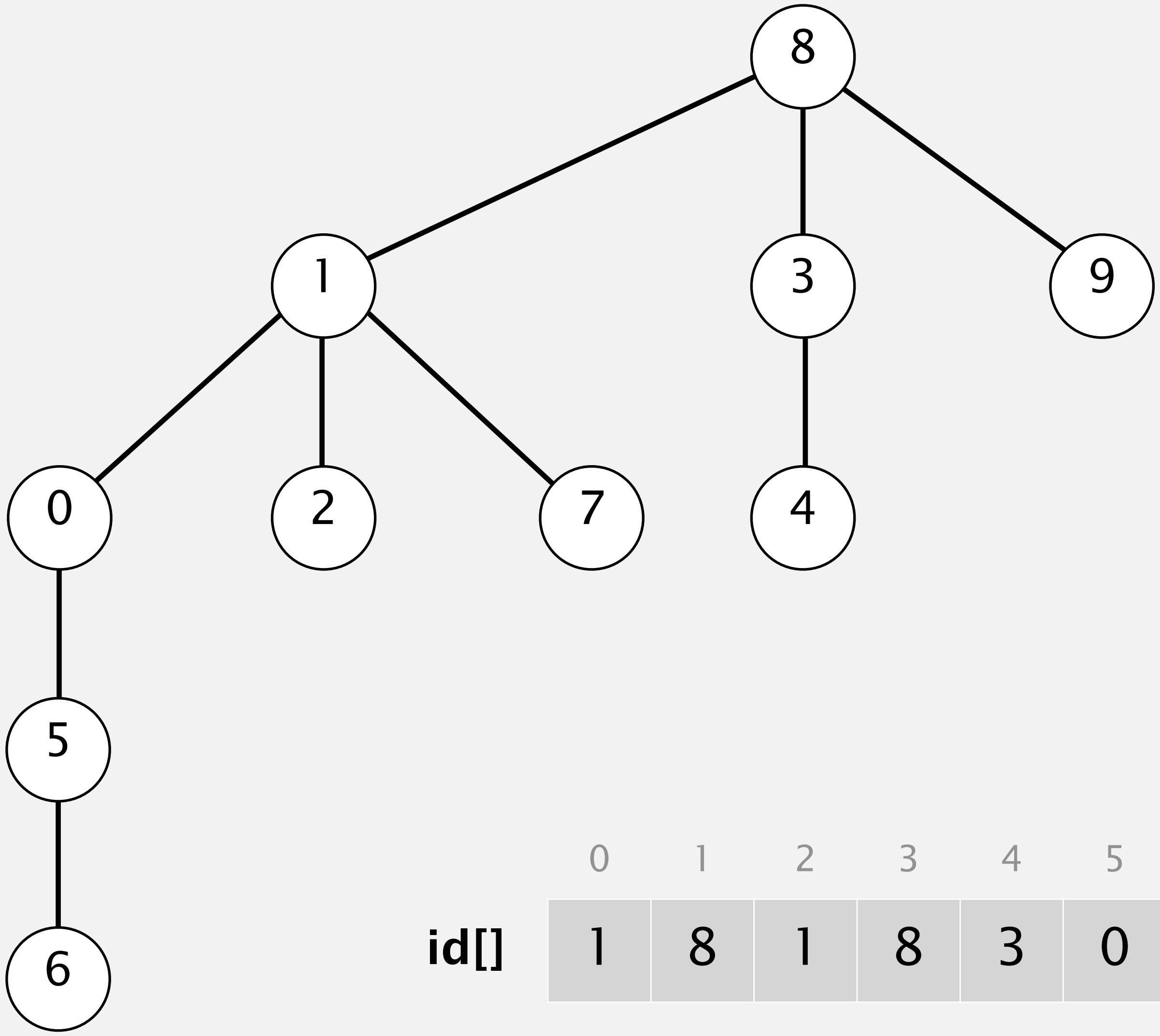
# Quick-union demo

union(7, 3)



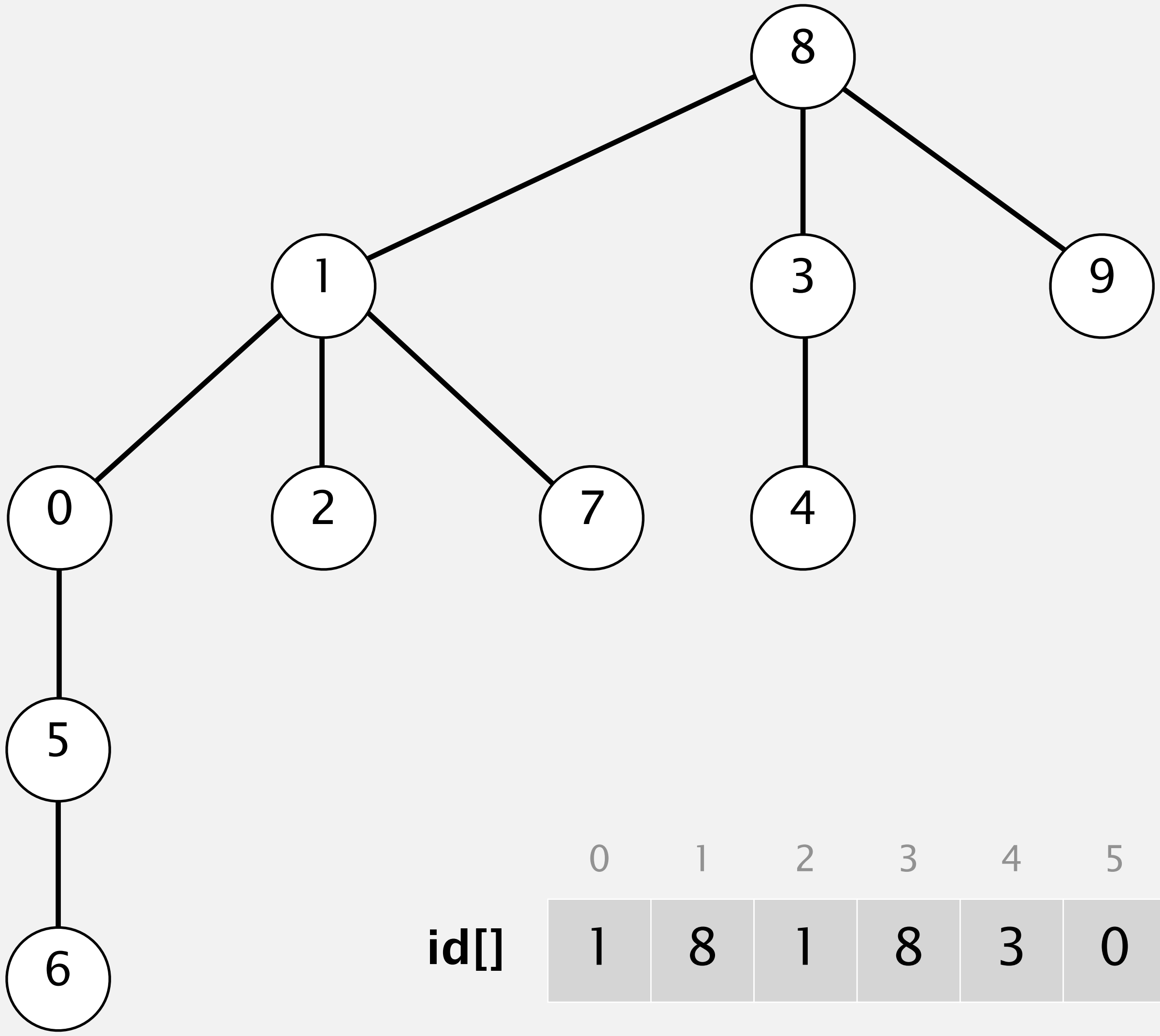
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| id[] | 1 | 8 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

# Quick-union demo



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 8 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

# Quick-union demo



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **id[]** | 1 | 8 | 1 | 8 | 3 | 0 | 5 | 1 | 8 | 8 |

# Quick-union:  Java implementation

```
public class QuickUnionUF
{
   private int[] id;

   public QuickUnionUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   public int find(int i)
   {
      What to write here? 3 mins.
   }
















}
```

set id of each object to itself
(N array accesses)

chase parent pointers until reach root
(depth of i array accesses)

# Quick-union: Java implementation

```
public class QuickUnionUF
{
   private int[] id;

   public QuickUnionUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   public int find(int i)
   {
      while (i != id[i]) i = id[i];
      return i;
   }

   public boolean connected(int p, int q)
   {
      return find(p) == find(q);
   }









}
```

set id of each object to itself
(N array accesses)

chase parent pointers until reach root
(depth of i array accesses)

do p and q have the same root?
(depth of p and q array accesses)

# Quick-union: Java implementation

```java
public class QuickUnionUF
{
   private int[] id;

   public QuickUnionUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   public int find(int i)
   {
      while (i != id[i]) i = id[i];
      return i;
   }

   public boolean connected(int p, int q)
   {
      return find(p) == find(q);
   }

   public void union(int p, int q)
   {
      What to write here? 2 mins.
   }
}
```

set id of each object to itself
(N array accesses)

chase parent pointers until reach root
(depth of i array accesses)

do p and q have the same root?
(depth of p and q array accesses)

change root of p to point to root of q
(depth of p and q array accesses)

# Quick-union: Java implementation

```java
public class QuickUnionUF
{
   private int[] id;

   public QuickUnionUF(int N)
   {
      id = new int[N];
      for (int i = 0; i < N; i++) id[i] = i;
   }

   public int find(int i)
   {
      while (i != id[i]) i = id[i];
      return i;
   }

   public boolean connected(int p, int q)
   {
      return find(p) == find(q);
   }

   public void union(int p, int q)
   {
      int i = find(p);
      int j = find(q);
      id[i] = j;
   }
}
```

set id of each object to itself
(N array accesses)

chase parent pointers until reach root
(depth of i array accesses)

do p and q have the same root?
(depth of p and q array accesses)

change root of p to point to root of q
(depth of p and q array accesses)

# Quick-union is also too slow

Cost model. Number of array accesses (for read or write).

| algorithm | initialize | union | find | connected |
|---|---|---|---|---|
| **quick-find** | N | N | 1 | 1 |
| **quick-union** | N | N † | N | N |

worst case

† includes cost of finding roots

Quick-find defect.

- Union too expensive ($N$ array accesses).
- Trees are flat, but too expensive to keep them flat.

Quick-union defect.

- Trees can get tall.
- Find/connected too expensive (could be $N$ array accesses).